CS 320 : Functions

Marco Gaboardi

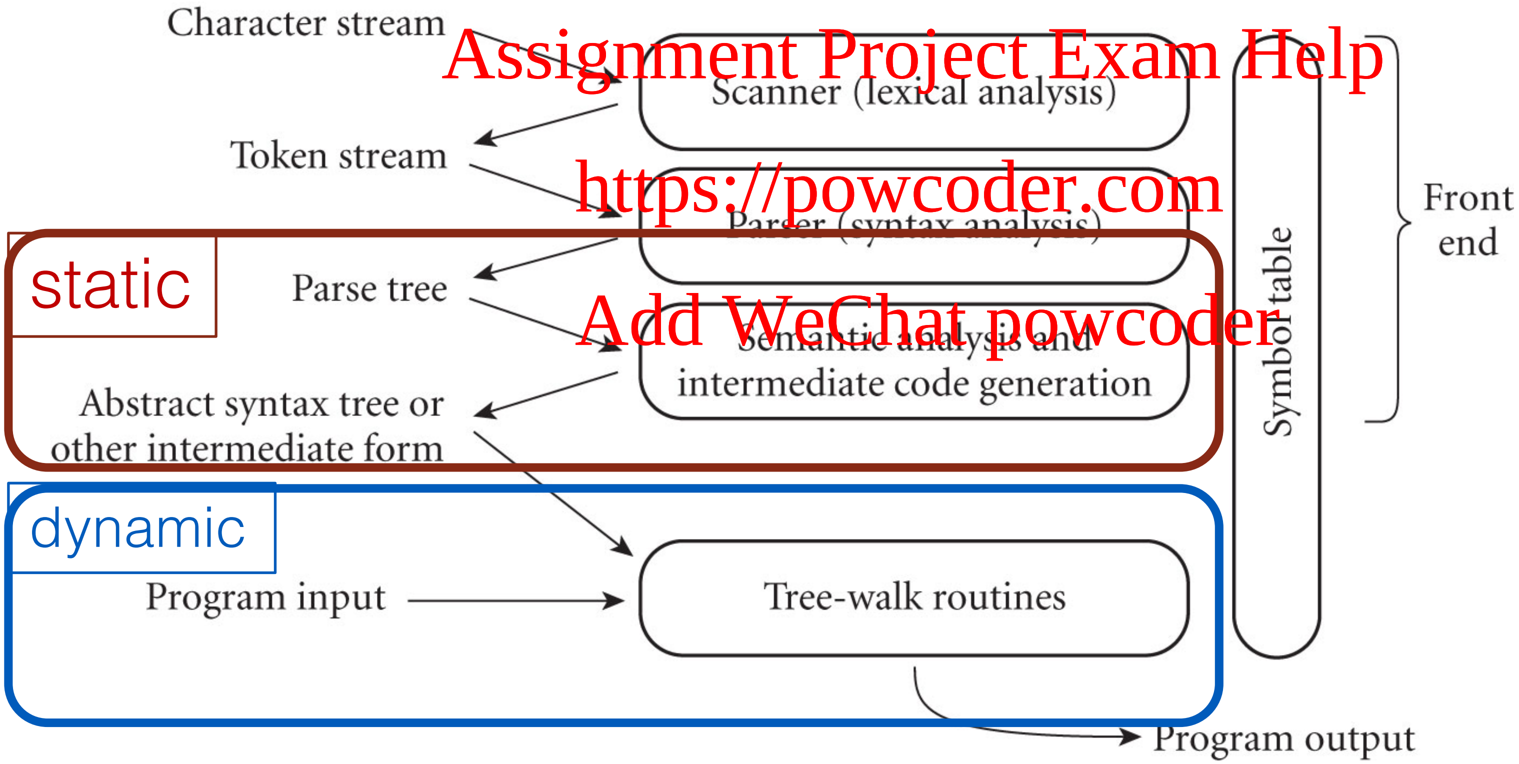
MSC 116

gaboardi@bu.edu

Announcements

- Interpreter part 3 due Sunday (last programming assignment)
- Last Theory Assignment due the 10th (deadline extension)

Parsing and semantic analysis

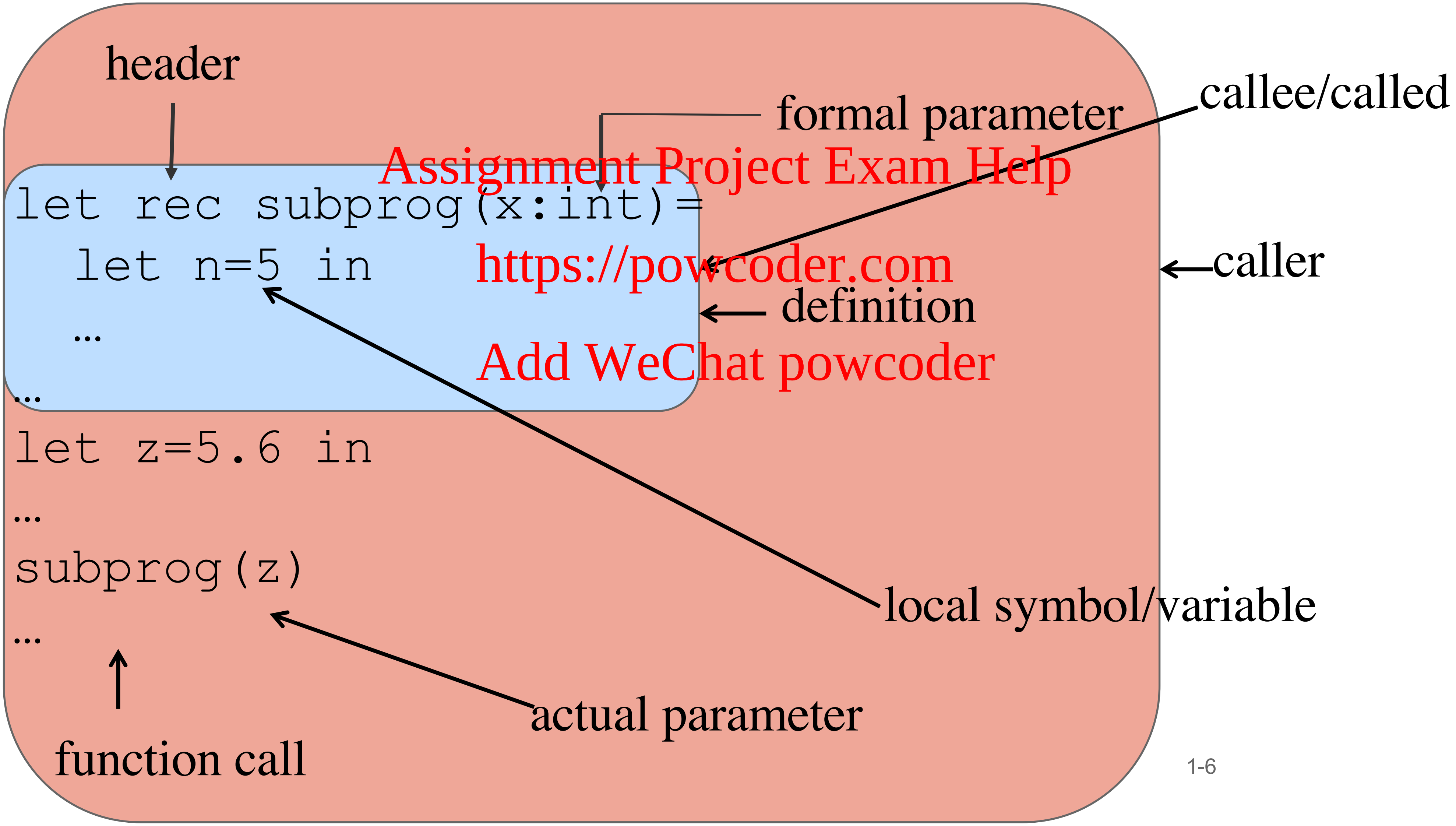


Today plan

- More on Functions

Functions

Terminology



```
let rec subprog(x:int)=
   let n=5 in
   …
…
let z=5.6 in
…
subprog(z)
…
```

header

formal parameter

callee/called

caller

definition

local symbol/variable

actual parameter

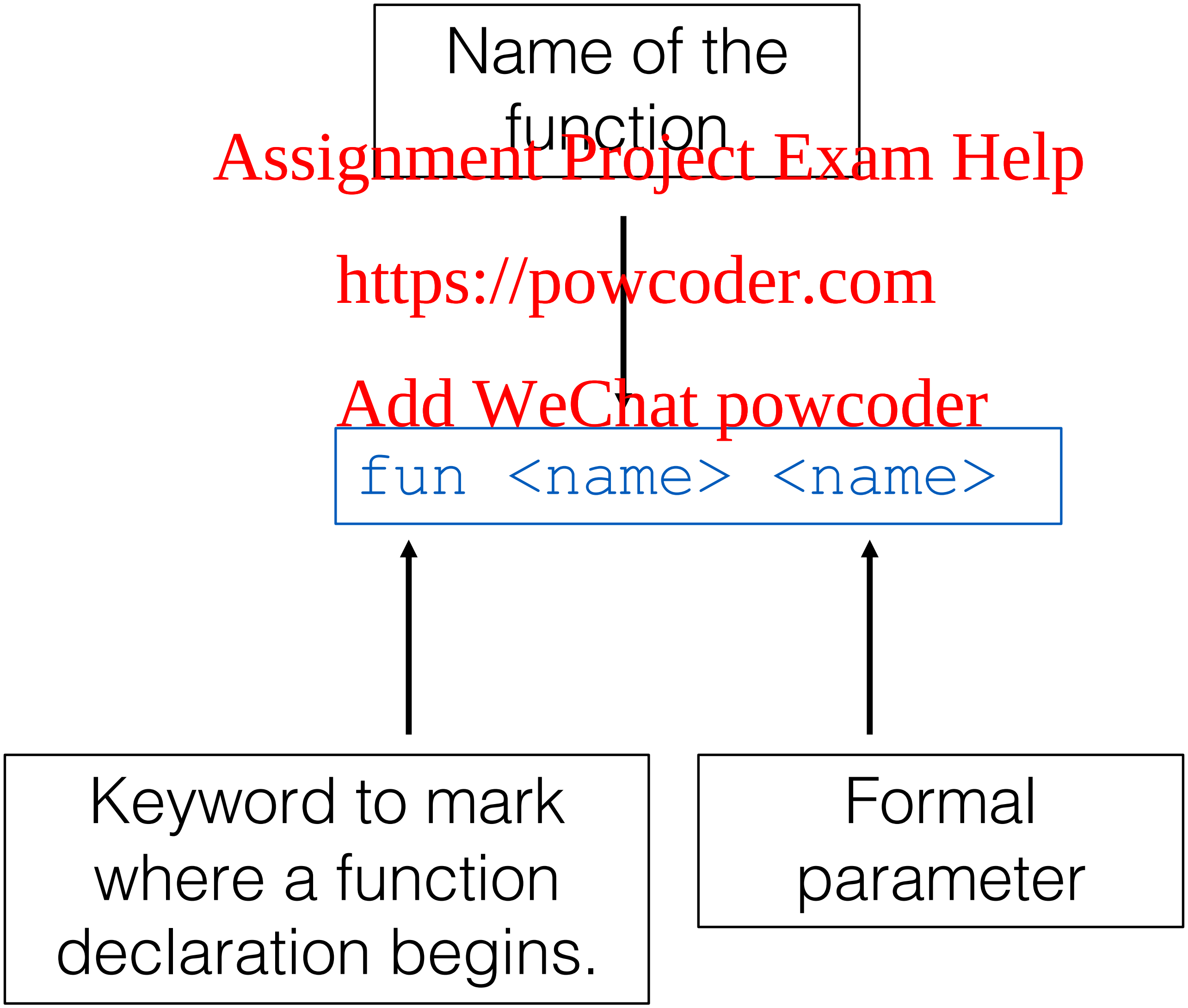
function call

Language for basic stack manipulations with local variables definitions and functions

```
…

<com>    ::= push <const> | add | sub | mul | div
           | neg | rem | swap | let | end | bind
           | fun <name> <name> | funEnd |return |call
```

Language for basic stack manipulations with local variables definitions and functions



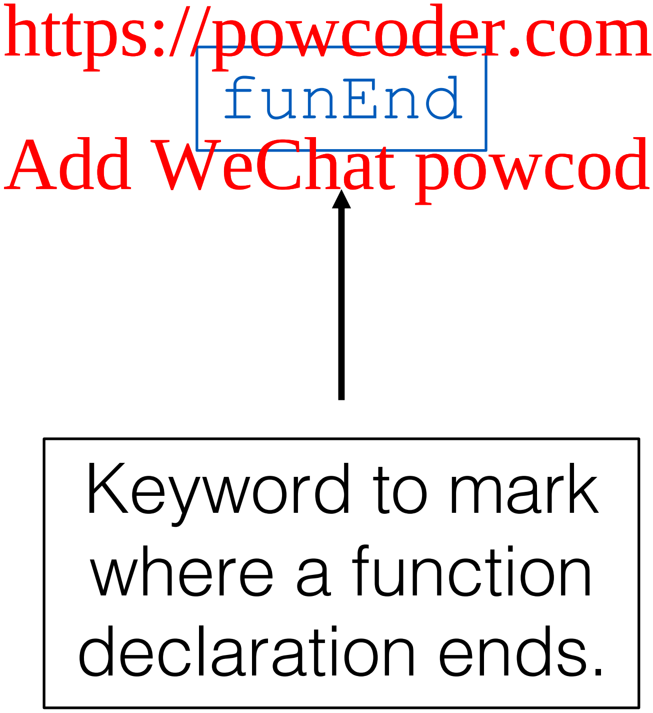
Name of the function

```
fun <name> <name>
```

Keyword to mark where a function declaration begins.

Formal parameter

Language for basic stack manipulations with local variables definitions and functions



`funEnd`

Keyword to mark where a function declaration ends.

Language for basic stack manipulations with local variables definitions and functions

```
return
```

Keyword to mark when a function needs to return a value.

Language for basic stack manipulations with local variables definitions and functions

`call`

Keyword to mark when a function needs to be called.

Language for basic stack manipulations with local variables definitions and functions

input

```
  fun identity x
  push x
  return
  funEnd

push identity
push 1
call
quit
```

Actual parameter

What are the design considerations for functions?

