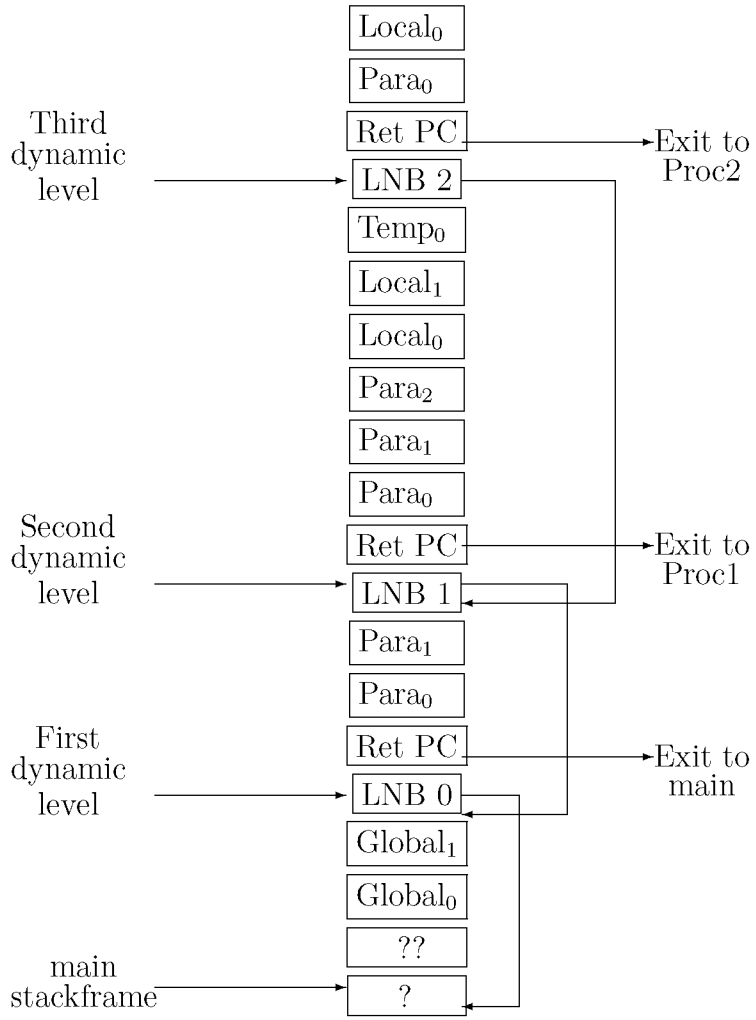
15 Procedure calling and object activation

Figure 1: Simple forward going stack



This is deliberately misleading in its choice of names. The variables declared inside main() are of course not true globals, but in fact restricted to the scope of function main(). Figure 3 shows the same program in Pascal, but now we need to be able to see variables Global0 and Global1 from the inside of all the other procedures, yet there is no obvious way to achieve this with the stackframe model we have. The dynamic links merely record the sequence of calling, but do not correspond to the textual levels at which procedures were declared.

Figure 2: C program corresponding to Figure 1

```
void Proc3(int Para0)
{
   float Local0;
   (* Snapshot taken here *)
}

void Proc2(float Para0, Para1, Para2)
{
   int Local0, Local1;
   Proc3(1);
}

void Proc1(int Para0,Para1)
{
   int Local0, Local1;
   Proc2(3.0,2.56);
}

void main()
{
   int Global0, Global1;
   Proc1(1,2);
}
```

Figure 3: Pascal version of simple program

```
program Main;

var
   Global1, Global2 : integer;

   procedure Proc3(Para0 : Boolean);
   var
      Local0 : real;
   begin
      { * Snapshot taken here * }
   end;

   procedure Proc2(Para0, Para1, Para2 : real);
   var
      Local0,Local1 : {Red, Green, Blue};
   begin
      Proc3(True);
   end;

   procedure Proc1(Para0, Para1 : integer);
   var
      Local0, Local1 : integer;
   begin
      Proc2(3.0, 2.56);
   end;

begin
   Proc1(2, 4);
end.
```

