CHAPTER 10

Implementing Subprograms

Prepared By:

Lawas, Denzel John

Stewart, Christian Anthony

Woogue, Ivan Ric

Topics

The General Semantics of Calls and Returns

The subprogram call and return operations of a language are together called its **subprogram linkage**

The General Semantics of Calls and Returns

General semantics of calls to a subprogram

- Parameter passing methods
- Stack-dynamic allocation of local variables
- Save the execution status of calling program
- Transfer of control and arrange for the return
- If subprogram nesting is supported, access to nonlocal variables must be arranged

The General Semantics of Calls and Returns

General semantics of subprogram returns:

- In mode and inout mode parameters must have their values returned
- Deallocation of stack-dynamic locals
- Restore the execution status
- Return control to the caller

Implementing “Simple” Subprograms

Call Semantics:

- Save the execution status of the caller
- Pass the parameters
- Pass the return address to the called
- Transfer control to the called

Implementing “Simple” Subprograms

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
- If it is a function, move the functional value to a place the caller can get it
- Restore the execution status of the caller
- Transfer control back to the caller

Implementing “Simple” Subprograms

Required Storage:

- Status information
- Parameters
- Return address
- Return value for functions
- Temporaries

Distribution of Call and Return Actions

CALL ACTIONS

Saving the execution status	either caller or called
Pass the parameter	caller
Pass the return address to the called	
Transfer control to the called	

Distribution of Call and Return Actions

RETURN ACTIONS

If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to or made available to the corresponding actual parameters	called
If the subprogram is a function, the functional value is moved to a place accessible to the caller	
The execution status of the caller is restored	either caller or called
Control is transferred back to the caller	called

Distribution of Call and Return Actions

In general, the linkage actions of the called can occur at two different times, either at the beginning of its execution or at the end.

These are sometimes called the **prologue** and **epilogue** of the subprogram linkage. In the case of a simple subprogram, all of the linkage actions of the one being called occur at the end of its execution, so there is no need for a prologue.

Implementing “Simple” Subprogram

Two separate parts:

- the actual code (constant)
- the non-code part (local variables and data that can change)

Implementing “Simple” Subprogram

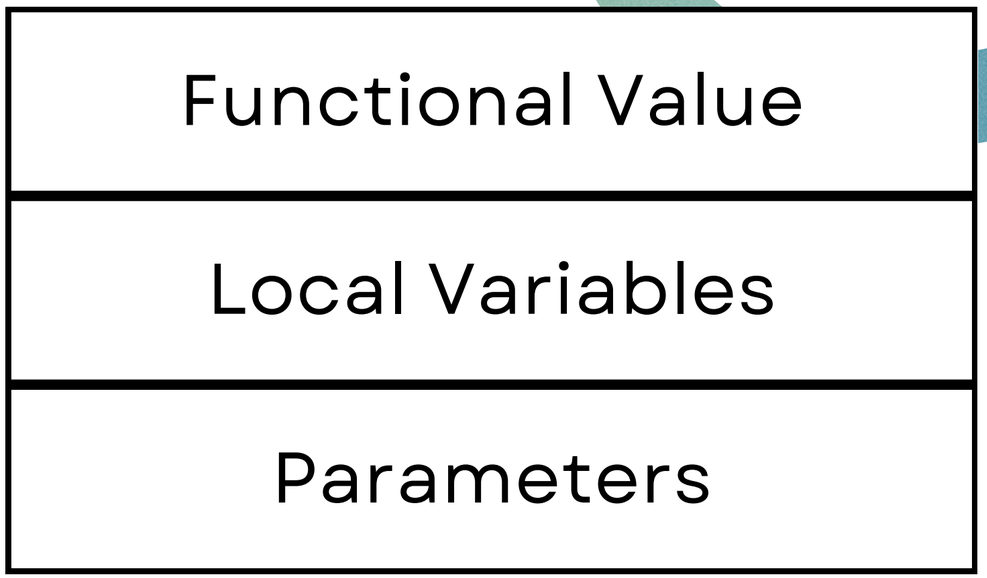
The format, or layout, of the non-code part of an executing subprogram is called an ***activation record***

An ***activation record instance*** is a concrete example of an activation record

---

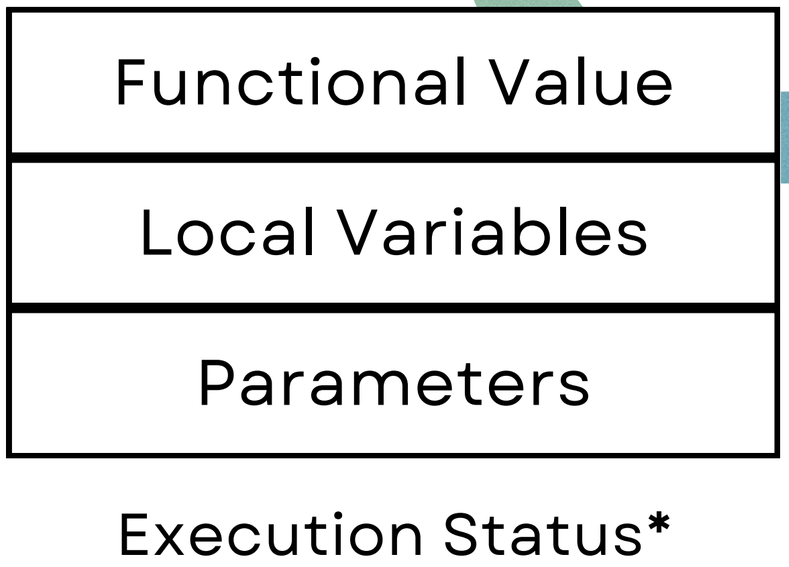
An Activation Record for “Simple” Subprograms

- Do not support recursion
- Only one active version of a subprogram at a time
- Only one instance of the activation record



An Activation Record for “Simple” Subprograms

* The saved execution status of the caller is omitted here and in the remainder of this chapter because it is simple and not relevant to the discussion



Example C Program

```
int simple() {

  static int count = 0;
  count++;

  return count;
}
```

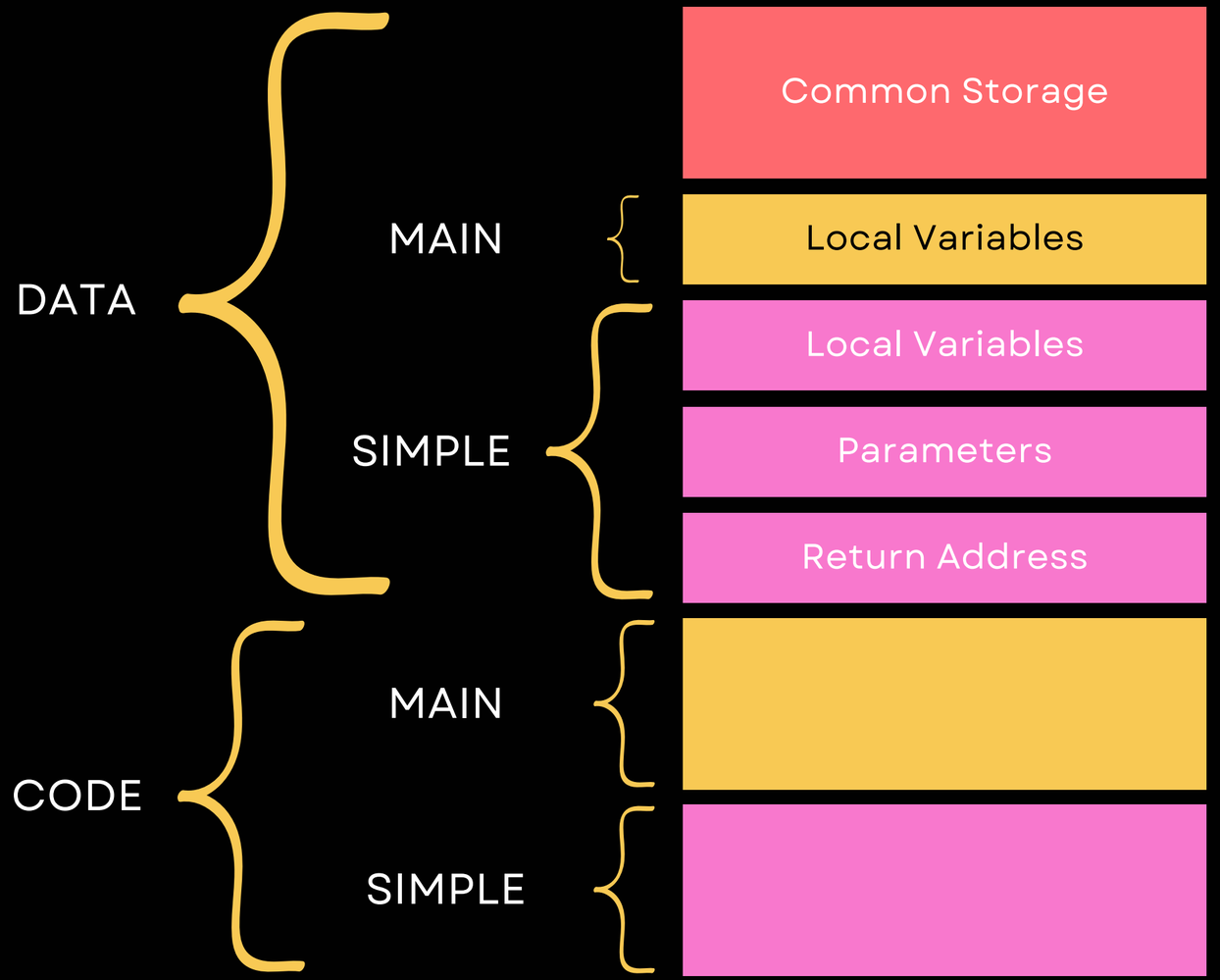
```
int main() {

  printf(" %d \n", simple() ); // prints 1
  printf(" %d \n", simple() ); // prints 2
  printf(" %d \n", simple() ); // prints 3

  return 0;
}
```

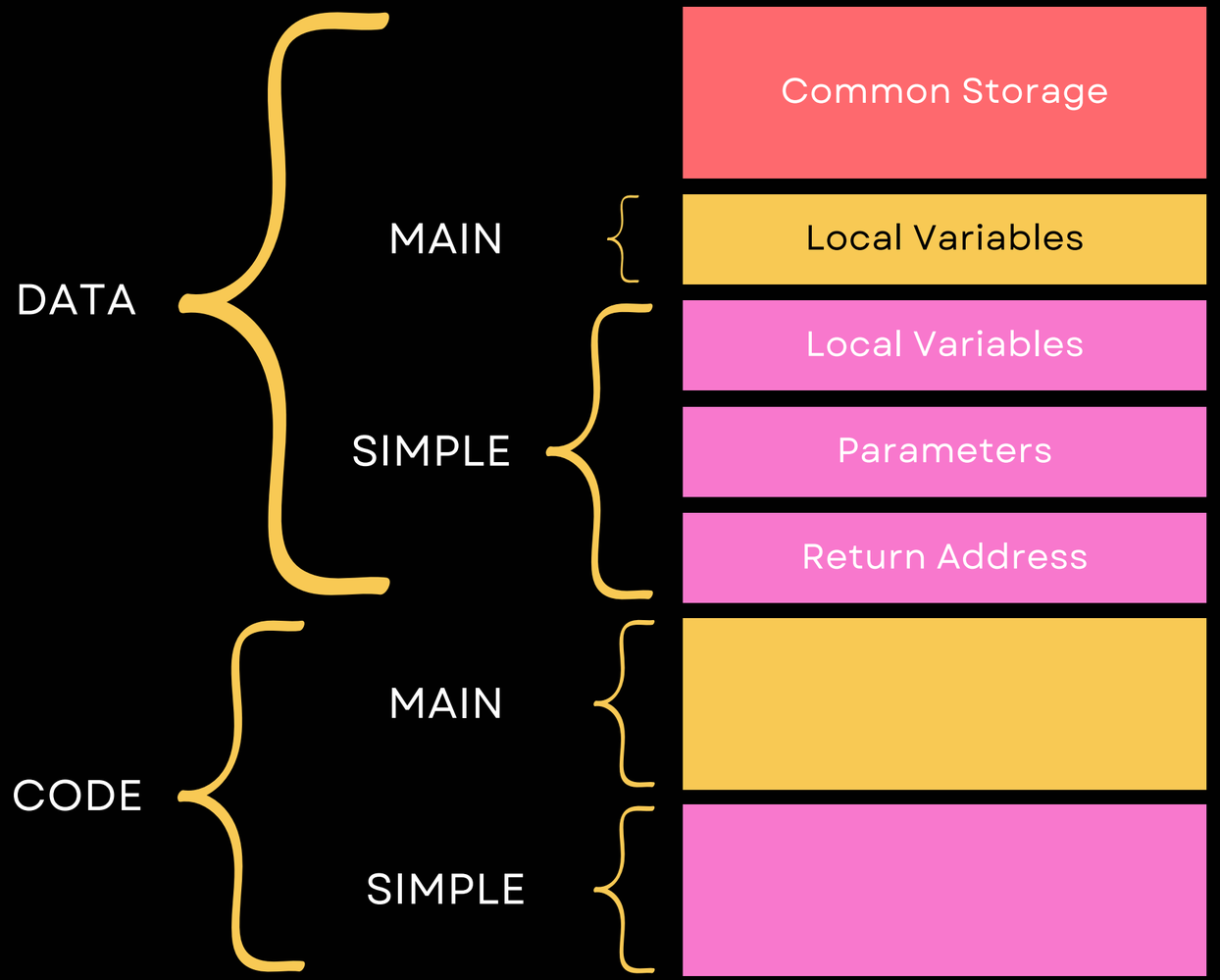
Code and Activation Records of a Program with “Simple” Subprogram

Note that there are no nested subprograms (only 1 subprogram) and all local variables are static (declared using the static keyword in C)