We need to think about:

- parameter passing
- parameters returning
- variables: local vs global
- scope of variables
- nesting of subprograms
- referencing environment

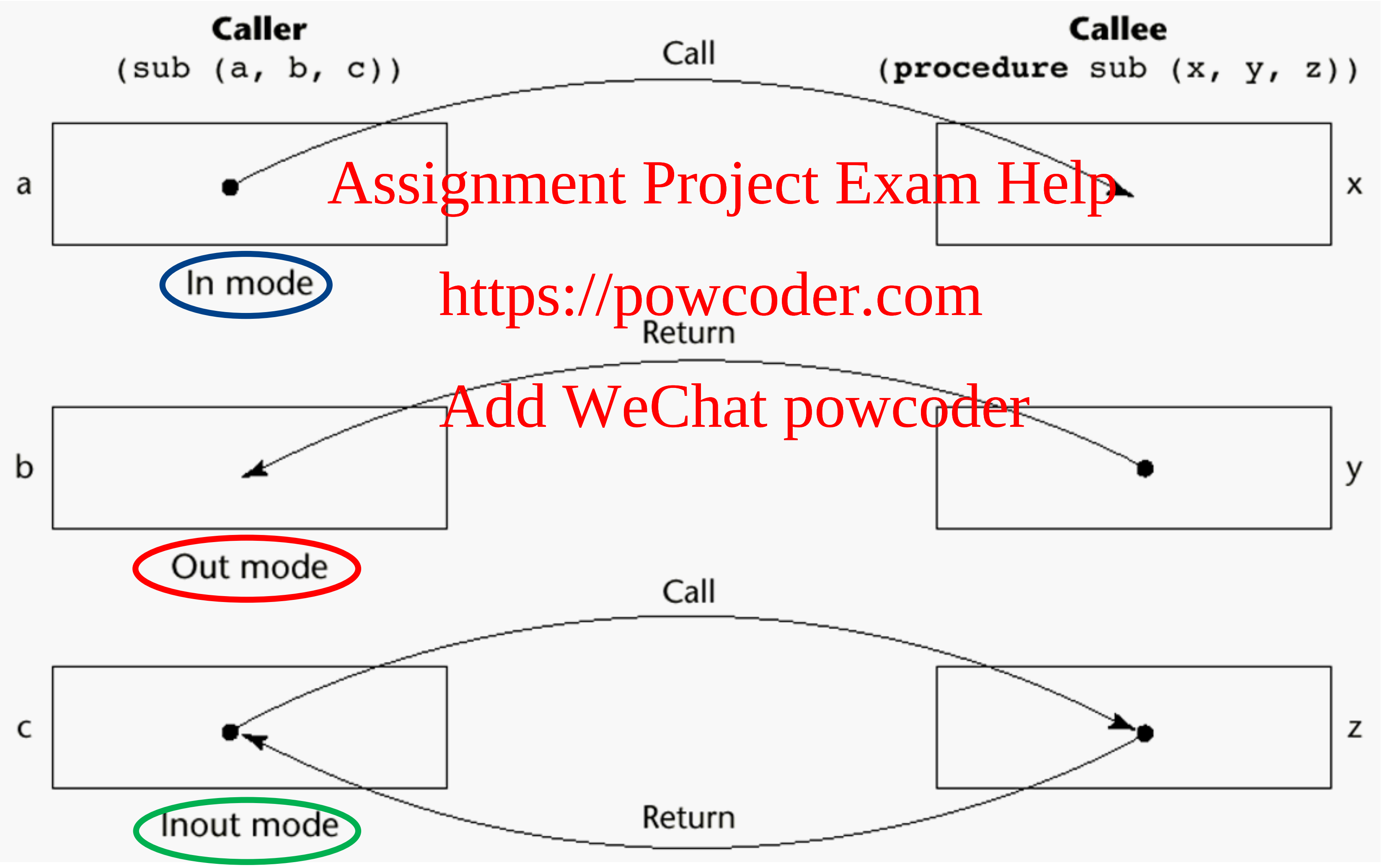
Parameter Passing

Parameter passing methods are ways in which parameters are transmitted to and from sub programs.

- Semantic Models of Parameter Passing

- Implementation Models for these semantic models

Semantic Modes of Parameter Passing



How to transfer a value

- We have different ways to provide access to a value to a subprogram
  - Physically move a value
  - An access path is transmitted (e.g. pointer or reference)
- These are orthogonal to the mode of the parameters

TopHat Q15-Q18

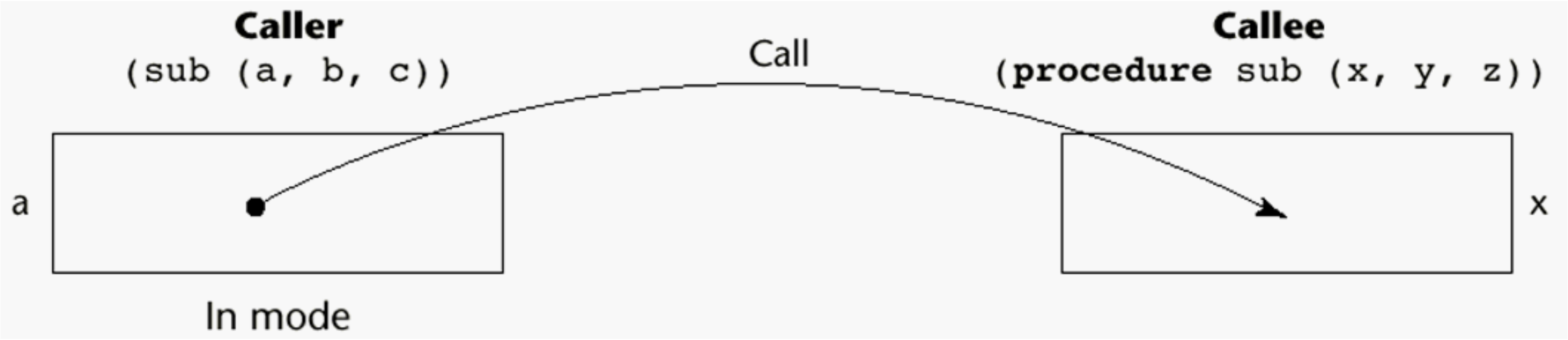
Implementation Models

Techniques used for parameter passing :

- Call by Value (In mode)
- Call by Result (Out mode)
- Call by Value-Result (In-out mode)
- Call by Reference (In-out mode)

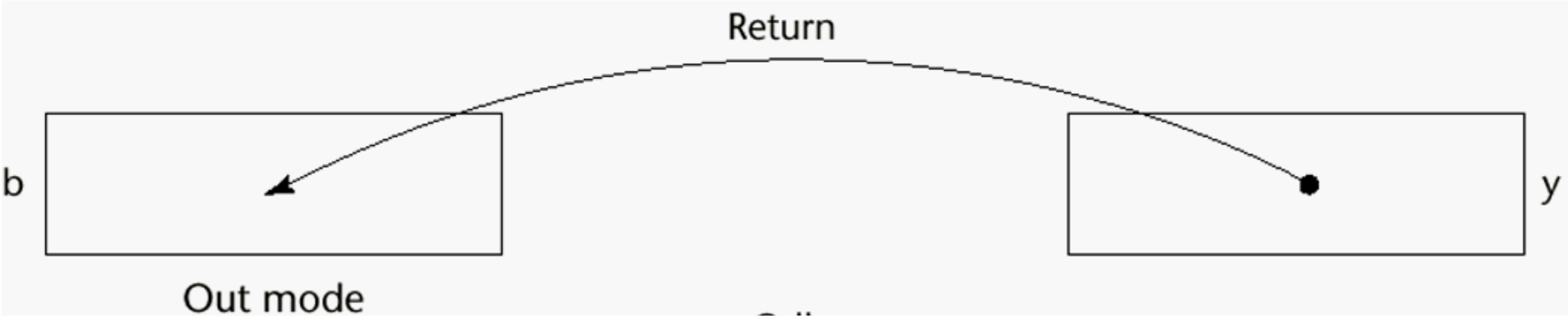
Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialized the corresponding formal parameter
  - Normally implemented by copying
  - Can be implemented by transmitting an access path but then one need to enforce write protection.



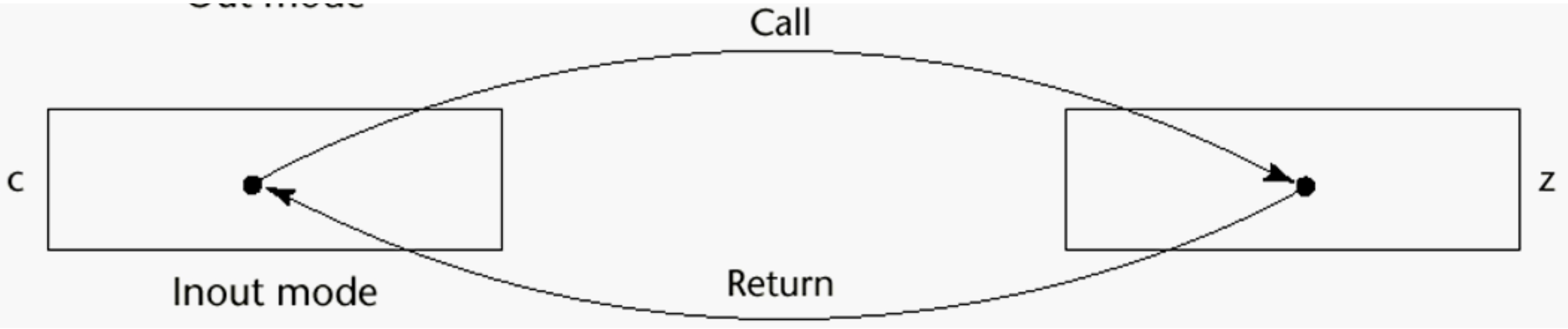
Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram;
  -the corresponding formal parameter acts as a local variable;
  -its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move



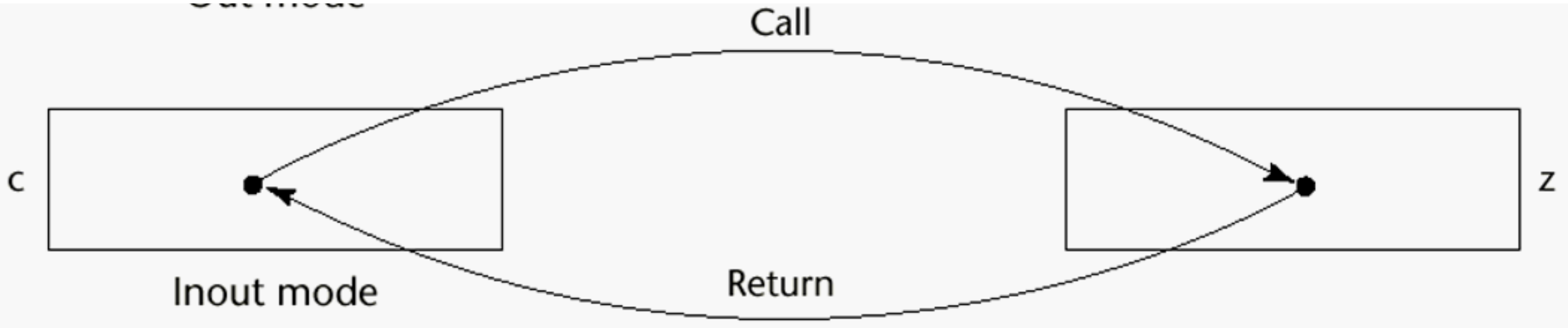
Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Actual values are copied in both directions.
- Formal parameters have local storage



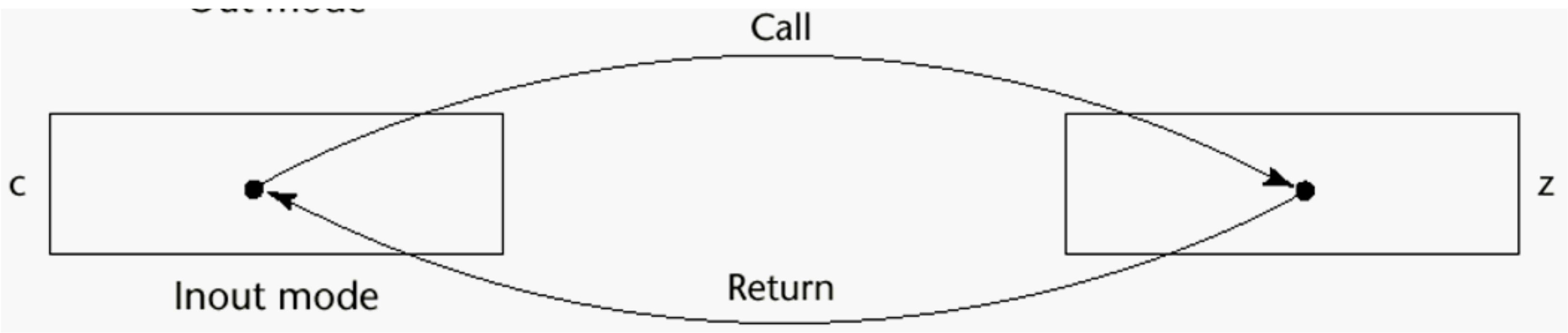
Pass-by-Reference (Inout Mode)

- Pass an access path to the value
- Passing process is efficient (no copying and no duplicated storage)
- Slower accesses (compared to pass-by-value) to formal parameters
- Potentials for unwanted side effects (collisions)
- Unwanted aliases (access broadened)



Pass-by-Name (Inout Mode)

- By textual substitution
- Formal parameters are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment



Implementing Parameter-Passing Methods

- In most languages parameter communication takes place through the run-time stack (more in the future)

- Pass-by-value parameters have their values copied into stack locations.

