Chapter 10

Function calls

10.1 Introduction

In chapter 7 we have shown how to translate the body of a single function. Function calls were left (mostly) untranslated by using the `CALL` instruction in the intermediate code. Nor did we in chapter 8 show how the `CALL` instruction should be translated.

We will, in this chapter, remedy these omissions. We will initially assume that all variables are local to the procedure or function that access them and that parameters are *call-by-value*, meaning that the *value* of an argument expression is passed to the called function. This is the default parameter-passing mechanism in most languages, and in many languages (*e.g.*, C or SML) it is the only one.

10.1.1 The call stack

A single procedure body uses (in most languages) a finite number of variables. We have seen in chapter 9 that we can map these variables into a (possibly smaller) set of registers. A program that uses recursive procedures or functions may, however, use an unbounded number of variables, as each recursive invocation of the function has its own set of variables, and there is no bound on the recursion depth. We can not hope to keep all these variables in registers, so we will use memory for some of these. The basic idea is that only variables that are local to the active (most recently called) function will be kept in registers. All other variables will be kept in memory.

When a function is called, all the live variables of the calling function (which we will refer to as the *caller*) will be stored in memory so the registers will be free for use by the called function (the *callee*). When the callee returns, the stored variables are loaded back into registers. It is convenient to use a stack for this storing and loading, pushing register contents on the stack when they must be saved and popping them back into registers when they must be restored. Since a stack is (in principle) unbounded, this fits well with the idea of unbounded recursion.

The stack can also be used for other purposes:

- Space can be set aside on the stack for variables that need to be spilled to memory. In chapter 9, we used a constant address ($address_x$) for spilling a variable x. When a stack is used, $address_x$ is actually an offset relative to a stack-pointer. This makes the spill-code slightly more complex, but has the advantage that spilled registers are already saved on the stack when or if a function is called, so they do not need to be stored again.
- Parameters to function calls can be passed on the stack, *i.e.*, written to the top of the stack by the caller and read therefrom by the callee.
- The address of the instruction where execution must be resumed after the call returns (the *return address*) can be stored on the stack.
- Since we decided to keep only local variables in registers, variables that are in scope in a function but not declared locally in that function must reside in memory. It is convenient to access these through the stack.
- Arrays and records that are allocated locally in a function can be allocated on the stack, as hinted in section 7.9.2.

We shall look at each of these in more detail later on.

10.2 Activation records

Each function invocation will allocate a chunk of memory on the stack to cover all of the function's needs for storing values on the stack. This chunk is called the *activation record* or *frame* for the function invocation. We will use these two names interchangeably. Activation records will typically have the same overall structure for all functions in a program, though the sizes of the various fields in the records may differ. Often, the machine architecture (or operating system) will dictate a *calling convention* that standardises the layout of activation records. This allows a program to call functions that are compiled with another compiler or even written in a different language, as long as both compilers follow the same calling convention.

We will start by defining very simple activation records and then extend and refine these later on. Our first model uses the assumption that all information is stored in memory when a function is called. This includes parameters, return address and the contents of registers that need to be preserved. A possible layout for such an activation record is shown in figure 10.1.

	... Next activation records
	Space for storing local variables for spill or preservation across function calls
	Remaining incoming parameters
	First incoming parameter / return value
FP $\longrightarrow$	Return address
	Previous activation records ...

Figure 10.1: Simple activation record layout

FP is shorthand for "Frame pointer" and points to the first word of the activation record. In this layout, the first word holds the return address. Above this, the incoming parameters are stored. The function will typically move the parameters to registers (except for parameters that have been spilled by the register allocator) before executing its body. The space used for the first incoming parameter is also used for storing the return value of the function call (if any). Above the incoming parameters, the activation record has space for storing other local variables, *e.g.*, for spilling or for preserving across later function calls.

10.3 Prologues, epilogues and call-sequences

In chapter 7, we generated code corresponding to the body of a single function. We assumed parameters to this function were already available in variables and the generated code would just put the function result in a variable.

But, since parameters and results are passed through the activation record, we need in front of the code for the function body to add code that reads parameters from the activation record into variables. This code is called the *prologue* of the function. Likewise, after the code for the function body, we need code to store the calculated return value in the activation record and jump to the return address that was stored in the activation record by the caller. This is called the *epilogue* of the function. For the activation-record layout shown in figure 10.1, a suitable prologue and epilogue is shown in figure 10.2. Note that, though we have used a notation similar to the intermediate language introduced in chapter 7, we have extended this a bit: We have used $M[]$ and `GOTO` with general expressions as arguments.