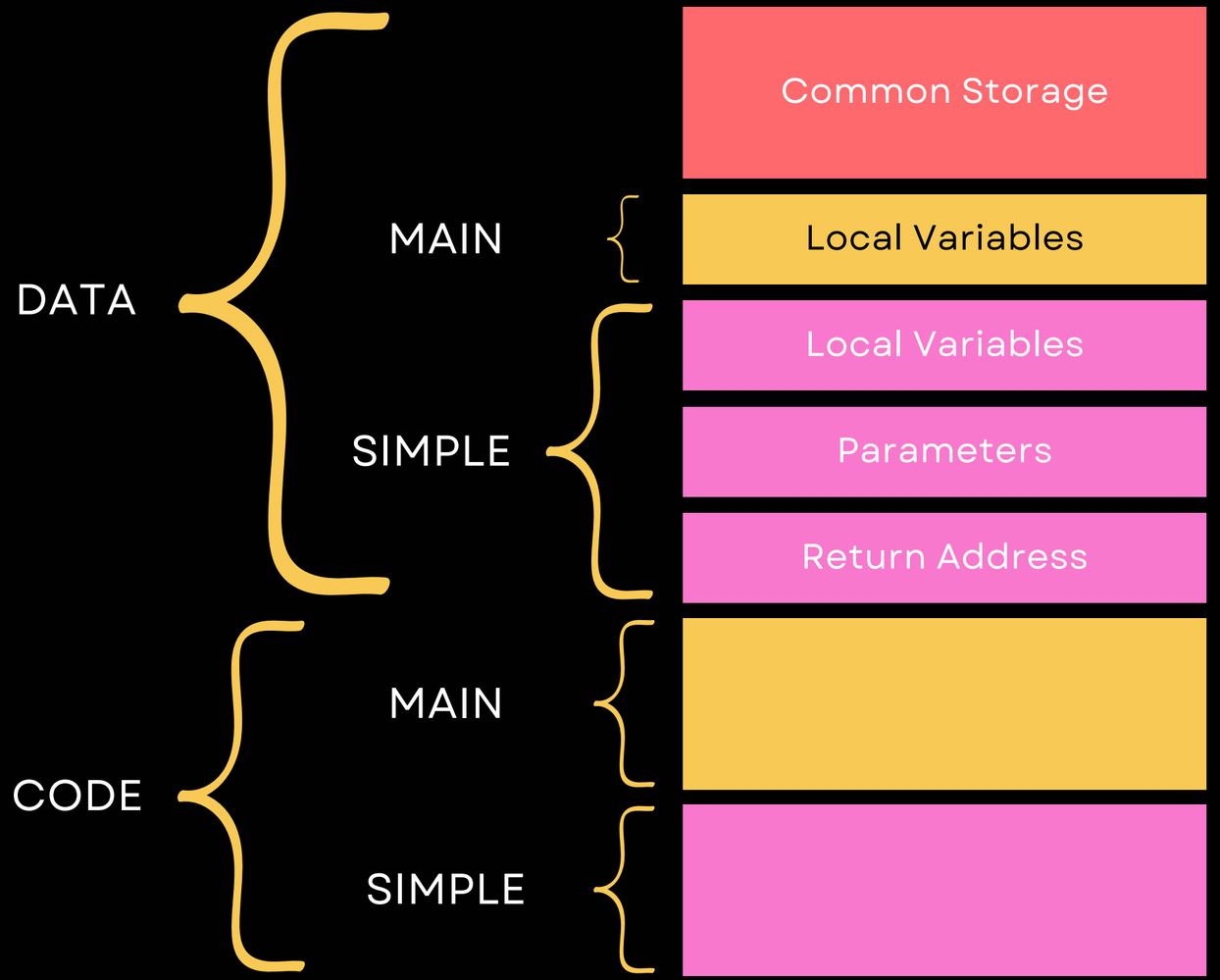
Code and Activation Records of a Program with “Simple” Subprogram

- not done entirely by compiler



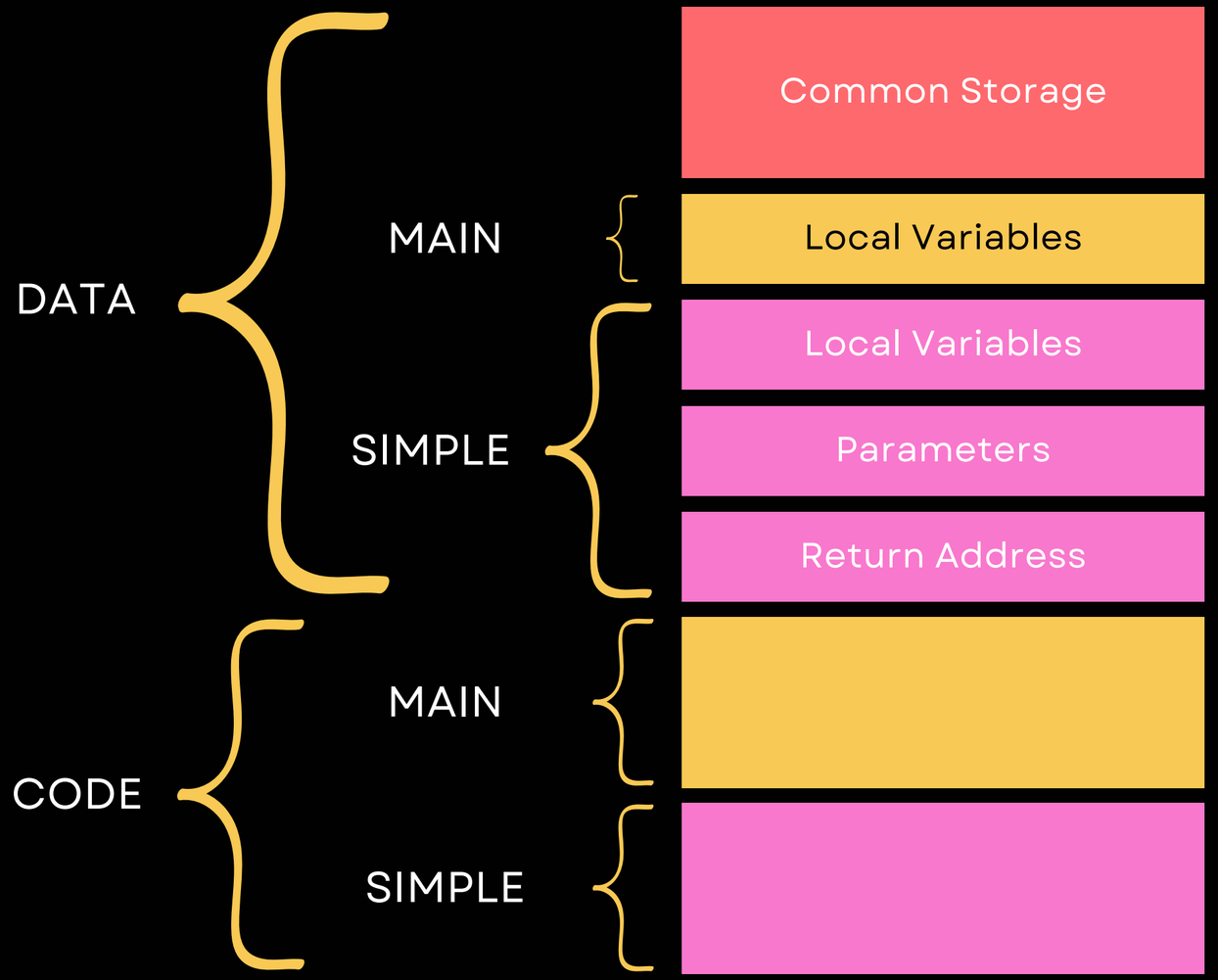
Code and Activation Records of a Program with “Simple” Subprogram

- not done entirely by compiler
- if a language allows independent compilation, program units may be compiled at different times



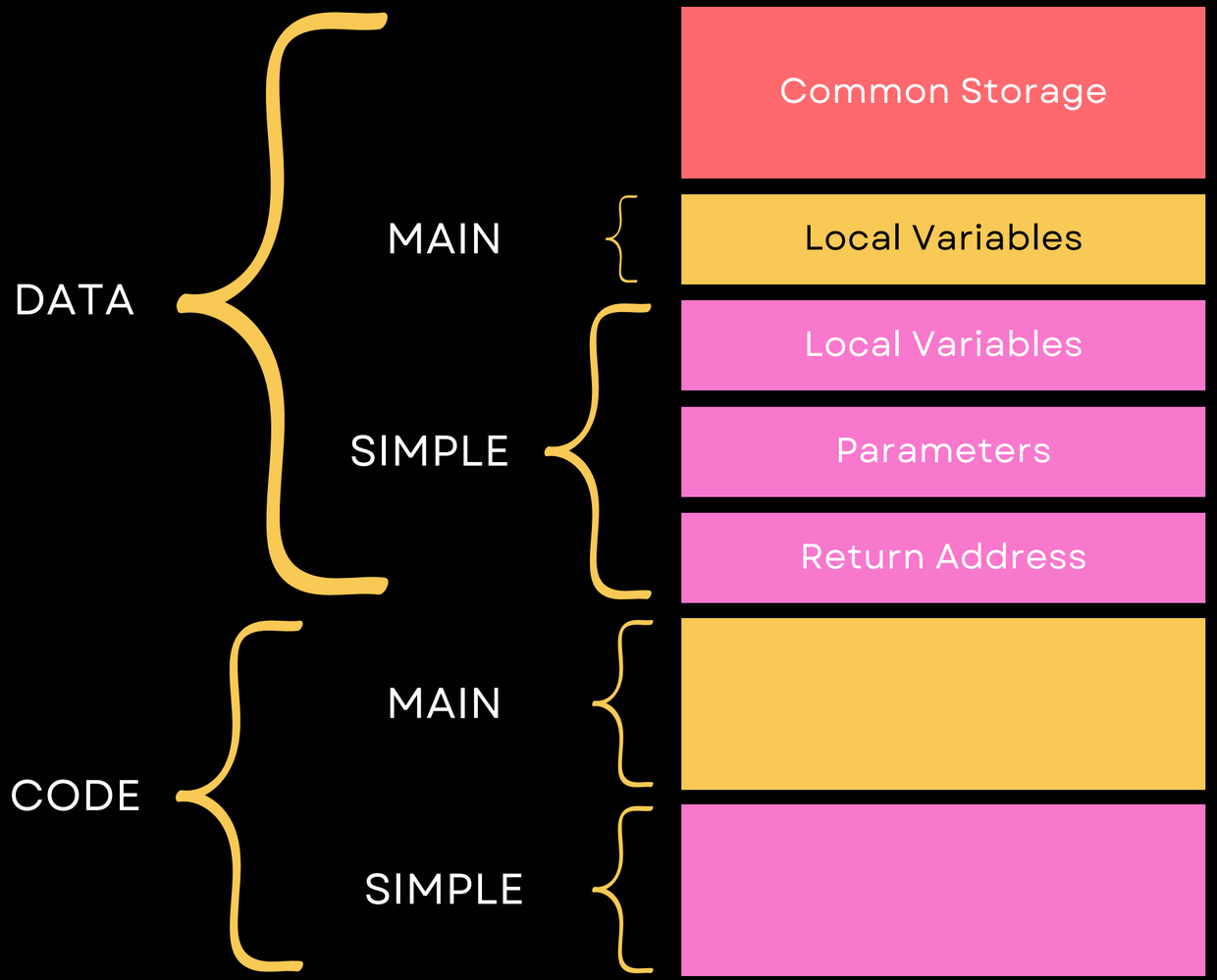
Code and Activation Records of a Program with “Simple” Subprogram

- not done entirely by compiler
- if a language allows independent compilation, program units may be compiled at different times
- during compilation, machine code and a reference list to external subprograms are written to a file



Code and Activation Records of a Program with “Simple” Subprogram

- not done entirely by compiler
- if a language allows independent compilation, program units may be compiled at different times
- during compilation, machine code and a reference list to external subprograms are written to a file
- ***LINKER*** - puts together the executable program



Code and Activation Records of a Program with “Simple” Subprogram

LINKER

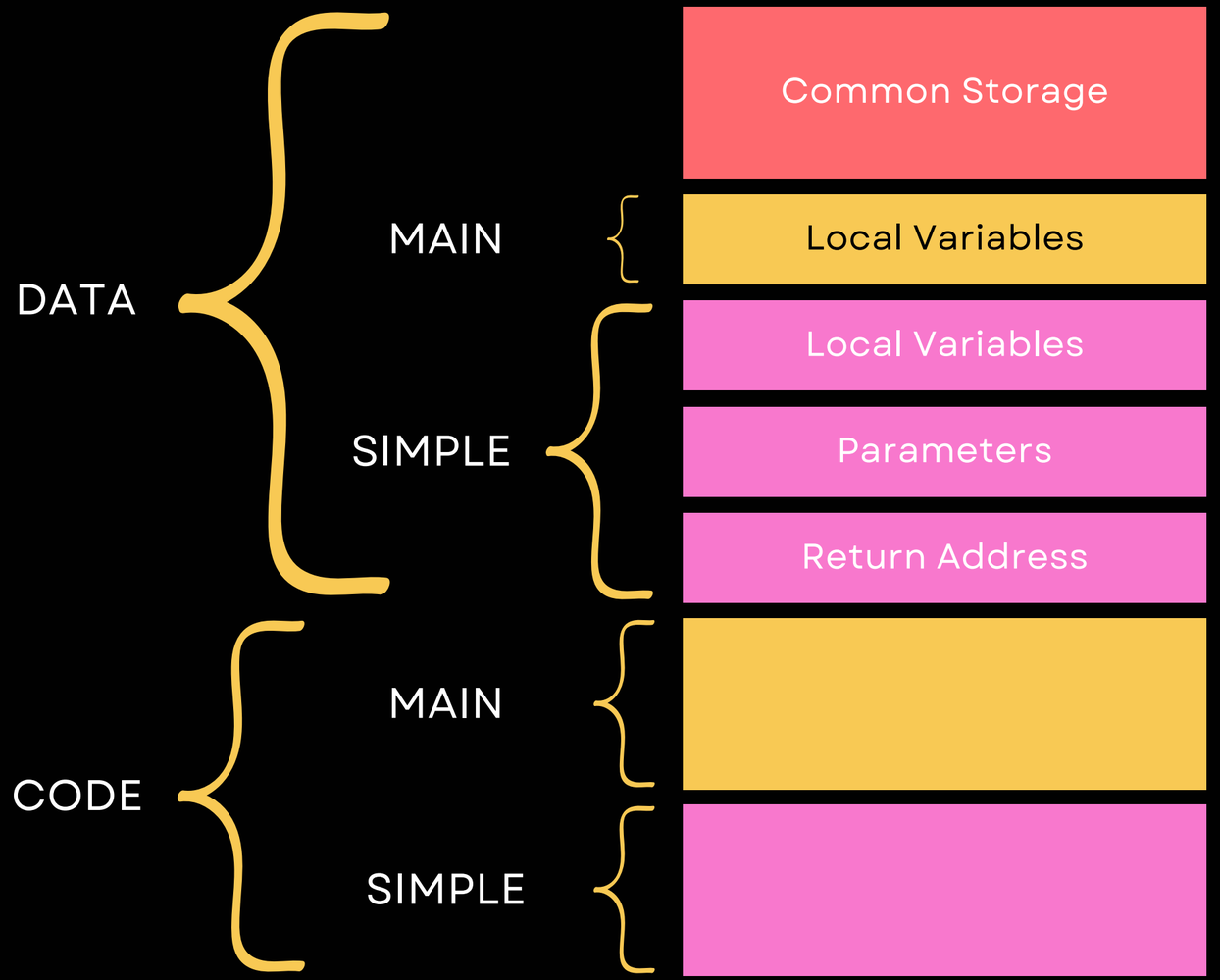
(Also called loader or link editors)

