Procedure and Function Calls,  
— *Part II* —  
Comp 412
# Lab 3 Schedule

<table>
<thead>
<tr>
<th>Sunday</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
<th>Saturday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Start work</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Complete &amp; Correct</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Dependence Graph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Correct Schedule</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Improve Schedules</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lab 3 Due Date</td>
<td>Thanksgiving Break</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **You are here:**
  - Monday, 27th of October
  - Start work
  - Tutorial 4PM
  - Lab 3 Lecture

- **The goal is here:**
  - Thanksgiving Break

- **Lab 3 Due Date:**
  - Monday, 24th of November

- **Late day:**
  - Monday, 31st of October

- **Not a late day:**
  - Tuesday, 1st of November
  - Wednesday, 2nd of November
  - Thursday, 3rd of November
  - Friday, 4th of November
  - Saturday, 5th of November

- **Walk:**
  - Sunday, 21st of October

**Thank you.**
The linkage must:

- Preserve & protect the caller’s environment from callee (& its callees)
  - For example, it must preserve values in registers that are live across the call
- Create a new environment for the callee (new name space)
  - At runtime, it creates a local data area for the callee & ties it to the context
- Map names & resources from caller into the callee, as needed

To accomplish these goals:

- Convention divides responsibility between caller & callee
- Caller & callee must use the same set of conventions
  - Implies limited opportunities for customization & optimization

We expect compiled code to work even when created with different compilers, perhaps in different languages.
Linkage Convention

A procedure or function has two distinct representations

**Code to implement the procedure**

- Code for all the statements
- Code to implement the linkage convention
- Labels on entry points and return points
  - Use name mangling to create consistent labels

**Data to represent an invocation**

- Activation record (AR)
- Local data + saved context of caller
- Control information
- Storage to preserve caller’s env’t
  - Chain ARs together to create context
- Static data area (name mangling)
The Meta Issue

One of the most difficult aspects of understanding how compilers work is keeping track of what happens at compiler-design time, compiler-build time, compile time, and run time.

• At compile time, the compiler emits code for the application program, including all of its procedures (and their prologs, epilogs, & sequences)
  – The compiler uses symbol tables to keep track of names
  – Those tables are compile-time data structures

• At runtime, those sequences (prolog, epilog, pre-call, & post-return) create, populate, use, and destroy ARs for invocations of procedures
  – The running code stores data (both program data and control data) in the ARs
  – Those ARs are run-time data structures

To confuse matters further, the compiler may preserve the symbol tables so that the runtime debugging environment can locate variables and values, and locate the various data areas and the ARs
Each procedure has
- A standard prolog
- A standard epilog

Each call uses
- A pre-call sequence
- A post-return sequence

The code for the pre-call and post-return sequences must be completely predictable from the call site. It depends on the number & the types of the actual parameters.

Note: The gap between pre-call & post-return code is artistic license.
In most systems, Activation Records have a similar layout:

- **Space for parameters**: Space for parameters to the current procedure.
- **Register save area**: Contents of saved registers.
- **Slot for return value**: If function, space for return value.
- **Slot for return address**: Address to resume caller.
- **Slot for addressability**: Help with non-local access.
- **Slot for caller’s ARP**: To restore caller’s ARP on return.
- **Space for local variables**: Space for local variables, temporaries, & spills.

One AR for each invocation of a procedure.
One register dedicated to hold the current ARP.
Procedure Linkages

Pre-call Sequence
• Sets up callee’s basic AR
• Helps preserve its own environment

The Details
• Allocates space for the callee’s AR
  – except space for local variables
• Evaluates each parameter & stores value or address
• Saves return address & caller’s ARP into callee’s AR
• Addressability maintenance, if necessary
• Saves any caller-save registers that are live across the call
  – Save into space in caller’s AR
• Jumps to address of callee’s prolog code

Pre-call sequence is a lot of work

Where do parameter values reside?
Conventional wisdom says:
• In registers (First 3 or 4)
• In callee’s AR (the rest)
Procedure Linkages

Post-return Sequence
• Finishes restoring caller’s environment
• Places any value back where it belongs
  – e.g., any reference parameter or global that is kept in a register in the callee

The Details
• Copies return value from AR into a register, if needed
• Frees the callee’s AR
  – *If access links are used, this action discards callee’s link*
• Restores any caller-save registers
• Restores any call-by-reference parameters to registers, if needed
  – *Also copy back call-by-value/result parameters*
• Continues execution after the call
Procedure Linkages

Prolog Code
- Finishes setting up callee’s environment
- Preserves parts of caller’s environment that will be disturbed by the callee

The Details
- Preserves any callee-save registers
- Addressability maintenance, if necessary
- Allocates space for local data
  - *Easiest scenario is to extend the AR*
- Finds any static data areas referenced in the callee
- Handles any local variable initializations
  - *Not any cheaper to say “int x = 0;”*

With heap allocated AR, may need a separate heap object for local variables. (Must know size of local data area)
Procedure Linkages

Epilog Code
• Winds up the business of the callee
• Starts restoring the caller’s environment

The Details
• Stores return value?
  – Place it, or a pointer, in the return value slot of the caller’s AR\(^1\)
  – Other schemes are equally feasible
• Restore callee-save registers, as needed
• Addressability maintenance, if necessary
• Free space for local data, if necessary (on the heap)
• Restore return address & caller’s ARP from AR
• Jump to the return address

Takeaway point: procedure call overhead is many cycles.

