Procedural Languages

CHAPTERS 3 AND 6 OF PROGRAMMING LANGUAGES PRAGMATICS

Procedural Languages

- In these languages, computation typically consists of an ordered series of changes to the values of variables
  - Indirectly accessed memory
- Heavy emphasis on control flow
  - Program is a list of instructions
  - In what order should these instructions be executed?

Key Features

- Sequencing
  - Statements should be executed in a specified order, usually specified by the order they appear in the program text
- Selection/Alternation
  - Depending on runtime conditions, a choice is made between two or more expressions or statements
- Iteration
  - A code fragment is executed repeatedly, either a specified number of times or a number of times based on runtime conditions
- Procedural Abstraction
  - A collection of control constructs treated as a single unit, usually subject to parameterization

Key Features

- Recursion
  - Expressions defined in terms of themselves
- Concurrency
  - Two or more program fragments executed “at the same time”
- Exception handling/Speculation
  - A program fragment is executed optimistically, on the assumption that some expected condition will be true
- Nondeterminancy
  - Ordering or choice is sometimes deliberately left unspecified, implying any alternative will lead to correct results

Key Concepts

- Values are read from memory, and used to compute new values that are then written back to memory
  - x = y+z+w*v
- Expressions are used to produce values
  - Constants, variables, operators, function calls, ect
  - Some expression may have side effects, they change the state of memory (some would argue this is a bad idea)
- Statements do not produce values, and are used only because of their side effects
  - Classic example, an assignment statement
  - Expressions are evaluated, statements are executed

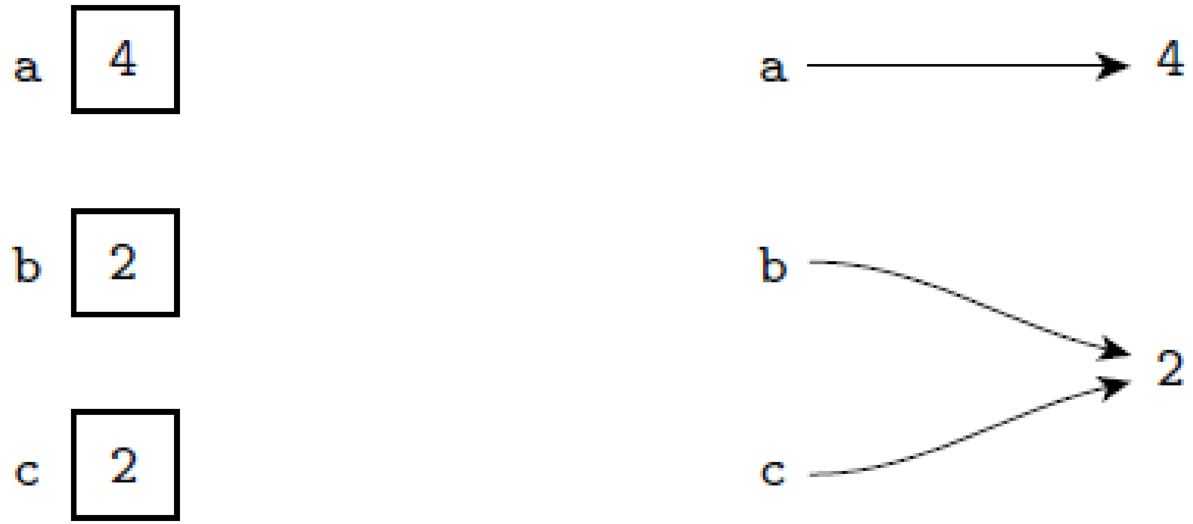
Outline

- Expressions
  - **l-values, r-values, pointers, references**
  - Side effects and order of evaluation
- Statements
  - Subroutines and scoping
  - Subroutine calls
  - Call stack/passing parameters
  - Lifetimes and memory management
  - Exceptions