Static linkage

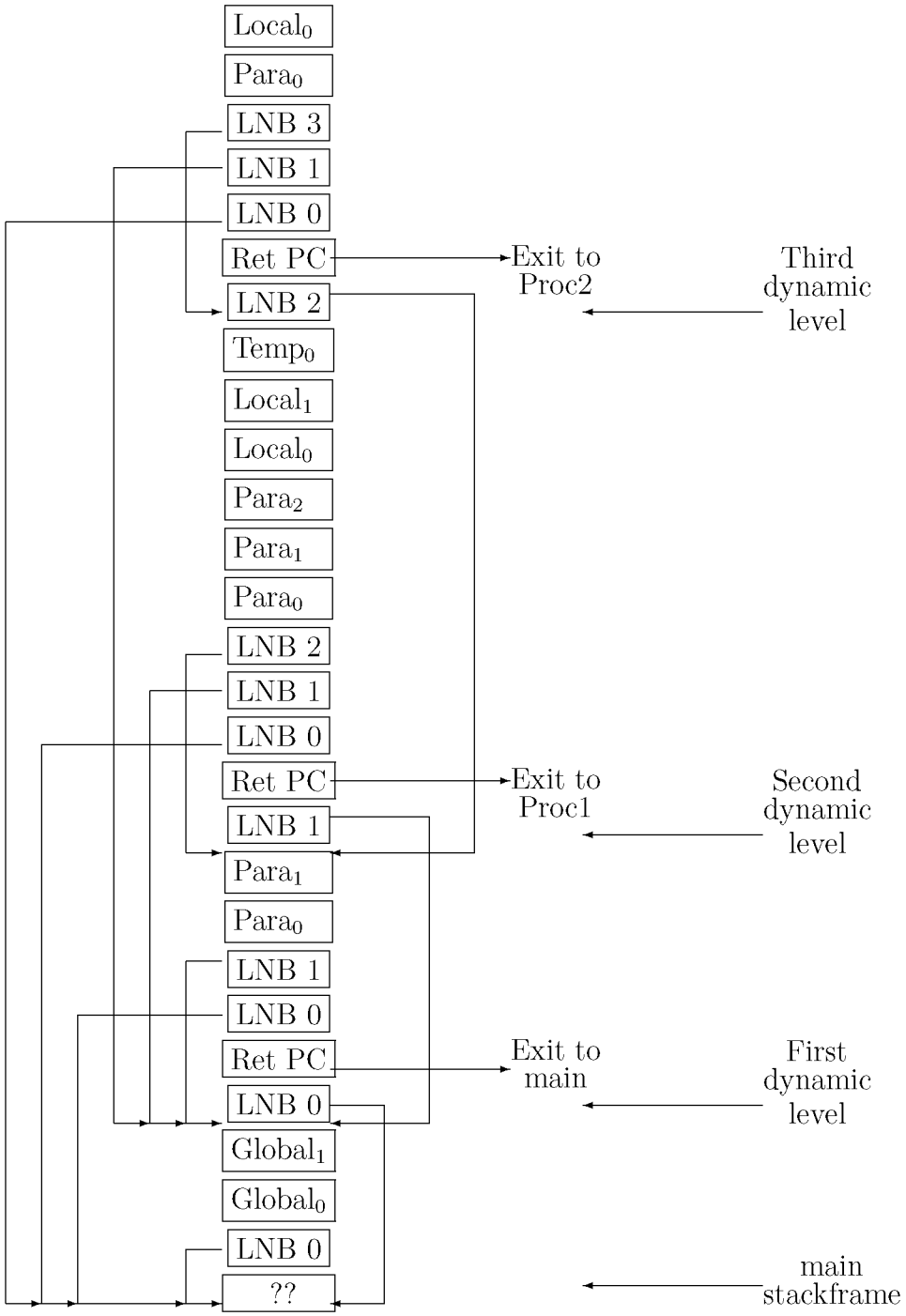
To more fully illustrate this problem and some solutions to it, consider the Pascal program in figure 4. This has one procedure, Proc1, declared at the main program level and two further procedures, Proc2 and Proc3, declared locally within Proc1. Proc 1 includes a call to Proc2. Since Proc2 is textually enclosed by Proc1, any declarations within Proc1 are still in scope inside Proc2 and some means of accessing them must be provided. These variables must also be available within Proc3, which is declared at the same textual level as Proc2, but Proc3 must not be able to access variables inside Proc2, since the scope rules of Pascal forbid seeing into a procedure from the same or any enclosing level.

It is important to see that this is different to the dynamic chain used to allow correct returning from calls. On the dynamic (calling) chain this program (at the point of the snapshot) has a stack with Proc3 on top. This is dynamically linked to Proc2, which is linked to Proc1, which is linked to the main program. Proc3's static (scope) chain, however, must miss Proc2 and link straight to Proc1 and then to the main program.

