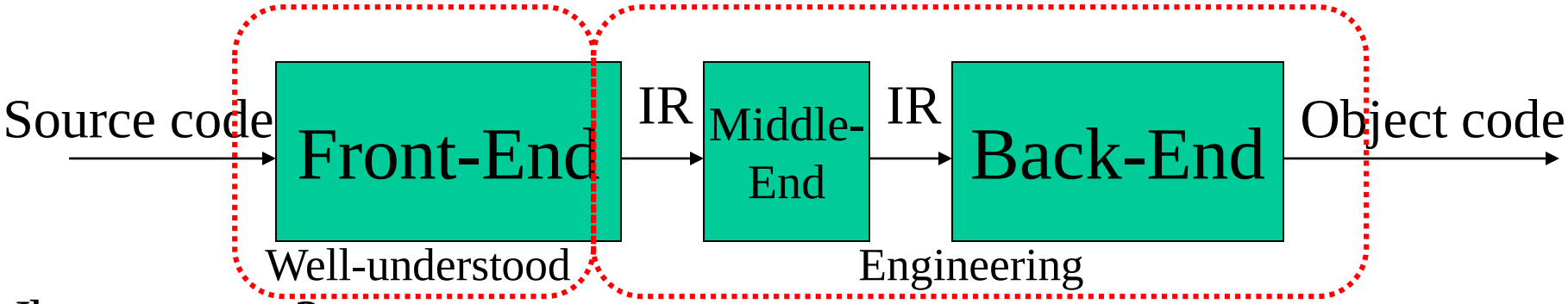
# The Procedure Abstraction; Run-Time Storage Organisation




Where are we?

- We crossed the dividing line between the application of well-understood technology and fundamental issues of design and engineering. The complications of compiling begin to emerge!
- The second half contains more open problems, more challenges, and more gray areas that the first half
  - This is compilation as opposed to parsing or translation (engineering as opposed to theory: imperfection, trade-off, constraints, optimisation)
  - Needs to manage target machine resources
  - This is where legendary compilers are made...

Today's lecture:

- The Procedure Abstraction and Run-Time Storage Organisation

# The Procedure

- **Procedures are the key to building large systems; they provide:**
  - Control abstraction: well-defined entries & exits.
  - Name Space: has its own protected name space.
  - External Interface: access is by name & parameters.
- **Requires system wide-compact:**
  - broad agreement on memory layout, protection, etc…
  - must involve compiler, architecture, OS
- **Establishes the need for private context:**
  - create a run-time “record” for each procedure to encapsulate information about control & data abstractions.
- **Separate compilation:**
  - allows us to build large systems; keeps compile-time reasonable

# The Procedure: A more abstract view

- A procedure is a collection of fictions.
- Underlying hardware supports little of this fiction:
  - well-defined entries and exits: mostly name-mangling
  - call/return mechanism: often done in software
  - name space, nested scopes: hardware understands integers!
  - interfaces: need to be specified.
- The procedure abstraction is a deliberate deception, produced collaboratively by the OS & the compiler.

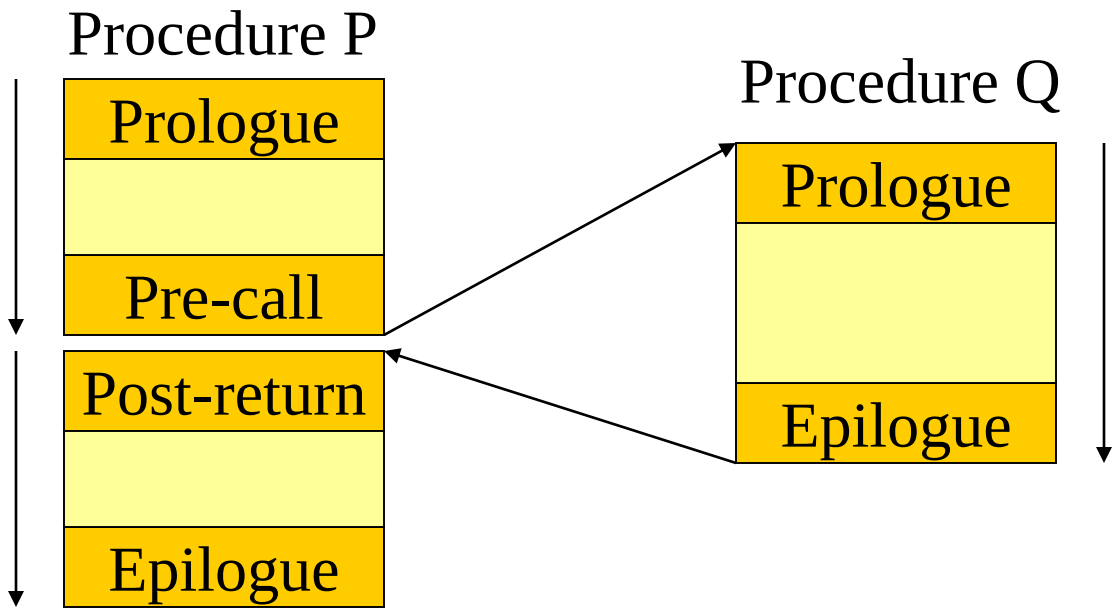
*One view holds that computer science is simply the art of realising successive layers of abstraction!*

