Procedure and Function Calls

Comp 412

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 6 in EaC2e
Expanding the Expression Grammar

The Classic, Left-Recursive, Expression Grammar

We have seen this slide before ...

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

Array reference
Function invocation
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 these abstractions
  — Compiler must ensure that what the manual says appears to be true.
• Part of the compiler’s job is to make that support efficient
  — Code optimization should remove unneeded overhead from implementation

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.

“We use the terms procedure & function interchangeably. Technically, a function is a procedure that returns a value.”

Scott Rixner, Coursera Python Course

“Functions are pieces of code that you write now and execute later.”

Scott Rixner, Coursera Python Course
A procedure is an abstract structure built of 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 & 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

Consider the following expression, where f is a function

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

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

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

- Compiler must generate code to invoke it & to handle its return

Programmers expect such calls to “just work”

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

The linkage convention is typically a system-wide convention

- Allows interoperability among languages & compilers
  - Code compiled with gcc can link with code from clang or g++ or gfortran
- Limits opportunities for customization & optimization

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)
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.
In essence, the procedure linkage code wraps around each unique procedure to give it a uniform interface for naming & control

⇒ It allows 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.

**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 AR with needed values & context
- Preserve parts of the caller’s environment that might be modified by callee
  - Contents of registers, pointer to its AR, addressability info ...
- Handle parameter passing
  - Evaluate parameters in caller
  - Make them available in callee
- Transfer control from caller to callee & back (**LIFO** behavior)
Preserving values for the caller:
Local values in the caller are either stored in the caller’s AR and, thus, 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>
</tr>
</thead>
<tbody>
<tr>
<td>PR0 to 5</td>
</tr>
<tr>
<td>Reserved for r_{ARP}, parameters, stacktop</td>
</tr>
<tr>
<td>PR6 to 18</td>
</tr>
<tr>
<td>Caller saves registers</td>
</tr>
<tr>
<td>PR19 to 31</td>
</tr>
<tr>
<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 → caller-saves register (needs no save)
- Value that is live across a call → callee-saves register (let callee do work)

With small procedures, can avoid some spurious saves & restores

1 If fee contains no calls, it can allocate a register save area that only holds the callee-saved registers.

2 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. They are created & destroyed at runtime by code that the compiler generates at compile time.

In Java, people refer to ARs as frames or stack 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

- 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

**ARP** ≡ Activation Record Pointer

- One AR for each invocation of a procedure
- One register dedicated to hold the current ARP

---

If ARs are stack allocated, the stack grows in this direction

If ARs are stack allocated, the stack grows in this direction

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

STOPPED