- Pass-by-reference are the simplest to implement; only an address is placed in the stack

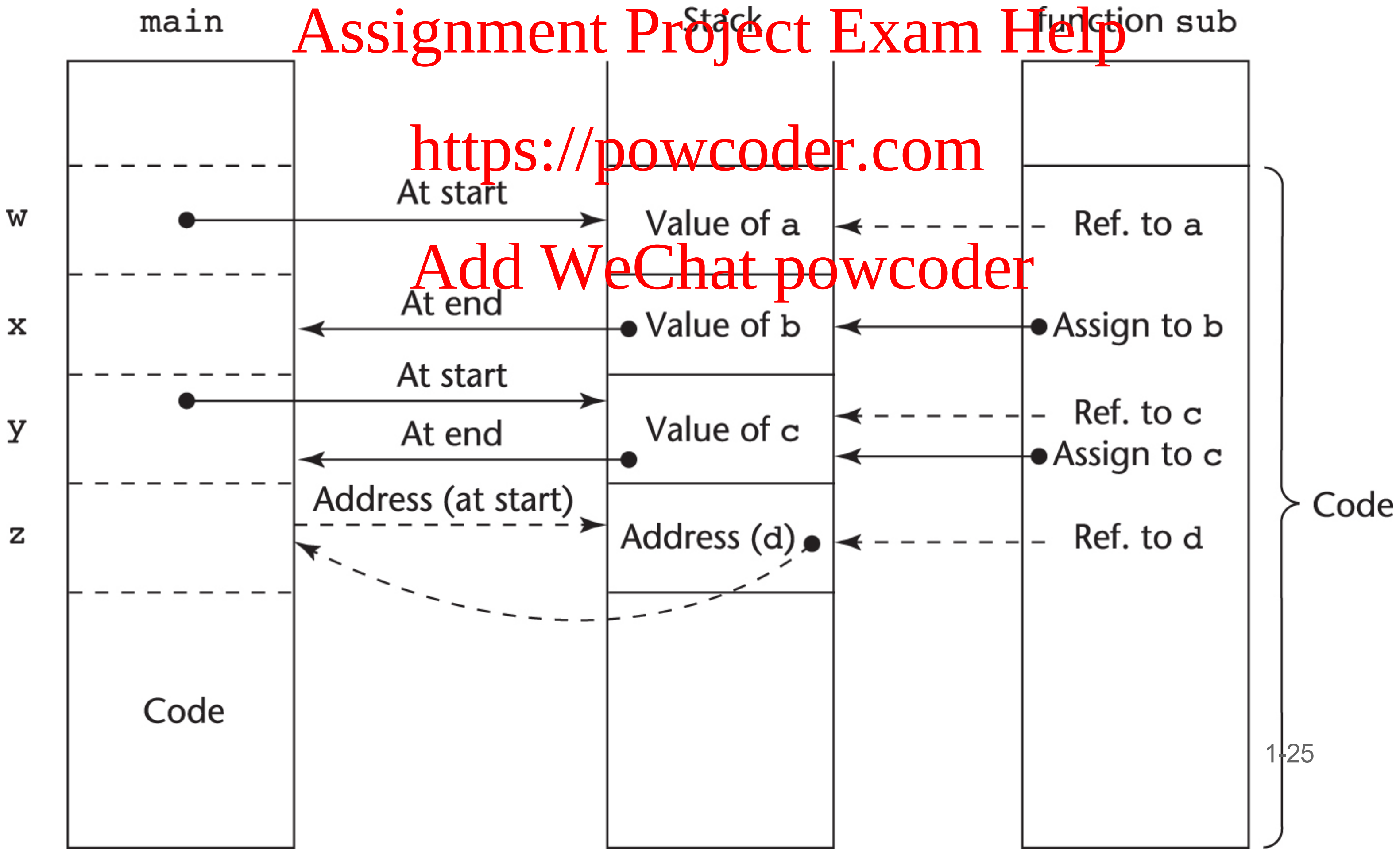
- In Pass-by-result the caller reads from the stack the final value of the parameter before the stack of the callee is disposed

Implementing Parameter-Passing Methods

Function header: `void sub(int a, int b, int c, int d)`

Function call: `sub(w, x, y, z)`

(pass `w` by value, `x` by result, `y` by value-result, `z` by reference)



Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```

Let assume that “simple” subprograms cannot be nested and that have only static local variables.

How a subprogram is executed? How can we implement the call-return procedure?

Function call

Call Semantics:

- Save the execution status of the caller
- In mode and inout mode parameters must be provided
- Pass the return address to the called subprogram
- Transfer control to the called subprogram

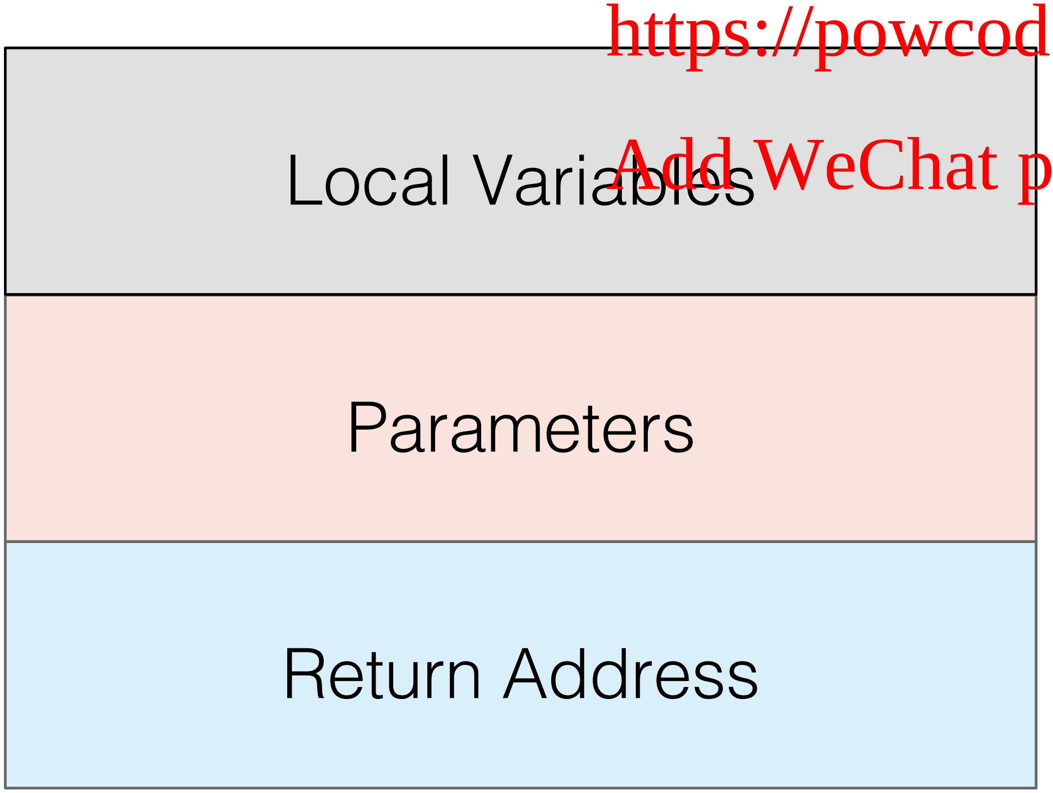
Function return

Return Semantics:

- If pass-by-value-result or out mode parameters are used, move the current values of those parameters to their corresponding actual parameters
- Restore the execution status of the caller
- Transfer control back to the caller

Activation Records for Simple Functions

- We need to store some information to guarantee the correct execution of the subprogram. This constitutes the activation record of the subprogram.



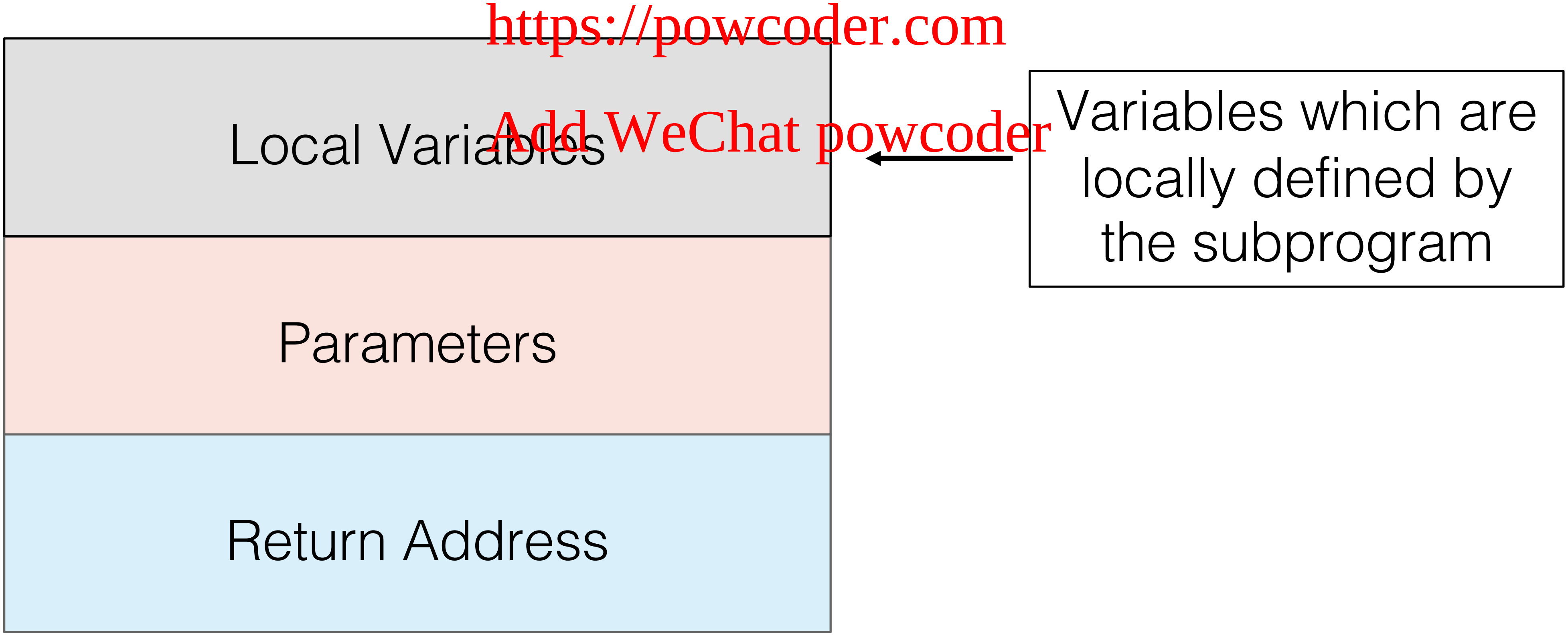
Local Variables

Parameters

Return Address

Activation Records for Simple Functions

- We need to store some information to guarantee the correct execution of the subprogram. This constitutes the activation record of the subprogram.