# The linkage convention

Procedures have well-defined control-flow behaviour:

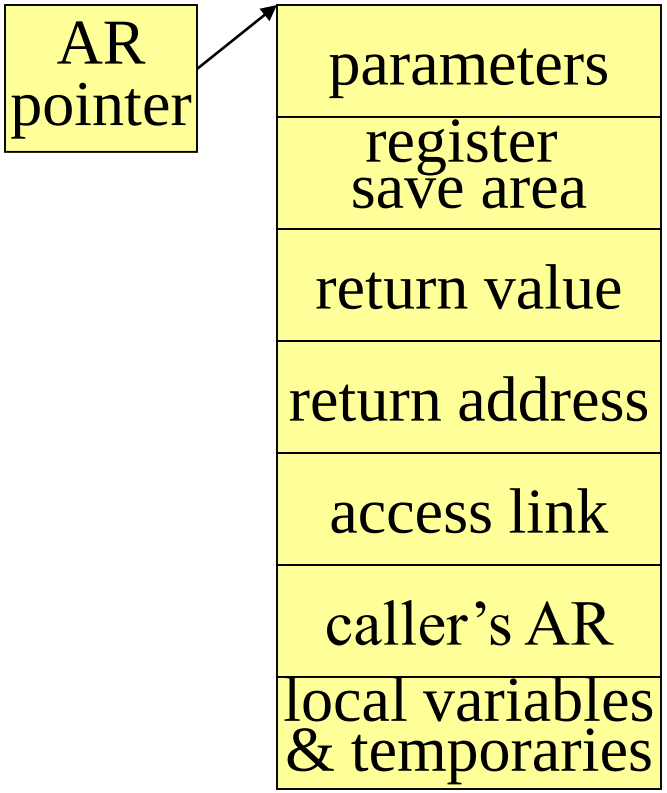
- A protocol for passing values and program control at procedure call and return is needed.
- The linkage convention ensures that procedures inherit a valid run-time environment and that they restore one for their parents.

- Linkages execute at run-time.
- Code to make the linkage is generated at compile-time.



# Storage Organisation: Activation Records

- Local variables require storage during the lifetime of the procedure invocation at run-time.
- The compiler arranges to set aside a region of memory for each individual call to a procedure (run-time support): activation record:




Activation Records (ARs here) are also known as stack frames.

In general, the compiler is free to choose any convention for the AR. The manufacturer may want to specify a standard for the architecture.

→ Address to resume caller

→ Help with non-local access

→ Pointer to caller's activation record

# Procedure linkages

(the procedure linkage convention is a machine-dependent contract between the compiler, the OS and the target machines to divide clearly responsibility)

**Caller (pre-call):**
- allocate AR
- evaluate and store parameters
- store return address
- store self's AR pointer
- set AR pointer to child
- jump to child

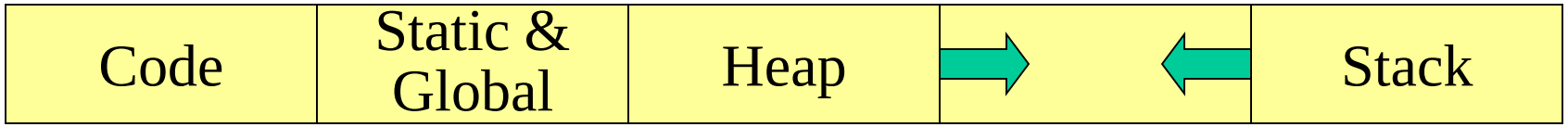
**Callee (prologue):**
- save registers, state
- extend AR for local data
- get static data area base address
- initialise local variables
- fall through to code