We use the names $parameter_1, \ldots, parameter_n$ for the intermediate-language variables used in the function body for the n parameters. *result* is the intermediate-language variable that holds the result of the function after the body have been executed.

$$\text{Prologue} \left\{ \begin{array}{l} \texttt{LABEL}\, \textit{function-name} \\ \textit{parameter}_1 := M[FP+4] \\ \cdots \\ \textit{parameter}_n := M[FP+4*n] \end{array} \right.$$

code for the function body

$$\text{Epilogue} \left\{ \begin{array}{l} M[FP+4] := \textit{result} \\ \texttt{GOTO}\, M[FP] \end{array} \right.$$

Figure 10.2: Prologue and epilogue for the frame layout shown in figure 10.1

In chapter 7, we used a single intermediate-language instruction to implement a function call. This function-call instruction must be translated into a *call-sequence* of instructions that will save registers, put parameters in the activation record, *etc.* A call-sequence suitable for the activation-record layout shown in figure 10.1 is shown in figure 10.3. The code is an elaboration of the intermediate-language instruction $x :=$ `CALL` $f(a_1,\ldots,a_n)$.

First, all registers that can be used to hold variables are stored in the frame. In figure 10.3, $R0$-Rk are assumed to hold variables. These are stored in the activation record just above the calling function's own m incoming parameters. Then, the frame-pointer is advanced to point to the new frame and the parameters and the return address are stored in the prescribed locations in the new frame. Finally, a jump to the function is made. When the function call returns, the result is read from the frame into the variable x, FP is restored to its former value and the saved registers are read back from the old frame.

Keeping all the parameters in register-allocated variables until just before the call, and only then storing them in the frame can require a lot of registers to hold the parameters (as these are all live up to the point where they are stored). An alternative is to store each parameter in the frame as soon as it is evaluated. This way, only one of the variables $a_1,\ldots,a_n$ will be live at any one time. However, this can go wrong if a later parameter-expression contains a function call, as the parameters to this call will overwrite the parameters of the outer call. Hence, this optimisation must only be used if no parameter-expressions contain function calls or if nested calls use stack-locations different from those used by the outer call.

In this simple call-sequence, we save on the stack all registers that can potentially hold variables, so these are preserved across the function call. This may save more registers than needed, as not all registers will hold values that are required after the call (*i.e.*, they may be dead). We will return to this issue in section 10.6.

```
$M[FP+4*m+4] := R0$
$\cdots$
$M[FP+4*m+4*(k+1)] := Rk$
$FP := FP + framesize$
$M[FP+4] := a_1$
$\cdots$
$M[FP+4*n] := a_n$
$M[FP] := returnaddress$
GOTO $f$
LABEL $returnaddress$
$x := M[FP+4]$
$FP := FP - framesize$
$R0 := M[FP+4*m+4]$
$\cdots$
$Rk := M[FP+4*m+4*(k+1)]$
```

Figure 10.3: Call sequence for $x :=$ `CALL` $f(a_1,\ldots,a_n)$ using the frame layout shown in figure 10.1

Suggested exercises: 10.1.

10.4 Caller-saves versus callee-saves

The convention used by the activation record layout in figure 10.1 is that, before a function is called, the caller saves all registers that must be preserved. Hence, this strategy is called *caller-saves*. An alternative strategy is that the called function saves the contents of the registers that need to be preserved and restores these immediately before the function returns. This strategy is called *callee-saves*.

Stack-layout, prologue/epilogue and call sequence for the callee-saves strategy are shown in figures 10.4, 10.5 and 10.6.

Note that it may not be necessary to store *all* registers that may potentially be used to hold variables, only those that the function actually uses to hold its local variables. We will return to this issue in section 10.6.

So far, the only difference between caller-saves and callee-saves is *when* registers are saved. However, once we refine the strategies to save only a subset of the registers that may potentially hold variables, other differences emerge: Caller-saves need only save the registers that hold *live* variables and callee-saves need only save the registers that the function actually uses. We will in section 10.6 return to how this can be achieved, but at the moment just assume these optimisations are made.