Values of Expression

- Normally, an expression E designates a value
  - This value is referred to the r-value of E: if E appears on the right-hand side of an assignment statement, E stants for this value
- But sometimes E designates a location in memory or a reference value
  - This depends on what model of memory the language is using, a value model of variables or a reference model of variables
  - Used when E appears on the left-hand side of an assignment
  - The l-value of E is that location in memory or that reference
- d = a;
  a = b + c;
  - In C, if the variable"a" is of type int, the r-value is the int number stored in memory, and the l-value is the chunck of memory (typically 4 bytes) where the number is stored

Value model vs reference model of memory



Pointers in C/C++

- Most values are the usual things: numbers, characters, structures, arrays, ect
- Special kind of values: Pointer values
  - A pointer is a “handle” to a chunk of memory
  - C implements this as the address of the first byte in memory
- Two pointer operators
  - Address of operator &
    - &E gives the l-value of E (the location in memory)
  - Dereference operator *
    - *E uses the r-value of E to access the value stored in memory

Pointers in C/C++

```
int x=1, y=2, z[10];
int *ip;
ip = &x;
y = *ip;
*ip = 0;
ip = &z[0];
*ip = *ip + 10;
y = *ip + 1;
*ip += 1;
```

References in Java

```
class Rectangle { public double height, width; }
main(…) {
   Rectangle x , y;
   x = new Rectangle(); // 1) Create a Rectangle object in memory
                        // 2) Produce a reference value which is
                        //     a handle to this object
                        // 3) Assign this reference value to x
   y = x;               // Copy the r-value of x
   y.width = 3.14;      // 1) Use the r-value of y to get to the object
}                       // 2) Assign based on the l-value of field width
```

Expressions

- Elements: variables, contants, function calls, ect
- Operators vs functions
  - Convention is that operators are built-in functions that use special, simple syntax
- Operators an operands
  - Arity: unary, binary, ternary
    - Unary: “++x” or “x++”
    - Binary: “x + y”
    - Ternary: “x ? y : z
  - Prefix, infix, postfix notation:
    - “+ x y” vs “x + y” vs “x y +”

Operator precedence and associativity

- When using prefix or postfix notation, not a problem
- Using infix notation, consider this Fortran expression (** for exponentiation): a+b*c**d**e/f
  - How should this be evaluated?
    - ((((a+b)*c)**d)**e)/f ?
    - a+(((b*c)**d)**(e/f)) ?
    - a+((b*(c**(d**e)))/f) ?

Operator precedence and associativity

- What about this code from Pascal, how should the condition be evaluated?
  if A < B and C < D then ....
  - (A < B) and (C < D) ?
  - A < (B and C) < D ?
- What about all the other operators languages introduce?
  - C has 45 if I counted correctly
  - Whenever you aren’t sure, use parantheses

Operator precedence and associativity

- Another fun example:
- In C, how is a-b-c-d evaluated?
  - (((a-b)-c)-d) or (a-(b-(c-d))) ?
- In C, how is a=b=c=d evaluated?
  - (((a=b)=c)=d) or (a=(b=(c=d))) ?

Operator precedence and associativity

- Another fun example:
- In C, how is a-b-c-d evaluated?
  - **(((a-b)-c)-d)** or (a-(b-(c-d))) ?
- In C, how is a=b=c=d evaluated?
  - (((a=b)=c)=d) or **(a=(b=(c=d)))** ?

Functions

- Built-in or programmer-defined
  - Example: Math library in C provides **double log(double x)**
  - Typically use prefix notation, function identifier first and then function arguments follow, contained in parenthese
    - **pow(e1, e2)** where **pow** if the function name, **e1** and **e2** are arguments
  - Typically, functions should not have side effects

Outline

- Expressions
  - l-values, r-values, pointers, references
  - **Side effects and order of evaluation**
- Statements
  - Subroutines and scoping
  - Subroutine calls
  - Call stack/passing parameters
  - Lifetimes and memory management
  - Exceptions

Side effects of expression evaluation

- Referential transparency
  - Can we replace an expression with the r-value of this expression?
    - This is a desirable property
  - Consider:
    `x = 5; y = 1 + x++; if (y == x) printf("OK");`
    vs
    `x = 5; y = 1 + 5; if (y == x) printf("OK");`
    If we replace the expression x++ with the value 5, the behavior changes
  - Referential transparency is not possible when we have side effects
- Expression in C
  - Operators **= ++ -- +=** and others all have side effects

Side effects and order of evaluation

- The presence of side effects must be considered when defining the order of evaluation, defining precedence and associativity is not enough
- In this expression
  a – f(b) – c * d
  - Will **f(b)** be evaluated before or after **a**?
  - Will **a – f(b)** be evaluated before or after **c * d**?
  - What if **f(b)** has side effects that could change **a**, **c**, or **d**?