**Callee (epilogue):**
- store return value
- restore registers, state
- unextend basic frame
- restore parent's AR pointer
- jump to return address

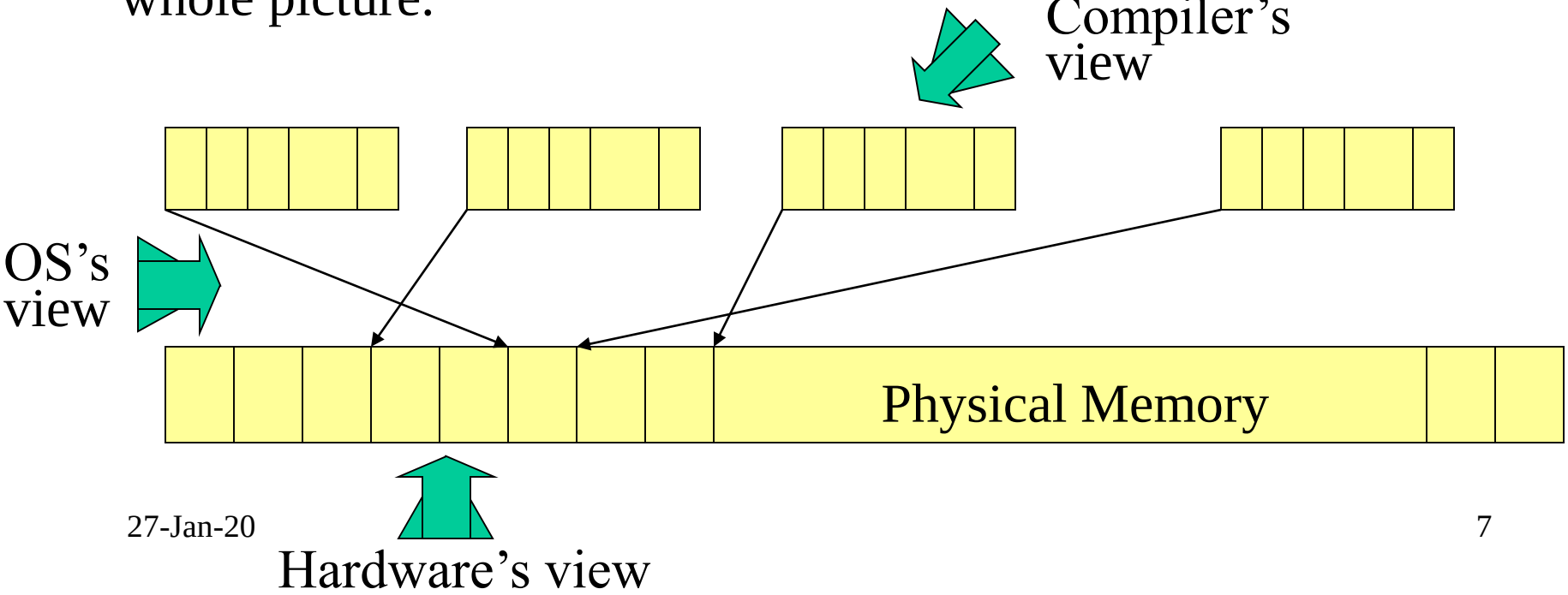
**Caller (post-return):**
- copy return value
- deallocate callee's AR
- restore parameters (if used for call-by reference)

# Placing run-time data structures

Single logical address space:




- Code, static, and global data have known size.
- Heap & stack grow towards each other.
- From the compiler's perspective, the logical address space is the whole picture.



# Activation Record Details

- How does the compiler find the variables?
  - They are offsets from the AR pointer.
  - Variable length-data: if AR can be extended, put it below local variables; otherwise put on the heap.
- Where do activation records live?
  - If it makes no calls (leaf procedure - hence, only one can be active at a time), AR can be allocated statically.
  - Place in the heap, if it needs to be active after exit (e.g., may return a pointer that refers to its execution state).
  - Otherwise place in the stack (this implies: lifetime of AR matches lifetime of invocation and code normally executes a “return”).
  - (in decreasing order of efficiency: static, stack, heap)

# Run-time storage organisation

The compiler must ensure that each procedure generates an address for each variable that it references:

- Static and Global variables:
  - Addresses are allocated statically (at compile-time). (relocatable)
- Procedure local variables:
  - Put them in the activation record if: sizes are fixed and values are not preserved.
- Dynamically allocated variables:
  - Usually allocated and deallocated explicitly.
  - Handled with pointers.