---

\(^1\) Return value needs storage that survives the call
Where Do Saved Registers Go?

- Space for parameters to the current procedure
- Contents of saved registers
- If function, space for return value
- Address to resume caller
- Help with non-local access
- To restore caller’s AR on return
- Space for local variables, temporary, & spills

If $p$ always saves registers into its own AR, then the compiler knows how much space to reserve in the AR for those purposes — $p$’s own caller saves registers (in **prolog**), $p$’s own callee saves registers (in **pre-call** when $p$ calls $q$), and $p$’s own spills in the register allocator run on $p$. (**prolog patched after allocation**)
Where Do Parameters Go?

- **Space for parameters**
  - register save area
  - slot for return value
  - slot for return address
  - slot for addressability
  - slot for caller’s ARP
- **Space for local variables**

**Parameters**
- Parameters from callee go here
- Parameters passed to another call go here

**ARP**
- Contents of saved registers
- If function, space for return value
- Address to resume caller
- Help with non-local access
- To restore caller’s ARP on return
- Space for local variables, temporaries, & spills

Callee’s activation record
Creating and Destroying Activation Records

The activation record embodies the runtime state of the procedure

• How are ARs created and destroyed?
  – Procedure call must allocate & initialize callee’s AR (preserve caller’s world)
  – Return must dismantle environment (and restore caller’s world)

• Caller & callee must collaborate on the problems
  – Caller alone knows some of the necessary state
    → Return address, parameter values, access to other scopes
  – Callee alone knows the rest
    → Size of local data area (with spills), registers it will use

Their collaboration forms the essence of the linkage convention

1. Linkage convention is defined at compiler design time
2. Linkage code is emitted at compile time
3. Linkage code executes at runtime, when a procedure is called.

Assume, for the moment, an Algol-60-like environment where the AR is dead after the return.
Storing Activation Records

If activation records are stored on the stack

• Easy to create and to destroy: push and pop on runtime stack
• Easy to extend — simply bump top of stack pointer
• Caller & callee share responsibility
  — Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own ARP
  — Callee can push space for local variables (fixed & variable size)
Address Space Layout

Most language runtimes layout the address space in a similar way:

- Pieces (stack, heap, code, & globals) may move, but all will be there.
- Stack and heap grow toward each other (if heap grows).

Java Memory Layout
Storing Activation Records

If activation records are stored on the stack
• Easy to create and to destroy: push and pop on runtime stack
• Easy to extend — simply bump top of stack pointer
• Caller & callee share responsibility

If activation records are stored on the heap
• Explicit allocation; either explicit or implicit free
• Hard to extend
• Several options
  – Caller passes everything in registers; callee allocates & fills AR
  – Store parameters, return address, etc., in caller’s AR!
  – Store callee’s AR size in a defined static constant

NAME MANGLING, AGAIN
Address Space Layout

Most language runtimes layout the address space in a similar way:

- Pieces (stack, heap, code, & globals) may move, but all will be there.
- Stack and heap grow toward each other (if heap grows).

Java Memory Layout
Storing Activation Records

If activation records are stored on the stack
• Easy to create and to destroy: push and pop on runtime stack
• Easy to extend — simply bump top of stack pointer
• Caller & callee share responsibility

If activation records are stored on the heap
• Explicit allocation; either explicit or implicit free
• Hard to extend
• Several options

Without recursion, activation records can be static
• Same observation applies to a procedure that contains no calls
Most language runtimes layout the address space in a similar way:

- Pieces (stack, heap, code, & globals) may move, but all will be there.
- Stack and heap grow toward each other (if heap grows).

Statically allocated ARs go into the “global” data area — which we would need to redraw in as a larger area ...

STOP?
Where Do Variables Live?

We have seen ARs, static data areas, global data areas, ...

How does the compiler decide where to place each variable?
• A combination of visibility and lifetime determines that placement

<table>
<thead>
<tr>
<th>Lifetime</th>
<th>Scope</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic</td>
<td>local</td>
<td>declaring procedure’s AR</td>
</tr>
<tr>
<td>static</td>
<td>any</td>
<td>named static data area for its lifetime</td>
</tr>
<tr>
<td>dynamic</td>
<td>any</td>
<td>heap</td>
</tr>
</tbody>
</table>

Variable length items?
• Keep descriptor in its “natural” place & allocate space in AR or heap
  — Requires one level of indirection (a “pointer”)
  — Allows uniform addressability
  — In AR if AR is extendible
  — On heap if AR is not extendible

Automatic: Lifetime matches procedure activation
Static: Lifetime may be as long as entire execution
Dynamic: Lifetime is under program control & not known at compile time

COMP 412, Fall 2018
How Do We Address Variables?

Local variables

• Need a mechanism to locate the local data area in appropriate AR
• Represent the variable as a static coordinate: <level, offset>
  – Level is the lexical nesting level of the procedure that declares the variable
  – Offset is the offset within the AR’s local data area for that procedure
• Mechanism takes static coordinate to run-time address

Static variables (including global)

• Need a base-address, offset pair (mangled name for base address)
  – Emit a load of the base address
  – Emit an add of offset and a load (loadA if offset is small enough)
• Must be recognizable as non-local case (represent differently)

Dynamic variables

• Programmer manages access through pointers, names, ...

STOP