

# CS406: Compilers

Spring 2022

## Week 9: IR Code for Functions, Local Optimizations

# Functions Typical Syntax and Usage

```
FUNCTION VOID bar(INT x, FLOAT y) BEGIN
```

```
END
```

Keywords

Return type

comma separated parameter declarations.

Declarations (string or variable decl) followed by statement declarations

```
FUNCTION void foo() BEGIN
```

```
INT a;
```

```
FLOAT b;
```

```
...
```

```
bar(a, b);
```

← Calls function bar

```
END
```

# Terms

```
void foo() {  
    int a, b;  
    ...  
    bar(a, b);  
}
```

```
void bar(int x, int y) {  
    ...  
}
```

- foo is the *caller*
- bar is the *callee*
- a, b are the *actual parameters* to bar
- x, y are the *formal parameters* of bar
- Shorthand:
  - *argument* = actual parameter
  - *parameter* = formal parameter

# Different Kinds of Parameters

- Value
- Reference
- Result
- Value-Reference
- Read-only
- Call-by-Name

# Value parameters

- “Call-by-value”
- Used in C, Java, default in C++
- Passes the value of an argument to the function
- Makes a copy of argument when function is called
- Advantages? Disadvantages?

Advantage: ‘side-effect’ free – caller can be sure that the argument is not modified by the callee

Disadvantage: Not efficient for larger sized arguments.

# Value parameters

```
int x = 1;  
void main () {  
    foo(x, x);  
    print(x);  
}
```

```
void foo(int y, int z) {  
    y = 2;  
    z = 3;  
    print(x);  
}
```

- What do the print statements print?

# Reference parameters

- “Call-by-reference”
- Optional in Pascal (use “var” keyword) and C++ (use “&”)
- Pass the *address* of the argument to the function
- If an argument is an expression, evaluate it, place it in memory and then pass the address of the memory location
- Advantages? Disadvantages?

Advantage: Efficiency – for larger sized arguments

Disadvantage: results in clumsy code at times (e.g. check for null pointers)

# Reference parameters

```
int x = 1;  
void main () {  
    foo(x, x);  
    print(x);  
}
```

- What do the print statements print?

```
void foo(int &y, int &z) {  
    y = 2;  
    z = 3;  
    print(x);  
    print(y);  
}
```



# Result Parameters

- To capture the return value of a function
- Copied at the end of function into arguments of the caller
- E.g. output ports in Verilog module definitions

# Result Parameters

```
int x = 1
void main () {
    foo(x, x);
    print(x);
}
```

- What do the print statements print?

```
void foo(int y, result int z) {
    y = 2;
    z = 3;
    print(x);
}
```

# Value-Result Parameters

- “Copy-in copy-out”
- Evaluate argument expression, copy to parameters
- After subroutine is done, copy values of parameters back into arguments
- Results are often similar to pass-by-reference, but there are some subtle situations where they are different

# Value-Result Parameters

```
int x = 1
void main () {
    foo(x, x);
    print(x);
}
```

•What do the print statements print?

```
void foo(int y, value result int z)
{
    y = 2;
    z = 3;
    print(x);
}
```

# Read-only Parameters

- Used when callee will not change value of parameters
- Read-only restriction must be enforced by compiler
- E.g. `const` parameter in C/C++
- Enforcing becomes tricky when in the presence of aliasing and control flow. E.g.

```
void foo(readonly int x, int y) {  
    int * p;  
    if (...) p = &x else p = &y  
    *p = 4  
}
```

# Call-by-name Parameters

- The arguments are passed to the function before evaluation
  - Usually, we evaluate the arguments before passing them
- Not used in many languages, but Haskell uses a variant

```
int x = 1
void main () {
    foo(x+2);
    print(x);
}
```

```
void foo(int y) {
    z = y + 3; //expands to z = x + 2 + 3
    print(z);
}
```

# Call-by-name Parameters

- Why is this useful?
  - E.g. to analyze certain properties of a program/function – termination

```
void main () {  
    foo(bar());  
}
```

```
void foo(int y) {  
    z = 3;  
    if(z > 3)  
        z = y + z;  
}
```

- Even if bar has an infinite loop, the program terminates.

# Program Layout in Memory

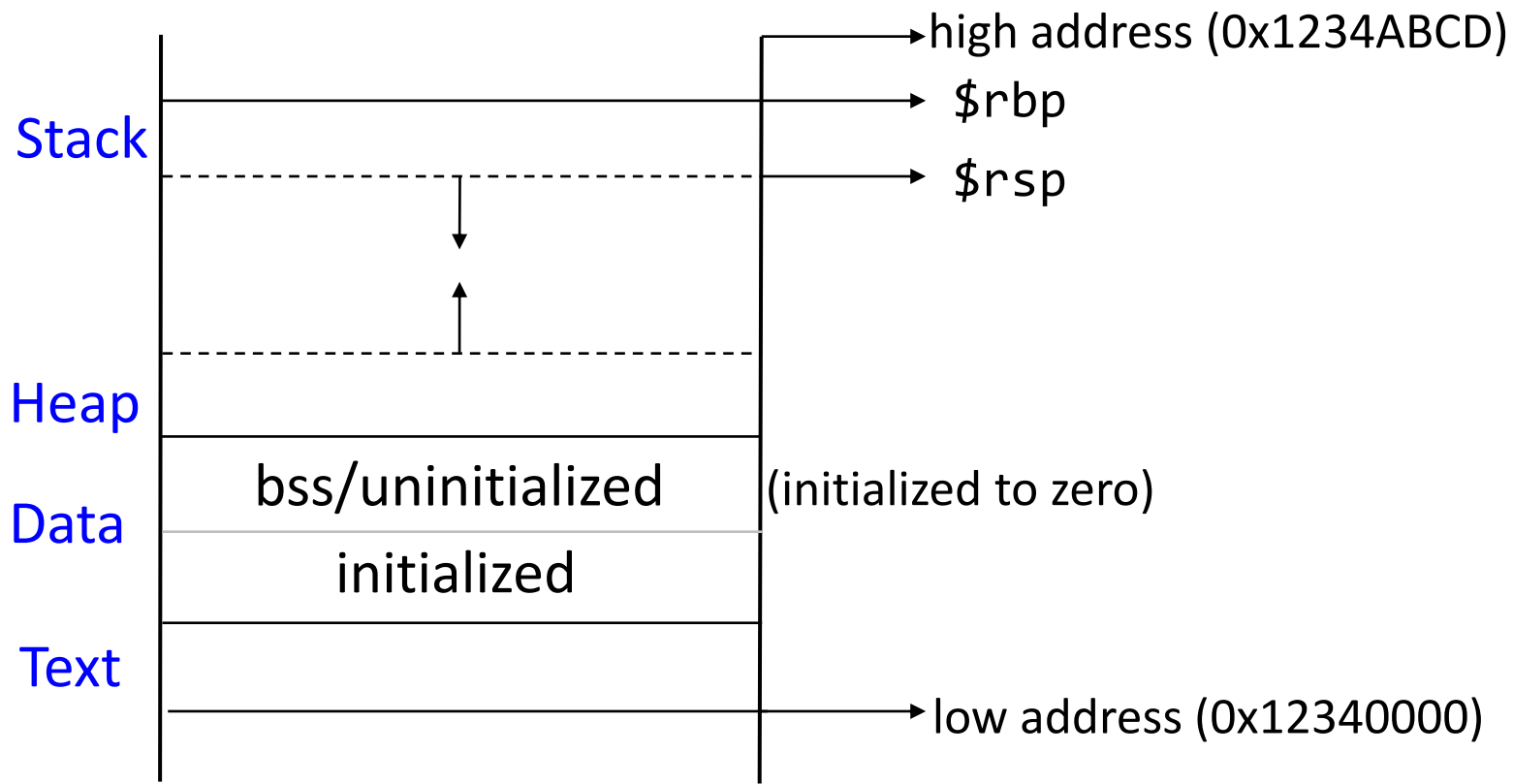
- Compiler assumes a *runtime environment* for execution of the program.
- A C/C++ program in Linux OS has 4 segments of memory
  - Every memory location is a *box* holding *data/instruction*



# Program Layout in Memory

- A program's memory space is divided into four segments:
  1. Text
    - source code of the program
  2. Data
    - Broken into *uninitialized* and *initialized* segments; contains space for global and static variables. E.g. `int x = 7; int y;`
  3. Heap
    - Memory allocated using `malloc/calloc/realloc/new`
  4. Stack
    - Function arguments, return values, local variables, [special registers](#).

