Runtime Management

The Compiler... so far

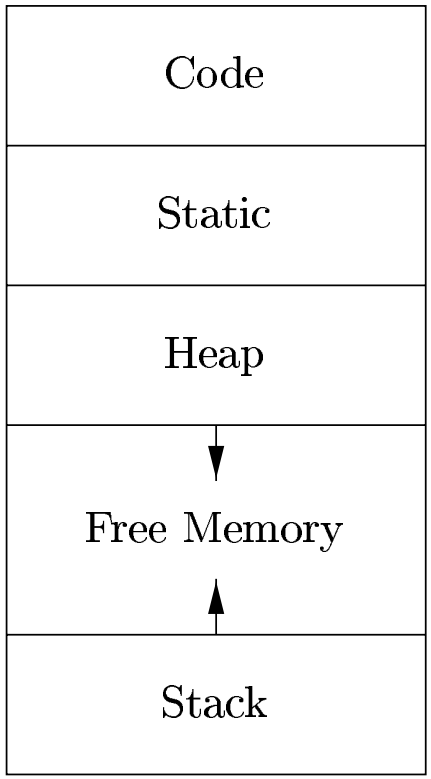
- We have covered all the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic Analysis
  - IR Generation
- We now begin the back-end phases
  - Runtime management
  - Register Allocation and Code Generation
  - Optimizations

Management of run-time resources

- Where are the variables stored?
- Compiler generates code for managing the layout and allocation of storage for variables.
  - The code is generated at compile-time, but executes at run-time to perform the actual management.
  - Must take into account HLL concepts such as scopes, bindings, data types, procedure abstractions, etc.
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e. “main”)
- From this point onwards, the responsibility of managing the space falls to the program.

Run-time storage organization

Typical memory layout



The classical scheme

- allows both stack and heap maximal freedom
- code and static data may be separate or intermingled

Run-time storage organization

Code space

- fixed size
- statically allocated

Static space

- fixed-sized data may be statically allocated, e.g. global variables.
- Advantage is that addresses of such variables can be directly compiled into the machine code.
- variable-sized data must be dynamically allocated.

Heap and Stack

- Heap stores dynamically allocated objects
- Stack stores data structures called ‘activation records’ for maintaining procedure abstraction

The procedure abstraction

Separate compilation:

- allows us to build large programs
- keeps compile times reasonable
- requires independent procedures

The linkage convention:

- a social contract
- machine dependent
- division of responsibility

The linkage convention ensures that procedures inherit a valid run-time environment and that they restore one for their parents.

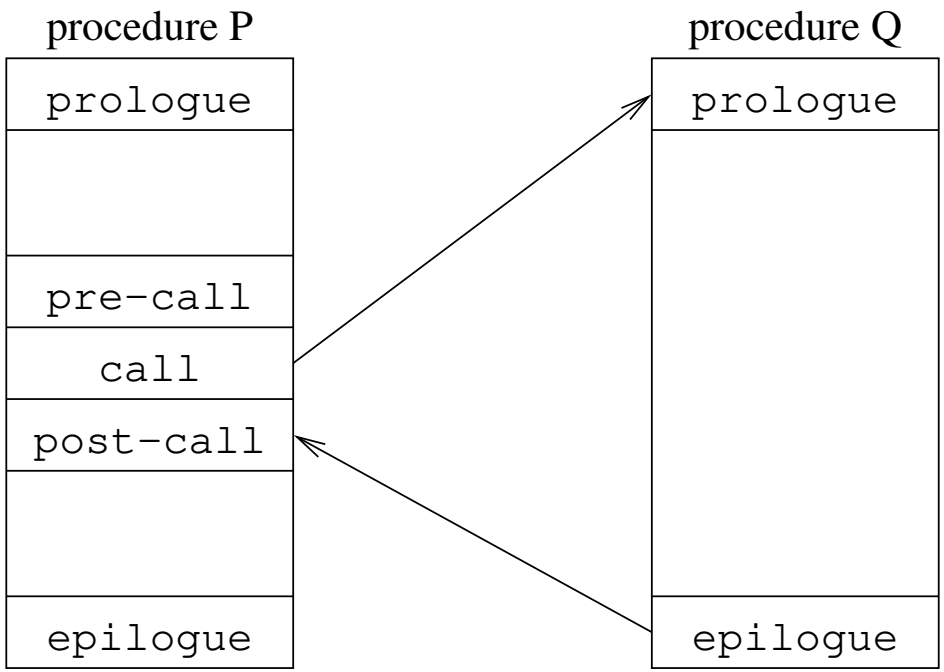
Linkages execute at run time

Code to make the linkage is generated at compile time

The procedure abstraction

The essentials:

- on entry, establish `p`'s environment
- at a call, preserve `p`'s environment
- on exit, tear down `p`'s environment
- in between, addressability and proper lifetimes



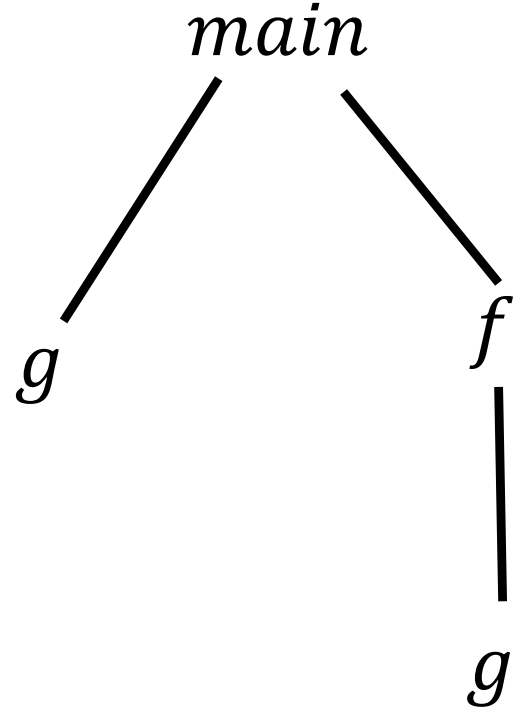
Each system has a standard linkage

Activations

- An invocation of procedure p is an activation of p.
- The lifetime of an activation of p consists of all the steps required to execute the activation.
  - Including all the steps in any procedure that p calls.
- Lifetimes of procedure activations are propertly nested.
  - Activation lifetimes can be depicted as a tree; called the activation tree.

Activation Tree: Example

```
class main{
public int g() { ... }
public int f() {
g();}
public int main() {
g(); f();
}
```



- Each node corresponds to an activation.
- Children of a node p correspond to activations of procedures called by the activation of p, in the order from left to right.

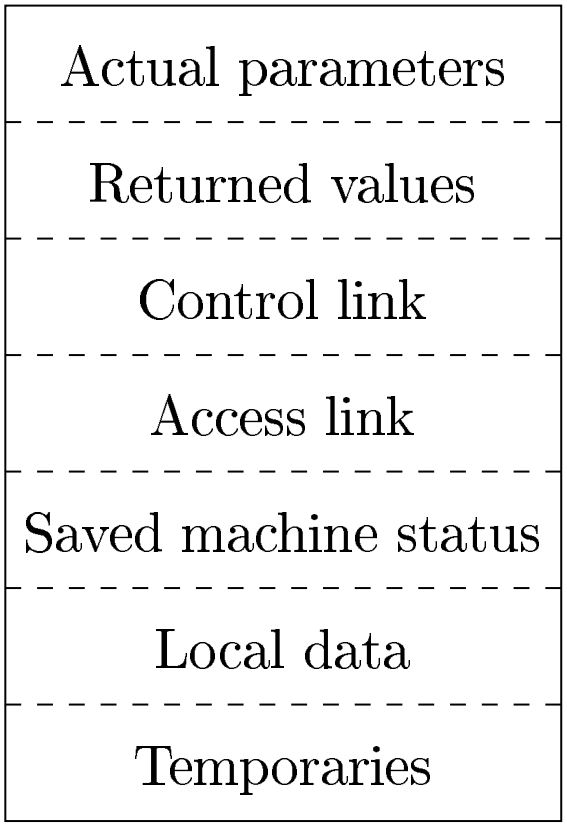
Activation Tree: Properties

- Activation tree is a dynamic structure; it depends on run-time behaviour.
  - The same program may have different activation trees in different executions.
- Several useful relationships exist between activation trees and runtime behaviour:
  - Sequence of procedure calls corresponds to pre-order traversal of the activation tree.
  - Sequence of returns corresponds to post-order traversal of the activation tree.
  - If the execution is currently in a particular activation p, then all activations that are currently live correspond to p and all its ancestors.
- We can use a stack to track live activations.