Linker Tasks

- Find file containing reference list of external subprograms and load them into memory
- Set target addresses of all calls to the subprograms to the entry addresses
  - Must be done for all loaded and library subprograms

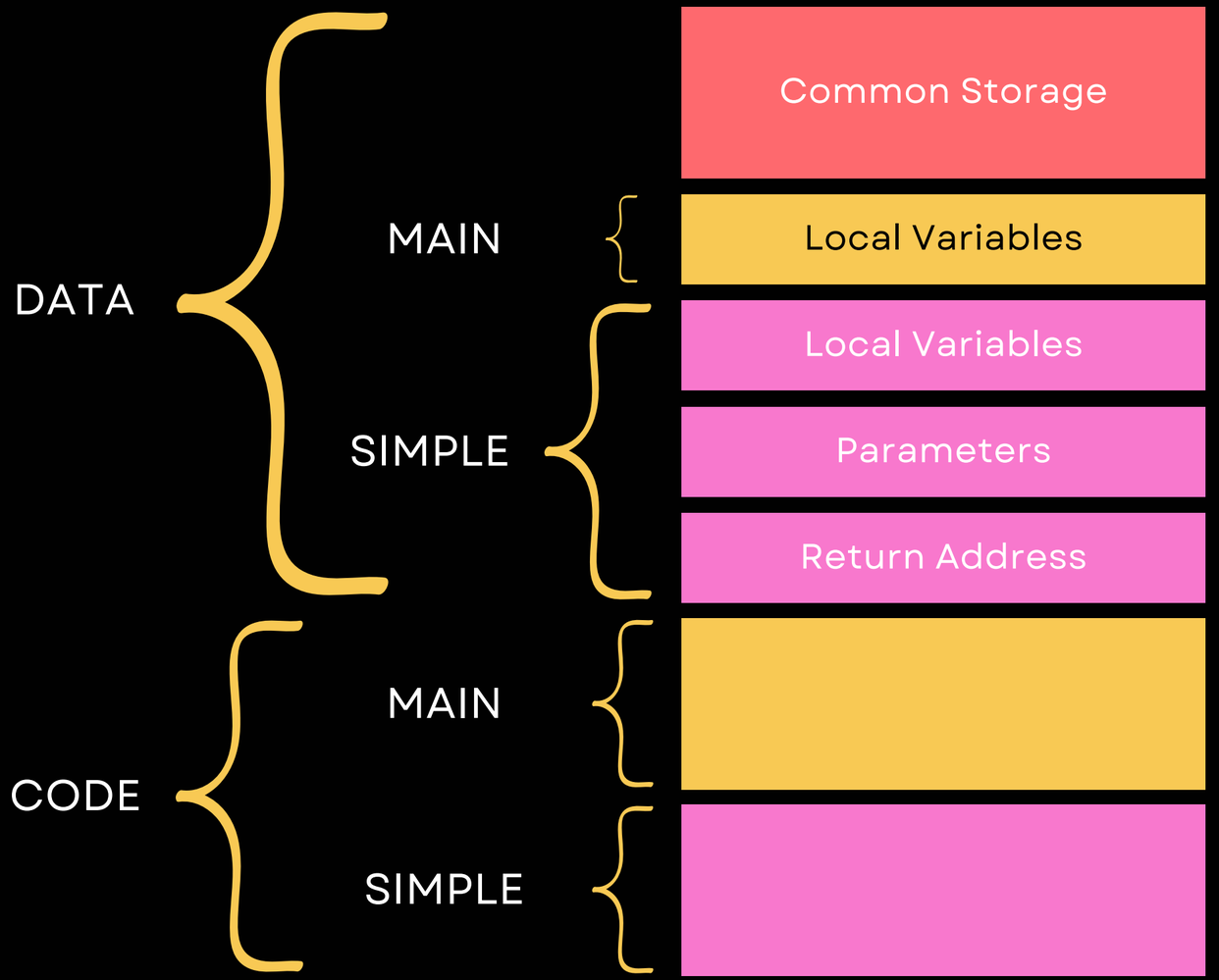
Code and Activation Records of a Program with “Simple” Subprogram

- Linker is called for MAIN



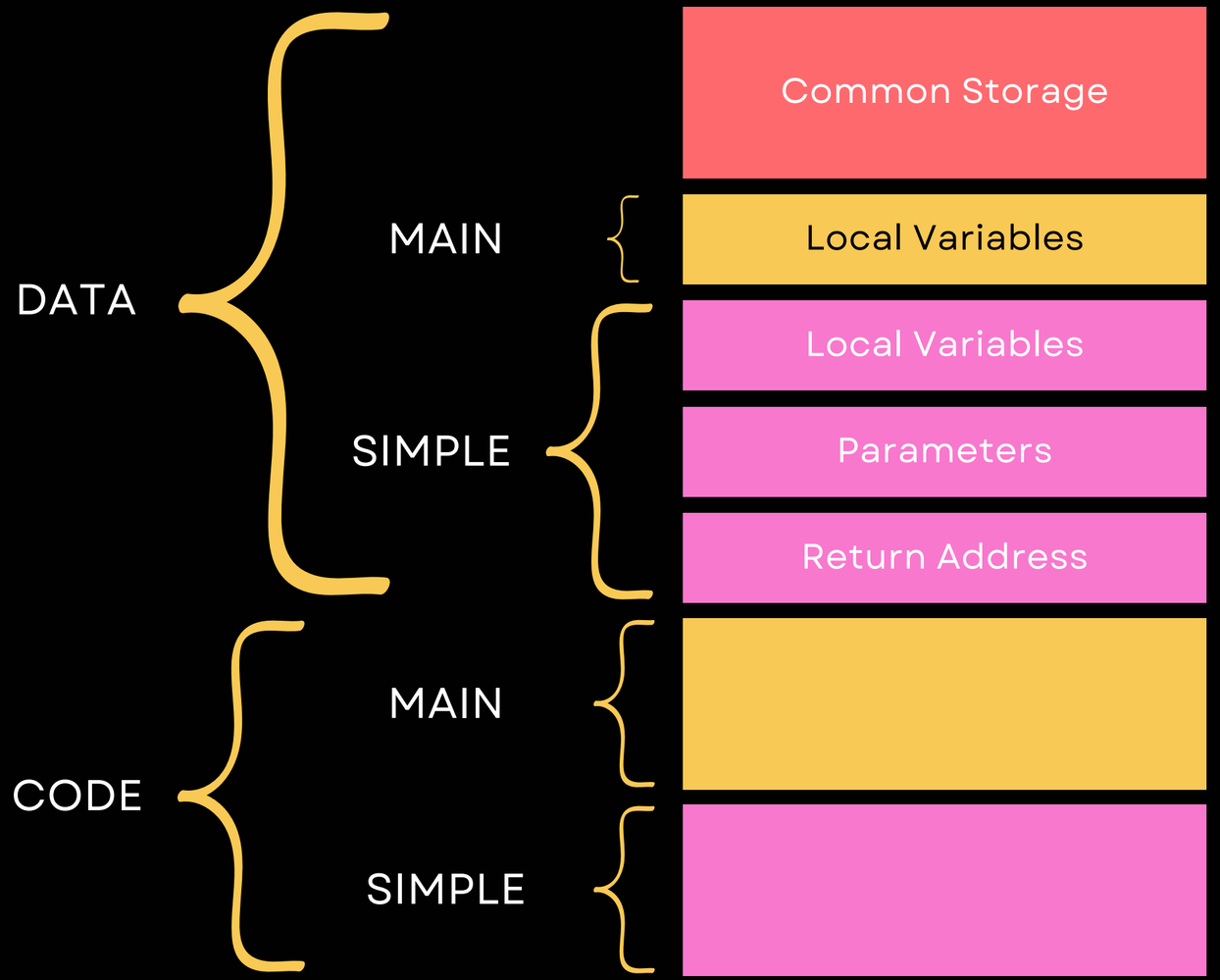
Code and Activation Records of a Program with “Simple” Subprogram

- Linker is called for MAIN
- Linker finds machine code for SIMPLE() and its activation record instances



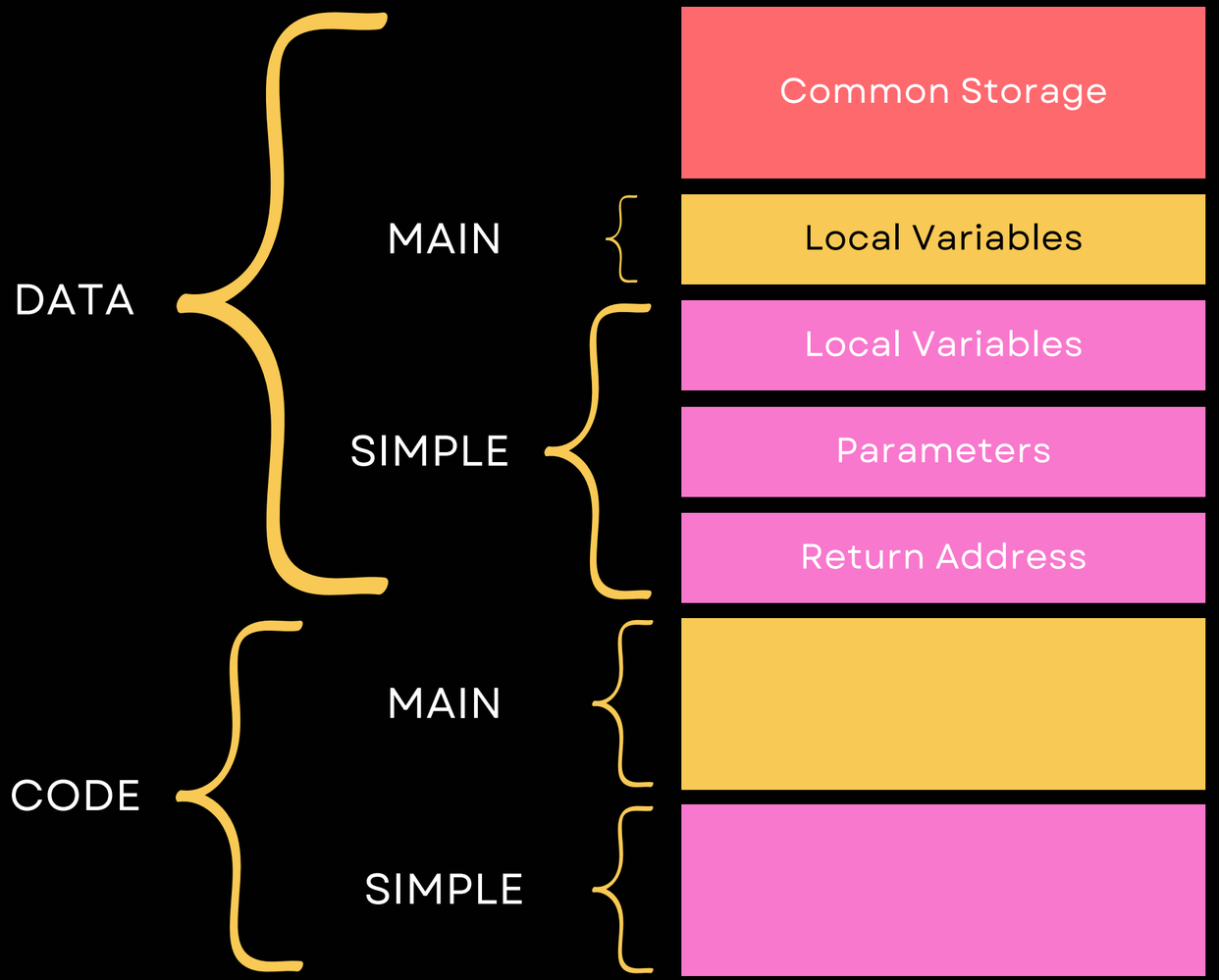
Code and Activation Records of a Program with “Simple” Subprogram

- Linker is called for MAIN
- Linker finds machine code for SIMPLE() and its activation record instances
- loads it into memory with code for MAIN



Code and Activation Records of a Program with “Simple” Subprogram

- Linker is called for MAIN
- Linker finds machine code for SIMPLE() and its activation record instances
- loads it into memory with code for MAIN
- target addresses for all calls to SIMPLE and any library subprograms are set



Implementing Subprograms with Stack-Dynamic Local Variables

most important advantage is the support for recursion. On the other hand, it's subprogram linkage is much more complex.