Local Variables

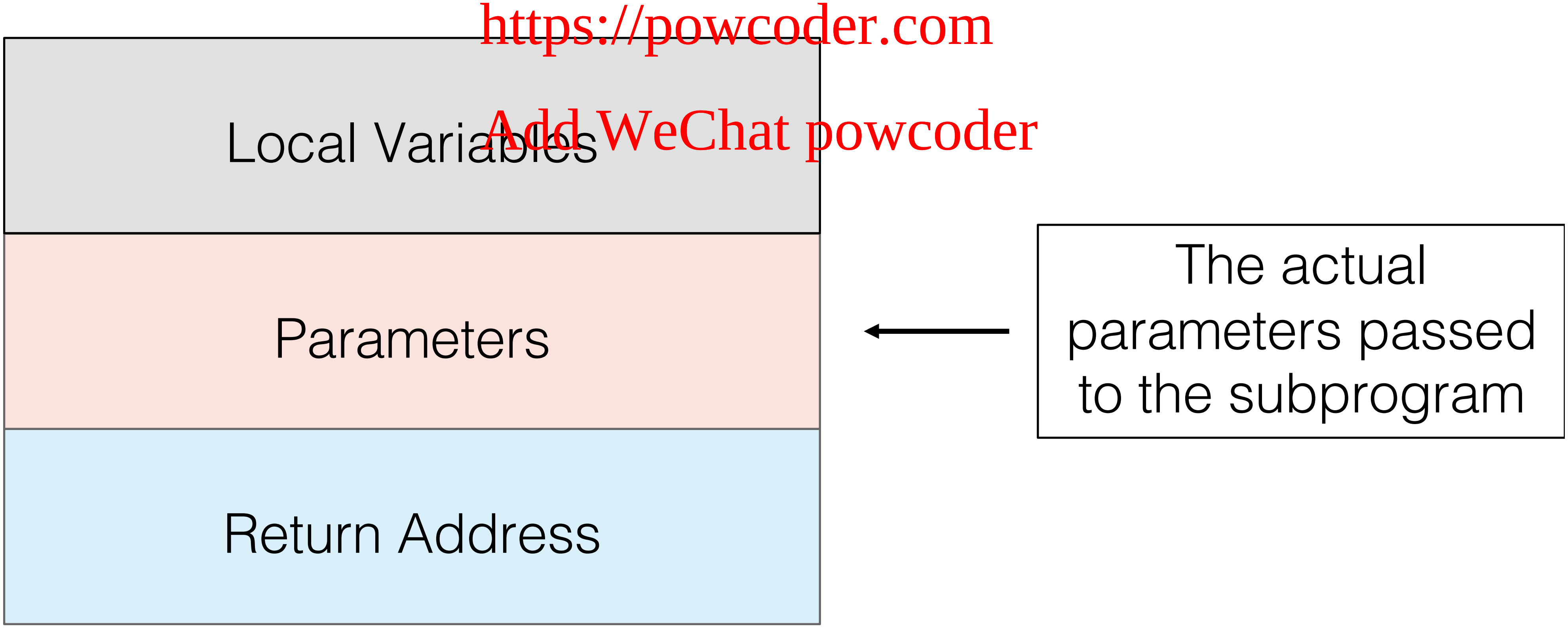
Parameters

Return Address

Variables which are locally defined by the subprogram

Activation Records for Simple Functions

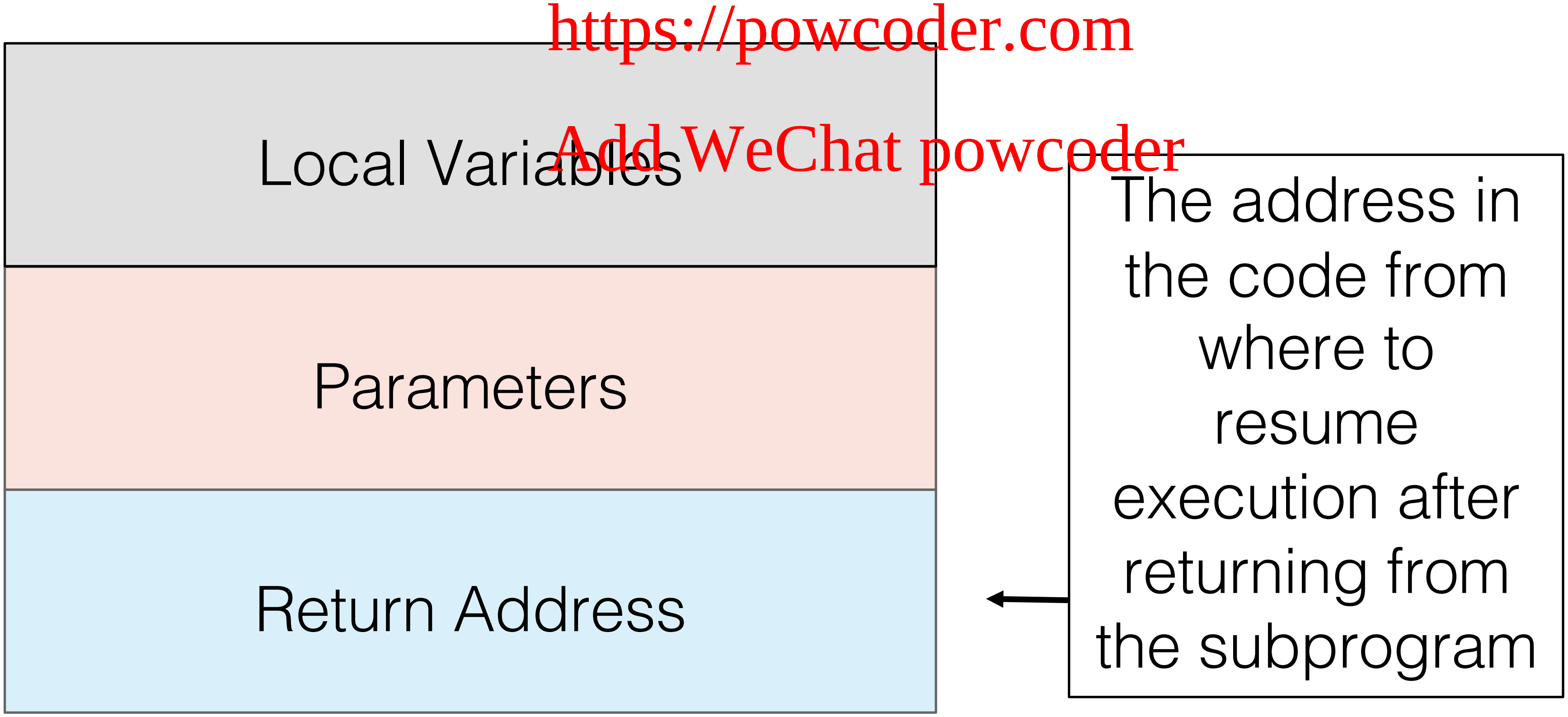
- We need to store some information to guarantee the correct execution of the subprogram. This constitutes the activation record of the subprogram.



The actual parameters passed to the subprogram

Activation Records for Simple Functions

- We need to store some information to guarantee the correct execution of the subprogram. This constitutes the activation record of the subprogram.



Local Variables

Parameters

Return Address

The address in the code from where to resume execution after returning from the subprogram

Information we need to store

- An activation record instance is a concrete example of an activation record (the collection of data for a particular subprogram activation)
- The activation record contains the non-code information that we need for the execution of the program.

Implementing Simple Functions: Activation Record

- The activation record format is static
- An activation record instance is dynamically created when a subprogram is called
- An activation record instance is dynamically deallocated when a subprogram returns
- Activation record instances reside on the run-time stack

For each activation record instance we need to maintain an Environment Pointer (EP) pointing at the base of the instance and used to deallocating it.

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```

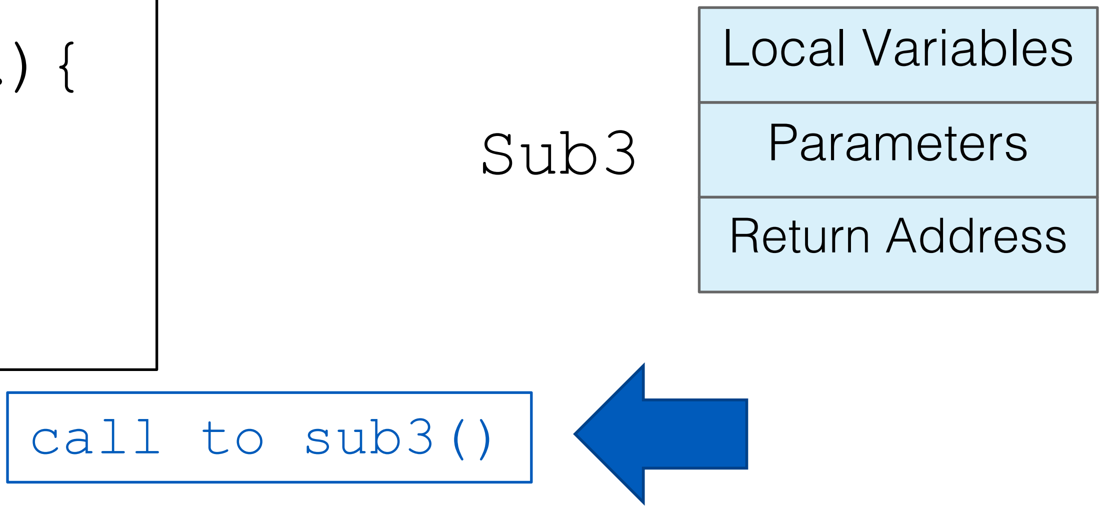
Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```

`call to sub3()`

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



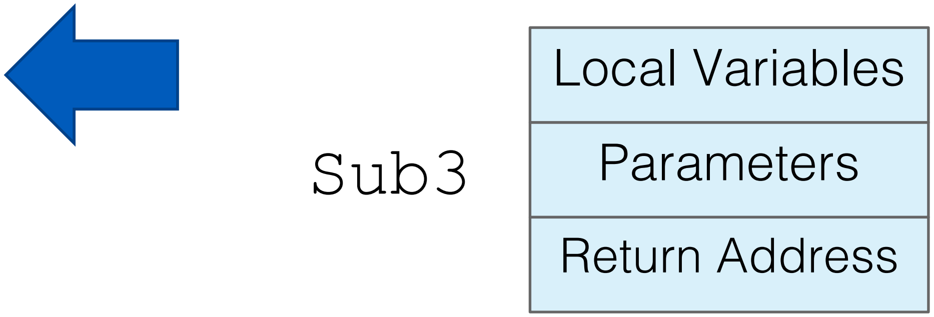
Sub3

Local Variables
Parameters
Return Address

`call to sub3()`

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



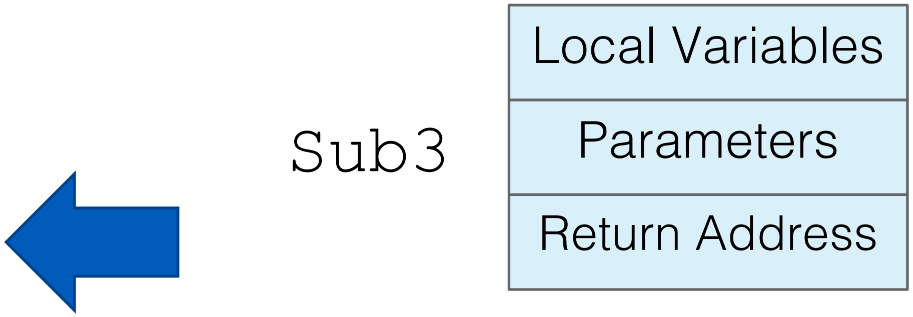
Sub3

Local Variables
Parameters
Return Address

call to sub3()

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



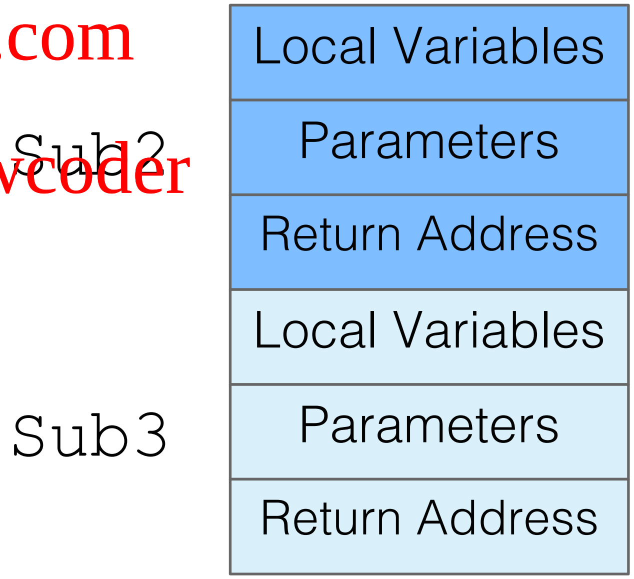
Sub3

Local Variables
Parameters
Return Address

call to sub3()

Example: Activation Records stack for Simple Functions

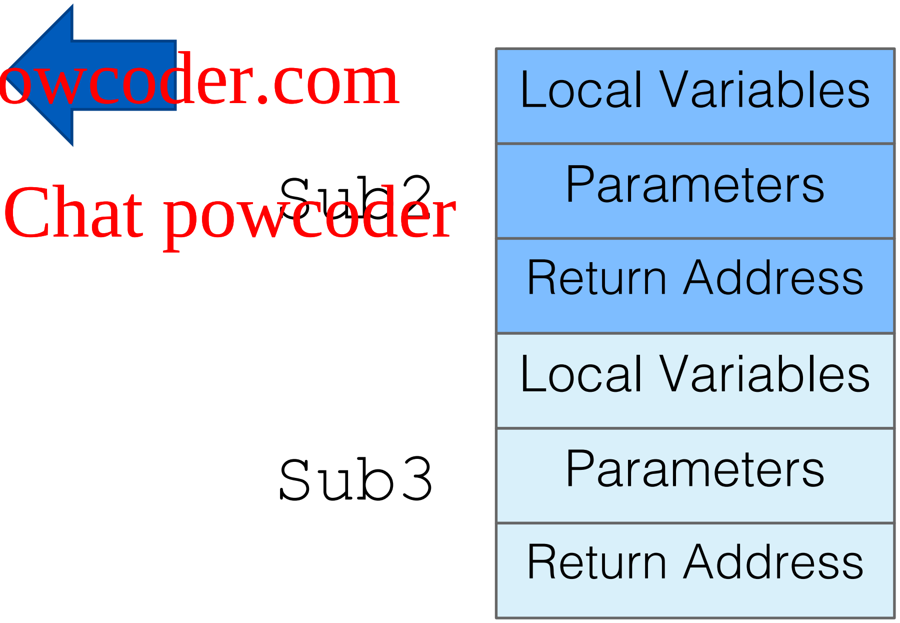
```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



call to sub3()

Example: Activation Records stack for Simple Functions

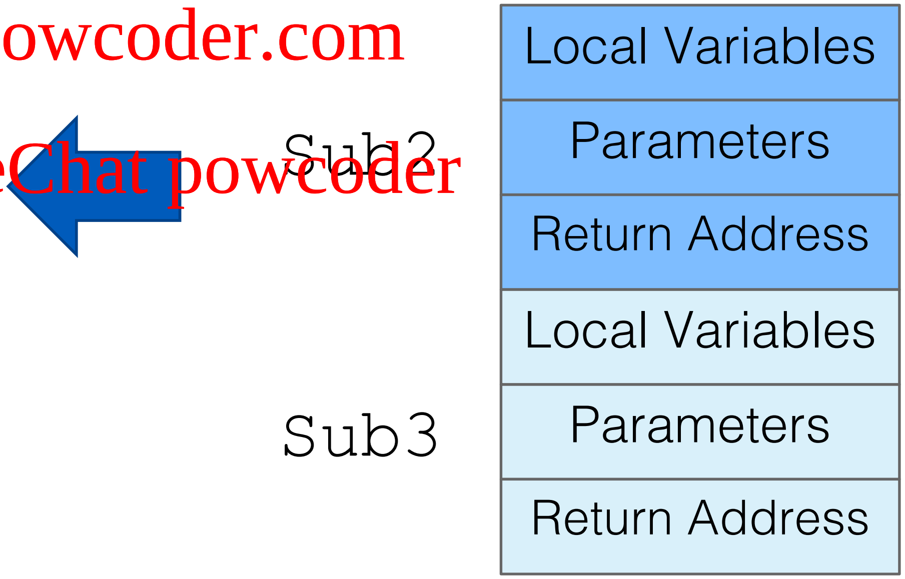
```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



call to sub3()