Side effects and order of evaluation

- We need a defined order for operands and function arguments to be evaluates
  - To clarify the behavior in the presence of side effects
  - To enable compiler optimizations:
    When evaluating the expression **a – f(b) – c * d,** computing **c*d** before **f(b)** requires a register to remember the value during the call to **f(b)**, this may be bad for performance
- C does not specify an order for operands/arguments (aim: performance)
- Java does specify this order (aim: correctness)

Outline

- Expressions
  - l-values, r-values, pointers, references
  - Side effects and order of evaluation
- Statements
  - **Subroutines and scoping**
  - Subroutine calls
  - Call stack/passing parameters
  - Lifetimes and memory management
  - Exceptions

Statements

- Assignment statements
- Control flow
  - Selection statements: if-then-else, switch, ect
  - Iteration statements: while, do-while, for, ect
  - Jump statements: goto, return, break, ect
- Unstructured control flow: goto allows arbitrarily complex behavior, but leads to “bad” code
- Structured control flow: Use standard “clean” abstractions such as if-then-else, while, ect

Subroutines

- Subroutines, procedures, functions, methods, ...
- For our purposes:
  - Subroutine is the general term
  - Procedure: a subroutine that does not returns a value
  - Function: a subroutine that returns a value
  - Method: a subroutine included in a class definition
- Some people use “procedure” as the general term
  - Hence the Procedural Languages: Imperative languages in which procedures are a major abstraction (C, Fortran, ect)
  - Procedural abstraction: A collection of statements is abstracted by
    - Name
    - List of formal parameters
    - A return value (optional)

Basic mechanism of subroutines

- A caller makes a call
  - The caller provides arguments (actual parameters), generally expressions that are evaluated immediately before the call
- Parameters are passed
  - The actual parameters are “mapped” to the formal parameters
  - Several ways of doing this
- Memory is allocated for the formal parameters and the local variables of the called subroutine (the callee)
- The flow of control enters the called subroutine
- When the callee is done, control returns to the caller
  - Or not, if there is an error/excepetion

Scopes in procedural languages

- Which entities (variables, subroutines, ...) are accessible in which parts of a program? What is their lifetime?
- Identifiers are bound to entities
- The textual region of the source code in which a binding is active is the scope of the binding

Scopes in procedural languages

- Basic – Only global scope, limited number of variables
- Fortran – Global and local scope
  - A Fortran program is a set of procedures
    - Procedure names are visible everywhere
    - Local variables are visible only in the declaring procedure
    - Global variables are visible everywhere

Main procedure	**Procedure S_1**	**…**	**Procedure S_n**

- In the C family of languages, scopes are delimited with **{ … }**

Referencing environment

- The referencing environment for a given point program execution is the set of active bindings
- Creating the referencing environment: Static vs Dynamic Scoping
- **Static Scoping** (C, C++, Java, ...): Bindings between names and entities are determined at compile time, by examining the text of the program
  - No consideration of the runtime flow of control
  - The current binding for a given name is found in the matching declaration most closely surrounding a given point in the program
- **Dynamic Scoping** (Lisp, Perl, Bash, ...): Bindings between names and entities are determined at runtime
  - Depend on the flow of control
  - Bindings are maintained on a stack, the current binding is found by searching from the top of the stack

Referencing environment

```
int n

procedure first()
        n = 1

procedure second()
        int n
        first()

procedure main()
        n=2
        if read() > 0
                second()
        else
                first()
        write(n)
```

- Static Scoping
  - This program always writes 1
- Dynamic Scoping
  - This program writes 2 if read() > 0, 1 otherwise
  - Why?