# Program Layout in Memory



# Activation

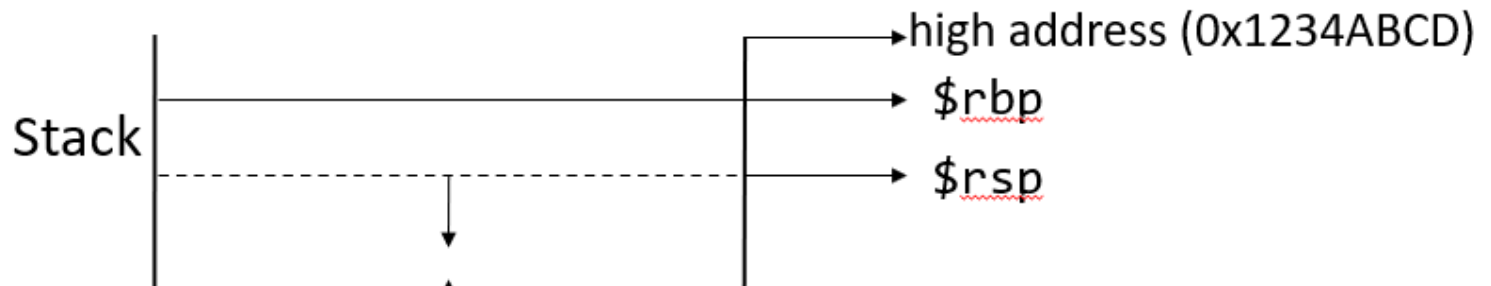
- A function call or invocation is termed an *activation*
- Calls to functions in a program form *activation tree*
  - Postorder traversal of the tree shows return sequence i.e. the order in which control returns from functions
  - Preorder traversal of the tree shows calling sequence
- In a sequential program, at any point in time, *control of execution is in any one activation*
  - All the ancestors of that activation are active i.e. have not returned

# Activation

- Activations are managed through the help of *control stack*
- A function call (activation) results in allocating a chunk of memory called *activation record* or *frame* on the stack (also called *stack frame*)

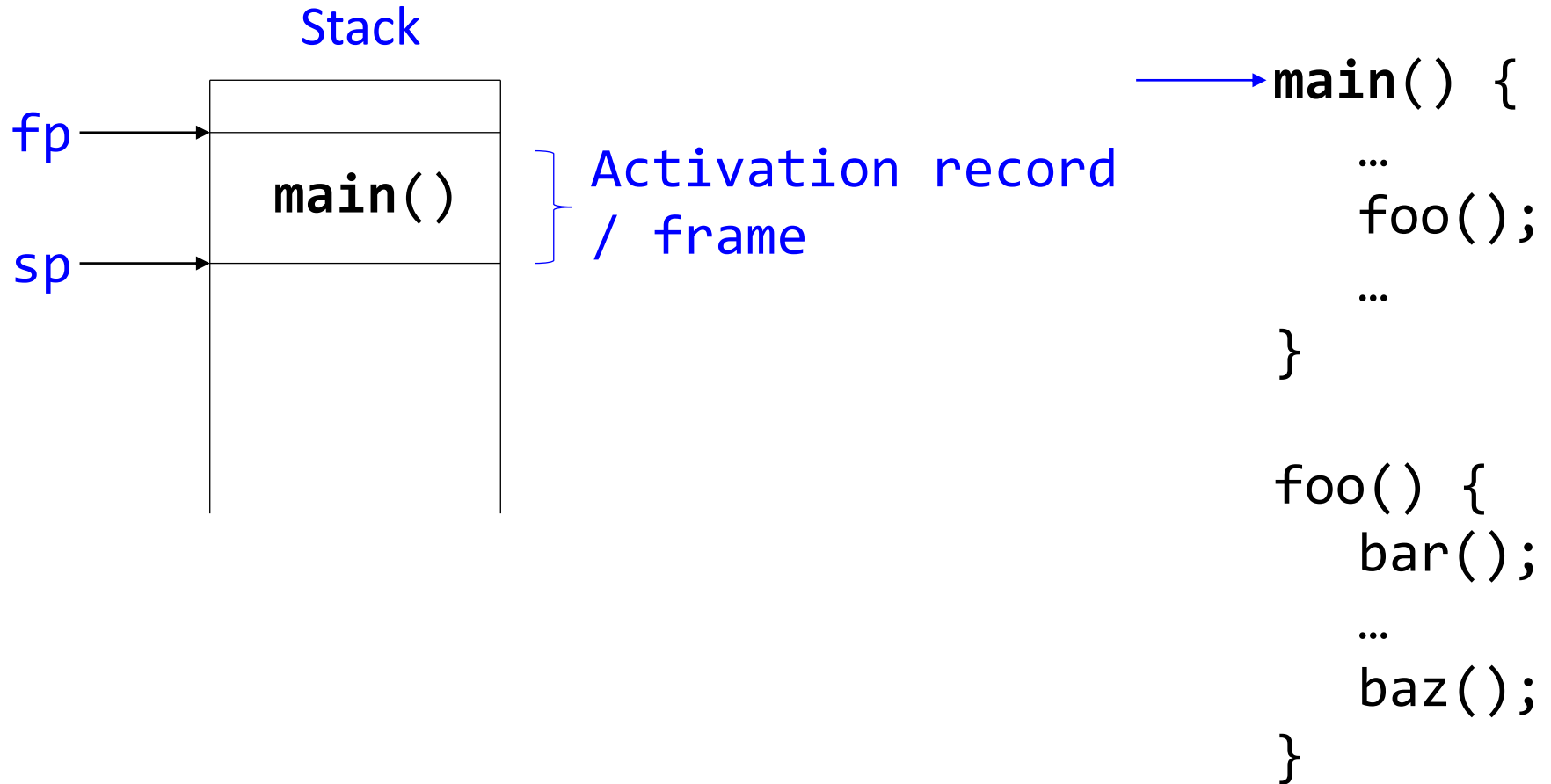
# Activation Record

- A *sub-segment* of memory on the stack
  - **Special registers** \$rbp and \$rsp track the bottom and top of the stack frame. These are the names in x86 architecture.

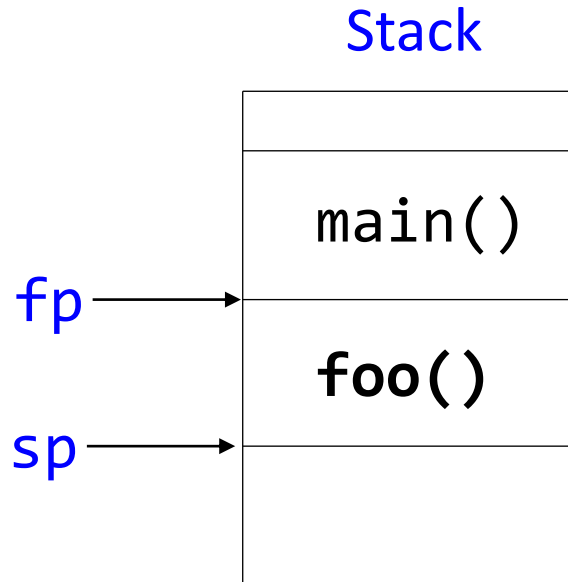


- \$rbp – base pointer or frame pointer (**fp**)
- \$rsp – stack pointer (**sp**)

# Activation Record - Example



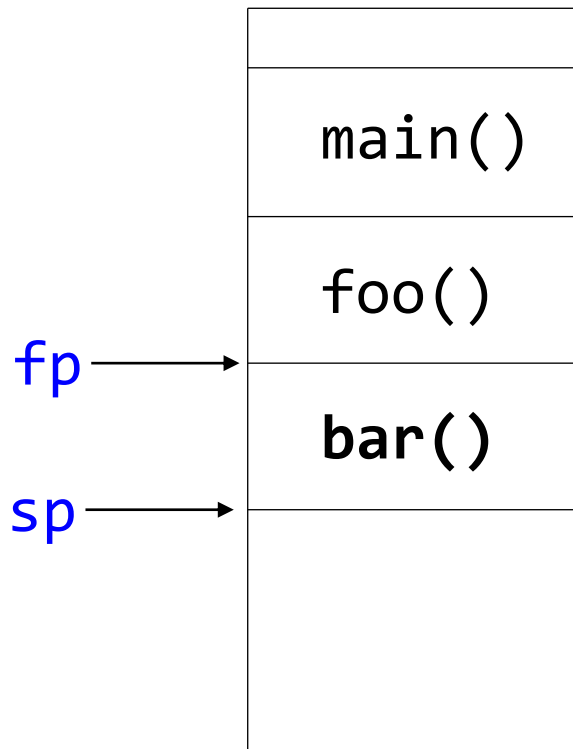
# Activation Record - Example



```
main() {  
    ...  
    → foo();  
    ...  
}  
  
foo() {  
    bar();  
    ...  
    baz();  
}
```

