Procedure and Function Calls

Comp 412
Expanding the Expression Grammar

The Classic, Left-Recursive, Expression Grammar

References to aggregates & functions

• First, the compiler must recognize expressions that refer to arrays, structures, and function calls
• Second, the compiler must have schemes to generate code to implement those abstractions
Conceptual Overview

Procedures and functions provide the fundamental abstractions that make programming practical & large software systems possible

• Information hiding
• Distinct and separable name spaces
• Uniform interfaces

Hardware does little to support these abstractions

• Part of the compiler’s job is to implement them
  – Compiler must ensure that what the manual says appears to be true.
• Part of the compiler’s job is to make that support efficient
  – Role of code optimization

“Functions are pieces of code that you write now and execute later..”
Scott Rixner, Coursera Python Course

Object-oriented Languages (OOLs) involve just a couple of twists on Algol-like Languages (ALLs). Those differences primarily involve naming. In subsequent lectures, we will look at the runtime support needed for naming in ALLs and OOLs.
Procedures & Functions

A procedure is an abstract structure constructed via software

Underlying hardware directly supports little of the abstraction—it understands bits, bytes, integers, reals, & addresses, but not:

• **Entries** and **exits**
• **Interfaces**
• **Call** and **return** mechanisms
  
  Typical ISA supports the transfer of control (call and return) but not the rest of the call / return sequence (e.g., preserving context)

• **Name space**
• **Nested scopes**

All these are established by carefully-crafted mechanisms provided by compiler, run-time system, linker, loader, and OS; together, we call these mechanisms the **linkage convention**
Function Calls

One possible leaf in an expression tree is a function call

\[ r \leftarrow f(x) + c \]

What code does the compiler generate for \( f(x) \)?

- Compiler may not have access to code for \( f() \)? (it may not be written yet)
- Another compilation or another compiler may have translated \( f() \)
  - \( f(x) \) might be a system call, such as \( \text{mmap}() \) or \( \text{malloc}() \)

Programmers expect such calls to “just work”

The linkage convention is the social contract that lets it “just work.”
Linkage Convention

The linkage convention plays a critical role in construction of systems

• Makes separate compilation possible
  – Compiler can generate code to the convention’s specifications
  – Adherence to the convention means that “things just work”

• Divides responsibility for the runtime environment & behavior among the caller, the callee, & the system
  – Naming and address space management
  – Control flow among functions & procedures (& to some extent processes)

The linkage convention is typically a system-wide convention

• Allows interoperability among languages & compilers
  – Code compiled with gcc can link with code from clang

• Admits only limited opportunities for customization & optimization
Linkage Convention: High-Level View

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

foe must operate in multiple distinct calling contexts.
In essence, the procedure linkage code wraps around each unique procedure to give it a uniform interface for naming & control. It allows the compiler to generate calls without seeing the callee.

Similar to building a brick wall rather than a rock wall.
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.

Procedure Linkages

Standard Procedure Linkage

procedure

prolog

... z = a*a + b*b
x = sqrt(y,z)

pre-call

L: post-return

... epilog
caller

function sqrt

prolog

... epilog
callee

Note: The gap between pre-call & post-return code is artistic license.
These four code sequences:

- Establish a local data area for the callee; we call it an activation record (AR)
- Populate that activation record with needed values & context
- Preserve parts of the caller’s environment that might be modified by callee
  - Contents of registers, pointer to its activation record, …
- Handle parameter passing
  - Evaluate parameters in caller
  - Make them available in callee
- Transfer control from caller to callee & back (LIFO behavior)
Procedure Linkages

Standard Procedure Linkage

Preserving values for the caller:
Any local values in the caller are either stored in the caller’s AR and, therefore, safe, \(^1\) or they reside in a register.

- Must preserve value in registers that are live across the call
- Store them in caller’s AR or in callee’s AR

Who is responsible for registers?
Conventional wisdom splits the task between caller and callee

- Set of **caller-saves** registers
- Set of **callee-saves** registers

Saver stores into its own AR

---

\(^1\) Local variables of the caller are safe because they live in the caller’s AR and the callee uses a new AR.
Caller-Saves vs. Callee-Saves Registers

Most linkage conventions (now) split the register set

- Take the registers available to the allocator
  - Assign half to the caller and half to the callee
  - Division is by PR name
- Half saved by caller; half saved by callee
- Procedure fee always saves into its own AR
  - The “register save area” holds fee’s own callee-saves register from the prolog and its own caller-saves registers from any pre-call sequences in fee

<table>
<thead>
<tr>
<th>Example Assignment</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PR0 to 5</td>
<td>Reserved for $r_{ARP}$, parameters, stacktop</td>
</tr>
<tr>
<td>PR6 to 18</td>
<td>Caller saves registers</td>
</tr>
<tr>
<td>PR19 to 31</td>
<td>Callee saves registers</td>
</tr>
</tbody>
</table>

This arrangement lets the compiler target VRs to specific kinds of PRs

- Value that is **not** live across a call $\rightarrow$ caller-saves register (*needs no save*)
- Value that is live across a call $\rightarrow$ callee-saves register (*let callee do work*)

With small procedures, can avoid some spurious saves & restores

---

1 As with many register use issues, this point is subtle and often misunderstood. When compiling fee, the compiler tries to avoid saves & restores in fee, whether or not that makes global sense. It is a simple greedy heuristic.
What is this “Activation Record”?