Figure 4: Pascal version of simple program

```
program Main;
var
   Global1, Global2 : integer;
   procedure Proc1(Para0, Para1 : integer);
   var
      Local0, Local1 : integer;
      procedure Proc3(Para0 : Boolean);
      var
         Local0 : real;
      begin
         { * Snapshot taken here * }
      end;
      procedure Proc2(Para0, Para1, Para2 : real);
      var
         Local0,Local1 : {Red, Green, Blue};
      begin
         Proc3(True);
      end;
   begin
      Proc2(3.0, 2.56);
   end;
begin
   Proc1(2, 4);
end.
```

Figure 5: Display in a forward going stack



The solution shown in figure 5 is know as a *display*. In effect an array of pointers to those stack frames in scope, indexed by their textual level, is maintained. To access any item in a textually enclosing block is now an indirect reference through the display element of appropriate level.

There are alternatives to the display, such as holding the textual level in each stackframe and chaining down the dynamic chain to find the first stackframe with the required level. This may seem less efficient, but a careful analysis is needed of the overhead in maintaining a display in every stackframe. This is done during the calling or entry sequence and requires that, in entering a procedure at textual level n the entries from the calling procedure's display numbered $0 \ldots n-1$ be pushed onto the stack, followed by the current stack frame's address or local name base (LNB).

The entry sequence for a procedure

The problems of maintaining a runtime stack are clearly complex. It is not surprising that several variants exist for a solution. The forward going stack shown above requires that the calling procedure make a *precall* before it starts building the display or pushing parameters. At this point the linkage information and return PC are stacked. This ensures that the parameters and locals are stored in a contiguous area with increasing addresses. The called procedure extends the stack's top to make room for its locals.

Other systems exist, with decreasing addressing on the stack, for instance. It is also common to first push the parameters, then make a call where the linkage is added and then allocate local space. This splits the local name space into two regions, which are typically addressed as negative and positive offsets (allowing for intervening linkage etc.) from a stack base register. If the parameters are pushed in reverse order in such an arrangement it is possible to accomodate the concept of variable numbers of parameters to a procedure, as in C.

On return, a stack with contiguous parameter/local space simply follows its dynamic chain. A split arrangement requires more information to be known and may involve an operand to the call instruction indicating the number of words pushed as parameters.

RISC calling

It is not strictly true to say that RISC machines must have very large numbers of registers. It is however very common for this to be the case. This is because of improvements in VLSI chip manufacture which have dropped the cost of onboard memory and allowed orders of magnitude increases in register numbers. Having more registers increases the probability that variables can be held in them rather than in stack or static locations. It is common to use a moving window on the register bank to replace the runtime stack. This has really been made possible because C and C++ have achieved dominance in new programming work. They, along with Fortran, require only the simplest stack arrangement, as described at the start of this lecture. They also tend to use only simple values (scalars and pointers) as parameters. It is much harder to use a register window where a display and complex values are commonly used in a language.

Nested stacks and parallel stacks

We are used to thinking of the stack as the most significant data structure at runtime in our programming languages. In fact it is merely a degenerate case of the more gneral concept of *nested stacks*. These are sometimes referred to as *cactus stacks*, since they can be visualised as irregular tree structures, rather like the giant cacti of North American deserts.

A single stack represents a sequence of procedure calls. In theory a program may reach an arbitrary depth of calling and so a stack may be of unbounded size. In practice the size reached by a stack depends on the data being processed and has an upper limit depending on the size of the area of memory allocated to it.

Because of its centrality in procedural programming, the stack is usually supported by the hardware of a computer. Call and return instructions are assumed to exist. Of course there are really many stacks on most machines. Each process typically has a user stack and a system stack. The latter is only accessible to programs with high levels of privilege. Then, on any multi-tasking machine there will exist a stack (0r pair of stacks) for every process running. At any time the operating system will have given control to one process and made its stack(s) the active ones, but the others will all have to be kept ready until their turn comes.

In languages, like SIMULA, which support co-routines there is a true cactus stack structure. In this approach to programming an active co-routine is similar to a procedure, but does not have to return to its point of call, releasing its stack frame and terminating. It may instead pass control to another coroutine and suspend itself, ready to resume when its turn comes. Within a coroutine there may be simple procedure calling, requiring a stack for that coroutine, or there may be a system of nested coroutines, requiring a system of concurrent stacks local to that coroutine.

Clearly the typical hardware of a computer does not support cactus stacks. (In fact Intel did prototype such an architecture, but abandoned it to pursue the more traditional and lucrative 8086 family.) To implement such languages a software solution is needed. There are various possibilities, but they essentially use a heap to store stackframes as irregular linked list structures, embodying the static and dynamic links of a conventional stack, but generalised to allow stacks to fork. When looking at this sort of arrangement it becomes clear that the hardware supported runtime stack is a limited mechanism, compromising for efficiency.