	$\cdots$ Next activation records
	Space for storing local variables for spill
	Space for storing registers that need to be preserved
	Remaining incoming parameters
	First incoming parameter / return value
FP $\longrightarrow$	Return address
	Previous activation records $\cdots$

Figure 10.4: Activation record layout for callee-saves

$$\text{Prologue} \begin{cases} \texttt{LABEL}\ \textit{function-name} \\ M[FP+4*n+4] := R0 \\ \cdots \\ M[FP+4*n+4*(k+1)] := Rk \\ parameter_1 := M[FP+4] \\ \cdots \\ parameter_n := M[FP+4*n] \end{cases}$$

code for the function body

$$\text{Epilogue} \begin{cases} M[FP+4] := result \\ R0 := M[FP+4*n+4] \\ \cdots \\ Rk := M[FP+4*n+4*(k+1)] \\ \texttt{GOTO}\ M[FP] \end{cases}$$

Figure 10.5: Prologue and epilogue for callee-saves

```
$FP := FP + framesize$
$M[FP+4] := a_1$
$\cdots$
$M[FP+4*n] := a_n$
$M[FP] := returnaddress$
GOTO $f$
LABEL $returnaddress$
$x := M[FP+4]$
$FP := FP - framesize$
```

Figure 10.6: Call sequence for $x :=$ CALL $f(a_1,\ldots,a_n)$ for callee-saves

Caller-saves and callee-saves each have their advantages (described above) and disadvantages: When caller-saves is used, we might save a live variable in the frame even though the callee does not use the register that holds this variable. On the other hand, with callee-saves we might save some registers that do not actually hold live values. We can not avoid these unnecessary saves, as each function is compiled independently and hence do not know the register usage of their callers/callees. We can, however, try to reduce unnecessary saving of registers by using a mixed caller-saves and callee-saves strategy:

Some registers are designated caller-saves and the rest as callee-saves. If any live variables are held in caller-saves registers, it is the caller that must save these to its own frame (as in figure 10.3, though only registers that are both designated caller-saves *and* hold live variables are saved). If a function uses any callee-saves registers in its body, it must save these before using them, as in figure 10.5. Only callee-saves registers that are actually used in the body need to be saved.

Calling conventions typically specify which registers are caller-saves and which are callee-saves, as well as the layout of the activation records.

10.5 Using registers to pass parameters

In both call sequences shown (in figures 10.3 and 10.6), parameters are stored in the frame, and in both prologues (figures 10.2 and 10.5) most of these are immediately loaded back into registers. It will save a good deal of memory traffic if we pass the parameters in registers instead of memory.

Normally, only a few (4-8) registers are used for parameter passing. These are used for the first parameters of a function, while the remaining parameters are passed on the stack, as we have done above. Since most functions have fairly short parameter lists, most parameters will normally be passed in registers. The registers

Register	Saved by	Used for
0	caller	parameter 1 / result / local variable
1-3	caller	parameters 2 - 4 / local variables
4-12	callee	local variables
13	caller	temporary storage (unused by register allocator)
14	callee	FP
15	callee	return address

Figure 10.7: Possible division of registers for 16-register architecture

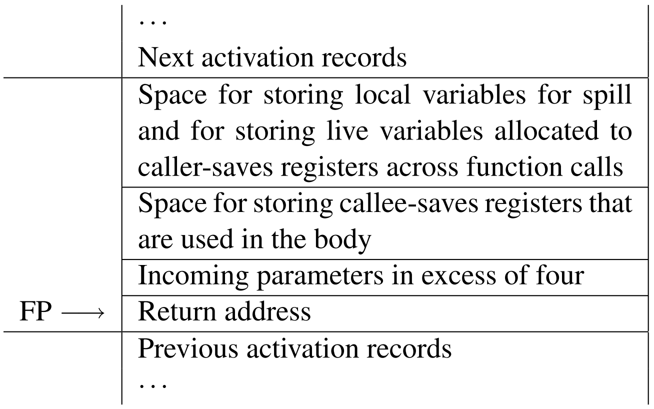
...
Next activation records
Space for storing local variables for spill and for storing live variables allocated to caller-saves registers across function calls
Space for storing callee-saves registers that are used in the body
Incoming parameters in excess of four
FP ⟶
Return address
Previous activation records
...

Figure 10.8: Activation record layout for the register division shown in figure 10.7

used for parameter passing are typically a subset of the caller-saves registers, as parameters are not live after the call and hence do not have to be preserved.

A possible division of registers for a 16-register architecture is shown in figure 10.7. Note that the return address is also passed in a register. Most RISC architectures have jump-and-link (function-call) instructions, which leaves the return address in a register, so this is only natural. However, if a function call is made inside the body, this register is overwritten, so the return address must be saved in the activation record before any calls. The return-address register is marked as callee-saves in figure 10.7. In this manner, the return-address register is just like any other variable that must be preserved in the frame if it is used in the body (which it is if a function call is made). Strictly speaking, we do not need the return address after the call has returned, so we can also argue that R15 is a caller-saves register. If so, the caller must save R15 prior to any call, *e.g.*, by spilling it.