Example: Activation Records stack for Simple Functions

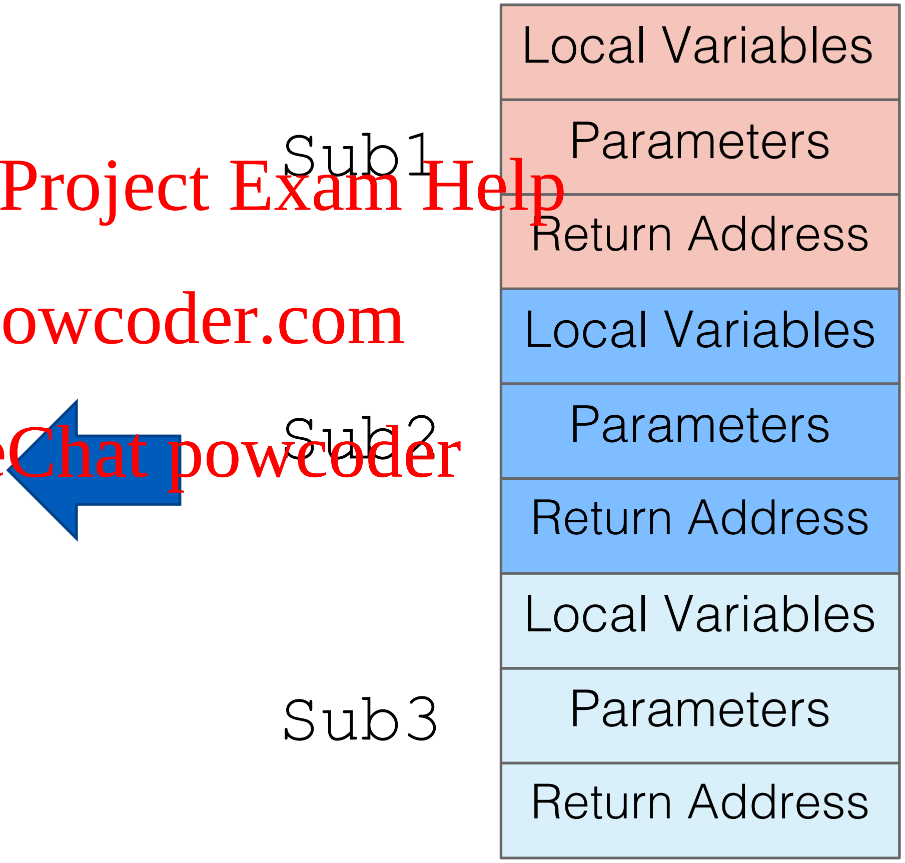
```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



call to sub3()

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



Sub1

Local Variables

Parameters

Return Address

Sub2

Local Variables

Parameters

Return Address

Sub3

Local Variables

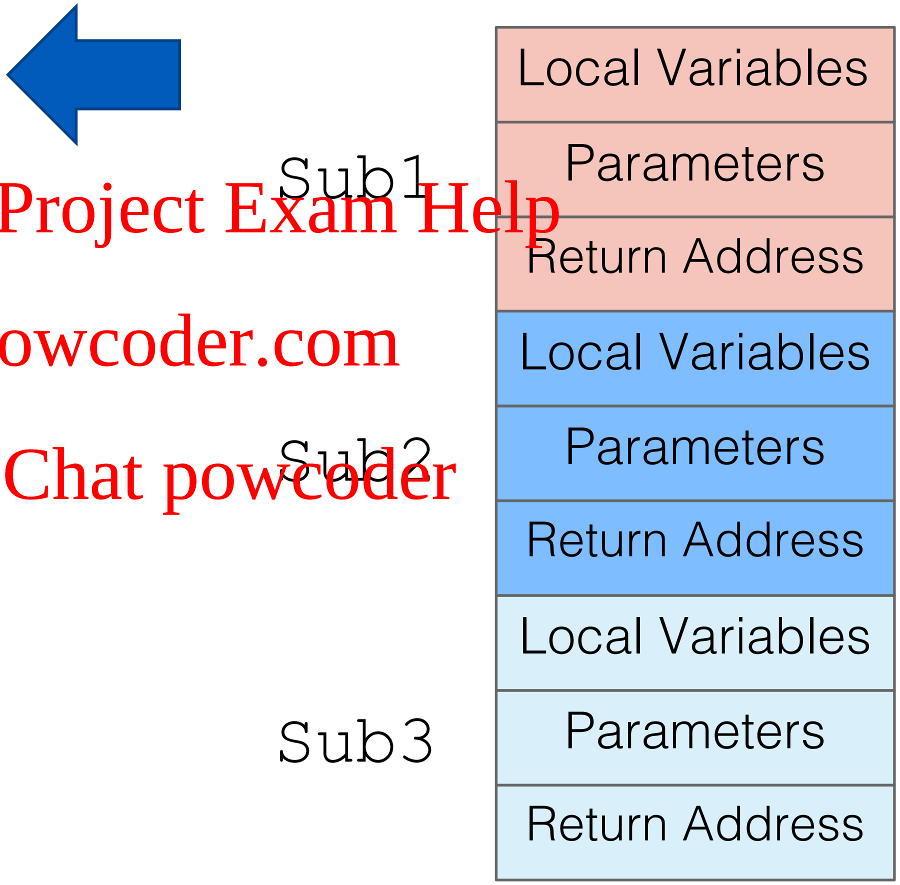
Parameters

Return Address

call to sub3()

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



Sub1

Local Variables
Parameters
Return Address

Sub2

Local Variables
Parameters
Return Address

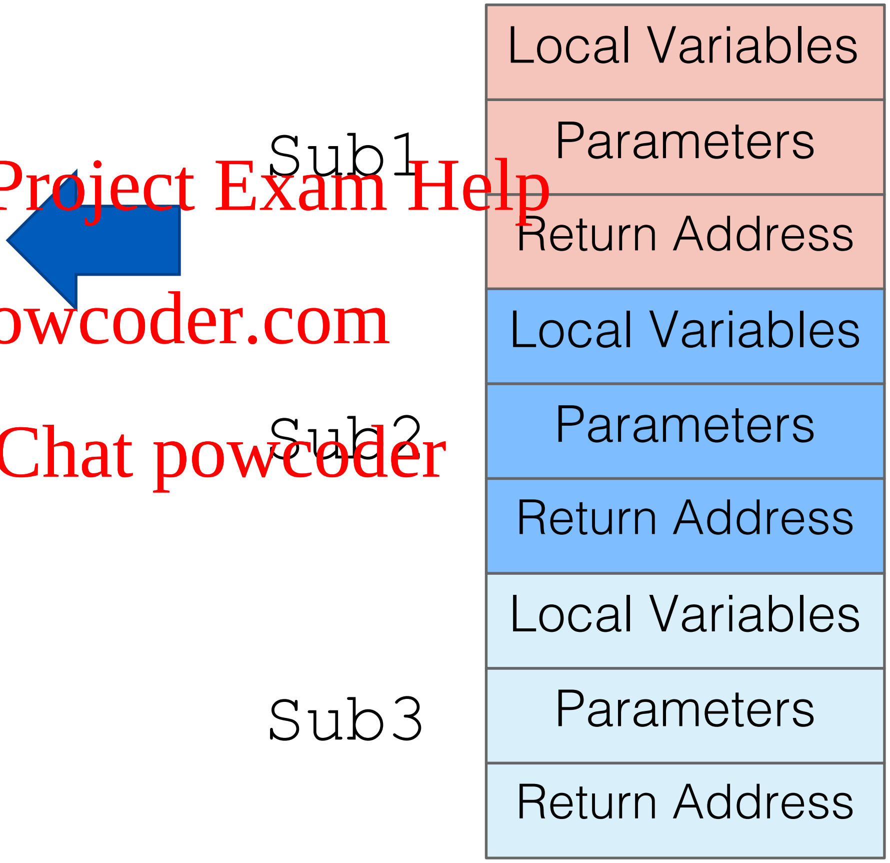
Sub3

Local Variables
Parameters
Return Address

`call to sub3()`

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```

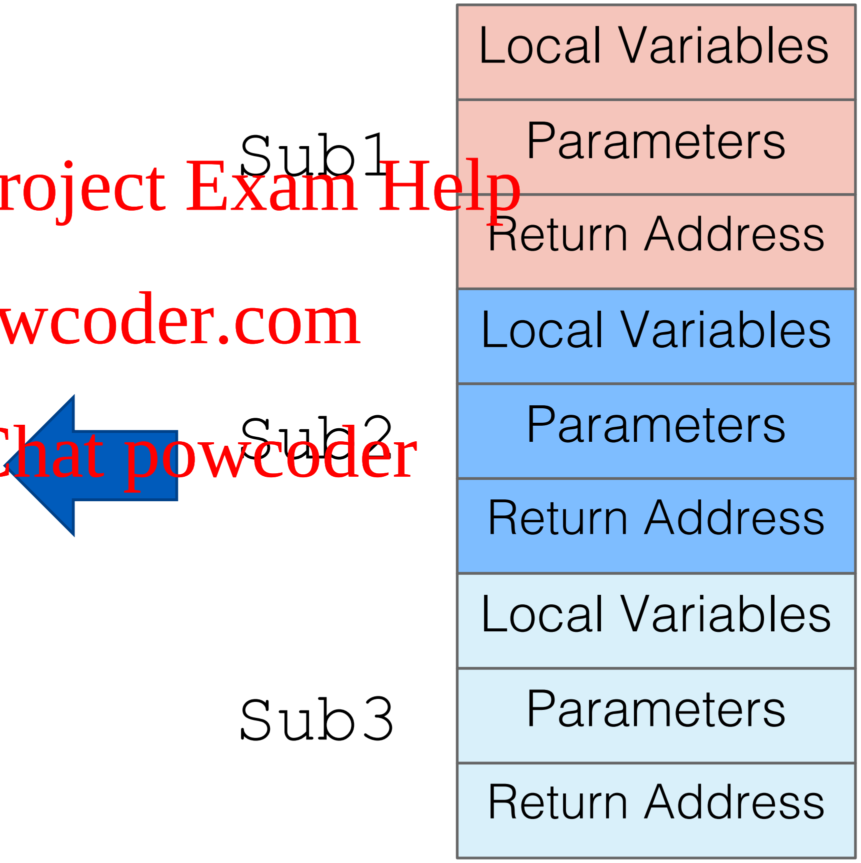


Sub1	Local Variables
	Parameters
	Return Address
Sub2	Local Variables
	Parameters
	Return Address
Sub3	Local Variables
	Parameters
	Return Address

`call to sub3()`

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



Sub1
Local Variables
Parameters
Return Address

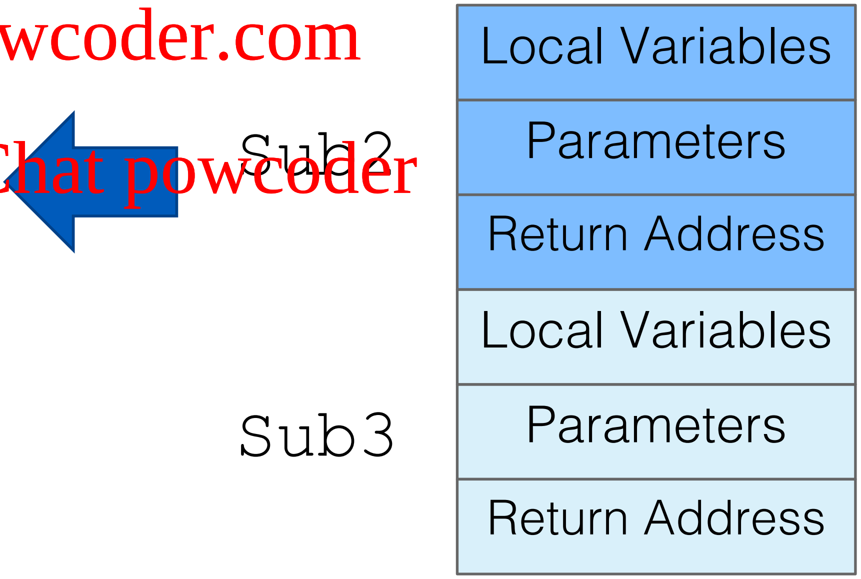
Sub2
Local Variables
Parameters
Return Address

Sub3
Local Variables
Parameters
Return Address

`call to sub3()`

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



Sub2

Local Variables
Parameters
Return Address

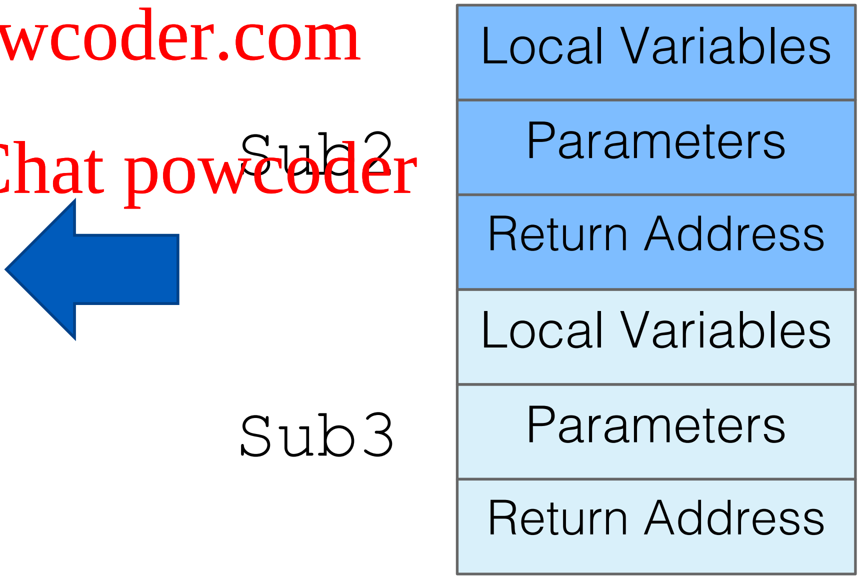
Sub3

Local Variables
Parameters
Return Address

```
call to sub3()
```

Example: Activation Records stack for Simple Functions

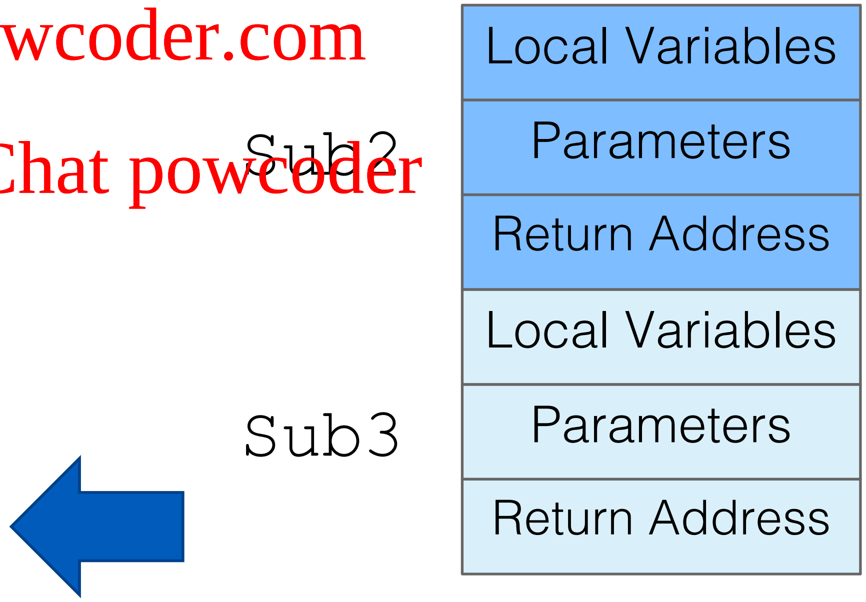
```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



```
call to sub3()
```