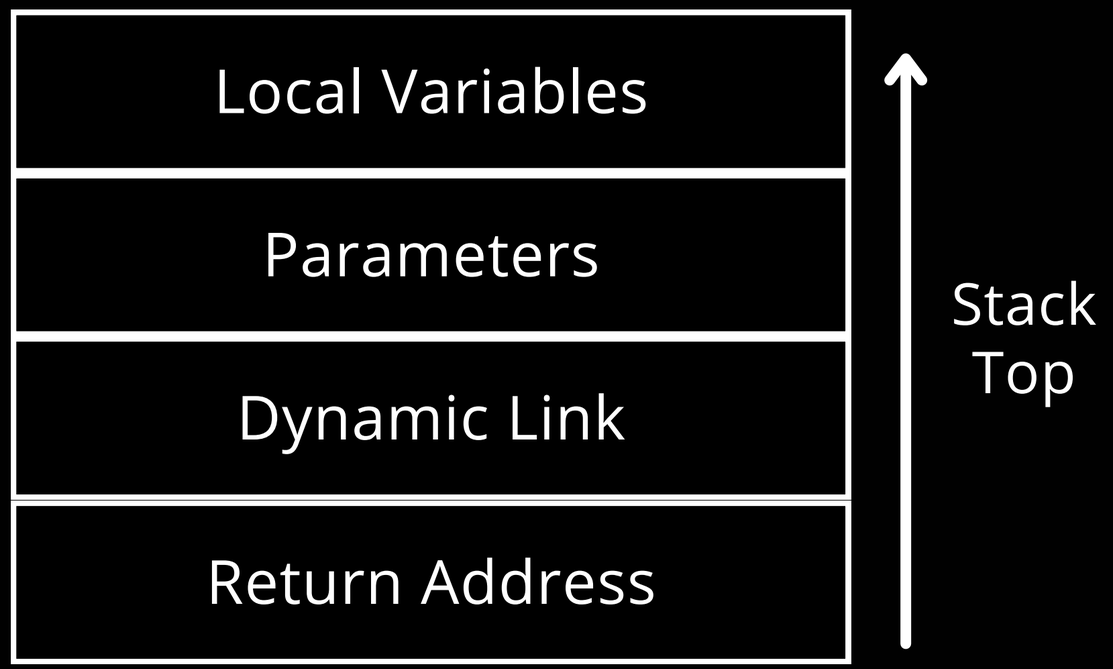
Implementing Subprograms with Stack-Dynamic Local Variables

More complex activation record

- The compiler must generate code to cause implicit allocation and deallocation of local variables
- Recursion must be supported (adds the possibility of multiple simultaneous activations of a subprogram)

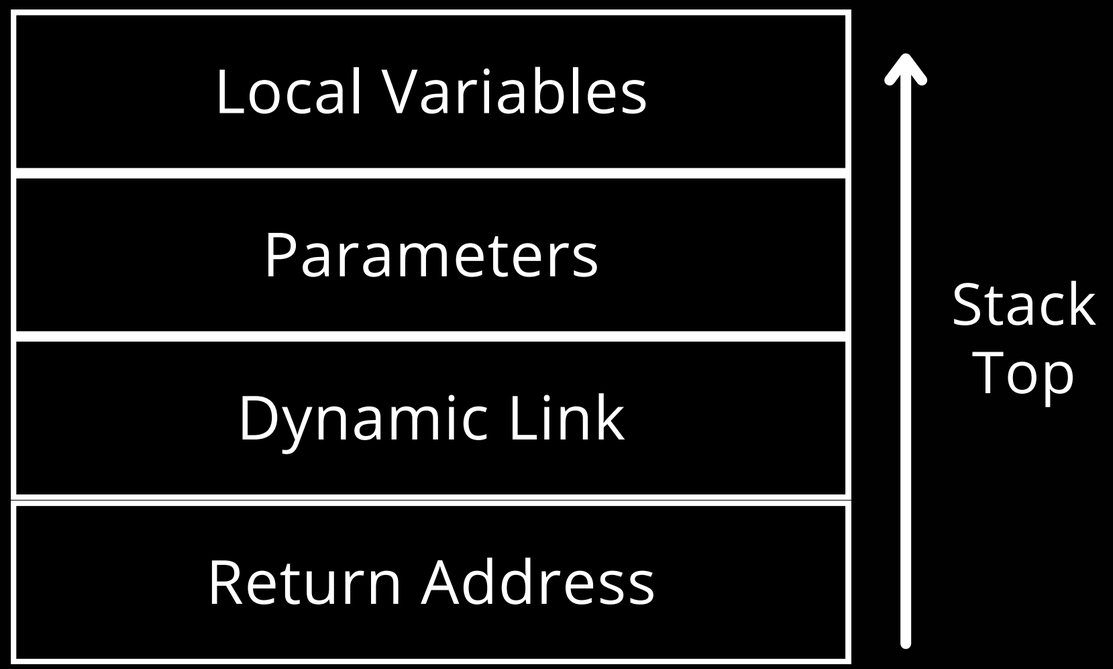
Typical Activation Record for a Language with Stack-Dynamic Local Variables

- format of an activation record is generally known at compile time.
- The activation record instance is dynamically created



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

- **Return Address** - pointer to instruction following the call in the calling program.
- **Dynamic Link** - points to the base of the activation record instance of the caller



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

- In static scope languages, link is used to provide traceback information for run-time errors.
- In dynamic-scoped languages, the link is used to access non-local variables

Dynamic Link

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

- **Actual Parameters** - the values or addresses provided by the caller

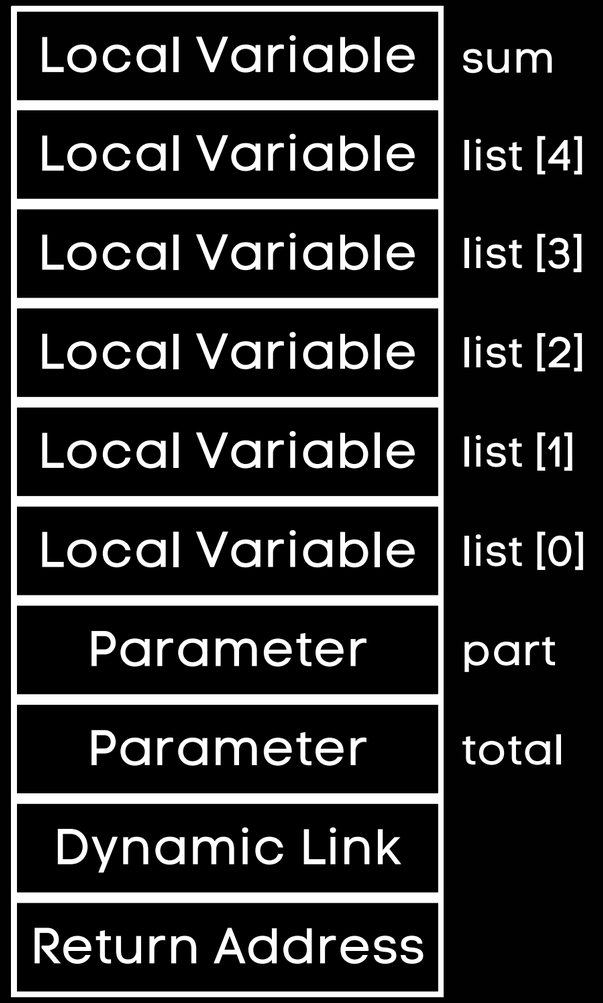
Typical Activation Record for a Language with Stack-Dynamic Local Variables

- **Local variables** that are structures are sometimes allocated elsewhere, and only their descriptors and a pointer to that storage are part of the activation record

- Local variables are allocated and possibly initialized in the called subprogram, so they appear last.

An Example: C Function

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

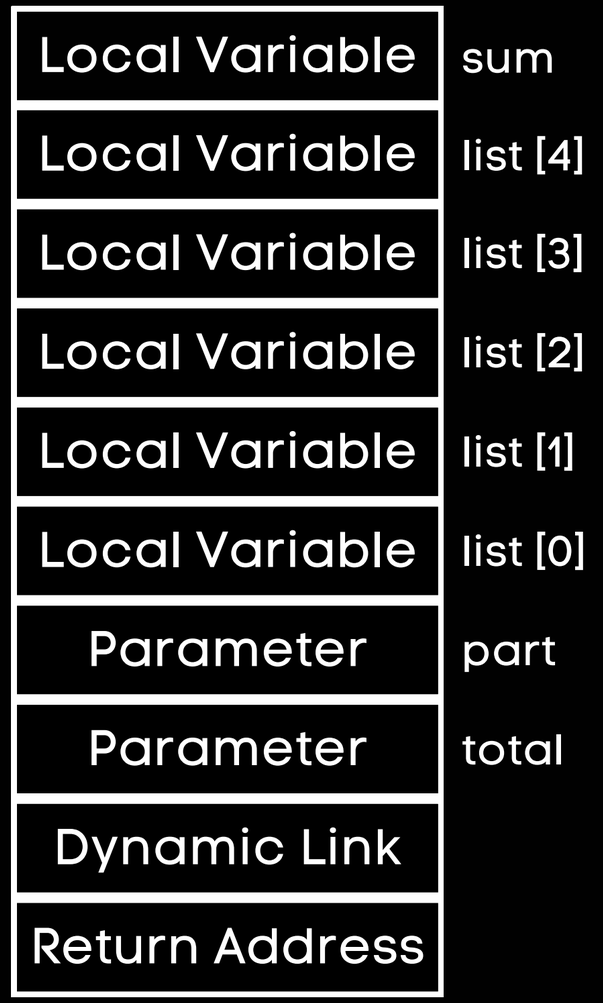


Activation Record

Activating a subprogram requires the dynamic creation of an ARI for the subprogram.

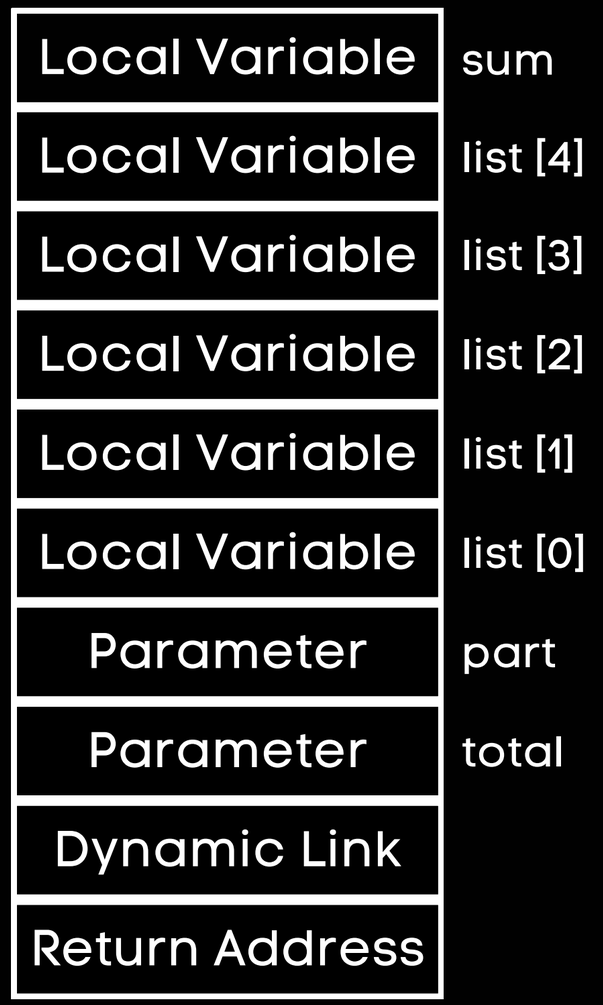
Why a stack?
the call and return semantics specify that the subprogram last called is the first to complete

Every subprogram activation, whether recursive or nonrecursive, creates a new instance of an activation record on the stack.