Activation Records

- The information needed to manage one procedure activation is called an activation record or frame.
- If a function f calls g, then g's activation record contains a mix of information about f and g.
  - f's execution is 'suspended' until g completes, after which, f must resume. Hence, g's AR contains information needed to resume execution of f.
  - g's AR will also contain space for return value, parameters and local variables of g.

A typical Activation Record



Note that this is one of many possible AR designs.
The compiler must fix the layout of the AR and generate code that correctly accesses locations in the AR.

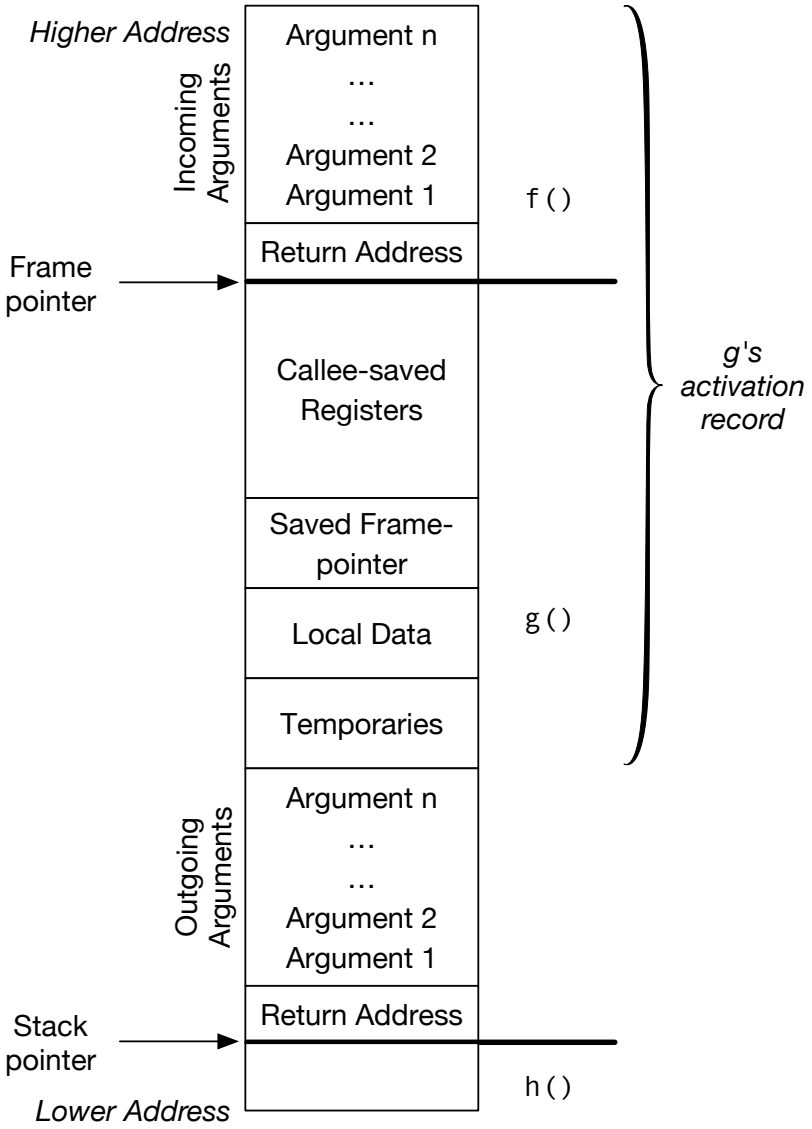
Procedure linkages

The linkage divides responsibility between caller and callee

	Caller	Callee
Call	pre-call	prologue
	1. allocate basic frame 2. evaluate & store params. 3. store return address 4. jump to child	1. save registers, state 2. store FP (dynamic link) 3. set new FP 4. Allocate space for local data
Return	post-call	epilogue
	1. copy return value 2. deallocate basic frame 3. restore parameters (if copy out)	1. store return value 2. restore state 3. restore parent's FP 4. jump to return address

At compile time, generate the code to do this.
At run time, that code manipulates the frame & data areas.

Activation records on the control stack



- Arguments are placed at the start of the AR; easy for caller to set them up.
- Temporaries are allocated at the end of the AR; current function can use as much space as necessary.
- The frame-pointer (FP) contains the base address of current frame
  - Arguments at constant positive offsets, local variables at constant negative offsets, relative to FP.