# Activation Record - Example

Stack

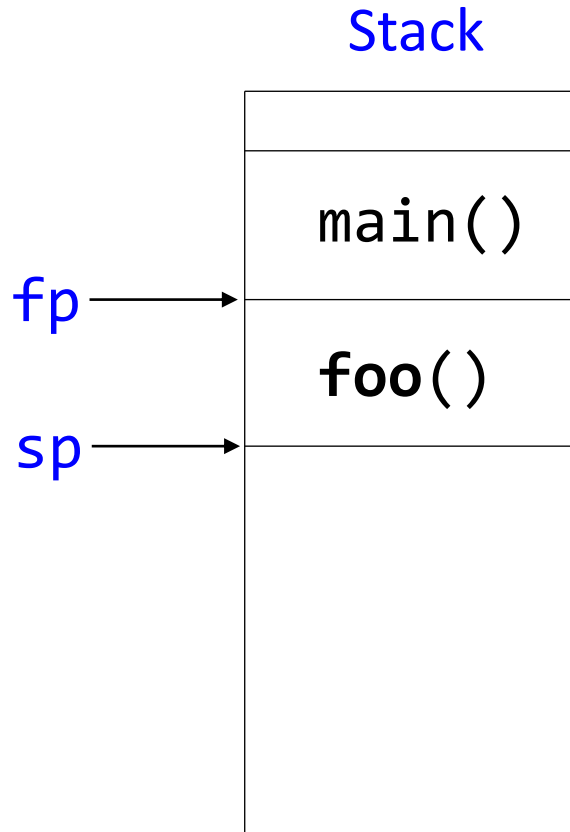


```
main() {  
    ...  
    foo();  
    ...  
}
```

```
foo() {  
    → bar();  
    ...  
    baz();  
}
```



# Activation Record - Example



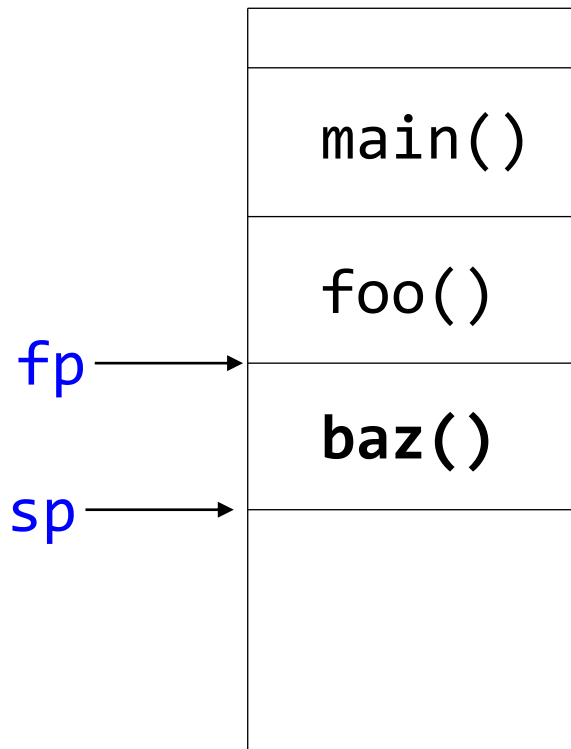
```
main() {  
    ...  
    foo();  
    ...  
}
```

```
foo() {  
    bar();  
    ...  
    baz();  
}
```

A blue arrow points to the `...` line within the `foo()` function body.

# Activation Record - Example

Stack



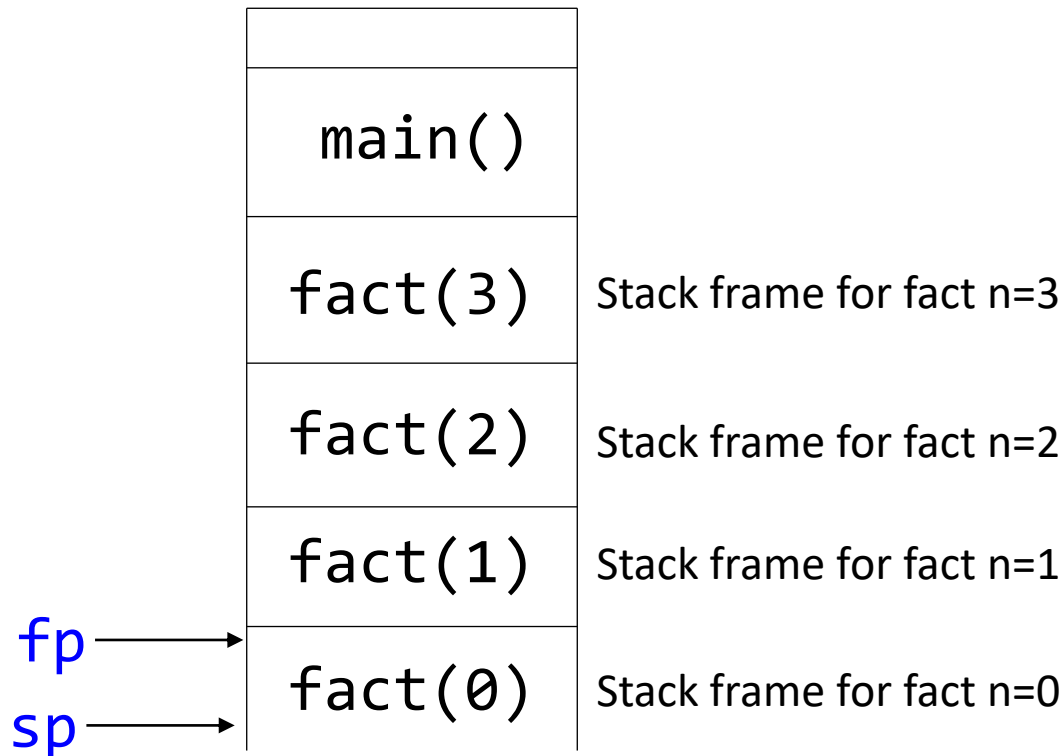
```
main() {  
    ...  
    foo();  
    ...  
}
```

```
foo() {  
    bar();  
    ...  
    baz();  
}
```

→

# Activation Record – Example (Recursive Functions)

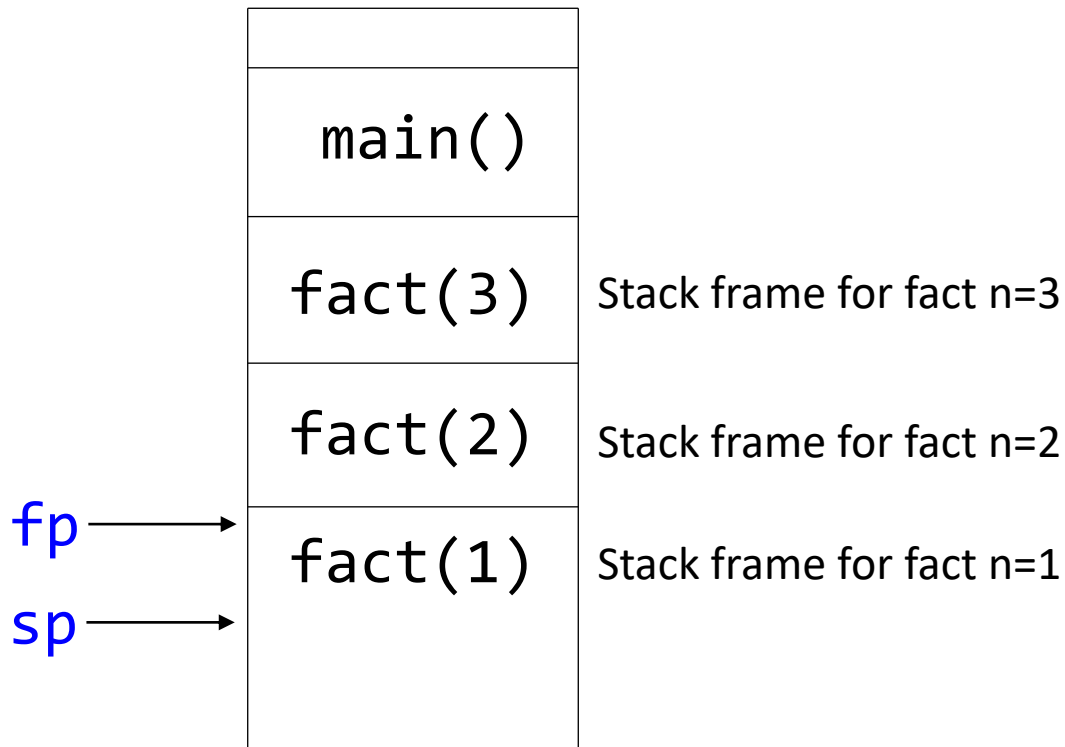
Stack



```
main() {  
    ...  
    fact(3);  
    ...  
}  
  
fact(int n) {  
    if (n=0) return 1  
    return n*fact(n-1)  
}
```