Activation Record

The Environment Pointer (EP) must be maintained by the run-time system. It always points at the base of the activation record instance of the currently executing program unit



Revised Semantic Call/Return Actions

CALLER ACTIONS

Create an activation record instance
Save the execution status of the current program unit
Compute and pass the parameters
Pass the return address to the called
Transfer control to the called

Revised Semantic Call/Return Actions

PROLOGUE ACTIONS (CALLED)

Save the old EP in the stack as the dynamic link and create the new value
Allocate local variables

Revised Semantic Call/Return Actions

EPILOGUE ACTIONS (CALLED)

If there are pass-by-value-result or out-mode parameters, the current values of those parameters are moved to the corresponding actual parameters
If the subprogram is a function, its value is moved to a place accessible to the caller
Restore the stack pointer by setting it to the value of the current EP-1 and set the EP to the old dynamic link
Restore the execution status of the caller
Transfer control back to the caller

An Example Without Recursion (C)

```
void fun1(float r) {
    int s, t;
    ...
    fun2(s);
    ...
}

void fun2(int x) {
    int y;
    ...
    fun3(y);
    ...
}
```

```
void fun3(int q) {
    ...
}
```

```
int main() {
    float p;
    ...
    fun1(p);
    ...
    return 0;
}
```

main calls fun1
fun1 calls fun2
fun2 calls fun3

An Example Without Recursion (JAVA)

```
void fun1(float r) {
 int s, t;
 // ...
 fun2(s);
 // ...
}

void fun2(int x) {
 int y;
 // ...
 fun3(y);
 // ...
}
```

```
 void fun3(int q) {
  // ...
 }
```

```
public static void
main(String[] args) {
 float p;
 // ...
 fun1(p);
 // ...
}
```

main calls fun1
fun1 calls fun2
fun2 calls fun3

An Example Without Recursion (PYTHON)

```
def fun1(r):
 s, t , 0
 # ...
 fun2(s)
 # ...

def fun2(x):
 y = 0
 # ...
 fun3(y)
 # ...
```

```
 def fun3(q):
  # ...
```

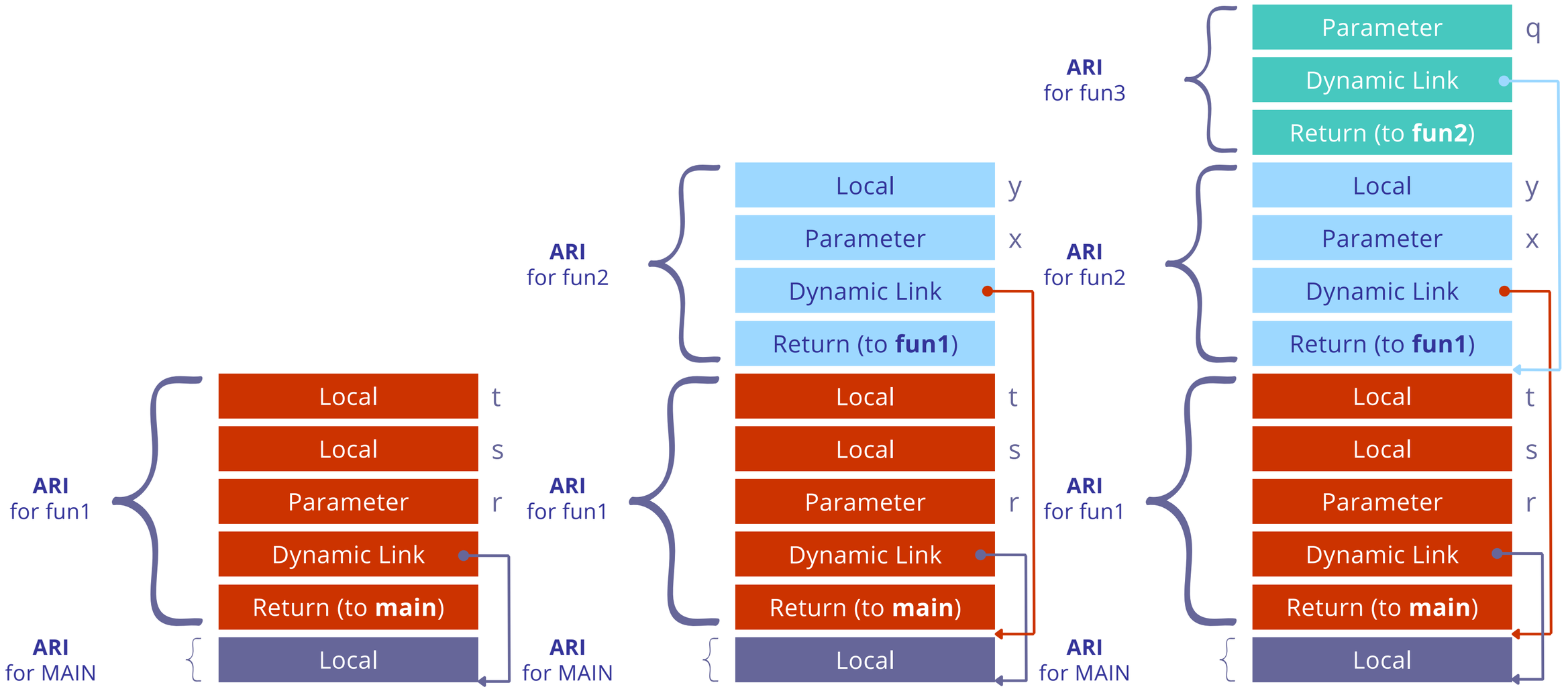
```
p = 0.0
# ...
fun1(p)
#...
```

main calls fun1
fun1 calls fun2
fun2 calls fun3

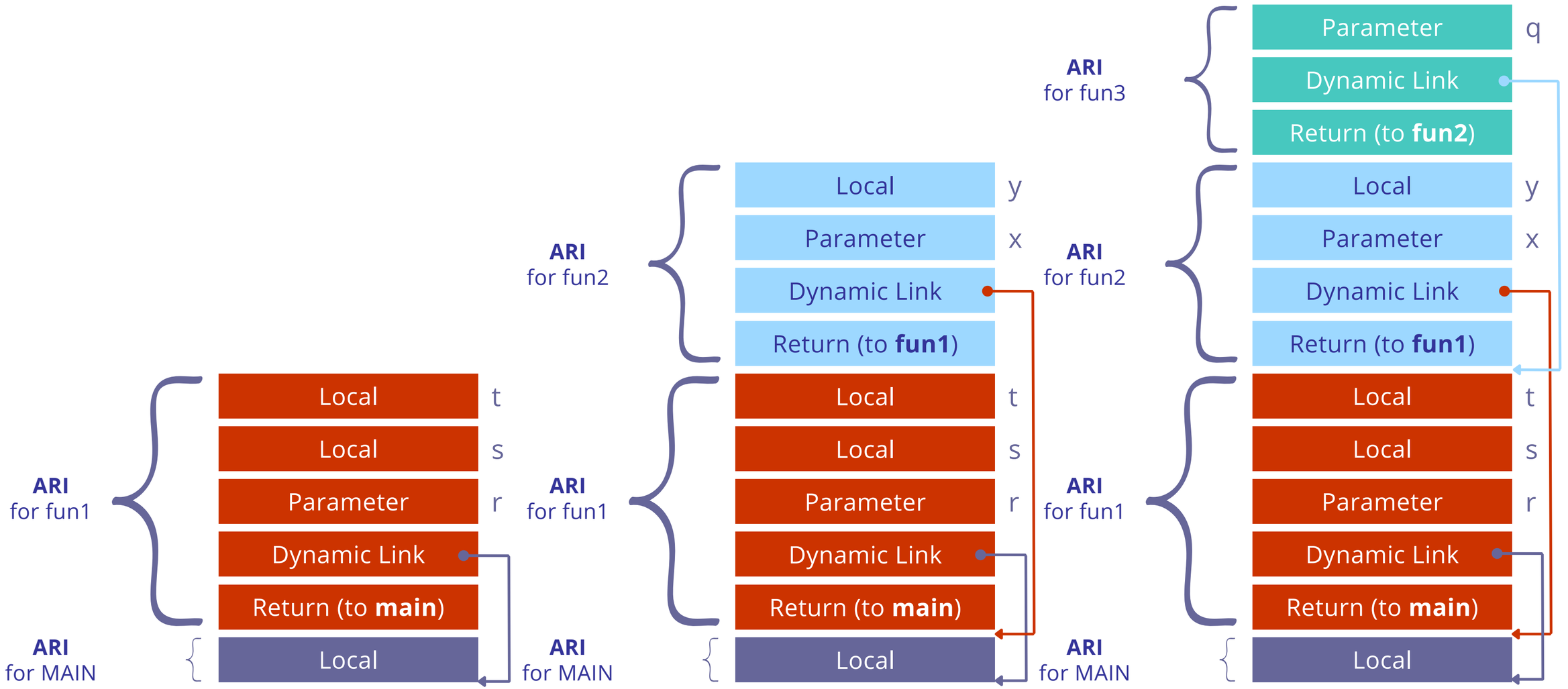
ENVIRONMENTAL POINTER

is used as the base of the offset addressing of the data contents of the activation record instance—parameters and local variables. Note that the EP currently being used is not stored in the run-time stack. Only saved versions are stored in the activation record instances as the dynamic link