Activation record layout, prologue/epilogue and call sequence for a calling convention using the register division in figure 10.7 are shown in figures 10.8, 10.9 and 10.10.

$$\begin{array}{ll}
\text{Prologue} & \left\{\begin{array}{l}
\texttt{LABEL}\ \textit{function-name} \\
M[FP+\textit{offset}_{R4}] := R4 \qquad \text{(if used in body)} \\
\cdots \\
M[FP+\textit{offset}_{R12}] := R12 \qquad \text{(if used in body)} \\
M[FP] := R15 \qquad \text{(if used in body)} \\
parameter_1 := R0 \\
parameter_2 := R1 \\
parameter_3 := R2 \\
parameter_4 := R3 \\
parameter_5 := M[FP+4] \\
\cdots \\
parameter_n := M[FP+4*(n-4)]
\end{array}\right. \\
 & \\
 & \textit{code for the function body} \\
 & \\
\text{Epilogue} & \left\{\begin{array}{l}
R0 := \textit{result} \\
R4 := M[FP+\textit{offset}_{R4}] \qquad \text{(if used in body)} \\
\cdots \\
R12 := M[FP+\textit{offset}_{R12}] \qquad \text{(if used in body)} \\
R15 := M[FP] \qquad \text{(if used in body)} \\
\texttt{GOTO}\ R15
\end{array}\right.
\end{array}$$

Figure 10.9: Prologue and epilogue for the register division shown in figure 10.7

```
$M[FP+\mathit{offset}_{live_1}] := live_1$    (if allocated to a caller-saves register)
$\dots$
$M[FP+\mathit{offset}_{live_k}] := live_k$    (if allocated to a caller-saves register)
$FP := FP + framesize$
$R0 := a_1$
$\dots$
$R3 := a_4$
$M[FP+4] := a_5$
$\dots$
$M[FP+4*(n-4)] := a_n$
$R15 := returnaddress$
GOTO $f$
LABEL $returnaddress$
$x := R0$
$FP := FP - framesize$
$live_1 := M[FP+\mathit{offset}_{live_1}]$    (if allocated to a caller-saves register)
$\dots$
$live_k := M[FP+\mathit{offset}_{live_k}]$    (if allocated to a caller-saves register)
```

Figure 10.10: Call sequence for $x :=$ CALL $f(a_1,\dots,a_n)$ for the register division shown in figure 10.7

Note that the offsets for storing registers are not simple functions of their register numbers, as only a subset of the registers need to be saved. R15 (which holds the return address) is, like any other callee-saves register, saved in the prologue and restores in the epilogue if it is used inside the body (*i.e.*, if the body makes a function call). Its offset is 0, as the return address is stored at offset 0 in the frame.

In a call-sequence, the instructions

$$
\begin{array}{l}
R15 := returnaddress \\
\texttt{GOTO}\ f \\
\texttt{LABEL}\ returnaddress
\end{array}
$$

can on most RISC processors be implemented by a jump-and-link instruction.

10.6 Interaction with the register allocator

As we have hinted above, the register allocator can be used to optimise function calls, as it can provide information about which registers need to be saved.

The register allocator can tell which variables are live after the function call. In a caller-saves strategy (or for caller-saves registers in a mixed strategy), only the (caller-saves) registers that hold such variables need to be saved before the function call.

Likewise, the register allocator can return information about which registers are used by the function body, so only these need to be saved in a callee-saves strategy.

If a mixed strategy is used, variables that are live across a function call should, if possible, be allocated to callee-saves registers. This way, the caller does not have to save these and, with luck, they do not have to be saved by the callee either (if the callee does not use these registers in its body). If all variables that are live across function calls are made to interfere with all caller-saves registers, the register allocator will not allocate these variables in caller-saves registers, which achieves the desired effect. If no callee-saves register is available, the variable will be spilled and hence, effectively, be saved across the function call. This way, the call sequence will not need to worry about saving caller-saves registers, this is all done by the register allocator.

As spilling may be somewhat more costly than local save/restore around a function call, it is a good idea to have plenty of callee-saves registers for holding variables that are live across function calls. Hence, most calling conventions specify more callee-saves registers than caller-saves registers.

Note that, though the prologues shown in figures 10.2, 10.5 and 10.9 load all stack-passed parameters into registers, this should actually only be done for parameters that are not spilled. Likewise, a register-passed parameter that needs to be spilled should be transferred to a stack location instead of to a symbolic register ($parameter_i$).

In figures 10.2, 10.5 and 10.9, we have moved register-passed parameters from the numbered registers or stack locations to named registers, to which the register allocator must assign numbers. Similarly, in the epilogue we move the function result from a named variable to $R0$. This means that these parts of the prologue and epilogue must be included in the body when the register allocator is called (so the named variables will be replaced by numbers). This will also automatically handle the issue about spilled parameters mentioned above, as spill-code is inserted immediately after the parameters are (temporarily) transferred to registers. This may cause some extra memory transfers when a spilled stack-passed parameter is first loaded into a register and then immediately stored back again. This problem is, however, usually handled by later optimisations.