Discussion

- The set of steps performed while calling a procedure is called calling sequence.
  - Set of steps performed while returning is called return sequence.
- Note that there is noting magic about the division of responsibility between caller and callee.
  - One can choose a different division, as long as all the steps in calling and return sequence are carried out.
  - Choose an organization which optimizes some (or all parameters): Execution time, code length, ease of code generation, etc.
- To optimize code length, it is better to delegate as much responsibility to the callee as possible.
  - If a procedure is called n times, then the portion assigned to the caller would be generated n times.

Run-time storage organization

Where do local variables go?
When can we allocate them on a stack?
Key issue is lifetime of local names

Downward exposure:

- called procedures may reference my variables
- dynamic scoping
- lexical scoping

Upward exposure:

- can I return a reference to my variables?
- functions that return functions
- continuation-passing style

With only downward exposure, the compiler can allocate the frames on the run-time call stack

Storage classes

Each variable must be assigned a storage class

Static variables:

- addresses compiled into code
- (usually) allocated at compile-time
- limited to fixed size objects

Global variables:

- almost identical to static variables
- layout may be important
- naming scheme ensures universal access

Storage classes (cont.)

Procedure local variables
Put them on the stack

- if sizes are fixed
- if lifetimes are limited
- if values are not preserved

Dynamically allocated variables
Must be treated differently

- call-by-reference, pointers, lead to non-local lifetimes
- (usually) an explicit allocation on the heap
- explicit or implicit deallocation

Access to non-local data

How does the code find non-local data at run-time?

Real globals

- visible everywhere
- naming convention gives an address
- initialization requires cooperation

Lexical nesting

- Allows nested procedure declarations
- A procedure can access variables of other procedures whose declaration surround its own declaration
- view variables as (nesting-depth,offset) pairs (compile-time)
- chain of non-local access links
- more expensive to find (at run-time)

Lexical Nesting: Example

```
 1) fun sort(inputFile, outputFile) =
        let
 2)         val a = array(11,0);
 3)         fun readArray(inputFile) = ···
 4)               ··· a ··· ;
 5)         fun exchange(i,j) =
 6)               ··· a ··· ;
 7)         fun quicksort(m,n) =
                let
 8)                  val v = ··· ;
 9)                  fun partition(y,z) =
10)                        ··· a ··· v ··· exchange ···
                in
11)                 ··· a ··· v ··· partition ··· quicksort
                end
        in
12)         ··· a ··· readArray ··· quicksort ···
        end;
```

Access to non-local data

Two important problems arise

- How do we map a name into a (nesting-depth,offset) pair?
  Use a block-structured symbol table
  - look up a name, want its most recent declaration
  - declaration may be at current level or any lower level
- Given a (nesting-depth,offset) pair, what's the address?
  Two classic approaches
  - access links (or static links)
  - displays