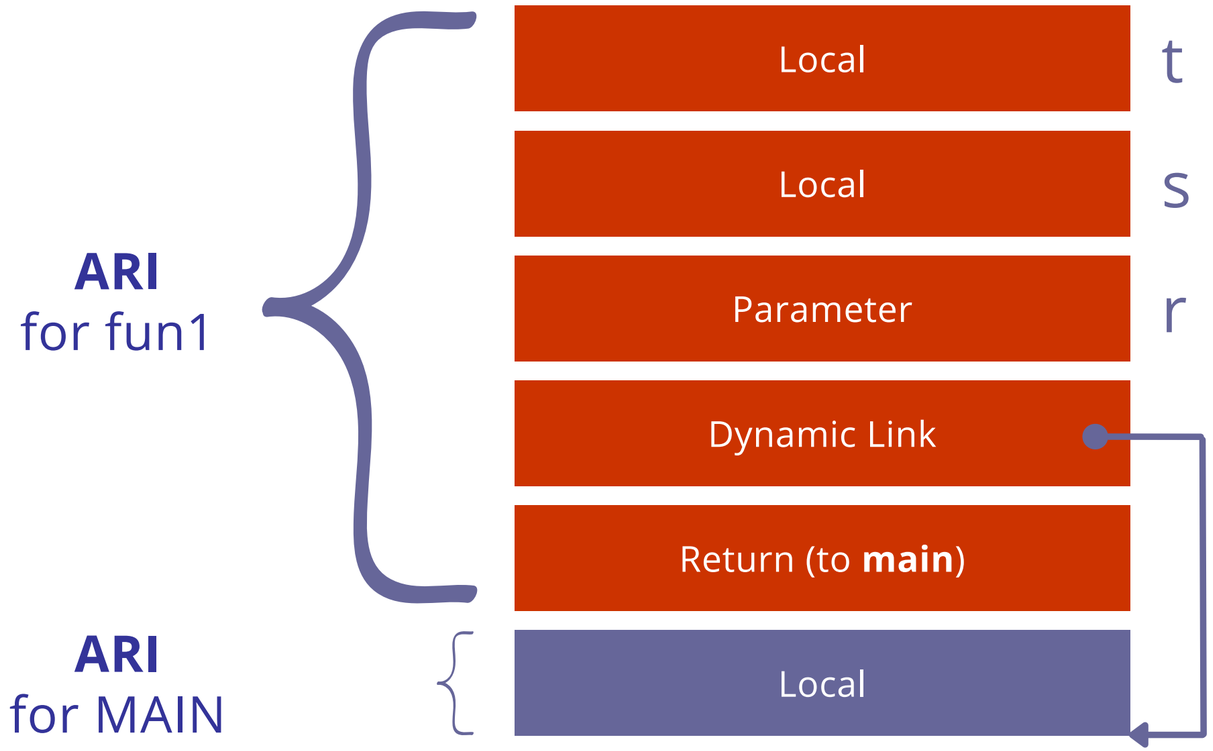
An Example Without Recursion



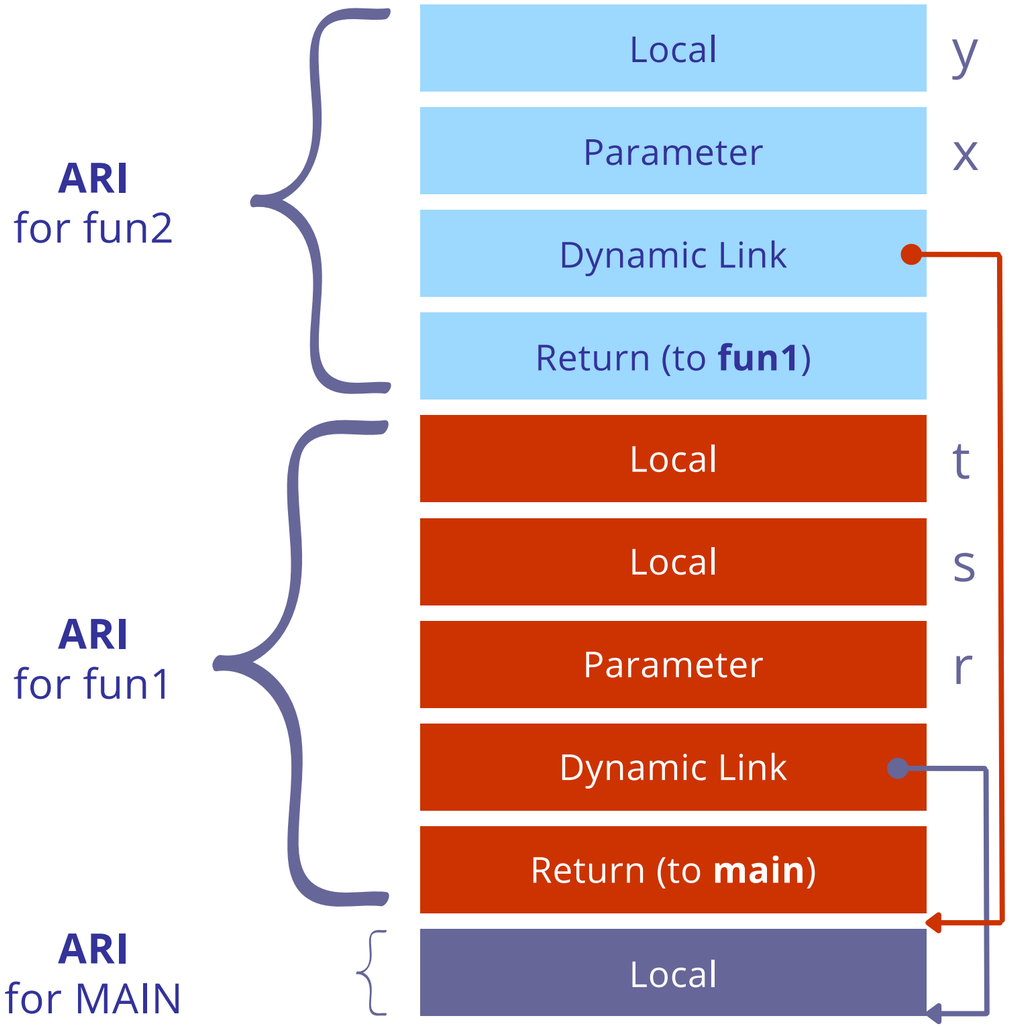
An Example Without Recursion



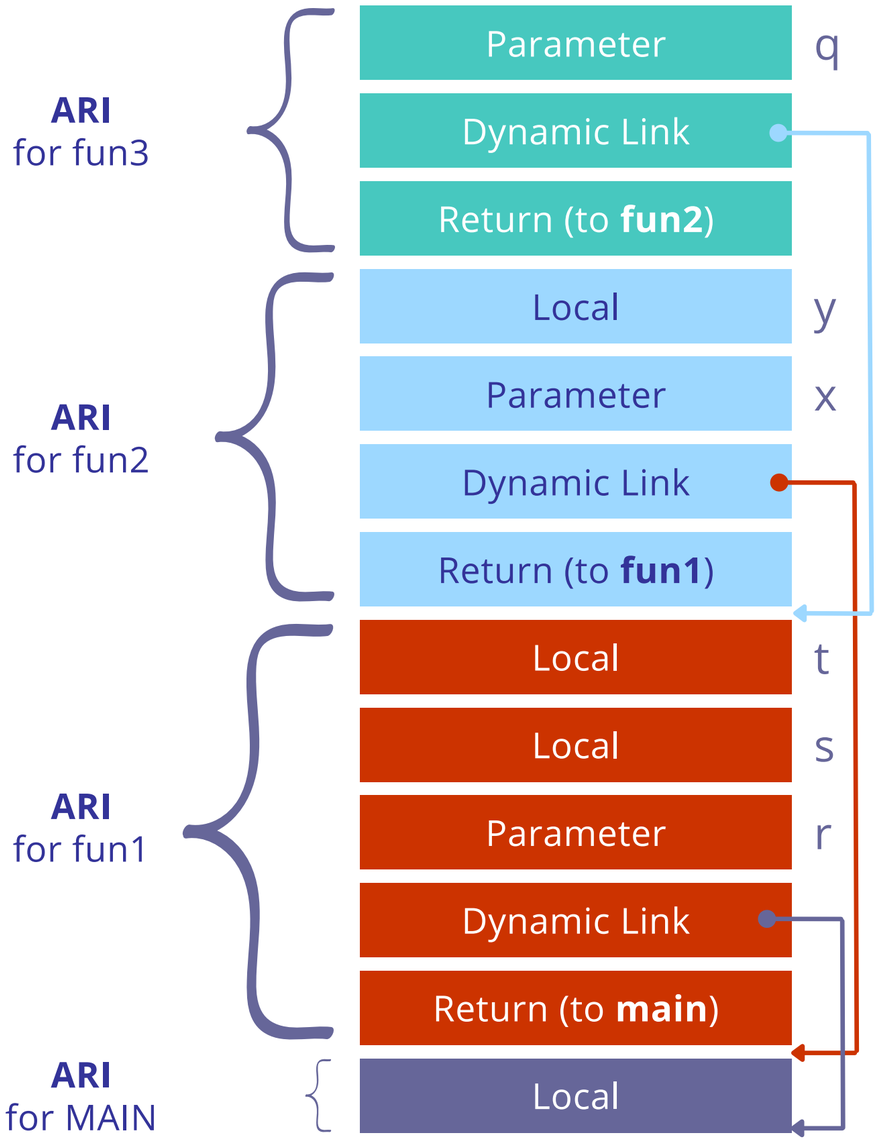
An Example Without Recursion



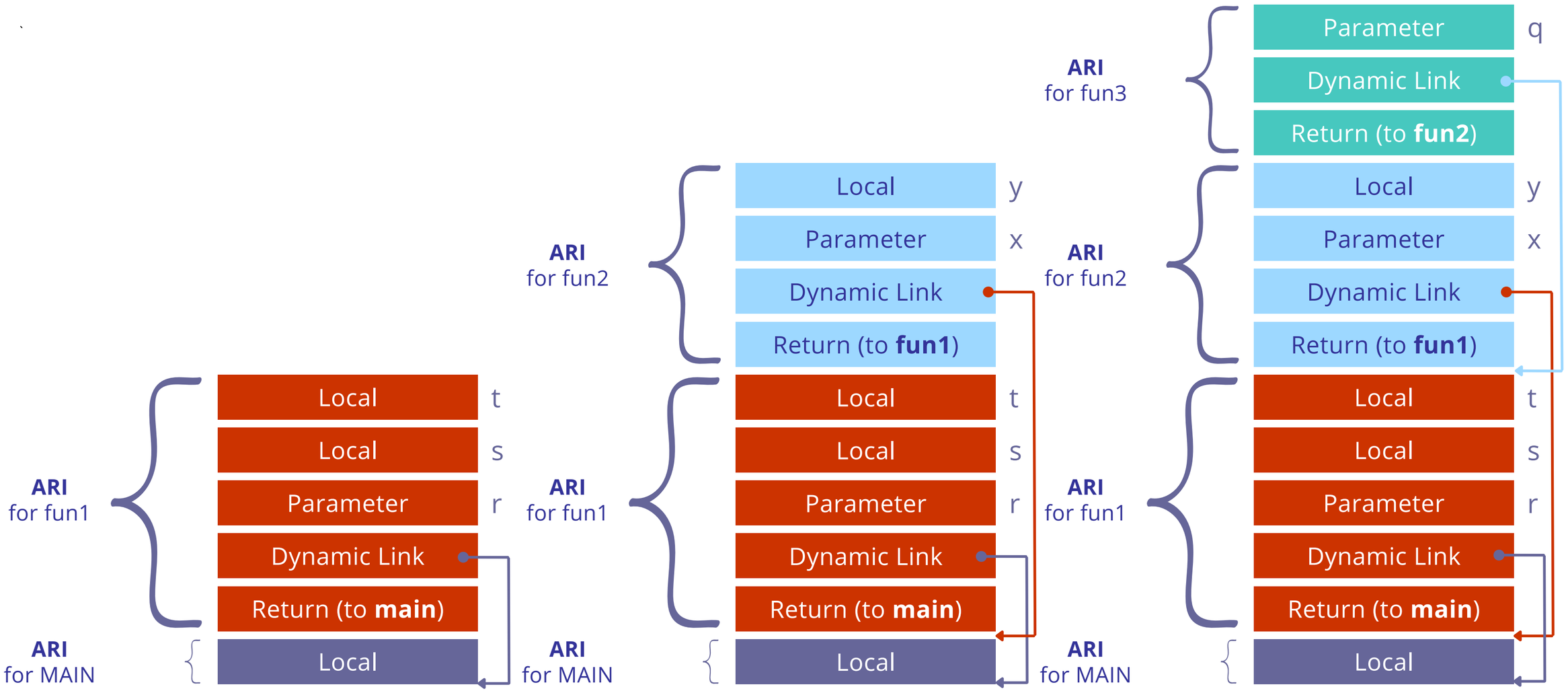
An Example Without Recursion



An Example Without Recursion



An Example Without Recursion



Dynamic Chain and Local Offset

- The collection of dynamic links in the stack at a given time is called the ***dynamic chain***, or ***call chain***
- Local variables can be accessed by their offset from the beginning of the activation record, whose address is in the EP. This offset is called the ***local_offset***
- The local_offset of a local variable can be determined by the compiler at compile time

An Example With Recursion(C)

- The activation record used in the previous example supports recursion

```
 int factorial (int n) {
    <-----------------------------1
  if (n <= 1) return 1;
  else return (n * factorial(n - 1));
    <-----------------------------2
}
```

```
void main() {
  int value;
  value = factorial(3);
}
```

An Example With Recursion(Java)

- The activation record used in the previous example supports recursion

```
class FactorialEx{
 static int factorial(int n){
 if(n == 0)
 return 1;
 else
 return(n*factorial(n-1));
 }
 public static void main(String args[]){
 int i, fact = 1;
 int number = 3;<--------------------number to calculate factorial
 fact = factorial(number);
 }
}
```

An Example With Recursion(Python)

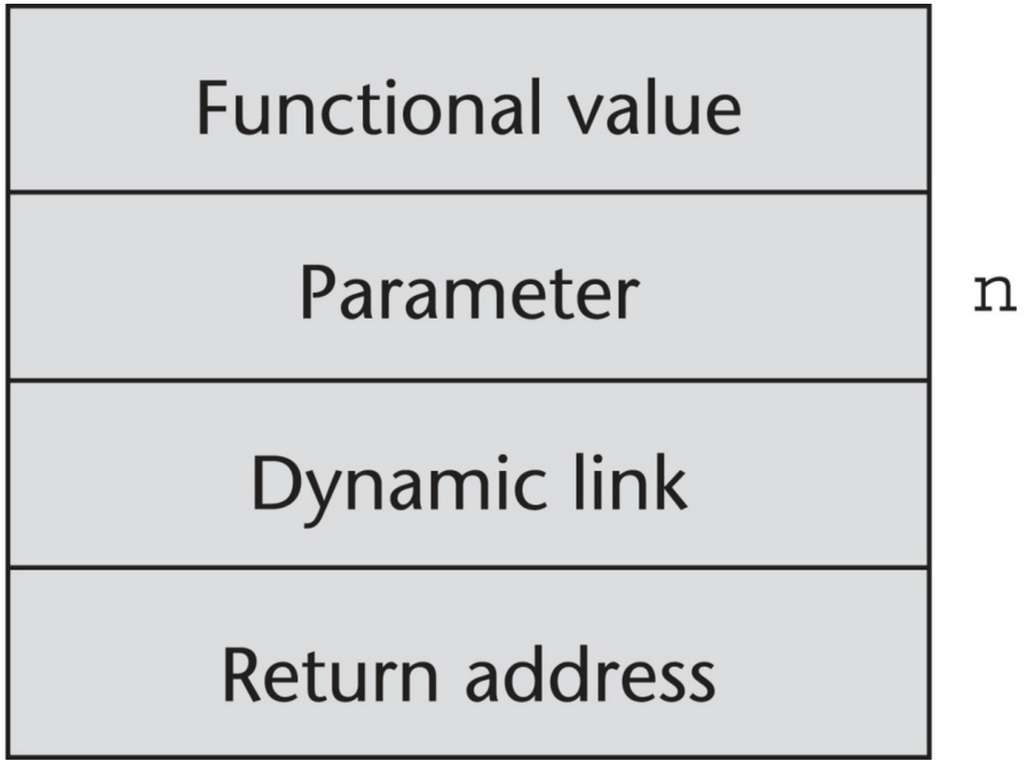
- The activation record used in the previous example supports recursion

```
def factorial(x):
 if x == 1:
  return 1
 else:
  return (x * factorial(x-1))
```