# Establishing addressability

- Local variables of current procedure:
  - If it is in the AR: use AR pointer and load as offset.
  - If in the heap: store in the AR a pointer to the heap (double indirection).
  - (both the above need offset information)
  - If in a register: well, it is there!
- Global and static variables:
  - Use a relocatable (by the OS's loader) label (no need to emit code to determine address at run-time).
- Local variables of other procedures:
  - Need to retrieve information from the "other" procedure's AR.

# Addressing non-local data

In a language that supports nested lexical scopes, the compiler must provide a mechanism to map variables onto addresses.

- The compiler knows current level of lexical scope and of variable in question and offset (from the symbol table).
- Needs code to:
  - Track lexical ancestry (not necessarily the caller) among ARs.
  - Interpret difference between levels of lexical scope and offset.
- Two basic mechanisms:
  - Access links
  - Global display.

# Access Links

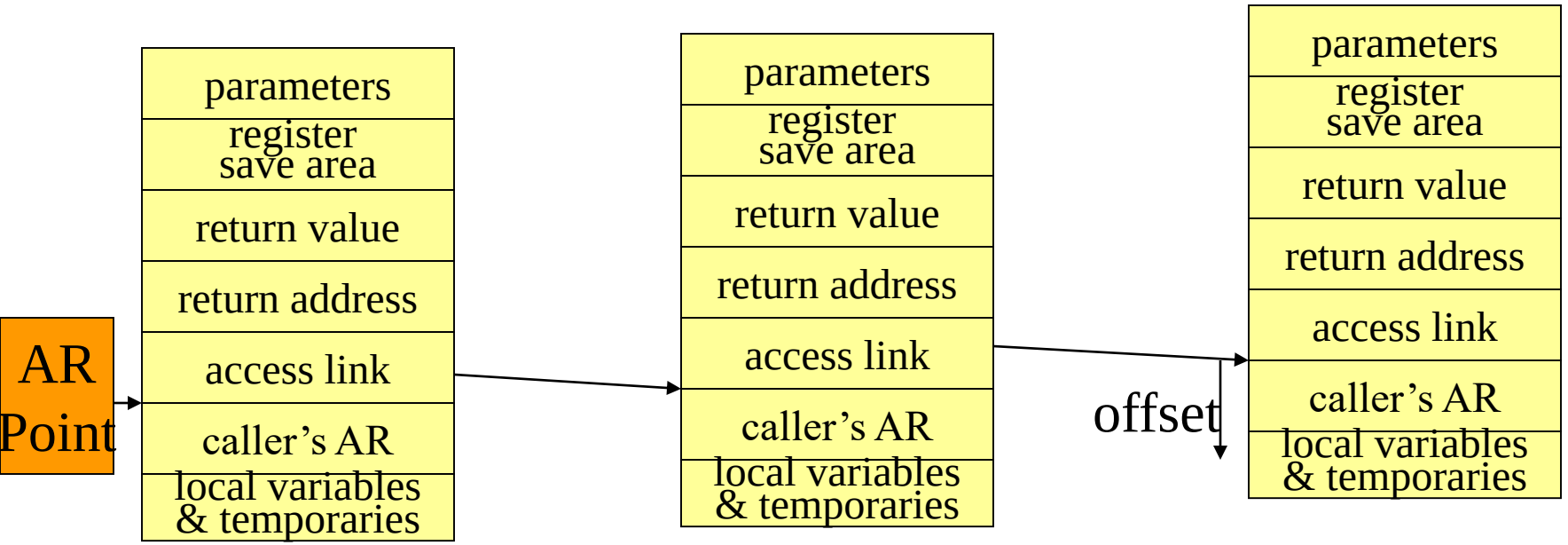
Idea: Each AR contains a pointer to its lexical ancestor.

Compiler needs to emit code to find lexical ancestor (if caller's scope=callee's scope+1 then it is the caller; else walk through the caller's ancestors)

Cost of access depends on depth of lexical nesting. Example:

(current level=2): needs variable at level=0, offset=16:

load r1,(ARP-4); load r1,(r1-4); load r2,(r1+16)



# Global Display

Idea: keep a global array to hold ARPs for each level.