# Activation Record – Example (Recursive Functions)

Stack

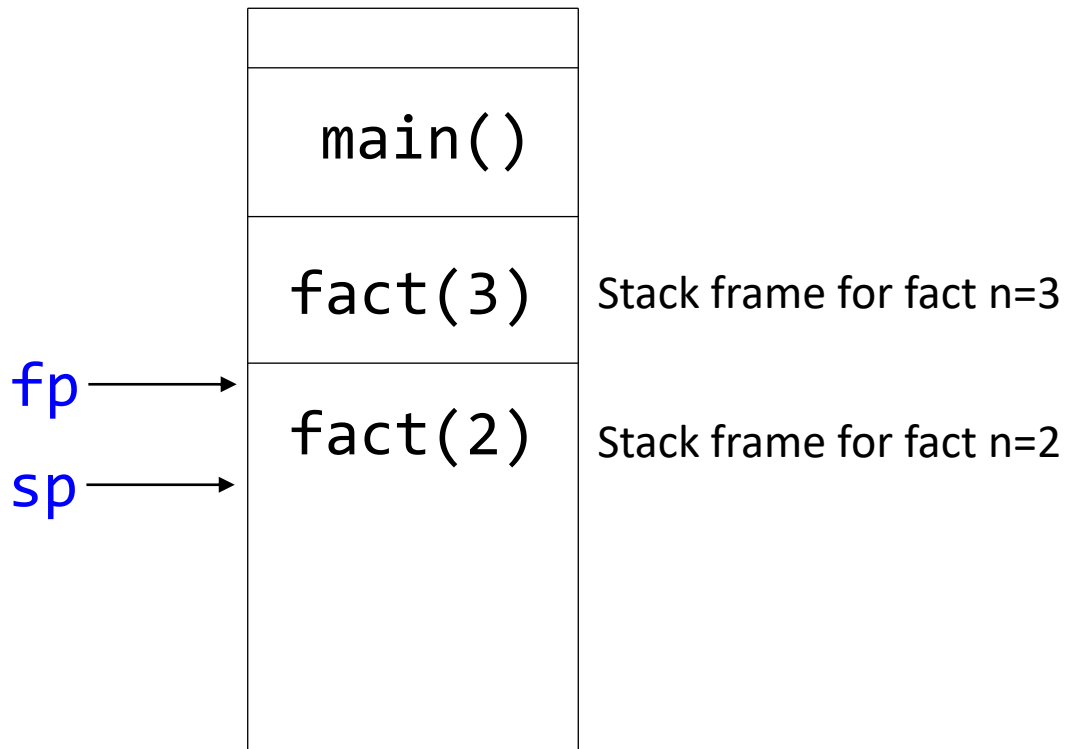


```
main() {  
    ...  
    fact(3);  
    ...  
}  
  
fact(int n) {  
    if (n=0) return 1  
    return n*fact(n-1)  
}
```

Stack frame for n=0 popped off. 1 Returned.

# Activation Record – Example (Recursive Functions)

Stack



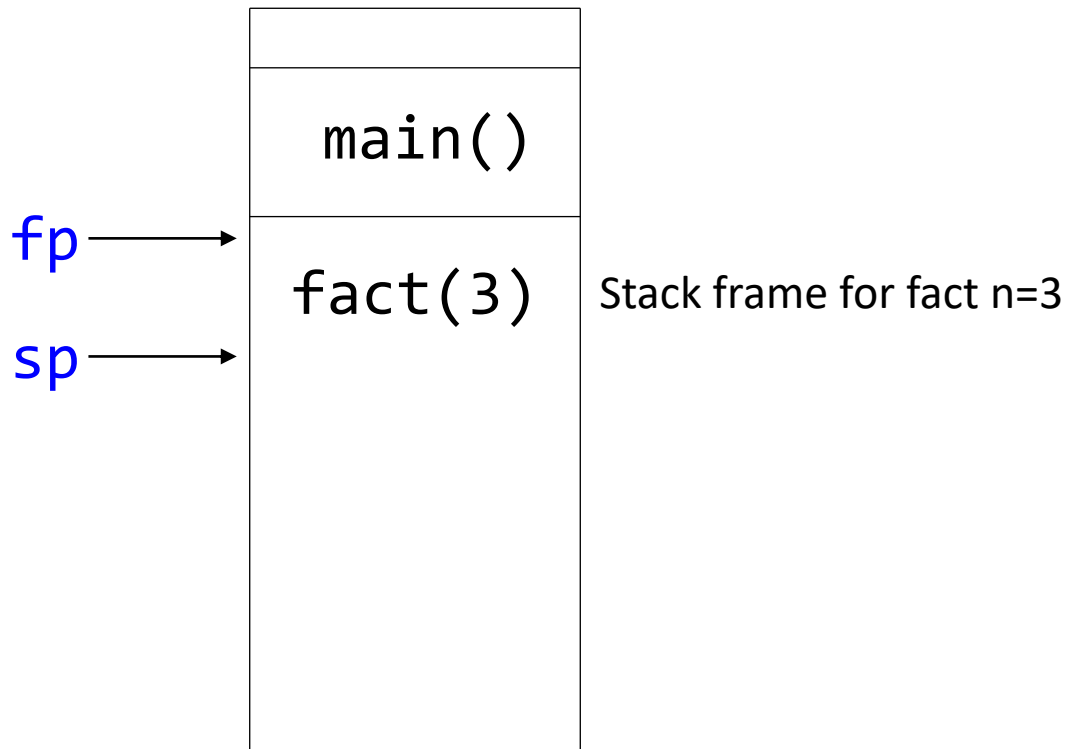
```
main() {  
    ...  
    fact(3);  
    ...  
}
```

```
fact(int n) {  
    if (n=0) return 1  
    return n*fact(n-1)  
}
```

Stack frame for n=1 popped off. 1 Returned.

# Activation Record – Example (Recursive Functions)

Stack

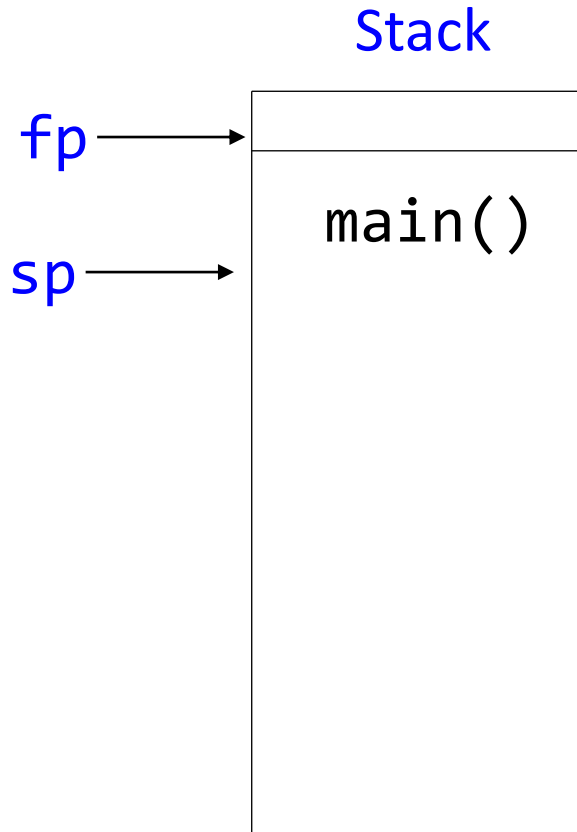


```
main() {  
    ...  
    fact(3);  
    ...  
}
```

```
fact(int n) {  
    if (n=0) return 1  
    return n*fact(n-1)  
}
```

Stack frame for n=2 popped off. 2 Returned.

