Phases of a compiler

Figure 1.6, page 5 of text
7.2.3 Calling Sequence

What happens during a function call?
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
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

During function call.
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

After function call.
Caller vs Callee responsibilities

"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
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
Caller writes arguments (actual parameters) past the end of its own invocation record.
Caller knows the offset of the eventual returned value. When callee returns the caller will look at this location for the returned value.
The caller stores its stack pointer here.
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. 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]

"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
top_sp
access link
saved machine status
...```

Move top_sp
Typical calling sequence [p. 437]

"3. The callee saves the register values and other status information."
caller's invocation record

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.
Relocatable object code

- Compiler produces relocatable object code: addresses are not absolute, but relative to known boundaries (e.g. Stack Pointer, start of record, Program Counter).

- Linker combines object code files into an executable file, in which static relative addresses are made absolute (in virtual address space).

- Loader copies contents of executable file into memory and starts execution.
Relocatable object code

- Compiler produces **relocatable object code**: addresses are not absolute, but relative to known boundaries (e.g. Stack Pointer, start of record, Program Counter).

- Linker combines object code files into an executable file, in which static relative addresses are made absolute (in virtual address space).

- Loader copies contents of executable file into memory and starts execution.

Leave relative offsets alone during translation.
Desirable characteristics of generated code:

- correctness (this is non-negotiable)
- small execution time
- small code size
- small power consumption
Desirable characteristics of generated code:

- correctness (this is non-negotiable)
- small execution time
- small code size
- small power consumption

Associate costs with each instruction, then "minimize" (lower) overall cost, with some balance since execution time and code size can be in conflict.
Significant tasks of code generator

- instruction selection
- register allocation and assignment
- instruction ordering
Significant tasks of code generator

- instruction selection
- register allocation and assignment
- instruction ordering

Which variables are kept in registers?
Significant tasks of code generator:

- instruction selection
- register allocation and assignment
- instruction ordering

Which specific register holds which value?
Significant tasks of code generator

- instruction selection
- register allocation and assignment
- instruction ordering

E.g., to minimize the number of registers needed.
If we generate code for each intermediate code instruction in isolation and string the results together the result may include redundant instructions.
Consider:
\[ x = y + z \]

This might be translated as:
- `LD R0, y` ← load the value of \( y \) into register \( R0 \)
- `ADD R0, R0, z` ← put into \( R0 \) the result of adding \( R0 \) and the value of \( z \)
- `ST x, R0` ← store the value of register \( R0 \) to \( x \)
Larger example [p. 509]

Consider applying the same template to a larger example:

\[
\begin{align*}
  a &= b + c \\
  d &= a + e
\end{align*}
\]

This might be translated as:

```
LD R0, b
ADD R0, R0, c
ST a, R0
LD R0, a
ADD R0, R0, e
ST d, R0
```
Consider applying the same template to a larger example:

\[
\begin{align*}
a &= b + c \\
d &= a + e
\end{align*}
\]

This might be translated as:

\[
\begin{align*}
LD & \; R0, \; b \\
ADD & \; R0, \; R0, \; c \\
ST & \; a, \; R0 \\
LD & \; R0, \; a \\
ADD & \; R0, \; R0, \; e \\
ST & \; d, \; R0
\end{align*}
\]

This instruction is redundant: it is loading into \( R0 \) the value that is already there.