Static scope rules

- Algol, Pascal,C, C++, Java, C#, ...
- Entities accessible in a scope:
  - Those declared in that scope
  - Those declared in the surrounding scope, minus those with name conflicts
  - Those declared in scopes surroundoing that scope, minus those with name conflicts
  - ...
- A name declared in an inner scope “hides” a name declared in a surrounding scope

C++ example

```
class Point {
  public: Point(double x, double y);
  virtual void print(); virtual void add(Point* q);
  private: double x,y;
}

Point::Point(double x, double y) { this->x = x; this->y = y; }

void Point::print() { cout<<x<<","<<y<<endl; }

void Point::add(Point* q) {
  q->print();
  {
    Point *q = new Point(100.0,100.0);
    this->x += q->x; this->y += q->y;
  }
  this->x += q->x; this->y += q->y;
}

int main(void) {
  Point* p1 = new Point(1.0,1.0); p1->print();
  Point* p2 = new Point(2.0,2.0); p1->add(p2); p1->print();
  return 0;
}
```

C++ example

```
class Point {
  public: Point(double x, double y);
  virtual void print(); virtual void add(Point* q);
  private: double x,y;
}

Point::Point(double x, double y) { this->x = x; this->y = y; }

void Point::print() { cout<<x<<","<<y<<endl; }

void Point::add(Point* q) {
  q->print();
  {
    Point *q = new Point(100.0,100.0);
    this->x += q->x; this->y += q->y;
  }
  this->x += q->x; this->y += q->y;
}

int main(void) {
  Point* p1 = new Point(1.0,1.0); p1->print();
  Point* p2 = new Point(2.0,2.0); p1->add(p2); p1->print();
  return 0;
}
```

- At compile time, we consider the scopes and their nesting
  - Determine which entities (variables, ect) are accessible in which parts of the code
  - Additional restrictions on accessibility may be imposed with “access modifiers”
    - For example private, protected, ect
- At run time, each scope has a lifetime
  - Anything declared in a particular scope becomes alive at the stat of the scope, and dies at the end of the scope

```
class Point {
  public: Point(double x, double y);
        virtual void print(); virtual void add(Point* q);
  private: double x,y;
};
Point::Point(double x, double y) { this->x = x; this->y = y; }
void Point::print() { cout<<x<<","<<y<<endl; }
void Point::add(Point* q) {
  q->print();
  {
    Point *q = new Point(100.0,100.0);
    this->x += q->x; this->y += q->y;
  }
  this->x += q->x; this->y += q->y;
}
int main(void) {
  Point* p1 = new Point(1.0,1.0); p1->print();
  Point* p2 = new Point(2.0,2.0); p1->add(p2); p1->print();
  return 0; }
```

Start of program

Global scope: **main**; class **Point** and all its methods (**Point**, **print**, **add**) and fields (**x**, **y**)

Local scope for main: **p1**, **p2** (locals)

Local scope for Point constructor: **this** (formal); **x**, **y** (formals)

Local scope for print: **this** (formal)

…

Local scope for add: **this** (formal); **q** (formal)

Local scope for block: q

…

End of program

Static Scoping: Declaration Order

- Can an expression E refer to any name declared in the current scope, or only names that are declared before E in that scope?
  - Is a forward reference allowed?
- Early Pascal rules:
  - Names must be declared before they are used
  - The scope of a declaration is the surrounding block
- Early rules imply a semantic error here
  - Later versions of Pascal allow this, the scope goes from the declaration to the end of the block

```
program scoping;
  const N = 10;
  procedure foo;
    const
      M = N;
      N = 20;
  begin
    writeln('Value of M:', M);
  end;
begin
  foo();
end.
```

Try these C# and Java examples

C#

```
using System;

class A {
  int N = 10;
  void foo() {
    int M = N;
    int N = 20;
  }
}

class MainClass {
  public static void Main (string[] args) {
    Console.WriteLine ("Hello World");
  }
}
```