# Activation Record – Example (Recursive Functions)



```
main() {  
    ...  
    fact(3);  
    ...  
}  
  
fact(int n) {  
    if (n=0) return 1  
    return n*fact(n-1)  
}
```

Stack frame for n=3 popped off. 6 Returned.

# Activation Record

- What happens when a function is called?
  1. fp and sp get adjusted
  2. Memory for the activation record is allocated on stack
    - The size of the memory allocated depends on local variables used by the called function (consult function's symbol table for this)
  3. Each invocation of a function has its own instantiation of local variables
- When the function call returns:
  - Memory for the activation record is destroyed when the function returns



# Activation Record

- What is stored in the activation record?

Depends on the language being implemented:

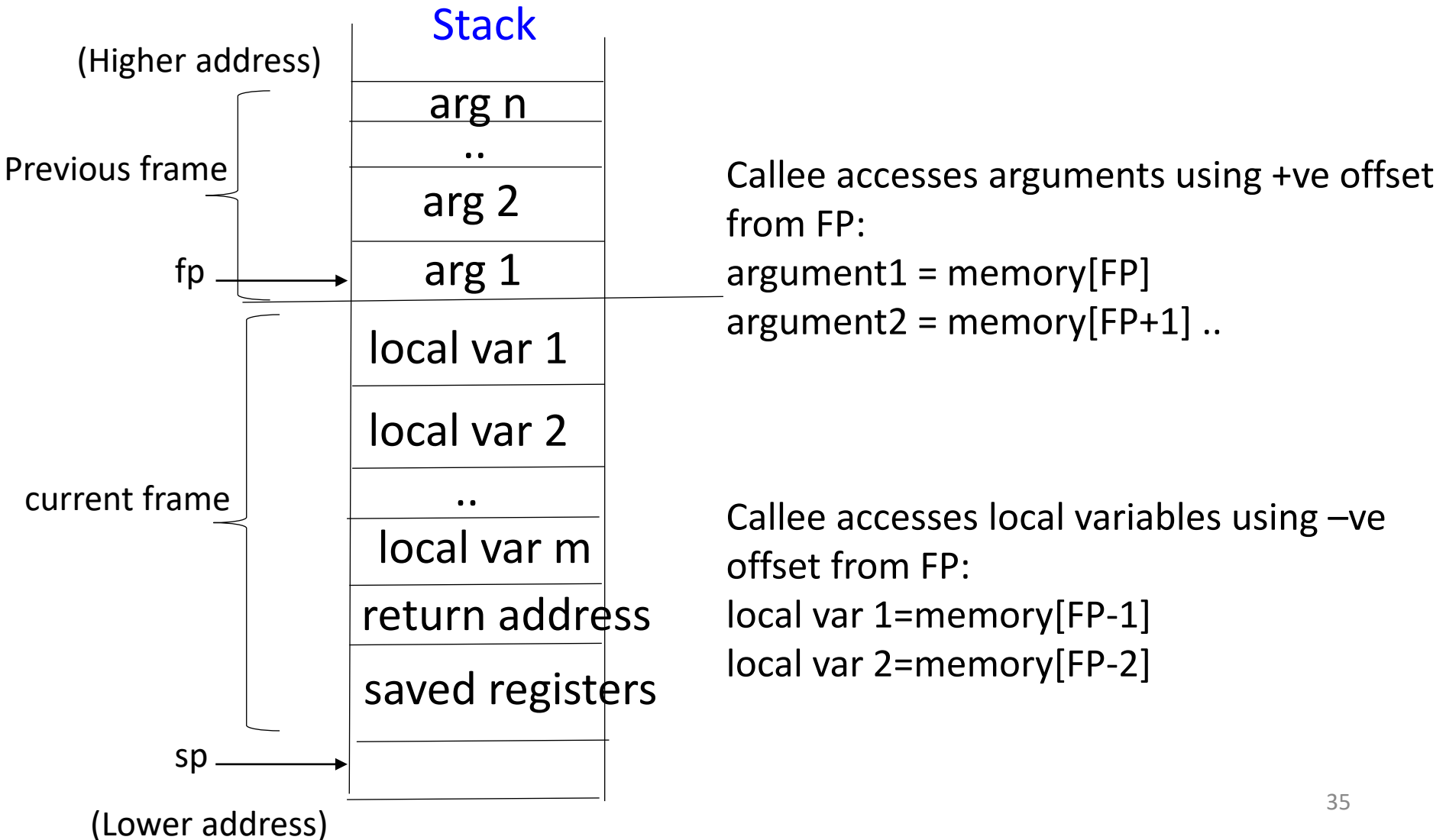
- Temporaries
  - Local vars
  - Saved registers
  - Return address, previous fp
  - Return value
  - Actual Params
- Who stores this information?
    - Caller } together execute *calling sequence* and *return*
    - Callee } *sequence*

# Application Binary Interface (ABI)

- How is data organized on the activation record?
  - ABI is the specification on how data is provided to functions
    - Caller saves or callee saves
  - ABI is meant to deliver interoperability between different compilers
    - Compile the function using one compiler to create an object code, Link object code with other code compiled using a different compiler

form the *calling convention*

# Typical Activation Record



# Function call: Peeking at Activation Record

- When `main` calls function `foo`

1. The following are pushed on to the stack:

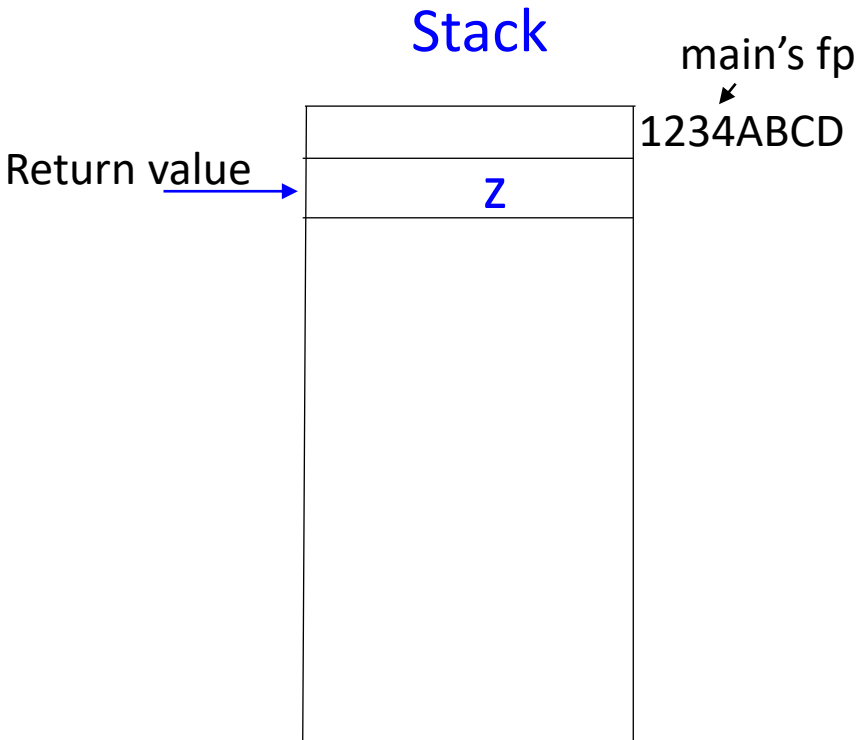
1. `foo`'s **arguments**
2. Space to hold `foo`'s **return value**
3. Address of the next instruction executed (in `main`) when `foo` returns (**return address**)
4. Current value of `$rbp` (**frame pointer**)

```
main() {  
    ...  
    foo();  
    ...  
}
```

`$rsp` is automatically updated (decremented) to point to current top of the stack.