Access to non-local data

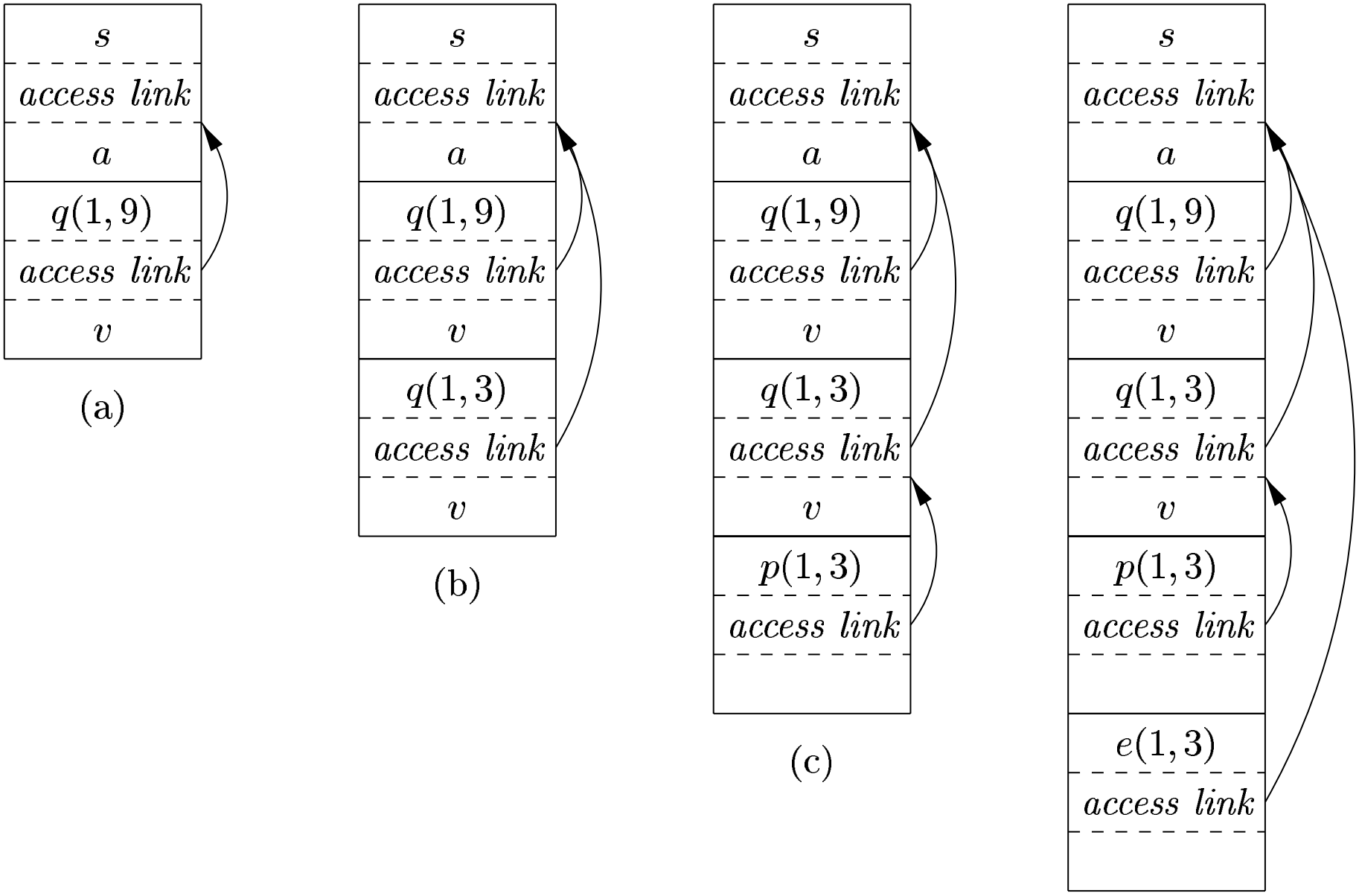
To find the value specified by (l, o)

- need current procedure nesting-depth, k
- $k = l \Rightarrow$ local value
- $k > l \Rightarrow$ find l's activation record
- $k < l$ cannot occur

Maintaining access links: (static links)

- calling nesting-depth $k+1$ procedure
  1. pass current procedure's FP as access link
  2. current procedure's backward chain will work for lower nesting-depths
- calling procedure at nesting-depth $l \leq k$
  1. find link to nesting-depth $l-1$ and pass it to the callee
  2. its access link will work for lower nesting-depths

Using Access Links: Example



The display

To improve run-time access costs, use a display:

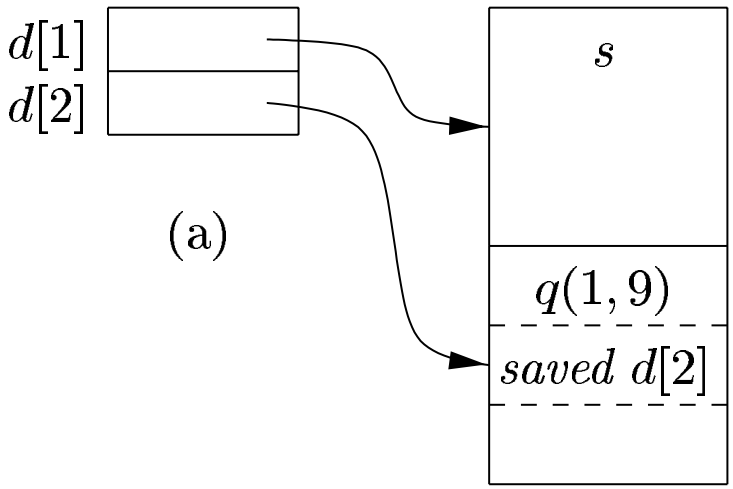
- table of access links for all nesting-dephts
- For each nesting-depth, the access link to the most recent AR at the depth is stored.
- Each AR also stores the link to the previous highest AR.
- Where to store the display? Can be stored separately or as part of the activation record.