JAVA

```
class A {
  int N = 10;
  void foo() {
    int M = N;
    int N = 20;
  }
}

class Main {
  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

Outline

- Expressions
  - l-values, r-values, pointers, references
  - Side effects and order of evaluation
- Statements
  - Subroutines and scoping
  - **Subroutine calls/Call Stack**
  - Passing parameters
  - Lifetimes and memory management
  - Exceptions

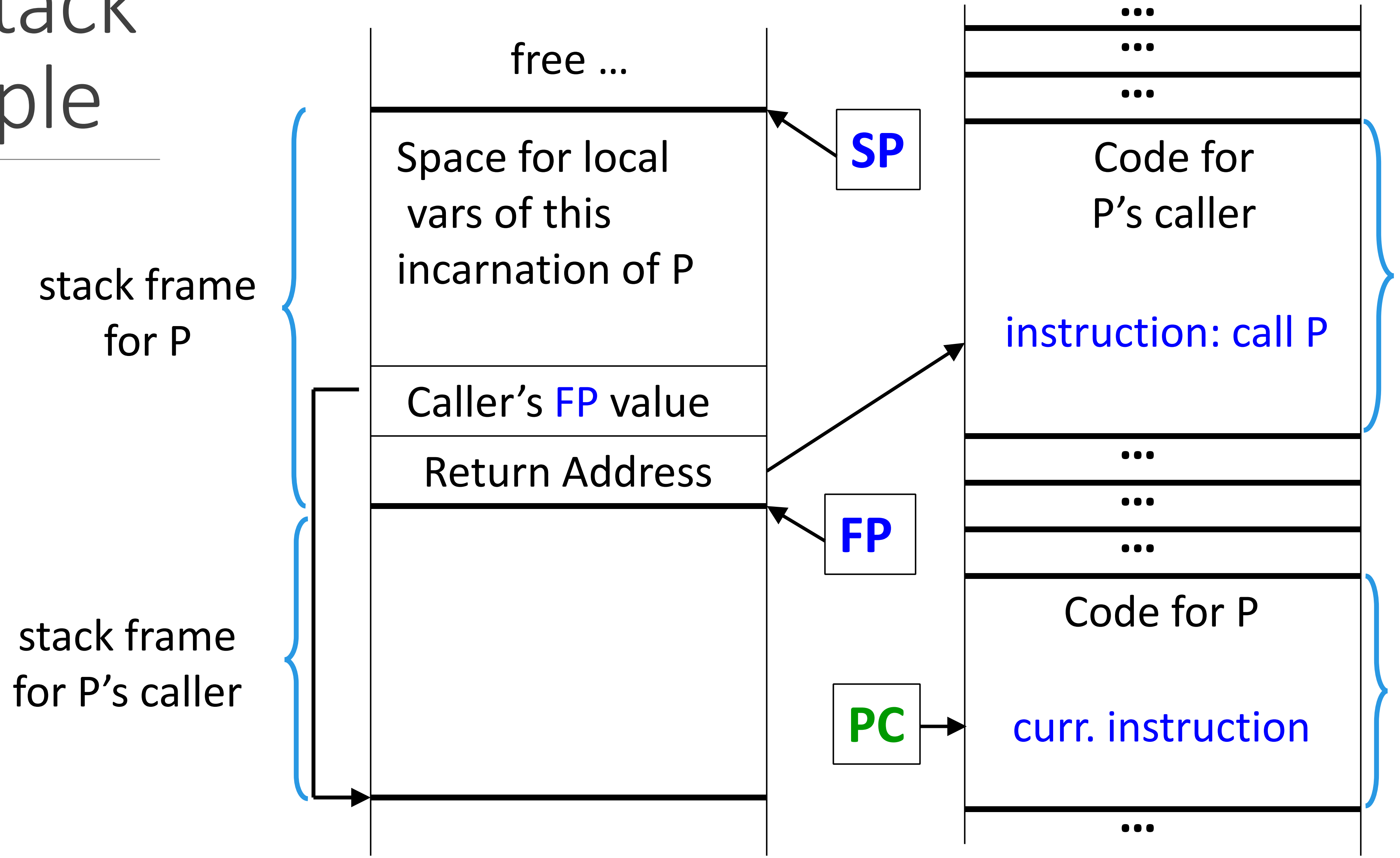
Storage Allocation Mechanisms

- Static/Global
  - Entities are given an absolute address that is retained throughout program execution
- Stack
  - Entities are allocated and decallocated in last-in, first-out order
  - Allocation typically happens at well defined points in execution (subroutine calls and returns)
- Heap
  - Entities may be allocated and deallocated at arbitrary times
  - This requires an expensive space management algorithm

Run-time Call Stack

- When a procedure P begins execution:
  - A stack frame/activation record for that instance of P is creates on the stack
    - The intent is to create space for local variables, but many additional values will go in the frame
  - During the execution of this instance of P, the frame pointer (FP) will contain the starting address of the frame for this instance of P
  - The stack pointer (SP) will contain the address of the location immediately beyond this frame, the start of “free space” on the stack
  - In many modern systems, FP and SP are reserved registers
- When this instance of P finishes execution:
  - Control returns to the caller
  - The record is “popped” off the stack

Call Stack Example



Compile-time Code Generation

- What code does the compiler need to produce to make this work?