2. `$rbp` is assigned the value of `$rsp`

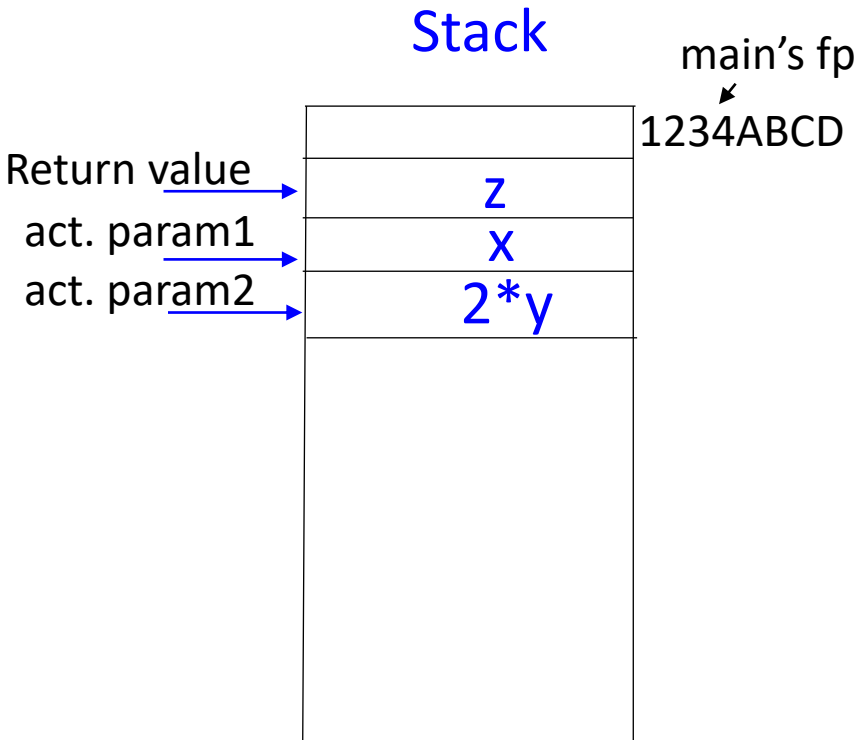
# Function call: Peeking at Activation Record



```
main() {  
    z=foo(x, 2*y);  
    return;  
}
```

```
int foo(int a, int b) {  
    int l1, l2  
    l1=a;  
    l2=b;  
    return l1+l2;  
}
```

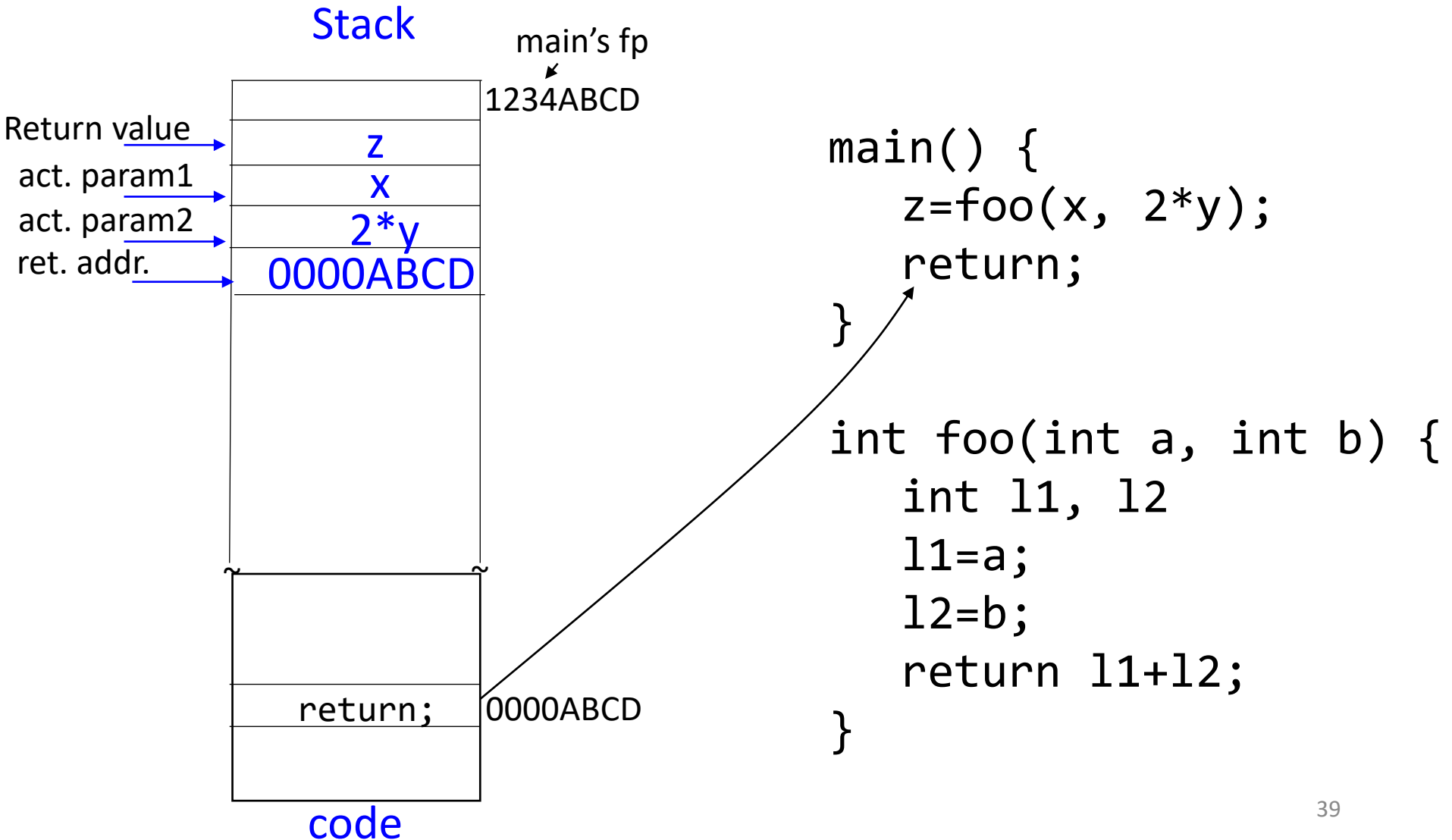
# Function call: Peeking at Activation Record



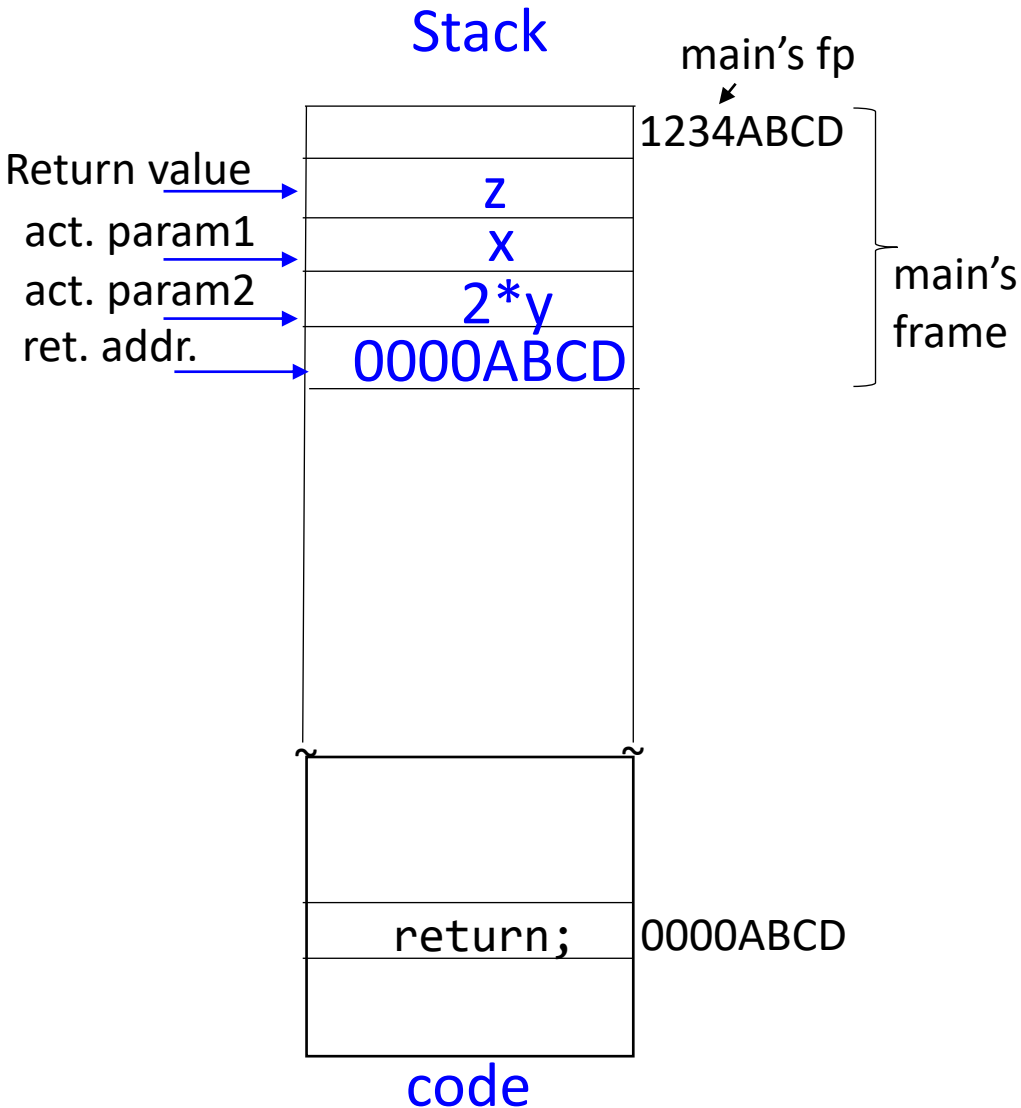
```
main() {  
    z=foo(x, 2*y);  
    return;  
}
```

```
int foo(int a, int b) {  
    int l1, l2  
    l1=a;  
    l2=b;  
    return l1+l2;  
}
```