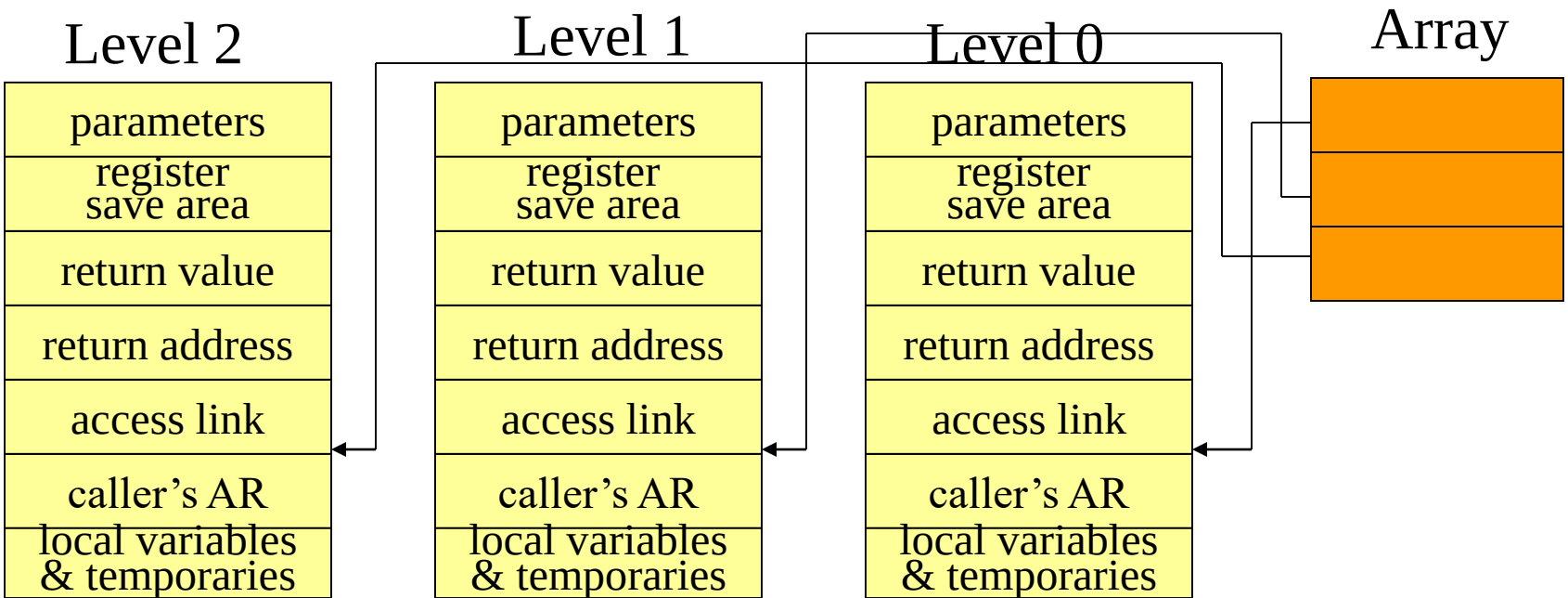
Compiler needs to emit code (when calling and returning from a procedure) to maintain the array.

Cost of access is fixed (table lookup + AR). Example:

(current level=2): needs variable at level=0, offset=16:

load r1,(DISPLAY_BASE+0); load r2,(r1+16)

Display vs access links trade-off. conventional wisdom: use access links when tight on registers; display when lots of registers.



# Other (storage related) issues

- Target machines may have requirements on where data items can be stored (e.g., 32-bit integers begin on a full word boundary). The compiler should order variables to take into account this.
- Cache performance: if two variables are used in near proximity in the code, the compiler needs to ensure that they can reside in the cache at the same time. Complex to consider more variables.
- Conventional wisdom: tight on registers: use access links; lots of registers: use global display.
- Memory to memory model vs register to register model.
- Managing the heap: first-fit allocation with several pools for common sizes (usually powers of 2 up to page size).
  - Implicit deallocation: reference counting (track the number of outstanding pointers referring to an object); garbage collection (when there is no space, stop execution and discover objects that can be reached from pointers stored in program variables; unreachable space is recycled).
- Object-oriented languages have more complex name spaces.

# Finally…

- The compiler needs to emit code for each call to a procedure to take into account (at run-time) procedure linkage.
- The compiler needs to emit code to provide (at run-time) addressability for variables of other procedures.
- **Inlining**: the compiler can avoid some of the problems related to procedures by substituting a procedure call with the actual code for the procedure. There are advantages from doing this, but it may not be always possible (can you see why?) and there are disadvantages too.

- To discuss Next:
  - Instruction selection/code generation; register allocation; instruction scheduling.
  - Code optimisations.