A procedure’s *activation record* (AR) combines the control information needed to manage procedure calls (*return address, register save area, addressability, & return value*) and the procedure’s local data area(s)

- Each time a procedure is invoked, or *activated*, it needs an AR
  - Allocated on a runtime stack or in the runtime heap *(depends on needs)*
- The pre-call & post-return sequences manage the ARs
  - Pre-call creates AR & initializes it
  - Post-return cleans up the AR & deallocates it
- The prolog & epilog sequences manipulate environment data in the AR
  - Create & destroy parts of the new environment
  - Preserve & restore parts of the old environment

ARs are *runtime structures*, laid out at compile time
  - Created & destroyed at runtime by code generated at compile time

In Java, people refer to ARs as *frames*. You should think of them as ARs, which is the more general term.

Meta-Issue: Activation records are created, used and destroyed at runtime. The code to maintain them is emitted at compile time.
Activation Record Basics

In most systems, Activation Records have a similar layout

<table>
<thead>
<tr>
<th></th>
<th>Space for parameters</th>
<th>Space for parameters to the current procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>register save area</td>
<td>Contents of saved registers</td>
</tr>
<tr>
<td></td>
<td>slot for return value</td>
<td>If function, space for return value</td>
</tr>
<tr>
<td></td>
<td>slot for return address</td>
<td>Address to resume caller</td>
</tr>
<tr>
<td></td>
<td>slot for addressability</td>
<td>Help with non-local access</td>
</tr>
<tr>
<td></td>
<td>slot for caller’s ARP</td>
<td>To restore caller’s AR on return</td>
</tr>
<tr>
<td></td>
<td>space for local variables</td>
<td>Space for local variables, temporaries, &amp; spills</td>
</tr>
</tbody>
</table>

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
• Allocate 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
• Save any caller-save registers
  – Save into space in caller’s AR
• Jump 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**
- Finish restoring caller’s environment
- Place any value back where it belongs
  - e.g., reference parameter or global that kept in a register in the callee

**The Details**
- Copy return value from callee’s AR, if necessary
- Free the callee’s AR
  - *If access links are used, this action discards callee’s link*
- Restore any caller-save registers
- Restore any call-by-reference parameters to registers, if needed
  - *Also copy back call-by-value/result parameters*
- Continue execution after the call
Procedure Linkages

Prolog Code
• Finish setting up callee’s environment
• Preserve parts of caller’s environment that will be disturbed by the callee

The Details
• Preserve any callee-save registers
• Addressability maintenance, if necessary
• Allocate space for local data
  — Easiest scenario is to extend the AR
• Find any static data areas referenced in the callee
• Handle 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.
Procedure Linkages

**Takeaway point:** procedure calls have many instructions of overhead.

**Epilog Code**

- Wind up the business of the callee
- Start restoring the caller’s environment

**The Details**

- Store return value?
  - *Value designated in a return statement*
  - *Epilog moves it to designated slot in caller’s AR*
- Restore callee-save registers
- Addressability maintenance, if necessary
- Free space for local data, if necessary *(on the heap)*
- Load return address from AR
- Restore caller’s ARP
- Jump to the return address

If ARs are stack allocated, deallocation is implicit (& cheap) when stacktop is reset to its pre-call value.
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, temporaries, & spills

If $p$ saves its own registers in its own RSA, then $p$ saves its own local data area.

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?

- **Parameters from callee go here**
  - **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**
  - **space for parameters**

- **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, temporaries, & spills**

---

Callee’s activation record

COMP 412, Fall 2017
Creating and Destroying Activation Records

The activation record embodies the state of the procedure

• How are ARs created and destroyed?
  – Procedure call must allocate & initialize
    (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 takes the form of a linkage convention

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 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)
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).
Storing Activation Records

If activation records are stored on the 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)

If activation records are stored on the heap

- 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)
Storing Activation Records

If activation records are stored on the 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)

If activation records are stored on the heap

• 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

Without recursion, activation records can be static

COMP 412, Fall 2017
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 ...