# Function call: Peeking at Activation Record



# Function call: Peeking at Activation Record

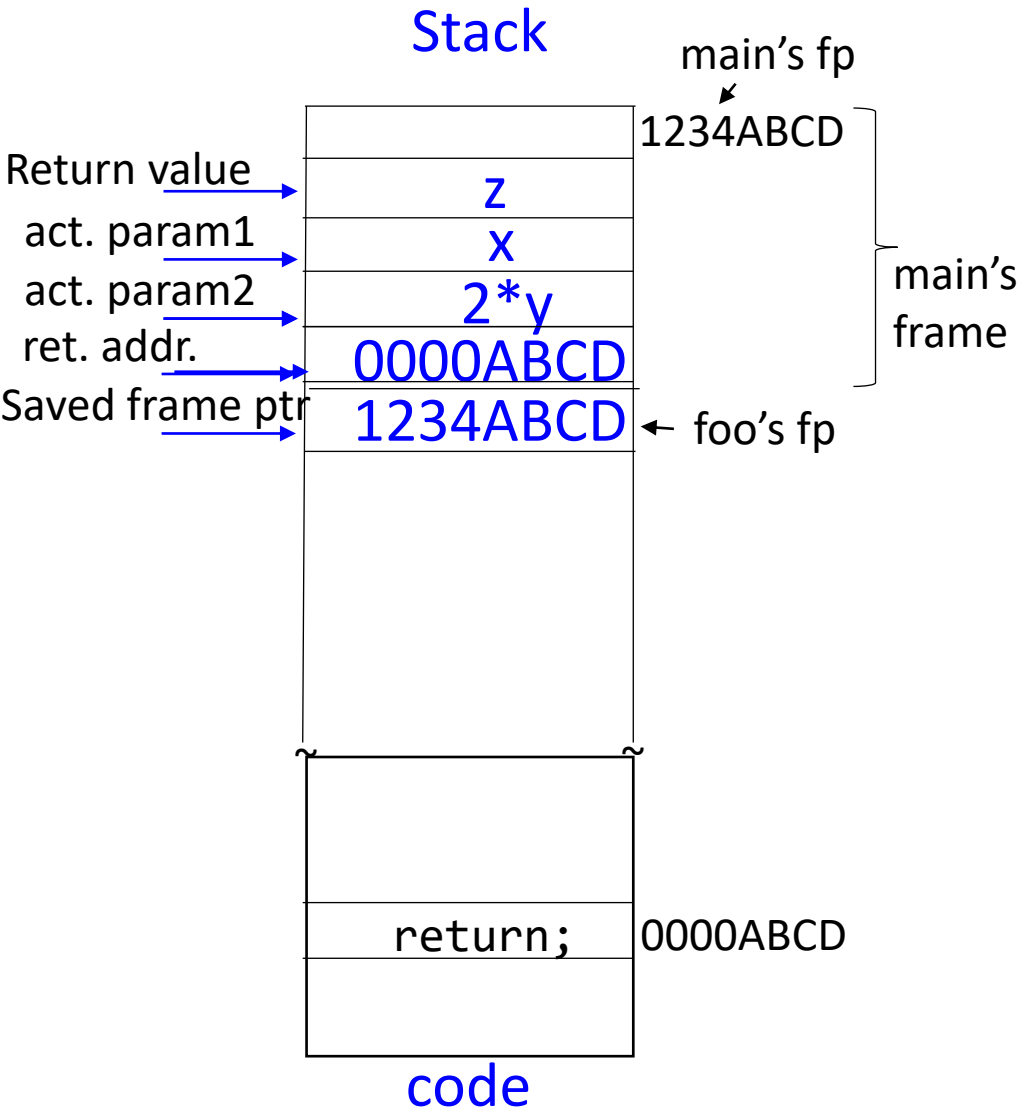


```
main() {  
    z=foo(x, 2*y);  
    return;  
}
```

```
int foo(int a, int b) {  
    int l1, l2  
    l1=a;  
    l2=b;  
    return l1+l2;  
}
```



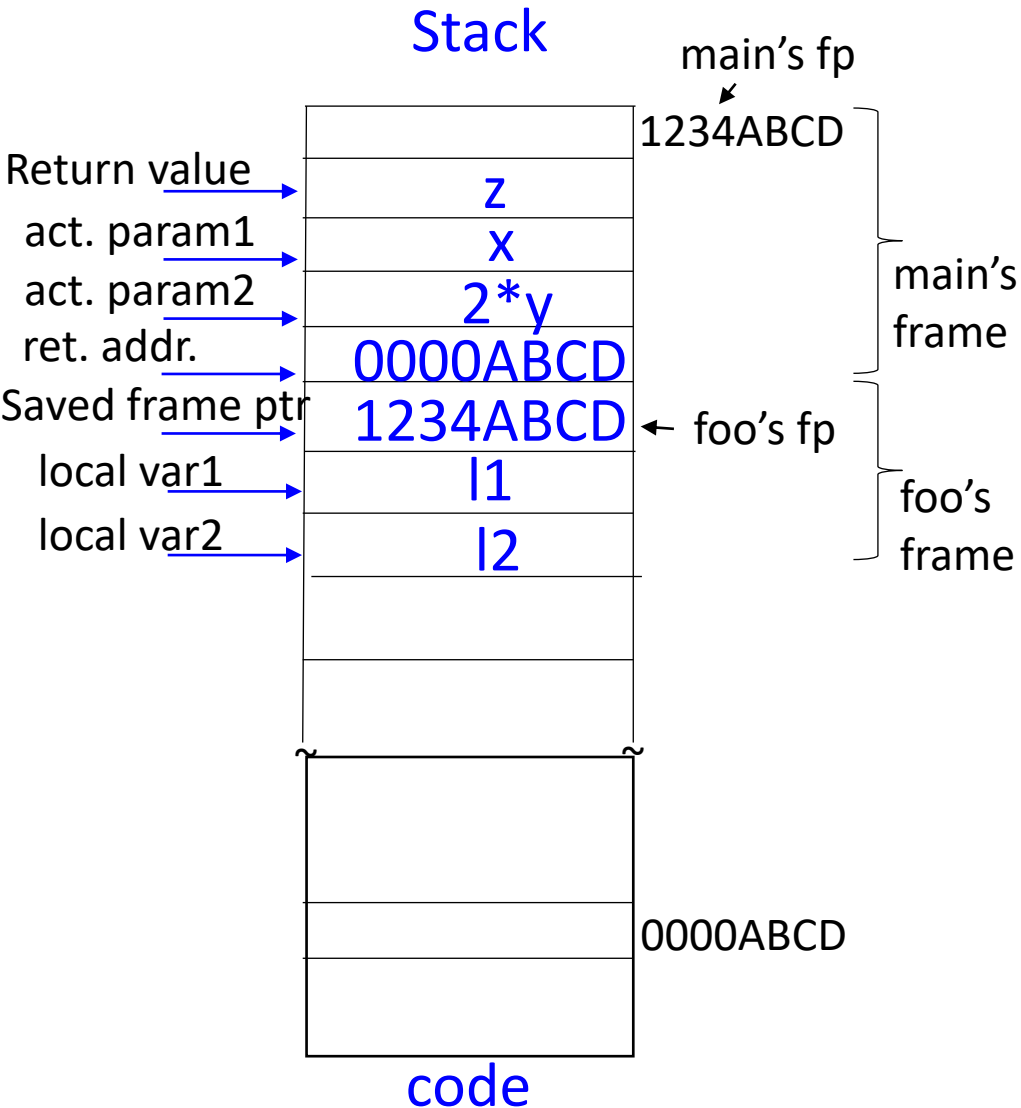
# Function call: Peeking at Activation Record



```
main() {  
    z=foo(x, 2*y);  
    return;  
}
```

```
int foo(int a, int b) {  
    int l1, l2  
    l1=a;  
    l2=b;  
    return l1+l2;  
}
```

