CSE443
Compilers

Dr. Carl Alphonse
alphonce@buffalo.edu
343 Davis Hall
exercises

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

How will you modify your grammar rules to generate intermediate code for function calls?

How will you modify your grammar rules to generate intermediate code for the switch/case statement?
What did you come up with?

- How will you modify your grammar rules to generate intermediate code for function calls?

- How will you modify your grammar rules to generate intermediate code for the switch/case statement?
previous semester exercise outcome

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

Target machine code generation

Figure 1.6, page 5 of text
Memory organization

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

<table>
<thead>
<tr>
<th>Memory Type</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>

- **code**: machine language instructions of the program
- **static**: free memory
- **heap**: stack
Memory organization

- code
- static
- heap
- free memory
- stack

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

code

static

heap

dynamically allocated memory (e.g. records, arrays)

free memory

stack
Memory organization

- code
- static
- heap
- free memory
- stack

heap grows towards stack
Memory organization

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

'Free memory' denotes the unallocated memory between heap and stack.
Memory organization

- code
- static
- heap
- free memory
- stack

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

- code
- static
- heap
- free memory
- stack grows towards heap
- stack
The size, layout and contents of both the code and static regions are determined at compile time.
Memory organization

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

- code
- static
- heap
- free memory
- stack

Heap allocation: reserve & release
Memory organization

Stack allocation:
function call
## Stack frame organization

<table>
<thead>
<tr>
<th>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>
# Stack frame organization

<table>
<thead>
<tr>
<th>Actual parameters</th>
<th>Returned value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(arguments)</td>
<td></td>
</tr>
<tr>
<td>Control link</td>
<td>Access link</td>
</tr>
<tr>
<td>(dynamic link)</td>
<td>(static link)</td>
</tr>
<tr>
<td>Saved machine status</td>
<td>(return address)</td>
</tr>
<tr>
<td>Local data</td>
<td>Temporaries</td>
</tr>
</tbody>
</table>

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>

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

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

The address of the caller's invocation record (stack frame).
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>

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 Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual parameters (arguments)</td>
<td>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).</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

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

Space for local variables.
## 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 temporary variables, and variable-length local data
- Temporaries may be in CPU registers.
7.2.3 Calling Sequence

What happens during a function call?
caller's invocation record

top_sp

top

Prior to function call.

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

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

callee's invocation record

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

top_sp

top
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]
After function call.

caller's invocation record

top_sp

top

actaul parameters

returned value

control link

access link

saved machine status

local data

temporaries

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

"1. The caller evaluates the actual parameters."

Recall:

- formal parameter == parameter
- actual parameter == argument
**caller's invocation record**

<table>
<thead>
<tr>
<th>saved machine status</th>
</tr>
</thead>
<tbody>
<tr>
<td>returned value</td>
</tr>
<tr>
<td>control link</td>
</tr>
<tr>
<td>access link</td>
</tr>
<tr>
<td>local data</td>
</tr>
<tr>
<td>temporaries</td>
</tr>
</tbody>
</table>

Prior to function call.

caller's invocation record

top_sp

top
Caller writes arguments (actual parameters) past the end of its own invocation record.
"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

top_sp

top

caller knows the offset of the eventual returned value. When callee returns the caller will look at this location for the returned value.
"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).
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's invocation record

- actual parameters
- returned value
- control link
- access link
- saved machine status
- local data
- temporaries
- actual parameters
- returned value
- \textcolor{red}{top-sp}
- access link
- saved machine status
  ...
  ...
  ...

\textcolor{red}{Move 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.