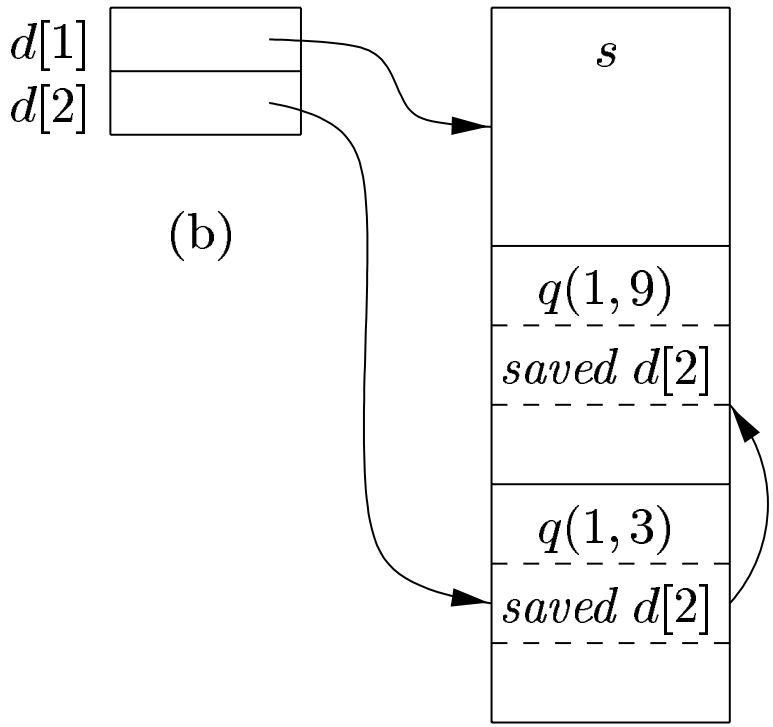
Access with the display
assume a value described by (l, o)

- find slot as `display`[l]
- add offset to pointer from slot (`display`[l][o])

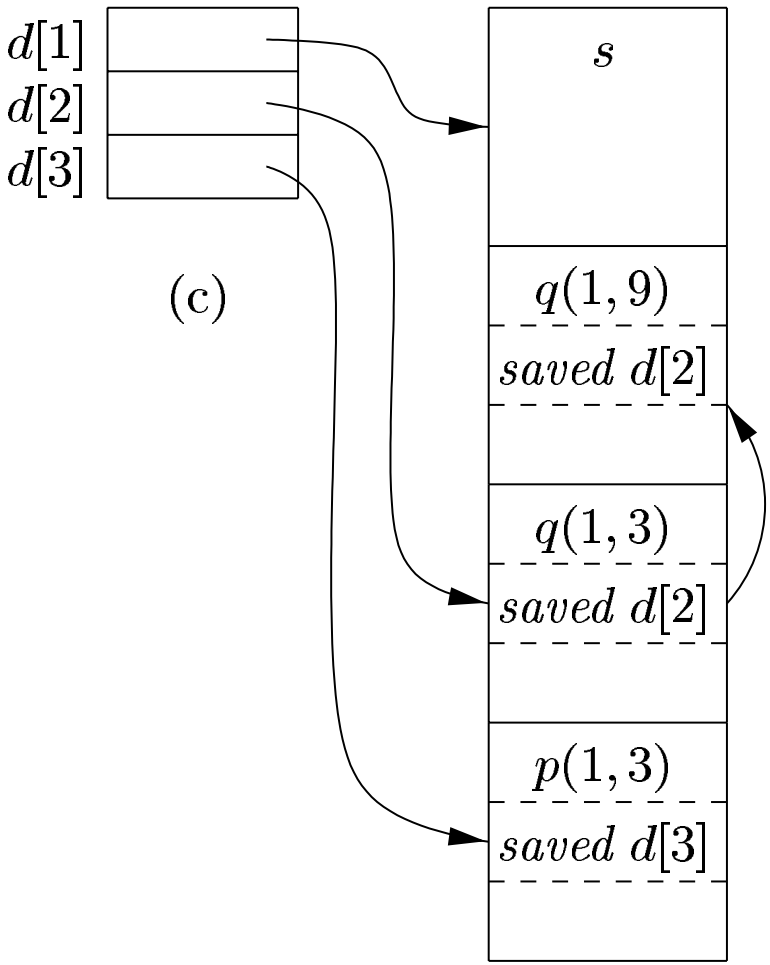
"Setting up the basic frame" now includes display manipulation

