Review

Last week

- Control Structures
- Selection
- Loops
Outline

- Subprograms
- Calling Sequences
- Parameter Passing
- Recursion

Sources:

PLP, 3.5, 6.6, 7.4, 8.1 - 8.3


Barrett. Lecture notes, Fall 2009.
Subprograms

• The basic abstraction mechanism

• *Functions* correspond to the mathematical notion of computation:

  \[ \text{input} \rightarrow \text{output} \]

• *Procedures* affect the environment, and are called for their side-effects

• Pure functional model possible but rare (HASKELL)

• Hybrid model most common: functions can have (limited) side effects
Activation Records

Recall that subroutine calls rely heavily on use of the stack.

Each time a subroutine is called, space on the stack is allocated for the objects needed by the subroutine. This space is called a stack frame or activation record.

The stack pointer contains the address of either the last used location or the next unused location on the stack.

The frame pointer points into the activation record of a subroutine so that any objects allocated on the stack can be referenced with a static offset from the frame pointer.
Activation Records

*Why not use an offset from the stack pointer to reference subroutine objects?*

The stack pointer changes, with nested scopes and function/procedure calls.

Also, there may be objects that are allocated on the stack whose size is unknown at compile time.

These objects get allocated above the frame pointer so that objects whose size is known at compile time can still be accessed quickly.
Activation Records

*Why not use an offset from the stack pointer to reference subroutine objects?*

There may be objects that are allocated on the stack whose size is unknown at compile time.

These objects get allocated last so that objects whose size is known at compile time can still be accessed quickly via a known offset from the frame pointer.

**Example**

```plaintext
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
  ...
end
```
# Typical Activation Record

<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>Frame pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack pointer</td>
<td>Variable-length Objects</td>
</tr>
<tr>
<td></td>
<td>Temporaries</td>
</tr>
<tr>
<td></td>
<td>Local Variables</td>
</tr>
<tr>
<td></td>
<td>Saved Values of Registers etc.</td>
</tr>
<tr>
<td></td>
<td>Return address</td>
</tr>
</tbody>
</table>
Managing Activation Records

When a subroutine is called, a new activation record is created and populated with data.

The management of this task involves both the caller and the callee.

- The calling sequence refers to code executed by the caller just before and just after a subroutine call.
- The prologue refers to activation record management code executed at the beginning of a subroutine.
- The epilogue refers to activation record management code executed at the end of a subroutine.

Sometimes the term calling sequence is used to refer to the combined operations of the caller, prologue, and epilogue.
Calling Sequence

Calling a subroutine

In the caller

- Store any caller-saved registers
- Place arguments in registers and/or stack
- Compute static link and pass as extra argument
- (Save return address on stack)
- Jump to subroutine

In the callee

- Allocate frame by changing stack pointer
- Save old frame pointer and update with new value
- Save any callee-saved registers
- Initialize objects
Calling Sequence

Finishing a subroutine

In the callee

- Move return values (if any) into registers and/or stack
- Restore callee-saved registers
- Restore frame and stack pointers
- Jump back to return address

In the caller

- Save return values
- Restore caller-saved registers
Calling a Subroutine

Stack pointer $\rightarrow$

Frame pointer $\rightarrow$

Calling Sequence (before)

Caller

Activation Record
Typical Calling Sequence

Calling Sequence (before)
1. Save caller-save registers
Typical Calling Sequence

- Stack pointer → Frame pointer

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
Typical Calling Sequence

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
### Typical Calling Sequence

#### Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

#### Prologue
1. Save old fp, set new fp

<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>Frame pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Pointer</td>
<td>Frame Pointer</td>
</tr>
</tbody>
</table>

- Saved fp
- Return address
- Arguments
- Caller-Saved Registers
- Caller Activation Record
Typical Calling Sequence

Stack pointer

Frame pointer

Callee-Saved

Registers

Saved fp

Return address

Arguments

Caller-Saved

Registers

Caller

Activation Record

**Calling Sequence (before)**

1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

**Prologue**

1. Save old fp, set new fp
2. Save callee-save registers
Typical Calling Sequence

Stack pointer →

<table>
<thead>
<tr>
<th>Variable-length Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporaries</td>
</tr>
<tr>
<td>Local Variables</td>
</tr>
<tr>
<td>Callee-Saved Registers</td>
</tr>
<tr>
<td>Saved fp</td>
</tr>
<tr>
<td>Return address</td>
</tr>
</tbody>
</table>

Frame pointer →

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caller-Saved Registers</td>
</tr>
<tr>
<td>Caller Activation Record</td>
</tr>
</tbody>
</table>

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
1. Save old fp, set new fp
2. Save callee-save registers
**Typical Calling Sequence**

<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>Frame pointer</th>
<th>Callee-Saved Registers</th>
<th>Saved fp</th>
<th>Return address</th>
<th>Arguments</th>
<th>Caller-Saved Registers</th>
<th>Caller Activation Record</th>
</tr>
</thead>
</table>