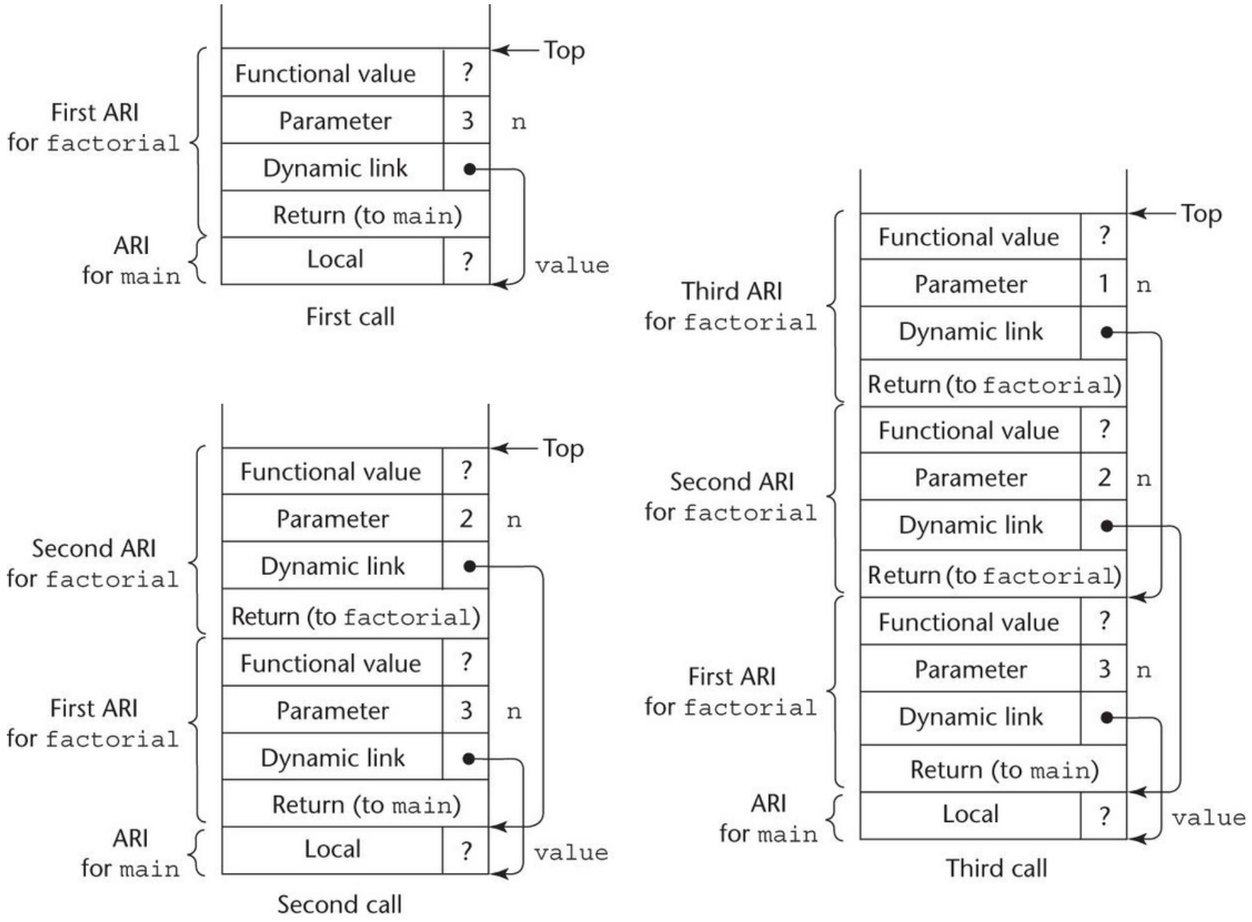
```
num=3

result = factorial(num)
```

Activation Record for factorial

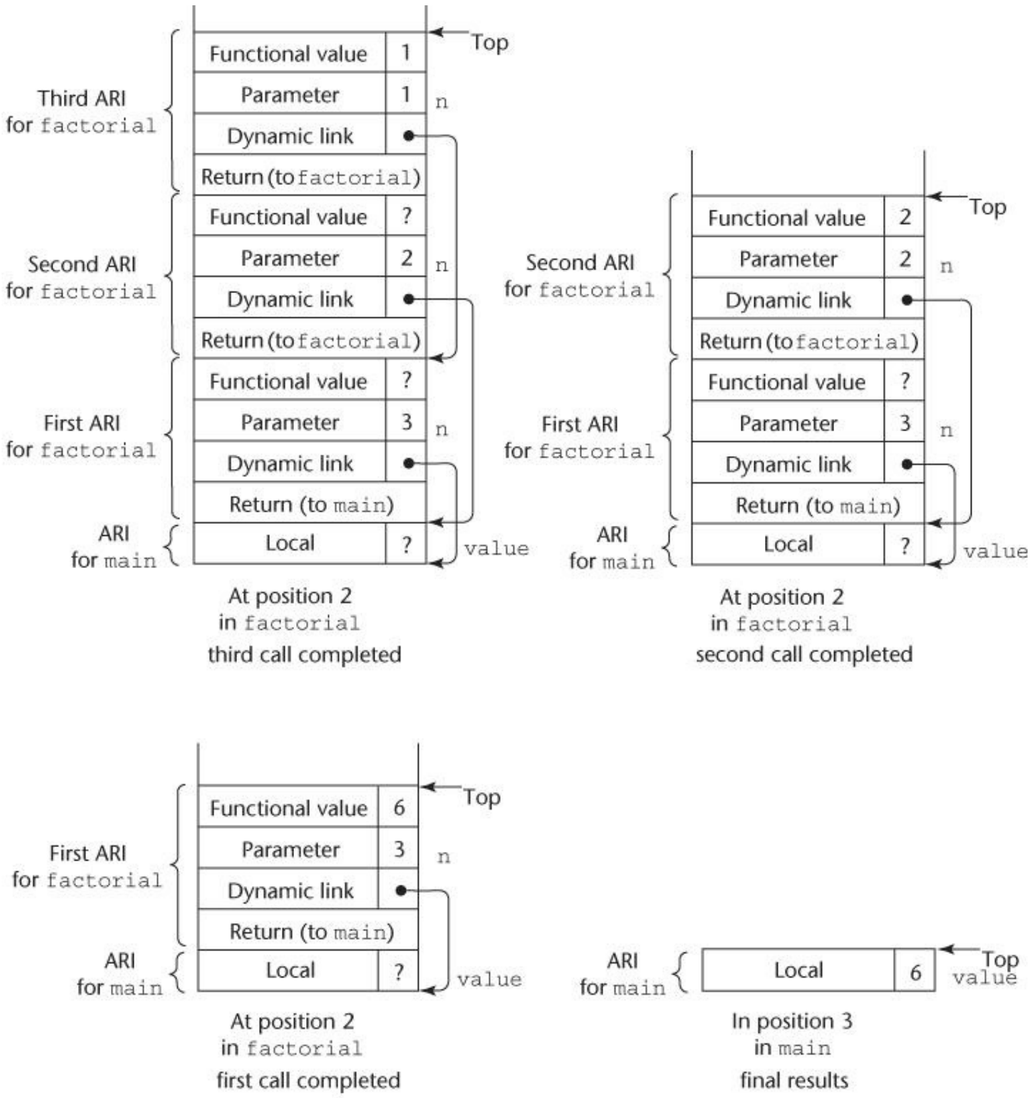


Stacks for calls to factorial



ARI = activation record instance

Stacks for returns from factorial



ARI = activation record instance

Nested Subprograms

- Fortran 95+, Ada, Python, Javascript, Ruby, Swift (non-C, static-scope)
  - stack-dynamic local variables
  - allow nested subprograms

Exploring common approaches for nested subprogram implementation...

Note: **Ignore closures**

Nested Subprograms

- All variables that can be **non-locally** accessed reside in some **activation record instance** in the stack
- Process of locating a non-local reference:
  1. Find the correct **activation record instance**
  2. Determine the correct **offset** within that activation record instance

Locating a Non-local Reference

Why activation record instance?

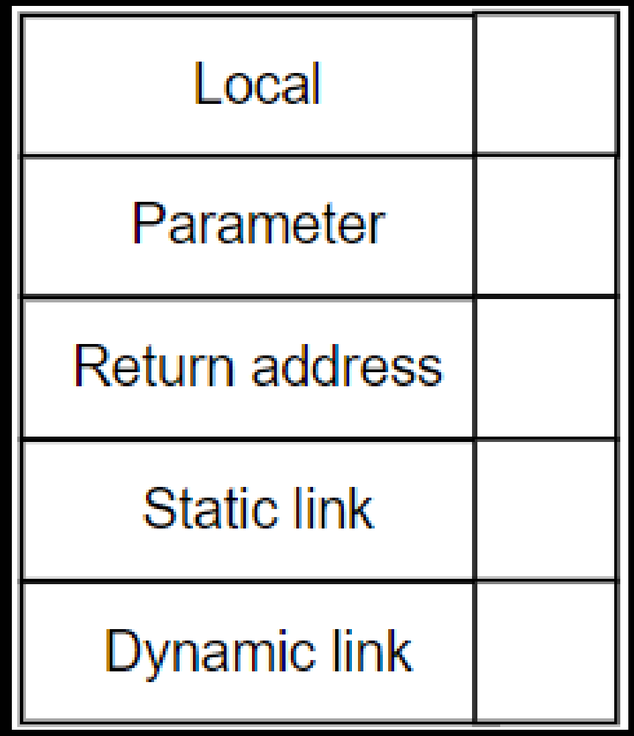
- Only nonlocal variables in **static ancestor scopes** are visible and can be accessed
- Guaranteed by **static semantic rules**
- Subprograms are **callable** when all its static ancestors are active
- Looks through the **most closely enclosing scopes** then the next
- **Possible** to find all activation record instances (ARI)

```
def top():
    a = 1
    def mid():
        b = 2
        def low():
            c = 3
            print(a)
            print(b)
```

Static Scoping

Static Link or Static Scope Pointer

- points to **bottom** of ARI
- appears below parameters typically
- original local offset changes



Static Scoping

- ***Static chain*** - a **lineage** of activation record instances
  - connects all static ancestors

- ***Static_depth*** - an integer indicating how deeply it is nested from the outermost scope
  - top - 0
  - mid - 1
  - low - 2

```
def top():
      a = 1
      def mid():
            b = 2
            def low():
                  c = 3
```

Static Scoping

- ***Chain_offset*** or ***nesting_depth*** of a nonlocal reference
  - static_depth of reference - static_depth of declaration

Find the chain_offset of reference a in low().

```
// global scope
def top():
 a = 1
 def mid():
  b = 2
  def low():
   print(a)
```

Static Scoping

- ***Chain_offset*** or ***nesting_depth*** of a nonlocal reference
  - static_depth of reference - static_depth of declaration

Find the chain_offset of reference a in low().

Ans: 2 // global scope is 0

```
// global scope
def top():
 a = 1
 def mid():
  b = 2
  def low():
   print(a)
```

Static Scoping

- A reference to a variable can be represented by the pair:

(chain_offset, local_offset)

Example Program

```
def top():
        a = 1
        b = 2
        def mid(num):
            c = 3
            d = 4
            print(f"a = {a} b = {b}")
            def low():
                e = 5
                f = 6
                print(f"num = {num}")
                print(f"a = {a} b = {b}")
                print(f"c = {c} d = {d}")
            low()
        mid(10)
top()
```

- Call sequence when program is called

program calls top
top calls mid
mid calls low

Stack Contents at Point 1

ARI = Activation Record Instance

Dynamic link

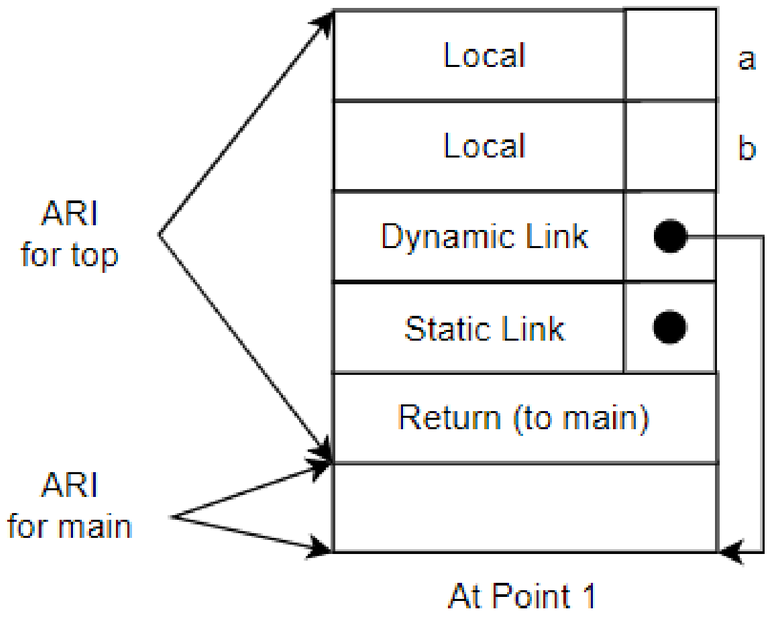
Static link

pair of a

-

pair of b

-




```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at Point 1

ARI = Activation Record Instance

Dynamic link

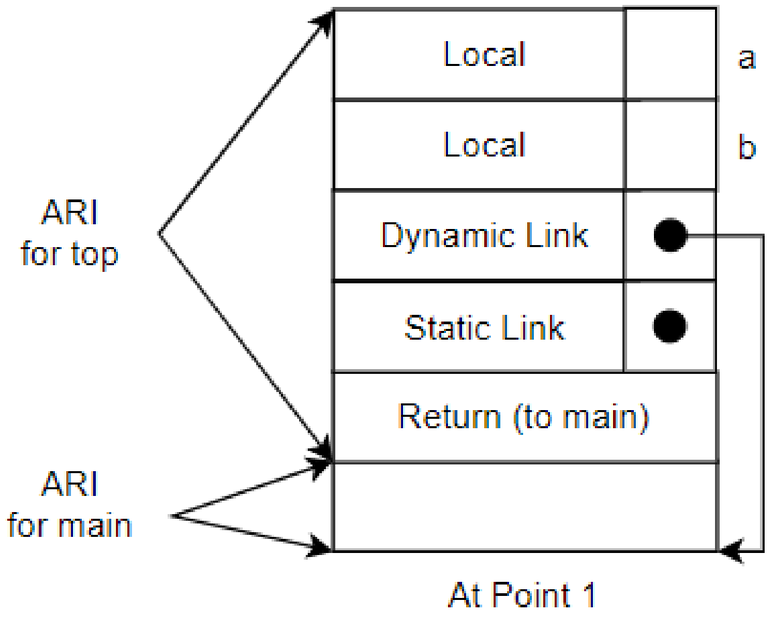
Static link

pair of a

- **(0, 4)**

pair of b

-



At Point 1

```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at Point 1

ARI = Activation Record Instance

Dynamic link

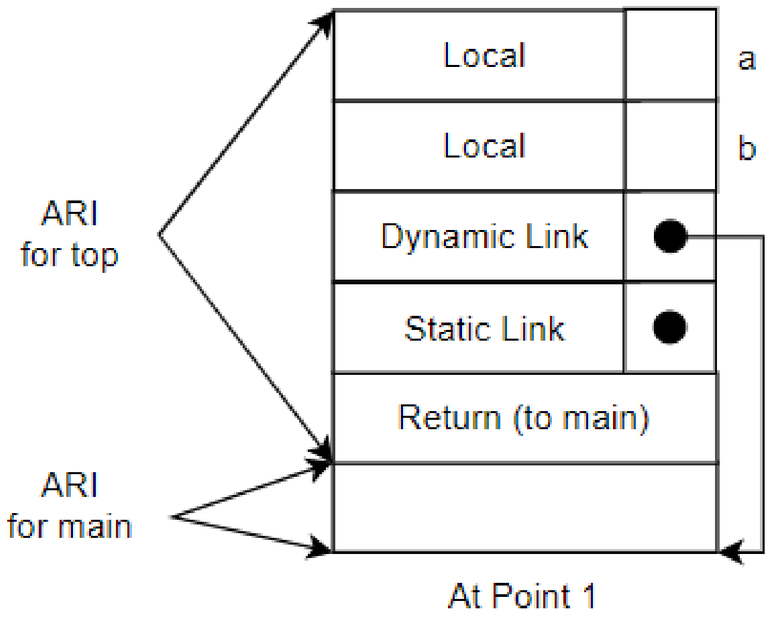
Static link

pair of a

- **(0, 4)**

pair of b

- **(0, 3)**




```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at Point 2

ARI = Activation Record Instance

Dynamic link

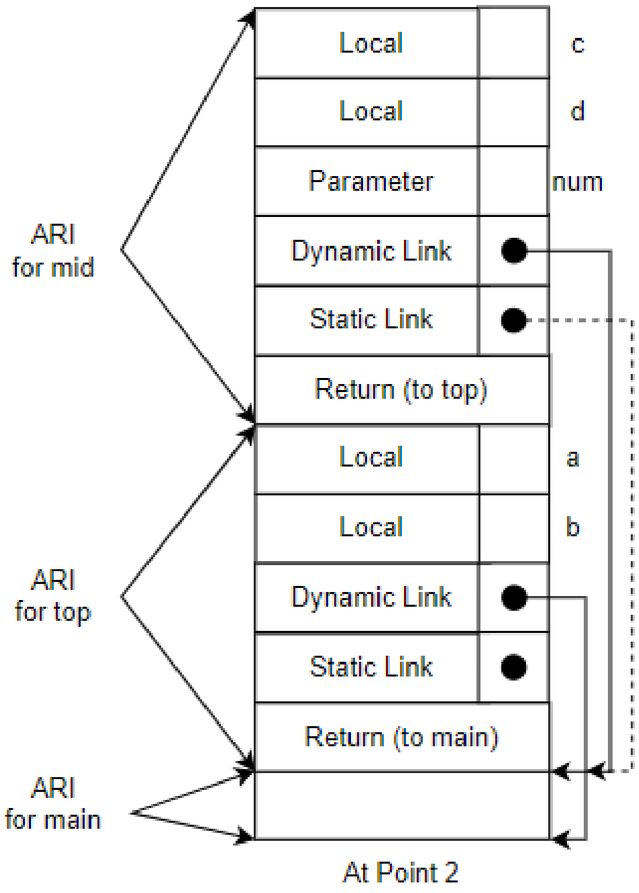
Static link

pair of a

-

pair of d

-



At Point 2