It may seem odd that we move register-passed parameters to named registers instead of just letting them stay in the registers they are passed in. But these registers may be needed for other function calls, which gives problems if a parameter allocated to one of these needs to be preserved across the call (as mentioned above, variables that are live across function calls should not be allocated to caller-saves registers). By moving the parameters to named registers, the register allocator is free to allocate these to callee-saves registers if needed. If this is not needed, the register allocator may allocate the named variable to the same register as the parameter was passed in and eliminate the (superfluous) register-to-register move. As mentioned in section 9.7, modern register allocators will eliminate most such moves anyway, so we might as well exploit this.

In summary, given a good register allocator, the compiler needs to do the following to compile a function:

1) Generate code for the body of the function, using symbolic names for variables (except precoloured temporary variables used for parameter-passing in call sequences or for instructions that require specific registers, see section 9.7.2).

2) Add code for moving parameters from numbered registers and stack locations into the named variables used for accessing the parameters in the body of the function, and for moving the function-result from a named register to the register used for function results.

3) Call the register allocator with this extended function body. The register allocator should be aware of the register division (caller-saves/callee-saves split) and allocate variables that are live across function calls only to callee-saves registers and should return both the set of used callee-saves registers and the set of spilled variables.

4) To the register-allocated code, add code for saving and restoring the callee-saves registers that the register allocator says have been used in the extended

function body and for updating the frame pointer with the size of the frame (including space for saved registers and spilled variables).

5) Add a function label at the beginning of the code and a return jump at the end.

10.7 Accessing non-local variables

We have up to now assumed that all variables used in a function are local to that function, but most high-level languages also allow functions to access variables that are not declared locally in the functions themselves.

10.7.1 Global variables

In C, variables are either global or local to a function. Local variables are treated exactly as we have described, *i.e.*, typically stored in a register. Global variables will, on the other hand, be stored in memory. The location of each global variable will be known at compile-time or link-time. Hence, a use of a global variable x generates the code

$$\begin{array}{l} x := M[address_x] \\ \text{instruction that uses } x \end{array}$$

The global variable is loaded into a (register-allocated) temporary variable and this will be used in place of the global variable in the instruction that needs the value of the global variable.

An assignment to a global variable x is implemented as

$$\begin{array}{l} x := \text{the value to be stored in } x \\ M[address_x] := x \end{array}$$

Note that global variables are treated almost like spilled variables: Their value is loaded from memory into a register immediately before any use and stored from a register into memory immediately after an assignment. Like with spill, it is possible to use different register-allocated variables for each use of x.

If a global variable is used often within a function, it can be loaded into a local variable at the beginning of the function and stored back again when the function returns. However, a few extra considerations need to be made:

- The variable must be stored back to memory whenever a function is called, as the called function may read or change the global variable. Likewise, the global variable must be read back from memory after the function call, so any changes will be registered in the local copy. Hence, it is best to allocate local copies of global variables in caller-saves registers.

- If the language allows *call-by-reference* parameters (see below) or pointers to global variables, there may be more than one way to access a global variable: Either through its name or via a call-by-reference parameter or pointer. If we cannot exclude the possibility that a call-by-reference parameter or pointer can access a global variable, it must be stored/retrieved before/after any access to a call-by-reference parameter or any access through a pointer. It is possible to make a global *alias analysis* that determines if global variables, call-by-reference parameters or pointers may point to the same location (*i.e.*, may be *aliased*). However, this is a fairly complex analysis, so many compilers simply assume that a global variable may be aliased with *any* call-by-reference parameter or pointer and that any two of the latter may be aliased.

The above tells us that accessing local variables (including call-by-value parameters) is faster than accessing global variables. Hence, good programmers will use global variables sparingly.

10.7.2 Call-by-reference parameters

Some languages, *e.g.*, Pascal (which uses the term **var**-parameters), allow parameters to be passed by *call-by-reference*. A parameter passed by call-by-reference must be a variable, an array element, a field in a record or, in general, anything that is allowed at the left-hand-side of an assignment statement. Inside the function that has a call-by-reference parameter, values can be assigned to the parameter and these assignments actually update the variable, array element or record-field that was passed as parameter such that the changes are visible to the caller. This differs from assignments to call-by-value parameters in that these update only a local copy.

Call-by-reference is implemented by passing the address of the variable, array element or whatever that is given as parameter. Any access (use or definition) to the call-by-reference parameter must be through this address.