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```

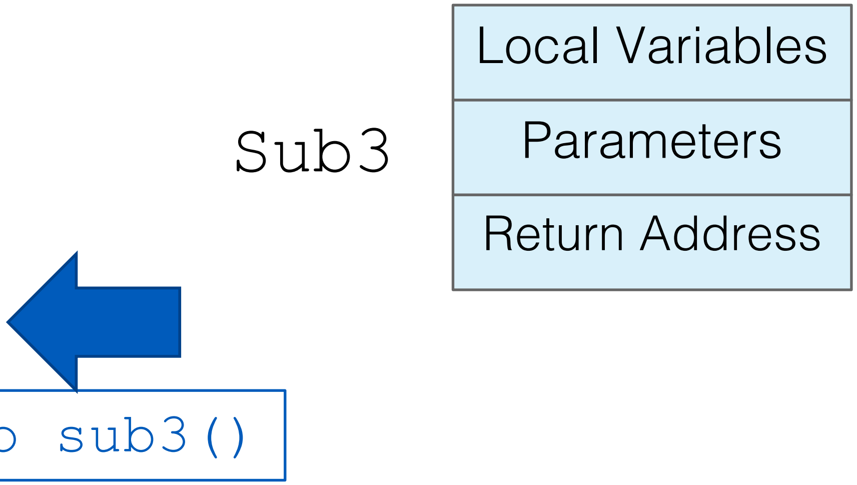


Sub2	Local Variables
	Parameters
	Return Address
Sub3	Local Variables
	Parameters
	Return Address

`call to sub3()`

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



Sub3

Local Variables

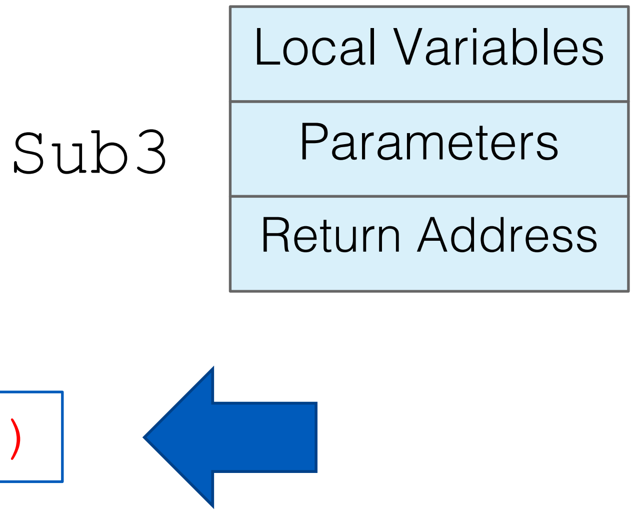
Parameters

Return Address

call to sub3()

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```



Sub3

Local Variables

Parameters

Return Address

call to sub3()

Example: Activation Records stack for Simple Functions

```
function sub1(…){
…
};
function sub2(…){
…
Sub1(…)
};
function sub3(…){
…
Sub2(…)
};
```

call to sub3()

TopHat Q6-Q10

Local variables

- Variables whose scope is usually the body of the subprogram in which they are defined

- They can be static or stack-dynamic

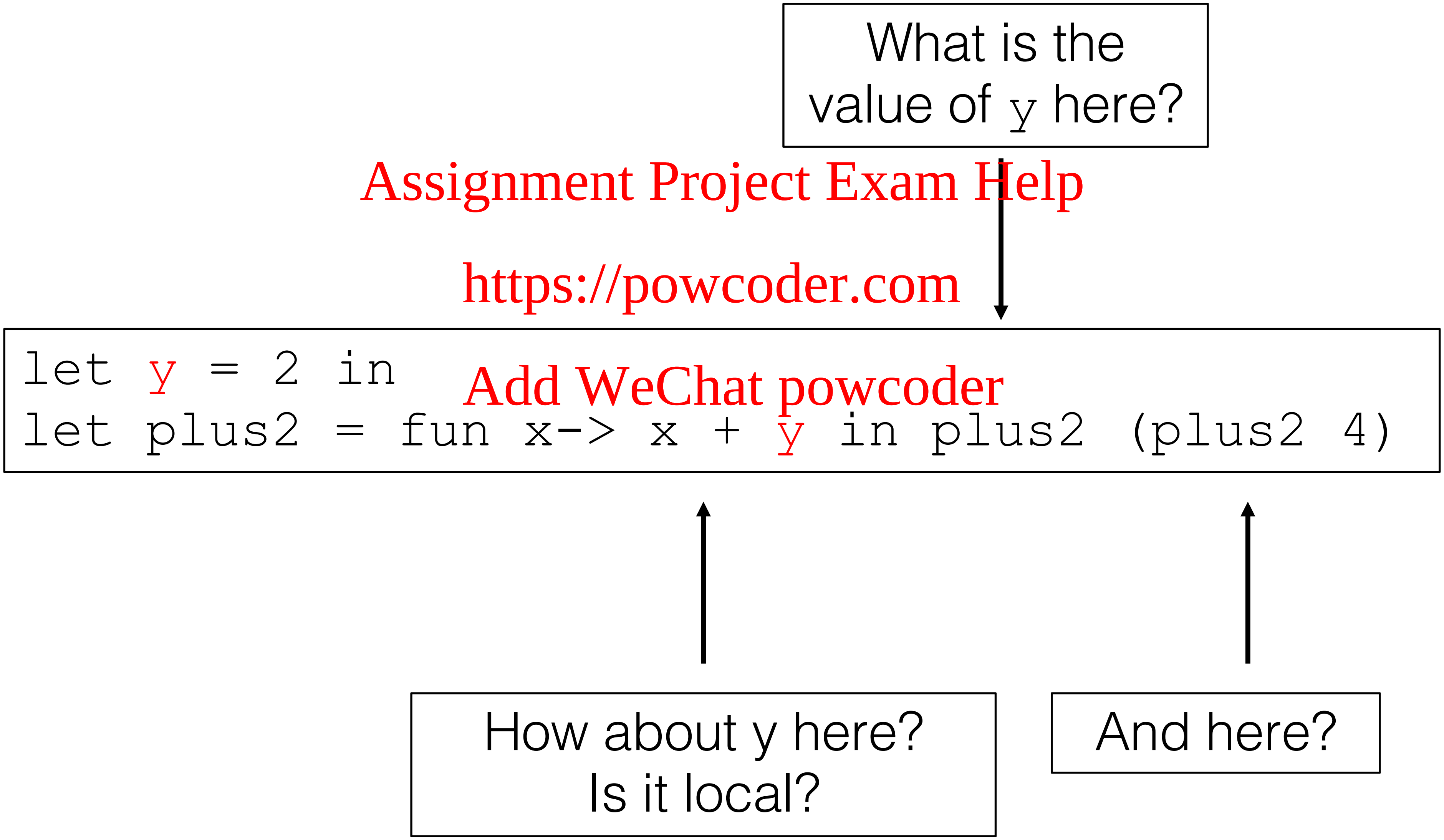
- A subprogram can use variables that are heap-dynamic but these usually do not count as local.

Local variables

```
let plus2 = fun x ->
             let y = 2 in x + y
```

Here y is a local variable to the function `plus2`

Local variables?



What is the value of y here?

```
let y = 2 in
let plus2 = fun x-> x + y in plus2 (plus2 4)
```

How about y here? Is it local?

And here?

Local variables?

input

```
push x
push 3
bind
    fun addX arg
    push x
    push arg
    add
    return
    funEnd
push x
push 5
bind
push a
push 3
bind
push addX
push a
call
quit
```

What is the value of x here?

Subprograms with stack-dynamic variables

Implementing Subprograms with Stack-Dynamic Local Variables: Activation Record

- The activation record format is static, but its size may be dynamic to deal with stack-dynamic variables.
- We need a dynamic link pointing to the base of the instance of the activation record of the caller – this will help us deallocate the activation record instance when it has dynamic size.
- The collection of dynamic links in the stack at a given time is called the dynamic chain, or call chain

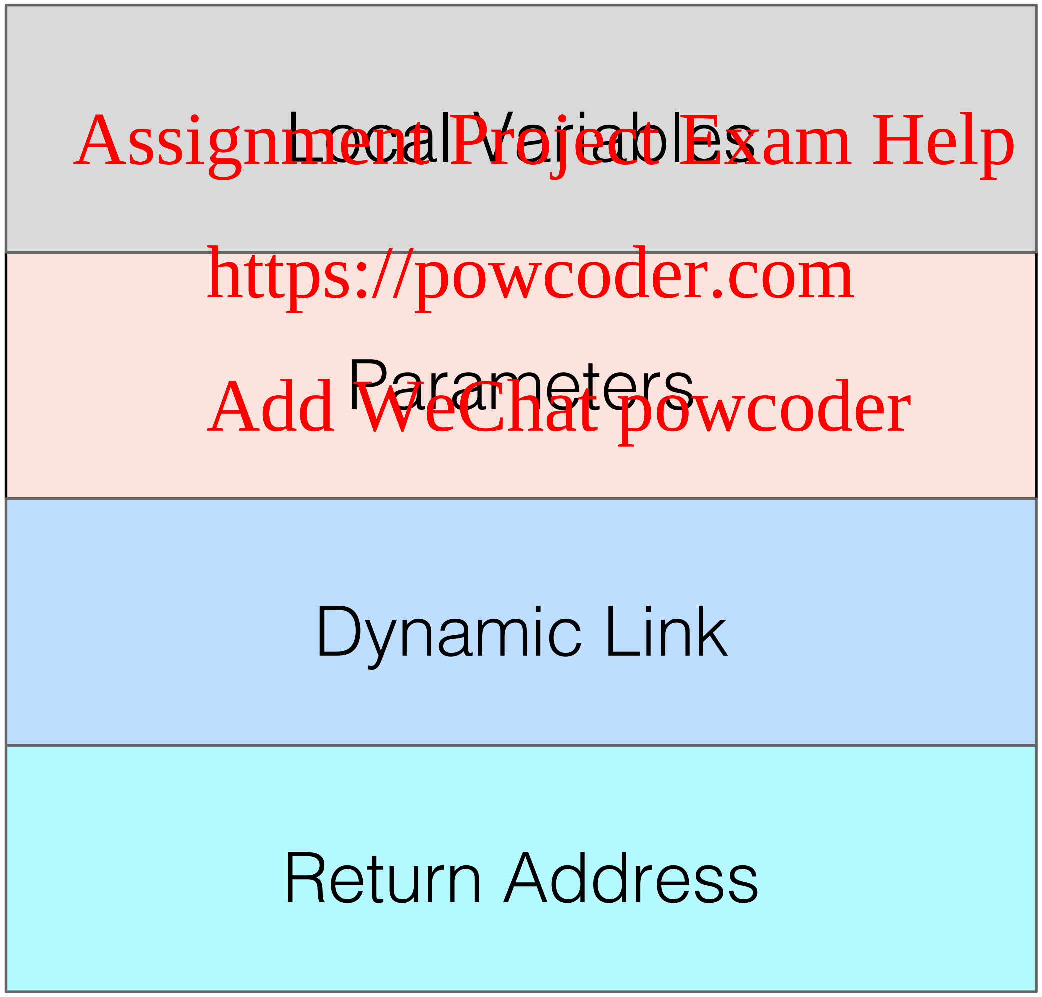
Function Calls for General Programs

- General semantics of calls to a subprogram
  - In mode and inout mode parameters must be provided
  - Stack-dynamic allocation of local variables
  - Save the execution status of the calling program
  - Transfer of control to the subprogram and arrange for the return
  - If subprogram nesting is supported, access to nonlocal variables must be arranged through a static link (we will see this next time).

Function Returns for General Programs

- General semantics of subprogram returns:
  - Out mode and inout mode parameters must have their values returned
  - Deallocation of stack-dynamic local variables
  - Restore the execution status
  - Return control to the caller

Typical Activation Record for a Language with Stack-Dynamic Local Variables



Local Variables

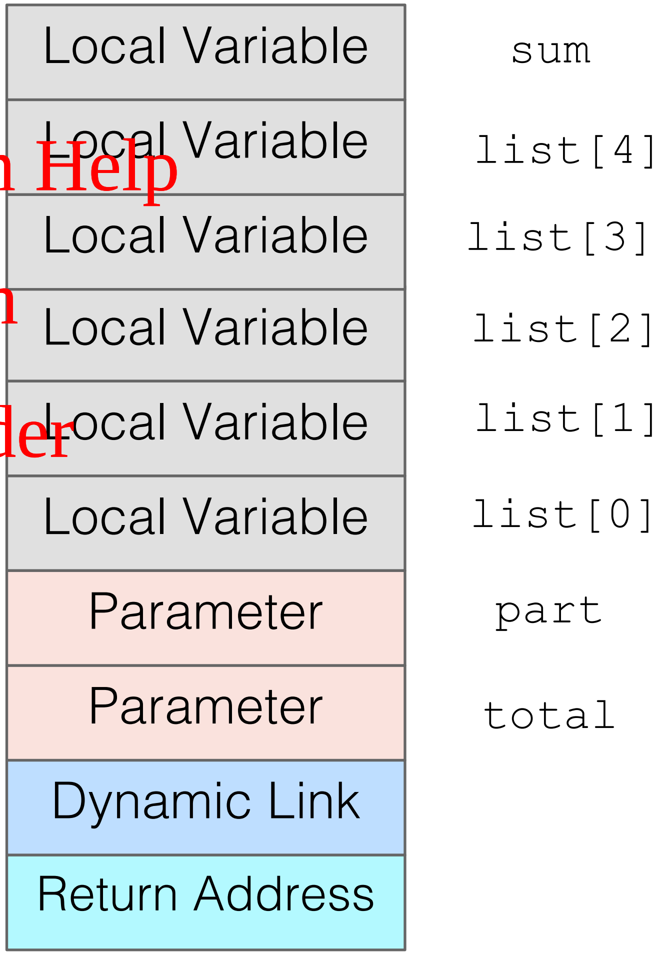
Parameters

Dynamic Link

Return Address

Example in C

```
void sub(float total, int part)
{
   int list[5];
   float sum;
   …
}
```



Local Variable	sum
Local Variable	list[4]
Local Variable	list[3]
Local Variable	list[2]
Local Variable	list[1]
Local Variable	list[0]
Parameter	part
Parameter	total
Dynamic Link	
Return Address	