  - Need code to allocate a frame
  - Need code to change FP, SP, PC (need to save these values?)
  - Need code to save values (i.e. registers) that are in danger of being overwritten and will be needed later
  - Need code to pass parameters
  - Need code to handle the return value
  - Need code to pop the frame and change FP, SP, PC
  - Need code to restore the saved registers
- Can the caller do all of this, or should these tasks be divided up?

Compile-time Code Generation

- Modern languages are designed to support separate compilation
  - Code divided into modules
  - Modules can be compiled at different times
  - A change in one module should not require recompiling another module
- This makes it necessary to divide the tasks between caller and callee
  - If caller was responsible for allocated the frame then the caller would need to be recompiled anytime the local variables in the callee are changed
  - If the callee is responsible for putting the return value then the callee would need to be recompled anytime the variable receiving the return value in the caller is changed

Calling Sequences

- We call the code responsible for maintaining the call stack the **calling sequence**
- The calling sequence includes:
  - Code executed by the caller immediately before the subroutine call
  - The prologue of the callee – code at the beginning of the subroutine
  - The epilogue of the callee – code at the end of the subroutine
  - Code executed by the caller immediately after the subroutine call
- Over the next few slides I will go over one **possible** way of structuring the calling sequence and dividing the tasks between caller and callee – but this is not the only way of doing this

(Possible) Calling Sequence