In C, there are no explicit call-by-reference parameters, but it is possible to explicitly pass pointers to variables, array-elements, *etc.* as parameters to a function by using the `&` (address-of) operator. When the value of the variable is used or updated, this pointer must be explicitly followed, using the `*` (de-reference) operator. So, apart from notation and a higher potential for programming errors, this is not significantly different from "real" call-by-reference parameters.

In any case, a variable that is passed as a call-by-reference parameter or has its address passed via a `&` operator, must reside in memory. This means that it must be spilled at the time of the call or allocated to a caller-saves register, so it will be stored before the call and restored afterwards.

It also means that passing a result back to the caller by call-by-reference or pointer parameters can be slower than using the function's return value, as the

```
procedure f (x : integer);
   var y : integer;
   function g(p : integer);
   var q : integer;
   begin
      if p<10 then y := g(p+y)
      else q := p+y;
      if (y<20) then f(y);
      g := q
   end;
begin
   y := x+x;
   writeln(g(y),y)
end;
```

Figure 10.11: Example of nested scopes in Pascal

return value can be passed in a register. Hence, like global variables, call-by-reference and pointer parameters should be used sparingly.

Either of these on their own have the same aliasing problems as when combined with global variables.

10.7.3 Nested scopes

Some languages, *e.g.*, Pascal and SML, allow functions to be declared locally within other functions. A local function typically has access to variables declared in the function in which it itself is declared. For example, figure 10.11 shows a fragment of a Pascal program. In this program, g can access x and y (which are declared in f) as well as its own local variables p and q.

Note that, since f and g are recursive, there can be many instances of their variables in different activation records at any one time.

When g is called, its own local variables (p and q) are held in registers, as we have described above. All other variables (*i.e.*, x and y) reside in the activation records of the procedures/functions in which they are declared (in this case f). It is no problem for g to know the offsets for x and y in the activation record for f, as f can be compiled before g, so full information about f's activation record layout is available for the compiler when it compiles g. However, we will not at compile-time know the position of f's activation record on the stack. f's activation record will not always be directly below that of g, since there may be several recursive invocations of g (each with its own activation record) above the last activation record for f. Hence, a pointer to f's activation record will be given as parameter to

```
function g(var fFrame : fRecord, p : integer);
var q : integer;
begin
   if p<10 then fFrame.y := g(fFrame,p+fFrame.y)
   else q := p+fFrame.y;
   if (fFrame.y<20) then f(fFrame.y);
   g := q
end;

procedure f (x : integer);
   var y : integer;
begin
   y := x+x;
   writeln(g(FP,y),y)
end;
```

Figure 10.12: Adding an explicit frame-pointer to the program from figure 10.11

g when it is called. When `f` calls g, this pointer is just the contents of FP, as this, by definition, points to the activation record of the active function (*i.e.*, `f`). When g is called recursively from g itself, the incoming parameter that points to `f`'s activation record is passed on as a parameter to the new call, so every instance of g will have its own copy of this pointer.

To illustrate this, we have in figure 10.12 added this extra parameter explicitly to the program from figure 10.11. Now, g accesses all non-local variables through the `fFrame` parameter, so it no longer needs to be declared locally inside `f`. Hence, we have moved it out. We have used record-field-selection syntax in g for accessing `f`'s variables through `fFrame`. Note that `fFrame` is a call-by-reference parameter (indicated by the `var` keyword), as g can update `f`'s variables (*i.e.*, `y`). In `f`, we have used `FP` to refer to the current activation record. Normally, a function in a Pascal program will not have access to its own frame, so this is not quite standard Pascal.

It is sometimes possible to make the transformation entirely in the source language (*e.g.*, Pascal), but the extra parameters are usually not added until the intermediate code, where FP is made explicit, has been generated. Hence, figure 10.12 mainly serves to illustrate the idea, not as a suggestion for implementation.

Note that all variables that can be accessed in inner scopes need to be stored in memory when a function is called. This is the same requirement as was made for call-by-reference parameters, and for the same reason. This can, in the same way, be handled by allocating such variables in caller-saves registers.

	. . . Next activation records
	Space for storing local variables for spill and for storing live variables allocated to caller-saves registers across function calls
	Space for storing callee-saves registers that are used in the body
	Incoming parameters in excess of four
	Return address
FP ⟶	Static link (SL)
	Previous activation records . . .

Figure 10.13: Activation record with static link

f:

	. . .
	y
	x
	Return address
FP →	SL (null)
	. . .

g:

	. . .
	q
	p
	Return address
FP →	SL (to f)
	. . .