**Calling Sequence (before)**
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

**Prologue**
1. Save old fp, set new fp
2. Save callee-save registers

**Epilogue**
1. Restore callee-save registers
### Typical Calling Sequence

- **Stack pointer** → **Frame pointer** →

  - **Saved fp**
  - **Return address**
  - **Arguments**
  - **Caller-Saved Registers**
  - **Caller Activation Record**

#### Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

#### Prologue
1. Save old fp, set new fp
2. Save callee-save registers

#### Epilogue
1. Restore callee-save registers
2. Restore frame pointer
## Typical Calling Sequence

<table>
<thead>
<tr>
<th>Stack pointer</th>
<th>Return address</th>
<th>Arguments</th>
<th>Caller-Saved Registers</th>
<th>Caller Activation Record</th>
</tr>
</thead>
</table>

### Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

### Prologue
1. Save old fp, set new fp
2. Save callee-save registers

### Epilogue
1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address
Typical Calling Sequence

**Calling Sequence (before)**
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

**Prologue**
1. Save old fp, set new fp
2. Save callee-save registers

**Epilogue**
1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

**Calling Sequence (after)**
Typical Calling Sequence

**Calling Sequence (before)**
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

**Prologue**
1. Save old fp, set new fp
2. Save callee-save registers

**Epilogue**
1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

**Calling Sequence (after)**
1. Restore caller-save registers
Typical Calling Sequence

Calling Sequence (before)
1. Save caller-save registers
2. Push arguments on stack
3. Jump to subroutine, saving return address on stack

Prologue
1. Save old fp, set new fp
2. Save callee-save registers

Epilogue
1. Restore callee-save registers
2. Restore frame pointer
3. Jump to return address

Calling Sequence (after)
1. Restore caller-save registers
Are there advantages to having the caller or callee perform various tasks?

If possible, have the callee perform tasks: task code needs to occur only once, rather than at every call site.

However, some tasks (e.g. parameter passing) must be performed by the caller.
Saving Registers

One difficult question is whether the caller or callee should be in charge of saving registers.

*What would the caller have to do to ensure proper saving of registers?*

Save all registers currently being used by caller

*What would the callee have to do to ensure proper saving of registers?*

Save all registers that will be used by callee

*Which is better?*

Could be either one—no clear answer.

In practice, many processors (including MIPS and x86) compromise: half the registers are caller-save and half are callee-save.

*Register Windows* offer an alternative: each routine has access only to a small *window* of a large number of registers; when a subroutine is called, the window moves, overlapping a bit to allow parameter passing.
Optimizations

Leaf routines

A *leaf routine* is one which does not call any subroutines.

Leaf routines can avoid pushing the return address on the stack: it can just be left in a register.

If a leaf routine is sufficiently simple (no local variables), it may not even need a stack frame at all.

Inlining

Another optimization is to *inline* a function: inserting the code for the function at every call site.

*What are advantages and disadvantages of inlining?*

- **Advantages**: avoid overhead, enable more compiler optimizations
- **Disadvantages**: increases code size, can’t always do it (i.e. recursive procedures)
Parameter Passing

Definitions

• *Formal parameters* are the names that appear in the declaration of the subroutine.

• *Actual parameters* or *arguments* refer to the expressions passed to a subroutine at a particular call site.

    // formal parameters: a, b, c
    function f (int a, int b, int c)
        ...

    // arguments: i, 2/i, g(i,j)
    f(i, 2/i, g(i,j));
Parameter passing

Modes

What does a reference to a formal parameter in the execution of a subroutine mean in terms of the actual parameters?

It depends on the parameter *mode*.

- **by value**: formal is bound to value of actual
- **by reference**: formal is bound to location of actual
- **by copy-return**: formal is bound to value of actual; upon return from routine, actual gets copy of formal
- **by name**: formal is bound to expression for actual; expression evaluated whenever needed; writes to parameter are allowed (and can affect other parameters!)
- **by need**: formal is bound to expression for actual; expression evaluated the first time its value is needed; cannot write to parameters

*What are the advantages of passing by need?*
Parameter passing

Modes

What does a reference to a formal parameter in the execution of a subroutine mean in terms of the actual parameters?

It depends on the parameter *mode*.