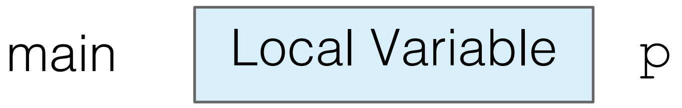
Example

```
void fun1(float r)
{
     int s, t;
     ...
     fun2(s);
     ...
}
void fun2(int x) {
     int y;
     ...
     fun3(y);
     ...
}
void fun3(int q) {
     ...
}
void main() {
     float p;
     ...
     fun1(p);
     ...
}
```

p

Example

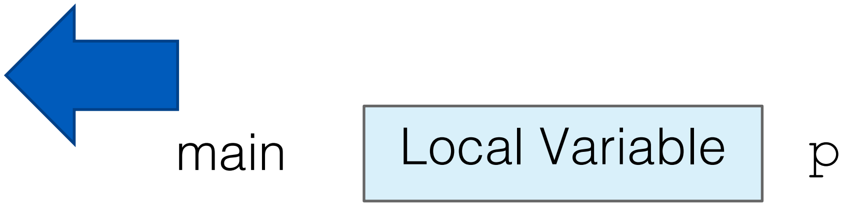
```
void fun1(float r)
{
    int s, t;
    ...
    fun2(s);
    ...
}
void fun2(int x) {
    int y;
    ...
    fun3(y);
    ...
}
void fun3(int q) {
    ...
}
void main() {
    float p;
    ...
    fun1(p);
    ...
}
```



main Local Variable p

Example

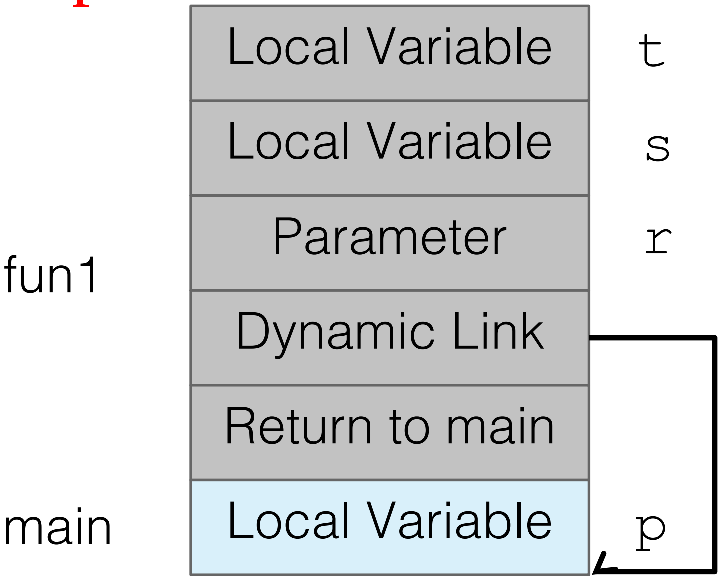
```
void fun1(float r)
{
      int s, t;
      ...
      fun2(s);
      ...
}
void fun2(int x) {
      int y;
      ...
      fun3(y);
      ...
}
void fun3(int q) {
      ...
}
void main() {
      float p;
      ...
      fun1(p);
      ...
}
```



main | Local Variable | p

Example

```
void fun1(float r)
{
        int s, t;
        ...
        fun2(s);
        ...
}
void fun2(int x) {
        int y;
        ...
        fun3(y);
        ...
}
void fun3(int q) {
        ...
}
void main() {
        float p;
        ...
        fun1(p);
        ...
}
```



Local Variable t

Local Variable s

Parameter r

fun1

Dynamic Link

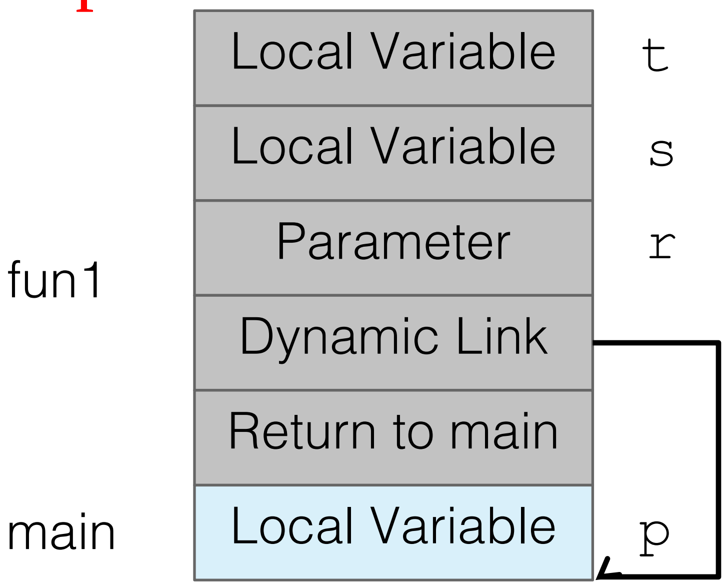
Return to main

main

Local Variable p

Example

```
void fun1(float r)
{
      int s, t;
      ...
      fun2(s);
      ...
}
void fun2(int x) {
      int y;
      ...
      fun3(y);
      ...
}
void fun3(int q) {
      ...
}
void main() {
      float p;
      ...
      fun1(p);
      ...
}
```



Local Variable t

Local Variable s

Parameter r

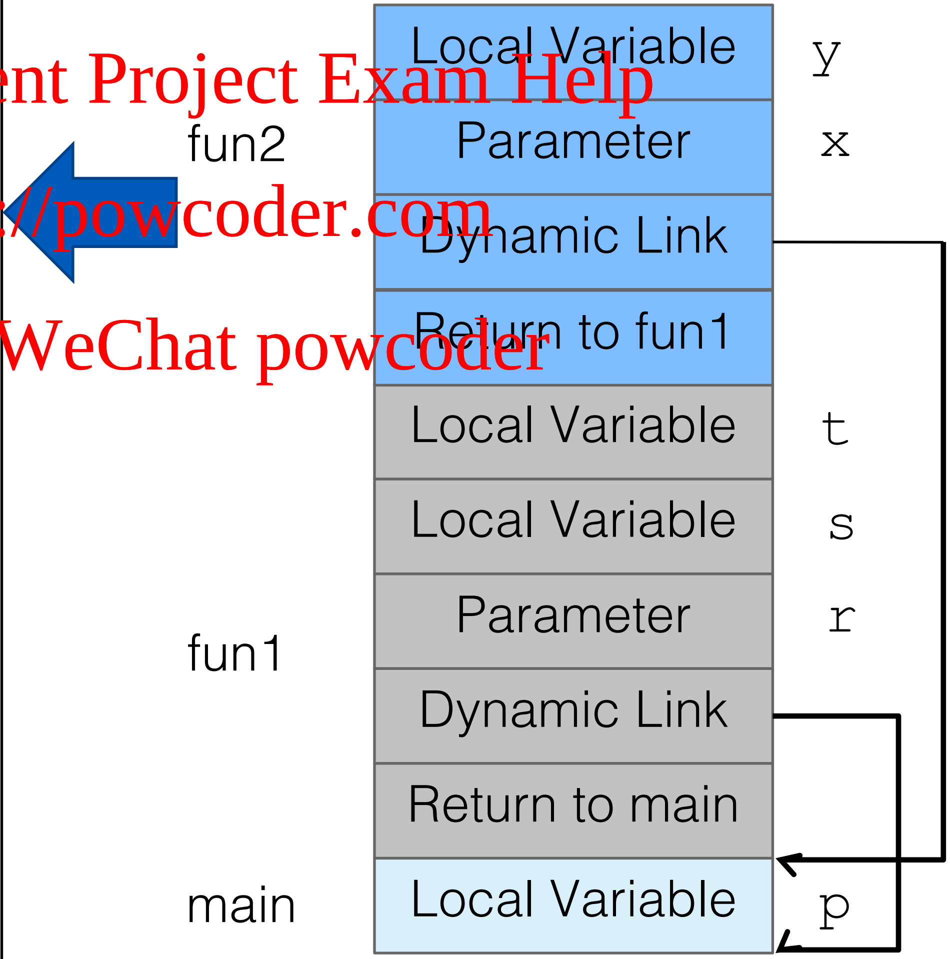
fun1 Dynamic Link

Return to main

main Local Variable p

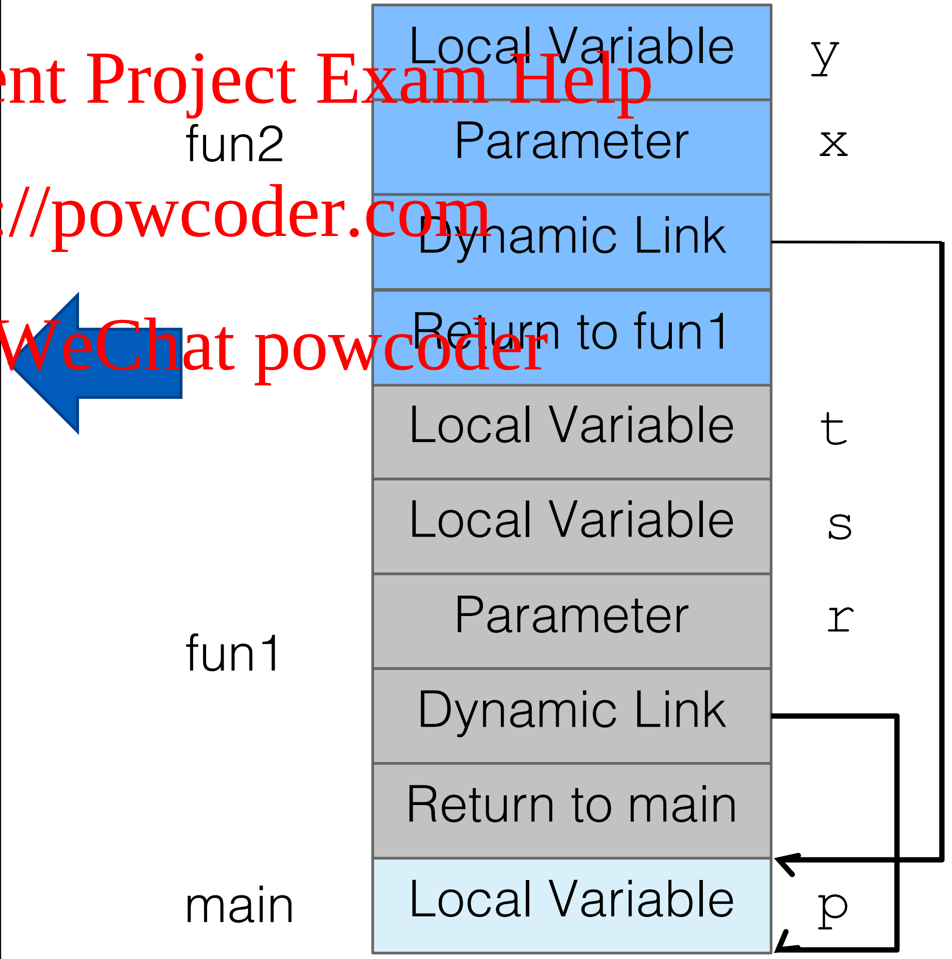
Example

```
void fun1(float r)
{
      int s, t;
      ...
      fun2(s);
      ...
}
void fun2(int x) {
      int y;
      ...
      fun3(y);
      ...
}
void fun3(int q) {
      ...
}
void main() {
      float p;
      ...
      fun1(p);
      ...
}
```



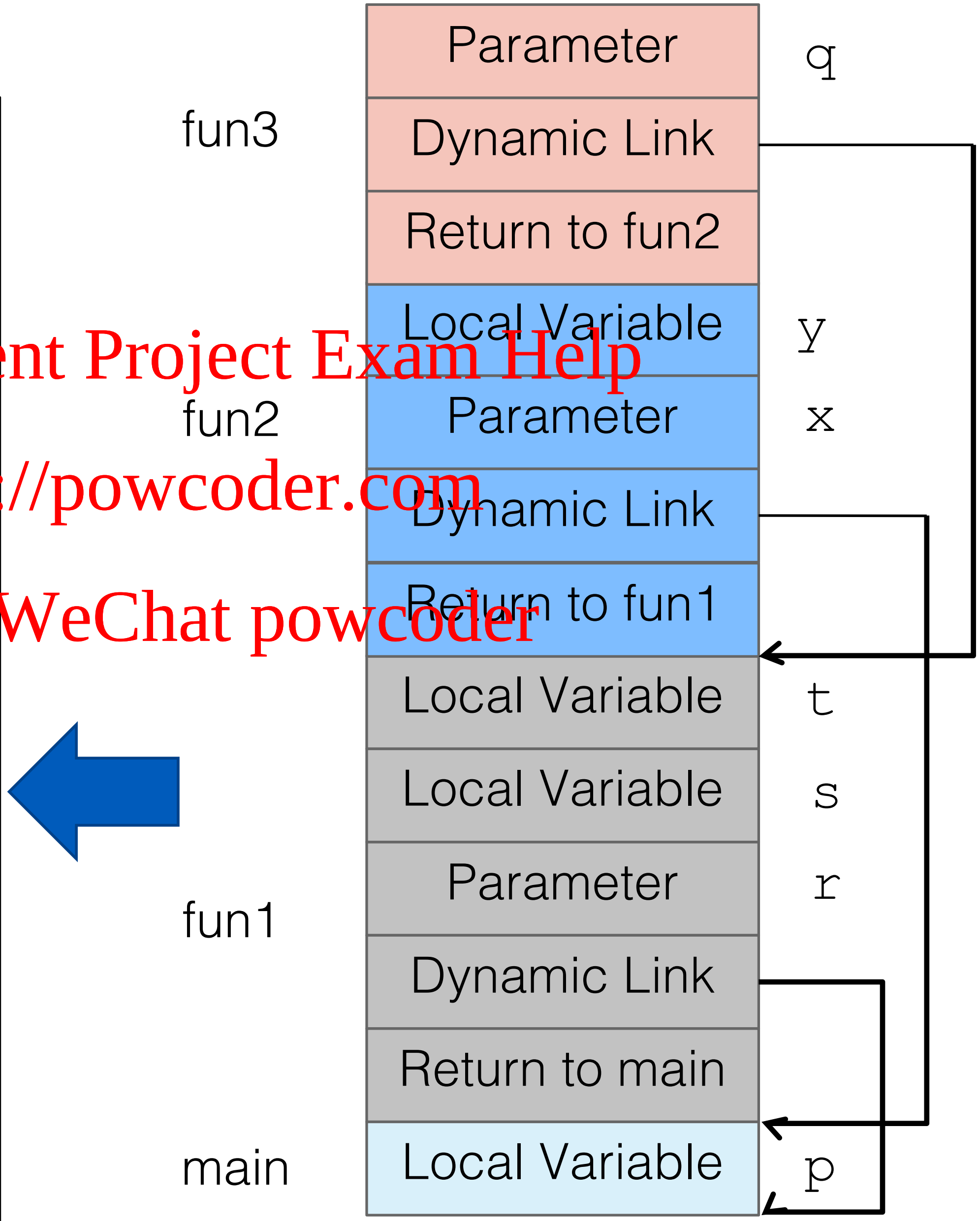
Example

```
void fun1(float r)
{
      int s, t;
      ...
      fun2(s);
      ...
}
void fun2(int x) {
      int y;
      ...
      fun3(y);
      ...
}
void fun3(int q) {
      ...
}
void main() {
      float p;
      ...
      fun1(p);
      ...
}
```