# Function call: Peeking at Activation Record



```
main() {
    z=foo(x, 2*y);
    return;
}
```

```
int foo(int a, int b) {
    int l1, l2
    l1=a;
    l2=b;
    return l1+l2;
}
```

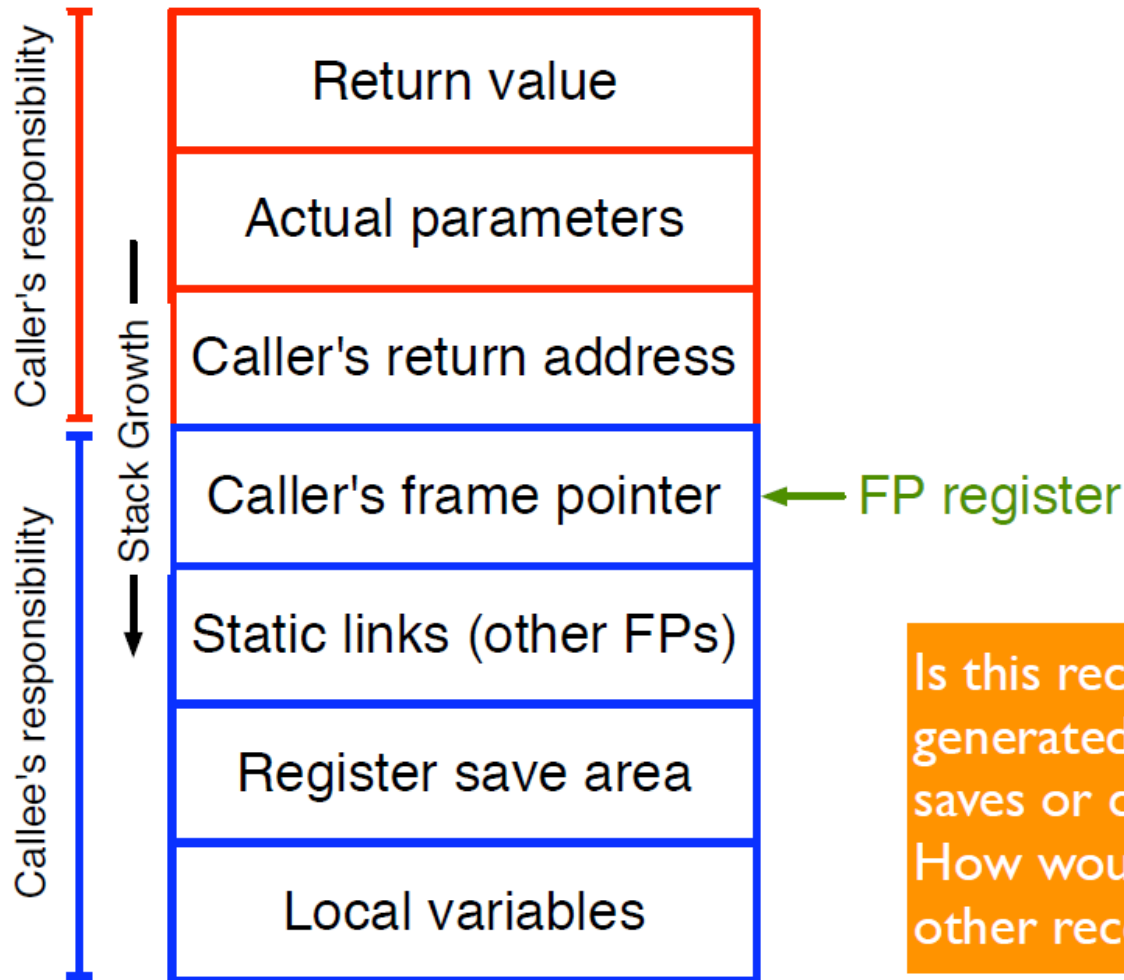
# Function calls – Register Handling

- **Did not use registers** in the previous example (for parameter passing)
- Registers are faster than memory. So, compiler should keep parameters in registers whenever possible
- Modern calling convention places first few arguments in registers (arg1 in r1, arg2 in r2, arg3 in r3...) and the remaining in memory.
  - In x86 C-ABI, first 6 arguments are passed in registers
- What if callee wants to use registers r1, r2, r3 etc. for local computation? Callee must save the registers in its stack frame.

# Function calls – Register Handling

- Two options: **caller saves** or **callee saves**
- Caller Saves
  - Caller pushes all the registers it is using on to the stack before calling the function
  - Restores the registers after the function returns
- Callee Saves
  - Callee pushes all the registers it is *going to use* on the stack immediately after being called
  - Restores the registers just before it returns

# Activation records



Is this record generated for callee-saves or caller-saves? How would the other record look?

# Activation Record – Return Address and Return Value

- Callee must be able to return to the caller when done
- Return address is the address of the instruction following the function call
- Return address can be placed on the stack or on register
- The `call` instruction on modern machines places the return address in a specific register
- Return value is placed in a specific register by the callee function

# The frame pointer

Push x:

```
stack_pointer--
```

```
Memory[stack_pointer] = x
```

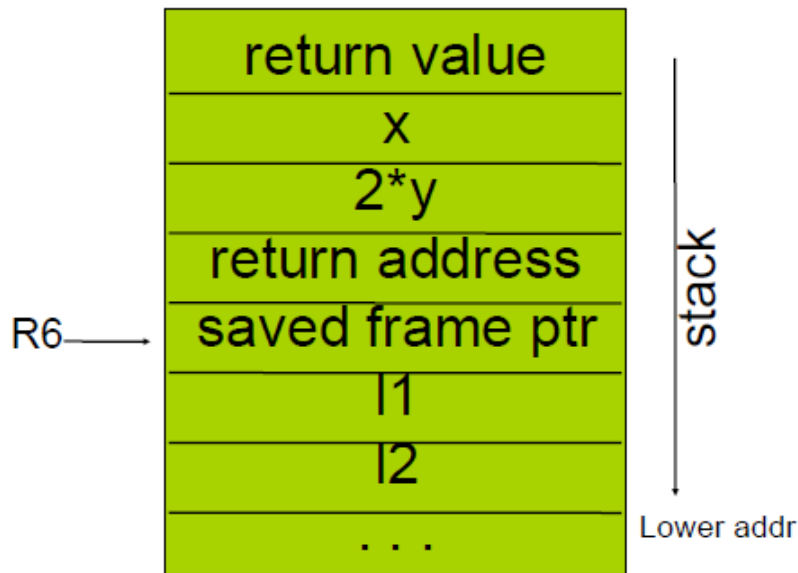
Pop x:

```
x = Memory[stack_pointer]
```

```
stack_pointer--
```

- Manipulate with instructions like `link` and `unlink`
- Link: push current value of FP on to stack, set FP to top of stack
- Unlink: read value at current address pointed to by FP, set FP to point to that value
- In other words: link pushes a new frame onto the stack, unlink pops it off

# Example Subroutine Call and Stack Frame



3-address code:

```
push
push x
mul 2 y t1
push t1
jsr SubOne
pop
pop
pop z
```

assembly code:

```
push
push x
load y R1
mul 2 R1
push R1
jsr SubOne
pop
pop
pop R1
store R1 z
```

```
z = SubOne(x, 2*y);
```

```
int SubOne(int a, int b) {
    int l1, l2;
    l1 = a;
    l2 = b;
    return l1+l2;
};
```

```
link 3
move $P1 $L1
move $P2 $L2
add $L1 $L2 t2
move t2 $R
unlink
ret
```

```
link R6 3
load 3(R6) R1
store R1 -1(R6)
load 2(R6) R2
store R2 -2(R6)
load -1(R6) R1
add -2(R6) R1
store R1 4(R6)
unlink
ret
```



# Questions ?

*Where are the command-line arguments stored?*

*How about environment variables such as LD\_LIBRARY\_PATH and PATH?*

**Challenge Q:** *are there scenarios where the activation record is required to be allocated on the heap?*