- **by value**: formal is bound to value of actual
- **by reference**: formal is bound to location of actual
- **by copy-return**: formal is bound to value of actual; upon return from routine, actual gets copy of formal
- **by name**: formal is bound to expression for actual; expression evaluated whenever needed; writes to parameter are allowed (and can affect other parameters!)
- **by need**: formal is bound to expression for actual; expression evaluated the first time its value is needed; cannot write to parameters

*How does reference differ from copy-return?*
Consider the following PASCAL program:

```pascal
var
global: integer := 10;
another: integer := 2;

procedure confuse (var first, second: integer);
begin
  first := first + global;
  second := first * global;
end;

begin
  confuse(global, another);  /* first and global */
  /* are aliased */
end
```

- different results if by reference or by copy-return
- such programs are considered erroneous in ADA
- passing by value with copy-return is less error-prone
Parameter Passing in C

- C: parameter passing is always by value: assignment to formal is assignment to local copy
- passing by reference can be simulated by using pointers

```c
void incr (int *x) {
    (*x)++;
}
incr(&counter); /* pointer to counter */
```

- no need to distinguish between functions and procedures: `void` indicates side-effects only
Parameter Passing in C++

- default is by-value (same semantics as C)

- explicit reference parameters also allowed:

  ```cpp
  void incr (int& y) {
    y++;
  }

  incr(counter);
  ```

- semantic intent can be indicated by qualifier:

  ```cpp
  // passed by reference, but call cannot
  // modify it
  void f (const double& val);
  ```
Parameter Passing in JAVA

- semantics of assignment to parameter differs for primitive types and for classes:
  - primitive types have value semantics
  - objects have reference semantics
- consequence: methods can modify objects
- for formals of primitive types: assignment allowed, only affects local copy
- for objects: \texttt{final} means that formal is read-only
Parameter Passing in Ada

- goal: separate semantic intent from implementation

- parameter modes:
  - \texttt{in}: read-only in subprogram
  - \texttt{out}: write in subprogram
  - \texttt{in out}: read-write in subprogram

- independent of whether binding by value, by reference, or by copy-return
  - \texttt{in}: bind by value or reference
  - \texttt{out}: bind by reference or copy-return
  - \texttt{in out}: bind by reference or by value/copy-return

- functions can only have \texttt{in} parameters
C and C++ allow parameters which are pointers to subroutines:

```c
void (*pf)(double);
// pf is a pointer to a function that takes
// a double argument and returns void

typedef void (*PROC)(int);
// type abbreviation clarifies syntax

void do_it(double d) { ... }

void use_it(PROC);

PROC ptr = &do_it;

use_it(ptr);
use_it(&do_it);
```

Are there any implementation challenges for this kind of subroutine call?
Passing Subroutines as Parameters

Not really: can be implemented in the same way as a usual subroutine call: in particular the *referencing environment* can stay the same.

*What if a nested subroutine is passed as a parameter?*

A closure must be created and passed in place of the subroutine.

A *closure* is a reference to a subroutine together with its referencing environment.

When a subroutine is called through a closure, the referencing environment from when the closure was created is restored as part of the calling sequence.
Passing Subroutines as Parameters

Remember your homework?

# Compose several parsers in sequence

def make_seq(*ps):
    def parse(s, i):
        for p in ps:
            i = p(s, i)
        if -1 == i: return -1
    return i
return parse

The `make_seq` function takes zero or more parser functions as its only argument. It defines a nested `parse` function, which applies these functions on some string and at some starting index in order. It returns the closure of nested function and environment, because `parse` needs to access `ps` to work.

(Using parser functions to compose other parser functions is a well-known technique, called *parser combinators*.)
Syntactic sugar

- Default values for in-parameters (ADA)

```plaintext
function Incr (Base: Integer;
    Inc: Integer := 1) return Integer;
```

- `Incr(A(J))` equivalent to `Incr(A(J), 1)`

- also available in C++

```plaintext
int f (int first,
    int second = 0,
    char *handle = 0);
```

- named associations (Ada):

```plaintext
Incr(Inc => 17, Base => A(I));
```
Variable number of parameters

\[ \text{printf("this is %d a format %d string", x, y);} \]

- within body of \texttt{printf}, need to locate as many actuals as placeholders in the format string
- solution: place parameters on stack in \emph{reverse} order

<table>
<thead>
<tr>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual 1 (format string)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>actual n-1</td>
</tr>
<tr>
<td>actual n</td>
</tr>
</tbody>
</table>
First-class functions: implementation implications

Allowing functions as first-class values forces heap allocation of activation records.

- environment of function definition must be preserved until the point of call: activation record cannot be reclaimed if it creates functions
- functional languages require more complex run-time management
- higher-order functions: functions that take (other) functions as arguments and/or return functions
  - powerful
  - complex to implement efficiently
  - imperative languages restrict their use
  - (a function that takes/returns pointers to functions can be considered a higher-order function)
Recursion

*In order to understand recursion, you must first understand recursion.*
Recursion

In order to understand recursion, you must first understand recursion.

Recursion is when a subroutine is called from within itself.

Example

```c
int fact(int n)
{
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```
Recursion

What are some advantages and disadvantages of using recursion?

- **Advantages**: often conceptually easier, and easier to understand code
- **Disadvantages**: usually slower, can lead to stack overflow

There is one case when recursion can be implemented without using a stack frame for every call:

A *tail recursive* subroutine is one in which no additional computation ever follows a recursive call.

For tail recursive subroutines, the compiler can *reuse* the current activation record at the time of the recursive call, eliminating the need to allocate a new one.