Figure 10.14: Activation records for `f` and `g` from figure 10.11

Static links

If there are more than two nested scopes, pointers to all outer scopes need to be passed as parameters to locally declared functions. If, for example, `g` declared a local function `h`, `h` would need pointers to both `f`'s and `g`'s activation records. If there are many nested scopes, this list of extra parameters can be quite long. Typically, a single parameter is instead used to hold a linked list of the frame pointers for the outer scopes. This is normally implemented by putting the links in the activation records themselves. Hence, the first field of an activation record (the field that FP points to) will point to the activation record of the next outer scope. This is shown in figure 10.13. The pointer to the next outer scope is called the *static link*, as the scope-nesting is static as opposed to the actual sequence of run-time calls that determine the stacking-order of activation records[1]. The layout of the activation records for `f` and `g` from figure 10.11 is shown in figure 10.14.

`g`'s static link will point to the most recent activation record for `f`. To read `y`, `g` will use the code

[1]Sometimes, the return address is referred to as the *dynamic link*.

$FP_f := M[FP]$	Follow g's static link
$address := FP_f + 12$	Calculate address of `y`
$y := M[address]$	Get `y`'s value

where y afterwards holds the value of `y`. To write `y`, `g` will use the code

$FP_f := M[FP]$	Follow g's static link
$address := FP_f + 12$	Calculate address of `y`
$M[address] := y$	Write to `y`

where y holds the value that is written to `y`. If a function `h` was declared locally inside `g`, it would need to follow two links to find `y`:

$FP_g := M[FP]$	Follow `h`'s static link
$FP_f := M[FP_g]$	Follow g's static link
$address := FP_f + 12$	Calculate address of `y`
$y := M[address]$	Get `y`'s value

This example shows why the static link is put in the first element of the activation record: It makes following a chain of links easier, as no offsets have to be added in each step.

Again, we can see that a programmer should keep variables as local as possible, as non-local variables take more time to access.

10.8 Variants

We have so far seen fixed-size activation records on stacks that grow upwards in memory, and where FP points to the first element of the frame. There are, however, reasons why you sometimes may want to change this.

10.8.1 Variable-sized frames

If arrays are allocated on the stack, the size of the activation record depends on the size of the arrays. If these sizes are not known at compile-time, neither will the size of the activation records. Hence, we need a run-time variable to point to the end of the frame. This is typically called the *stack pointer*, because the end of the frame is also the top of the stack. When setting up parameters to a new call, these are put at places relative to SP rather than relative to FP. When a function is called, the new FP takes the value of the old SP, but we now need to store the old value of FP, as we no longer can restore it by subtracting a constant from the current FP. Hence, the old FP is passed as a parameter (in a register or in the frame) to the new function, which restores FP to this value just before returning.

If arrays are allocated on a separate stack, frames can be of fixed size, but a separate stack-pointer is now needed for allocating/deallocating arrays.

If two stacks are used, it is customary to let one grow upwards and the other downwards, such that they grow towards each other. This way, stack-overflow tests on both stacks can be replaced by a single test on whether the stack-tops meet. It also gives a more flexible division of memory between the two stacks than if each stack is allocated its own fixed-size memory segment.

10.8.2 Variable number of parameters

Some languages (*e.g.*, C and LISP) allow a function to have a variable number of parameters. This means that the function can be called with a different number of parameters at each call. In C, the `printf` function is an example of this.

The layouts we have shown in this chapter all assume that there is a fixed number of arguments, so the offsets to, *e.g.*, local variables are known. If the number of parameters can vary, this is no longer true.

One possible solution is to have two frame pointers: One that shows the position of the first parameter and one that points to the part of the frame that comes after the parameters. However, manipulating two FP's is somewhat costly, so normally another trick is used: The FP points to the part of the frame that comes after the parameters, Below this, the parameters are stored at negative offsets from FP, while the other parts of the frame are accessed with (fixed) positive offsets. The parameters are stored such that the first parameter is closest to FP and later parameters further down the stack. This way, parameter number k will be a fixed offset $(-4*k)$ from FP.

When a function call is made, the number of arguments to the call is known to the caller, so the offsets (from the old FP) needed to store the parameters in the new frame will be fixed at this point.

Alternatively, FP can point to the top of the frame and all fields can be accessed by fixed negative offsets. If this is the case, FP is sometimes called SP, as it points to the top of the stack.

10.8.3 Direction of stack-growth and position of FP

There is no particular reason why a stack has to grow upwards in memory. It is, in fact, more common that call stacks grow downwards in memory. Sometimes the choice is arbitrary, but at other times there is an advantage to have the stack growing in a particular direction. Some instruction sets have memory-access instructions that include a constant offset from a register-based address. If this offset is unsigned (as it is on, *e.g.*, IBM System/370), it is an advantage that all fields in the activation record are at non-negative offsets. This means that either FP must point to the bottom of the frame and the stack grow upwards, or FP must point to the top of the frame and the stack grow downwards.