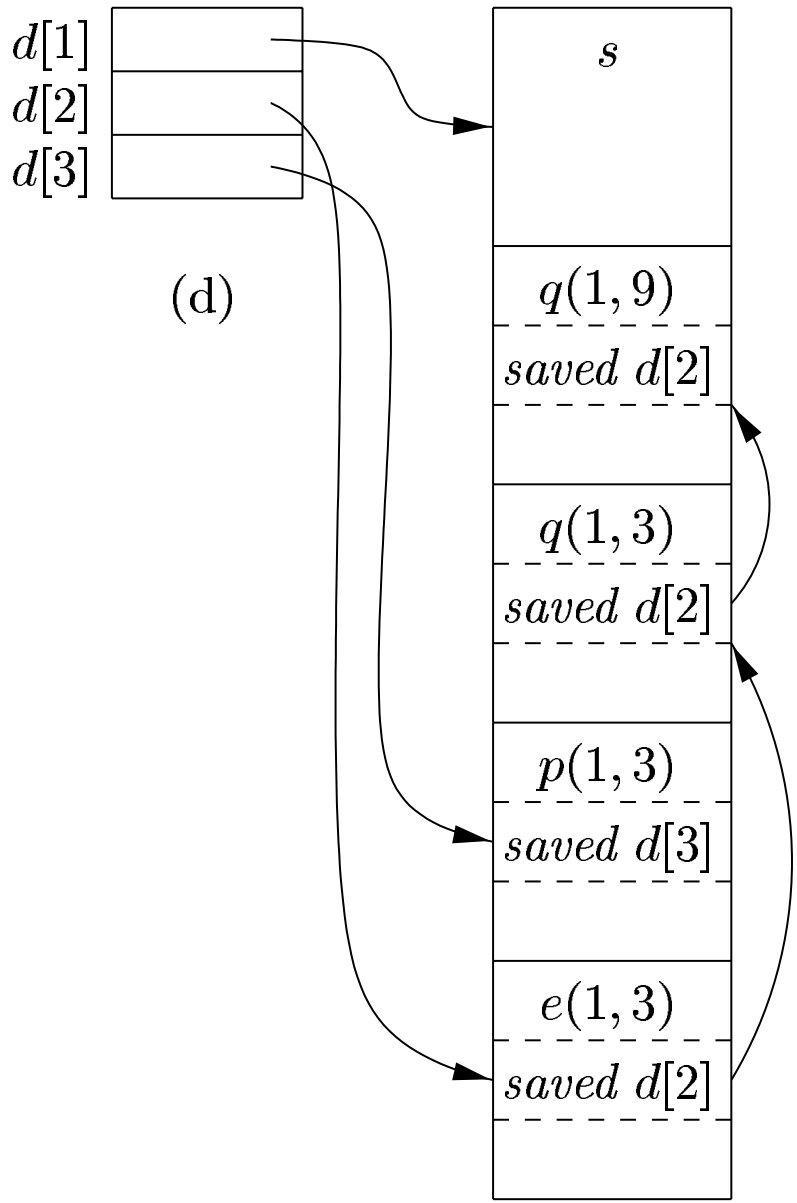
Display: Example





Display: Example





Parameter passing

What about parameters?

Call-by-value

- store values, not addresses
- never restore on return
- arrays, structures, strings are a problem

Call-by-reference

- pass address
- access to formal is indirect reference to actual

Parameter passing

What about variable length argument lists?

1. if caller knows that callee expects a variable number
   1. caller can pass number as 0^{th} parameter
   2. callee can find the number directly
2. if caller doesn't know anything about it
   1. callee must be able to determine number
   2. first parameter must be closest to `FP`

Consider `printf` :

- number of parameters determined by the format string
- it assumes the numbers match

Heap Management

- Data that outlives the procedure that creates it cannot be kept in the Activation record.
  - E.g. Objects created using `new` in Java, `malloc` in C/C++.
- Dynamically allocated data is stored on the heap.
- Heap Management techniques typically deal with allocation and deallocation problems.
  - Finding a contiguous segment of required size, avoiding fragmentation, etc.
  - Automatic/manual reclamation of space; Garbage Collection