- Caller: Immediately before the subroutine call
  - Save any registers the caller will need after the subroutine call returns
  - Compute the values of arguments and move them into the stack or reserved registers
    - Include as an extra, hidden parameter the callers FP value
  - Use a special instruction to jump the PC to the prologue of the callee
    - Somehow pass along the return address (for PC, not for return value)
- Callee: Prologue
  - Allocate a frame by modifying the value of SP
  - Save old FP value to the stack, update FP with new value
  - (Optional) save any registers that might be overwritten by the callee

(Possible) Calling Sequence

- Callee: Execute subroutine body
- Callee: Epilogue
  - Moves the return value into some reserved location
  - Restores registers if needed
  - Restores FP and SP
    - This deallocates/pops the record off the stack
  - Jump the PC back to the return address
- Caller: Immediately after the subroutine call
  - Moves the return value from reserved location to where it is needed
  - Restores registers if needed

Maintaing the Call Stack, Passing Control

- MEM is the memory – think of it as an array
- Caller
  - Save the return address: MEM[SP]=PC+4 (assuming 4 byte instructions)
  - Save the start location of callers FP: MEM[SP+4]=FP
  - Pass control to callee: PC= addrerss of first instruction of callee
- Callee
  - Prologue: Allocate the frame: FP=SP and SP=SP+**n**, where **n** is the size needed for the frame
  - Epilogue: Deallocate the frame: SP=FP, FP=MEM[FP+4], PC=MEM[SP]

Calling Sequence Example

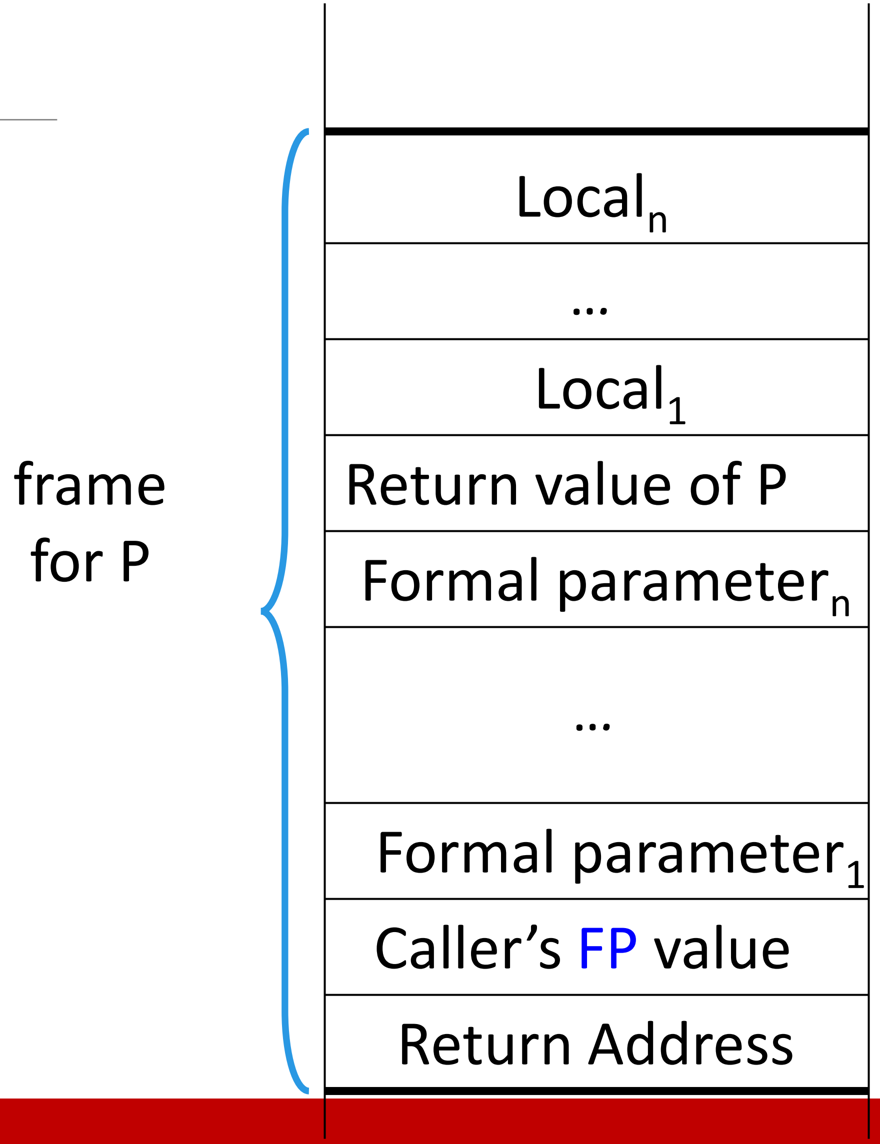
```
void fib(int n) {
	int result;
	if (n < 2) {
		result = n;
	} else {
		result = fib(n-1) + fib(n-2);
	}
}