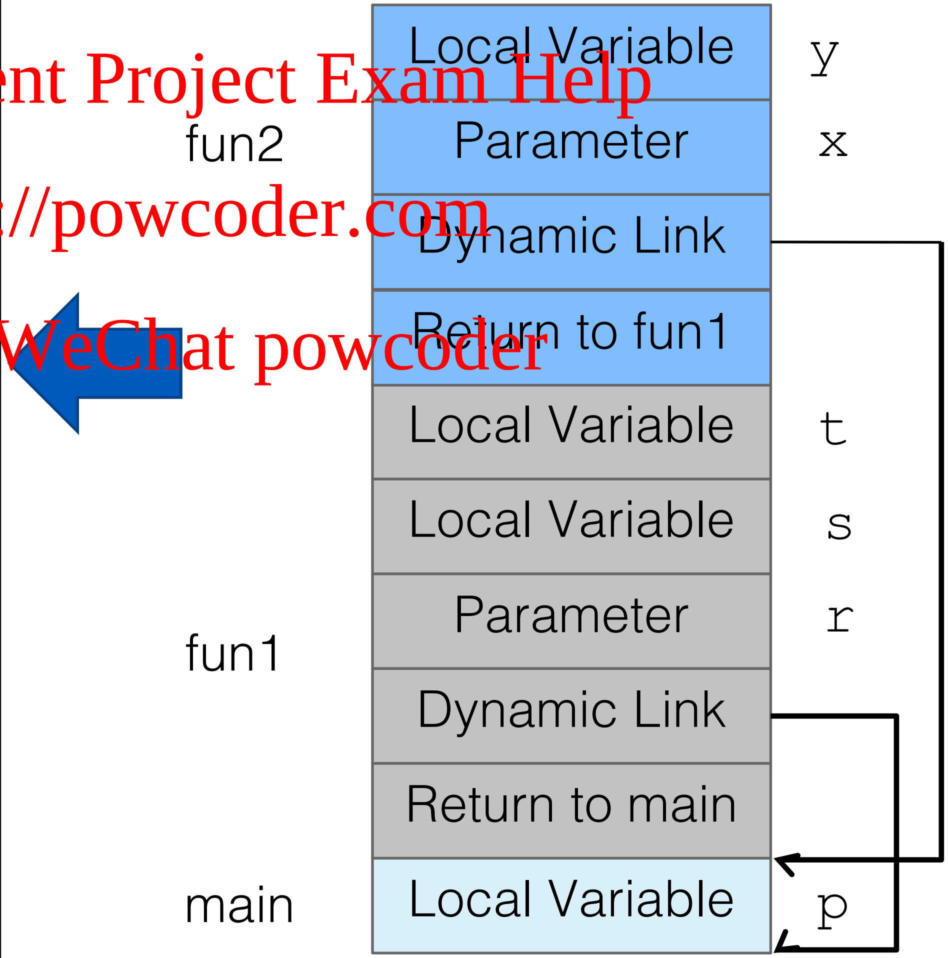
Example

```
void fun1(float r)
{
      int s, t;
      ...
      fun2(s);
      ...
}
void fun2(int x) {
      int y;
      ...
      fun3(y);
      ...
}
void fun3(int q) {
      ...
}
void main() {
      float p;
      ...
      fun1(p);
      ...
}
```



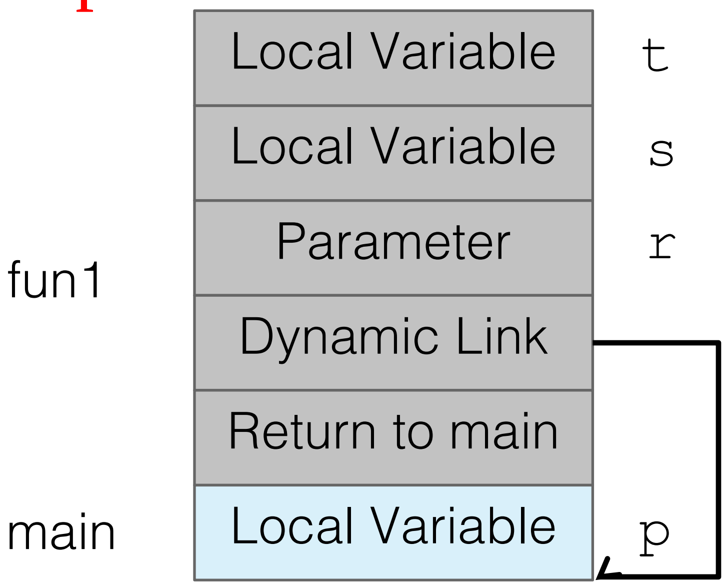
Example

```
void fun1(float r)
{
        int s, t;
        ...
        fun2(s);
        ...
}
void fun2(int x) {
        int y;
        ...
        fun3(y);
        ...
}
void fun3(int q) {
        ...
}
void main() {
        float p;
        ...
        fun1(p);
        ...
}
```



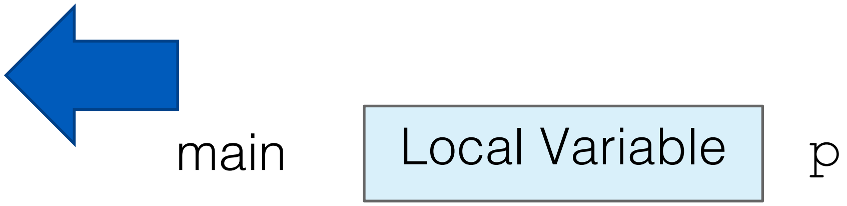
Example

```
void fun1(float r)
{
      int s, t;
      ...
      fun2(s);
      ...
}
void fun2(int x) {
      int y;
      ...
      fun3(y);
      ...
}
void fun3(int q) {
      ...
}
void main() {
      float p;
      ...
      fun1(p);
      ...
}
```



Example

```
void fun1(float r)
{
     int s, t;
     ...
     fun2(s);
     ...
}
void fun2(int x) {
     int y;
     ...
     fun3(y);
     ...
}
void fun3(int q) {
     ...
}
void main() {
     float p;
     ...
     fun1(p);
     ...
}
```



main

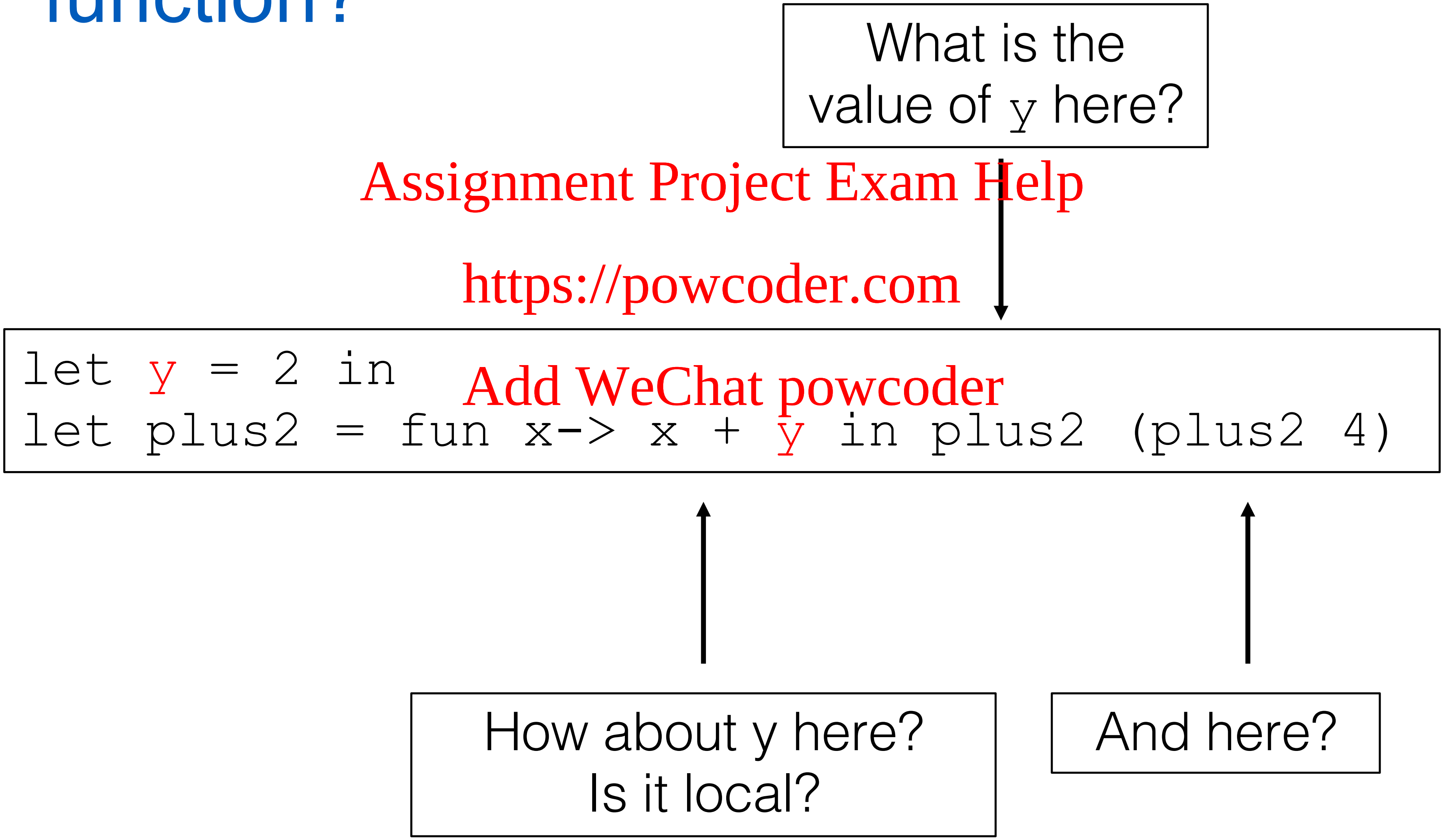
Local Variable

p

TopHat Q11-Q12

Passing functions as arguments

Local variables when passing a function?



What is the value of y here?

```
let y = 2 in
let plus2 = fun x-> x + y in plus2 (plus2 4)
```

How about y here?
Is it local?

And here?

Local variables?

input
push x
push 3
bind
fun addX arg
push x
push arg
add
return
funEnd
push x
push 5
bind
push a
push 3
bind
push addX
push a
call
quit

What is the value of x here?

Closures

- A closure is a pair consisting of the function code p and its referencing environment m:

$$(p, m)$$

- It is similar to a configuration but where p is the code of a function.
- The referencing environment is needed to provide values to the variables when the function (subprogram) can be called from an arbitrary place in the program
- Closures are needed if a (function (subprogram) can access variables in nesting scopes and it can be called from anywhere
- A static-scope language that does not permit nested subprograms doesn’t need closures

Example of closure

```
let y = 2 in
let plus2 = fun x-> x + y in plus2 (plus2 4)
```

What is the closure that will be created here?

```
(fun x-> x + y / y=2)
```

Local variables?

input

```
push x
push 3
bind
    fun addX arg
    push x
    push arg
    add
    return
    funEnd
push x
push 5
bind
push a
push 3
bind
push addX
push a
call
quit
```

What is the closure that will be created here?

`(fun arg -> push x; push arg; add; return / x=3)`

Closures vs Scope

```
let y = 2 in
let plus2 = fun x-> x + y in plus2 (plus2 4)
```

We said that we need a closure to find the value of y.

```
(fun x-> x + y in plus2 (plus2 4) / y=2)
```

What are we assuming here about the scope of y?

Tips for interpreter part 3:Language for basic stack manipulations with local variables definitions and functions

- Since in our interpreter we want to pass functions as arguments to other functions, when you encounter a function declaration you should construct a closure:

  1) the name of the function

  2) the name of the formal parameter

  3) the code in the function body

  4) the current environment

  (Notice that 3 and 4 constitute the function code.)

```
type value = …
|CLOSURE of(name * name *(command list) * (name*value)list)
```

Tips for interpreter part 3:Language for basic stack manipulations with local variables definitions and functions

How do we call a function?

```
push fun_name
push arg
call
```

1. Check if `fun_name` is bound to a closure.
2. Check if `arg` is a value or a name bound to a value (including closures for functions).
3. If both yes, then we can execute the body of the function – otherwise error.
4. We have to execute it in the environment we have in the closure with an additional binding between the formal parameter and the value of the actual (`arg`).
5. We also need to execute it using a new stack

Tips for interpreter part 3:Language for basic stack manipulations with local variables definitions and functions

Preparing for function evaluation (step 4)

```
push fun_name
push arg
call
```

We have to execute the code in the environment we have in the closure with an additional binding between the formal parameter and the value of the actual (`arg`).

We have to use the environment in the closure.

Tips for interpreter part 3:Language for basic stack manipulations with local variables definitions and functions

What to do when the function terminates?

```
…
…
funEnd
```

1. Restore the previous environment
2. Restore the previous stack
3. Push on the restored stack the last element on the function stack
4. Resume the execution from after the call instruction

Tips for interpreter part 3:Language for basic stack manipulations with local variables definitions and functions

What to do when the function terminates with a return?

```
…
…
return
```

1. Immediately stop the execution

2. Restore the previous environment

3. Restore the previous stack
4. Push on the restored stack the last element on the function stack
5. Resume the execution from after the call instruction