If, on the other hand, offsets are signed but have a small range (as on Digital's Vax, where the range is -128 – +127), it is an advantage to use both positive and negative offsets. This can be done, as suggested in section 10.8.2, by placing FP after the parameters but before the rest of the frame, so parameters are addressed by negative offsets and the rest by positive. Alternatively, FP can be positioned k bytes above the bottom of the frame, where $-k$ is the largest negative offset.

10.8.4 Register stacks

Some processors, *e.g.*, Suns Sparc and Intels IA-64 have on-chip stacks of registers. The intention is that frames are kept in registers rather than on a stack in memory. At call or return of a function, the register stack is adjusted. Since the register stack has a finite size, which is often smaller than the total size of the call stack, it may overflow. This is trapped by the operating system which stores part of the stack in memory and shifts the rest down (or up) to make room for new elements. If the stack underflows (at a pop from an empty register stack), the OS will restore earlier saved parts of the stack.

10.8.5 Functions as values

If you can pass a function as a parameter to another function, you need to pass its static link as well, so the function can access its non-local variables when it is called. If there are no local function definitions (and, hence, no need for static links) it is enough to pass the address of the function. This applies, for example, to the language C.

In Pascal, functions can be passed as parameters, but can not be returned as function results. When you pass a function as parameter, you pass a pair consisting of the address of the function's code and its static link. This pair is called a *thunk* or a *closure*. When the function is called and its frame is put on the stack, the static link is taken from this pair. Since you cannot return a functional value, the static link will always point to a frame that is further down the stack.

However, if you can return a functional value as the result of a function (as is possible in most functional languages), the frame that the static link points to can have been unstacked by the time the functional value is used. To prevent this, the part of the frame that contains variables is allocated on the heap instead of the stack and the static link in the closure points to the heap-allocated part of the frame.

Suggested exercises: 10.2.

10.9 Further reading

Calling conventions for various architectures are usually documented in the manuals provided by the vendors of these architectures. Additionally, the calling convention for the MIPS microprocessor is shown in [39].

In figure 10.12, we showed in source-language terms how an extra parameter can be added for accessing non-local parameters, but stated that this was for illustrative purposes only, and that the extra parameters are not normally added at source-level. However, [8] argues that it *is*, actually, a good idea to do this, and goes on to show how many advanced features regarding nested scopes, higher-order functions and even register allocation can be implemented mostly by source-level transformations.

Section 11.7 describes some optimisations that can be used to make function calls faster or use less stack space.

Exercises

Exercise 10.1

In section 10.3 an optimisation is mentioned whereby parameters are stored in the new frame as soon as they are evaluated instead of just before the call. It is warned that this will go wrong if any of the parameter-expressions themselves contain function calls. Argue that the *first* parameter-expression of a function call can contain other function calls without causing the described problem.

Exercise 10.2

Section 10.8.2 suggests that a variable number of arguments can be handled by storing parameters at negative offsets from FP and the rest of the frame at non-negative offsets from FP. Modify figures 10.8, 10.9 and 10.10 to follow this convention.

Exercise 10.3

Find documentation for the calling convention of a processor of your choice and modify figures 10.7, 10.8, 10.9 and 10.10 to follow this convention.

Exercise 10.4

Many functions have a body consisting of an if-then-else statement (or expression), where one or both branches use only a subset of the variables used in the body as a whole. As an example, assume the body is of the form

```
IF $cond$ THEN $label_1$ ELSE $label_2$
LABEL $label_1$
$code_1$
GOTO $label_3$
LABEL $label_2$
$code_2$
LABEL $label_3$
```

The condition *cond* is a simple comparison between variables (which may or may not be callee-saves).

A normal callee-saves strategy will in the prologue save (and in the epilogue restore) all callee-saves registers used in the body. But since only one branch of the if-then-else is taken, some registers are saved and restored unnecessarily.

We can, as usual, get information about variable use in the different parts of the body (*i.e.*, *cond*, $code_1$ and $code_2$) from the register allocator.

We will now attempt to combine the prologue and epilogue with a function body of the above form in order to reduce the number of *callee-saves* registers saved.

Replace in figure 10.9 the text "*code for function body*" by the above body. Then modify the combined code so parts of saving and restoring registers $R4$ – $R12$ and $R15$ is moved into the branches of the if-then-else structure. Be precise about which registers are saved and restored where. You can use clauses like "if used in $code_1$".