void main() {
	int n=3;
	fib(n);
}
```

Outline

- Expressions
  - l-values, r-values, pointers, references
  - Side effects and order of evaluation
- Statements
  - Subroutines and scoping
  - Subroutine calls/Call Stack
  - **Passing parameters**
  - Lifetimes and memory management
  - Exceptions

Call Stack: Parameters and Returns

- The callee stores formal parameters and return values are at offsets of the callees FP value that are known at compile time
- The caller can access them using the callers SP value before and after the call (even though technically that space is "deallocated")



Parameter Passing Modes

- Formal parameters are the **variables** that appear in the subroutine declaration to refer to the input of the subroutine
- The **variables and expressions** used in a call to the subroutine are the actual parameters
- How to map actual parameters to formal parameters depends on our model of variables
  - Value model: Variables are containers that hold values
  - Reference model: Variables are references (a link) to values

Parameter Passing Modes

- Call-by-value: C, Pascal, C++, Java, C#, ...
  - The formal parameter is essentially a local variable initialized with the actual parameter
  - The actual parameter may or may not have an l-value
    - int n=3; fib(n); fib(3);
- Call-by-sharing: Java, Lisp, Ruby, C#, ...
  - The parameter is a new reference to the value of the actual parameter; the value is shared
  - The actual parameter may or may not have an l-value
- Call by reference: C++, Pascal, Fortran, C#, ...
  - The formal parameter introduces a new name for the actual parameter
  - The actual parameter must have an l-value

Outline

- Expressions
  - l-values, r-values, pointers, references
  - Side effects and order of evaluation
- Statements
  - Subroutines and scoping
  - Subroutine calls/Call Stack
  - Passing parameters
  - **Lifetimes and memory management**
  - Exceptions

Lifetimes and Memory Management

- Static Allocation: Addresses determined once and retained throughout the execution of the program
  - Global variables in C, Pascal, ect
  - static fields in C++, Java, ect
  - Local variables in languages without recursion
    - For example early versions of Fortran
  - static local variables in C
  - Large constants
    - Not small constants? What is the cutoff?

Lifetimes and Memory Management

- Stack based allocation: Address determined when the call happens, lifetime ends when the call ends
  - Necessary for local variables in languages with recursion
  - Relative address within the stack frame is determined at compile time
- Heap based allocation: Space allocated and deallocated at arbitrary times
  - Deallocation can be manually done by the programmer
  - Garbage collections algorithms can automate the deallocation
  - C: A* a = (A*)malloc(sizeof(A)); .... free(a);
  - C++: A* a = new A(); .... delete a;
  - Java: A a = new A(); .... garbage collector decides when to deallocate

Outline

- Expressions
  - l-values, r-values, pointers, references
  - Side effects and order of evaluation
- Statements
  - Subroutines and scoping
  - Subroutine calls/Call Stack
  - Passing parameters
  - Lifetimes and memory management
  - **Exceptions**

Exceptions

- What do we do with “exceptional situations”?
  - Try to open a file but the files does not exist
  - Try to send a byte over a network socket but the connection was dropped
  - Try to allocate new memory (through malloc in C or new in C++/Java/C#) but we have run out of memory
  - Division by zero, use of null pointer/reference, ect
- Ad hoc solutions
  - Use a special return value to signify failure
    - For many C functions a return of 0 or -1 signifies an error
  - Set some global error flag
    - In C have the errno flag, which is just a global int
    - A call sqrt(-1) will return NaN (special float value for “not a number”) and will set errno to EDOM (an integer error code for “argument not in the domain of the function”)
  - Pass a closure (self contained block of instructions) to handle an error (not all language support this)

C Example

```
#include <stdio.h>
int main ()
{
  FILE* pFile;
  pFile=fopen("myfile.txt","r"); /* possible problem */
  if (pFile==NULL)
    perror ("Error opening"); /* perror prints a message based on  errno */
  else {
    fputc ('x',pFile);
    if (ferror (pFile))
      printf ("Error writing to myfile.txt\n");
    fclose (pFile);
  }
  return 0;
}
```

Java Example

```
import java.io.*;
class Main {
   public static void main(String[] args) {
            FileReader file = null;
            char c;
            try {
               file = new FileReader("myfile.txt"); // may throw FileNotFoundException
               c = (char) file.read(); // may throw IOException
               System.out.println("char: " + c);
            } catch (FileNotFoundException e) {
               System.err.println("Error opening");
            } catch (IOException e) {
               System.err.println("Error reading from myfile.txt");
            } finally {
               if (file != null) try { file.close(); } catch (IOException e) { }
            }
   }
}
```