```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at Point 2

ARI = Activation Record Instance

Dynamic link

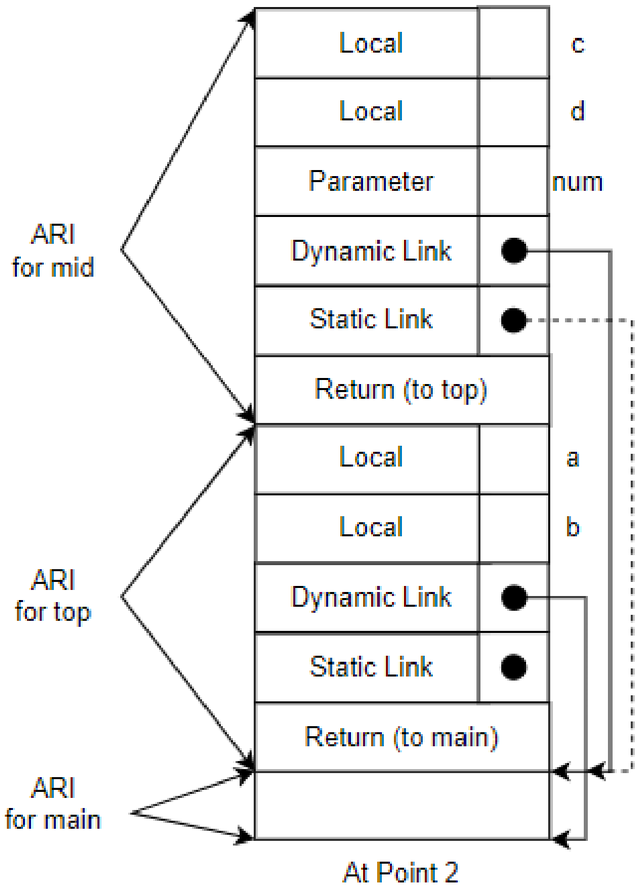
Static link

pair of a

- **(1, 4)**

pair of d

-




```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at Point 2

ARI = Activation Record Instance

Dynamic link

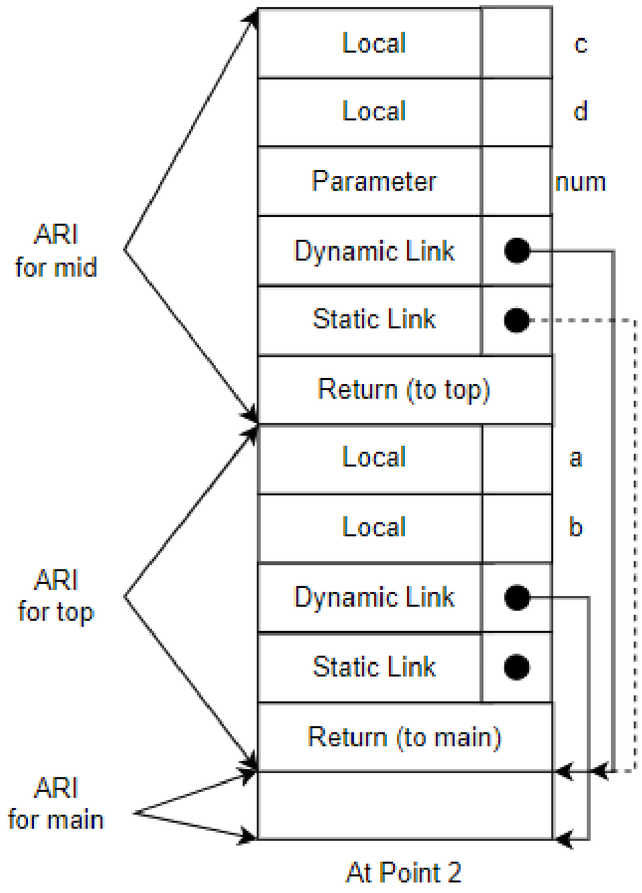
Static link

pair of a

- **(1, 4)**

pair of d

- **(0, 4)**



At Point 2

```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at Point 3

ARI = Activation Record Instance

Dynamic link

Static link

pair of a

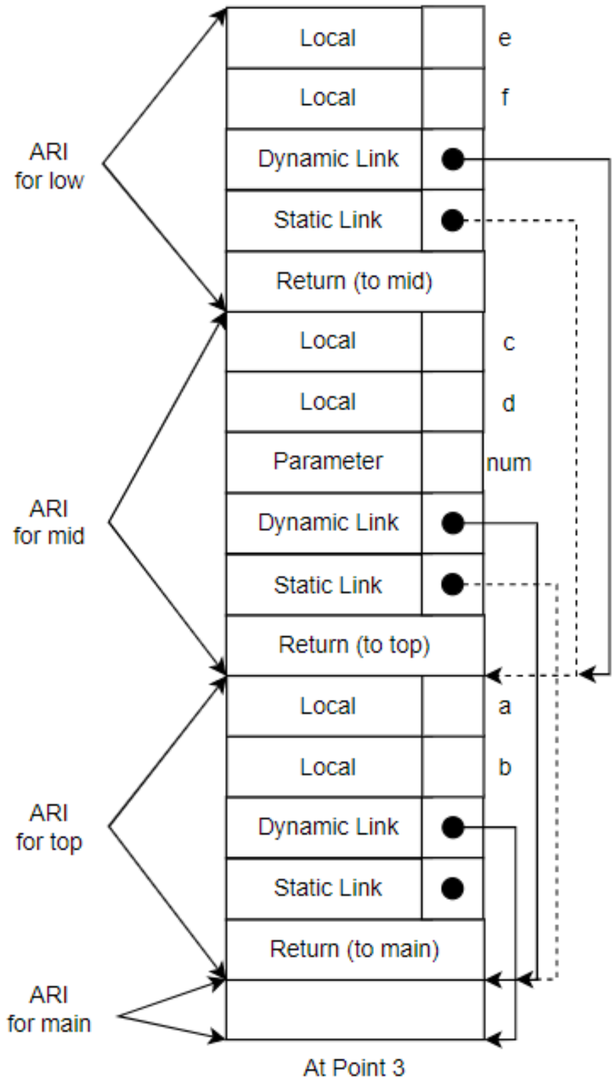
-

pair of d

-

pair of e

-



At Point 3

```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at Point 3

ARI = Activation Record Instance

Dynamic link

Static link

pair of a

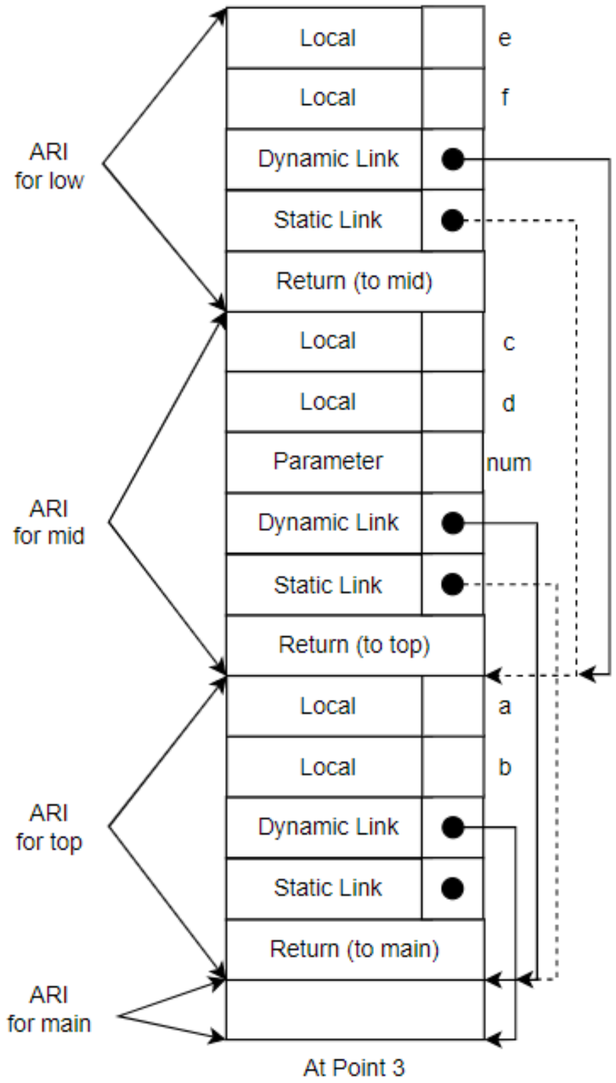
- **(2, 4)**

pair of d

-

pair of e

-



At Point 3

```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at Point 3

ARI = Activation Record Instance

Dynamic link

Static link

pair of a

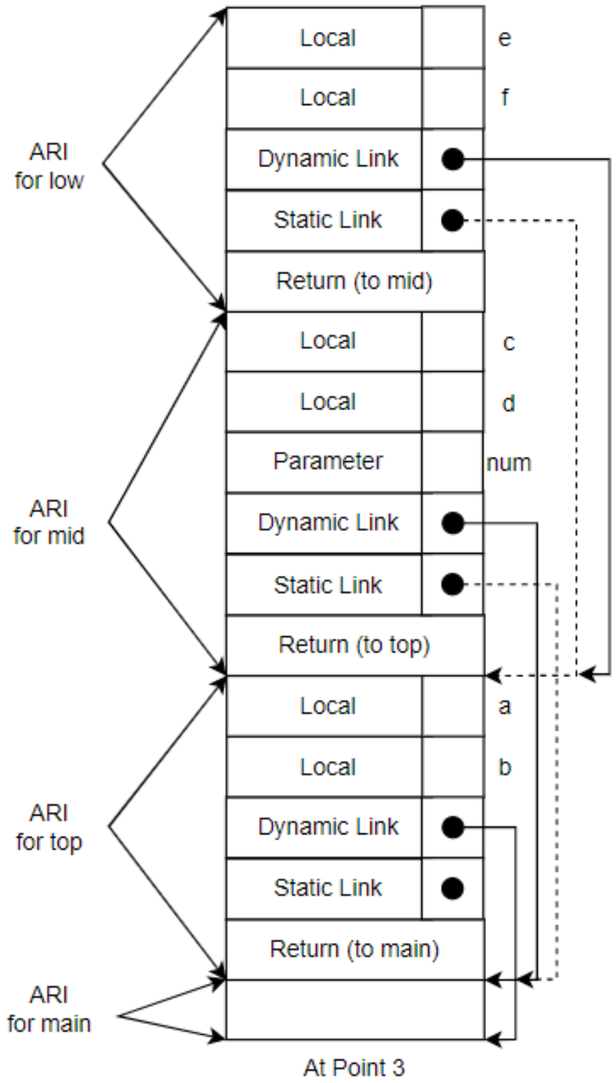
- **(2, 4)**

pair of d

- **(1,4)**

pair of e

-



At Point 3

```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at Point 3

ARI = Activation Record Instance

Dynamic link

Static link

pair of a

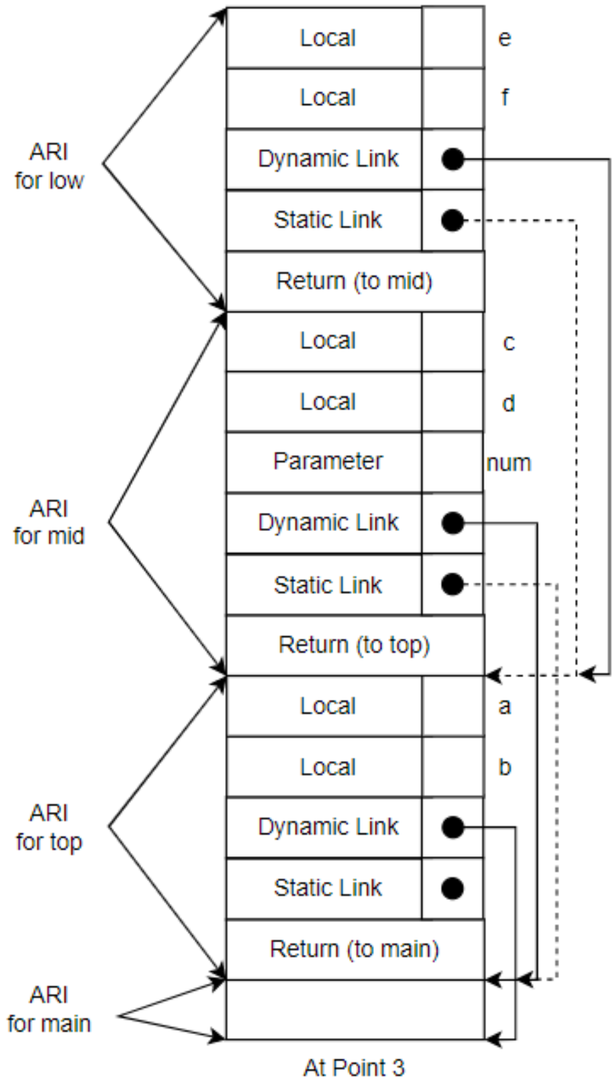
- **(2, 4)**

pair of d

- **(1,4)**

pair of e

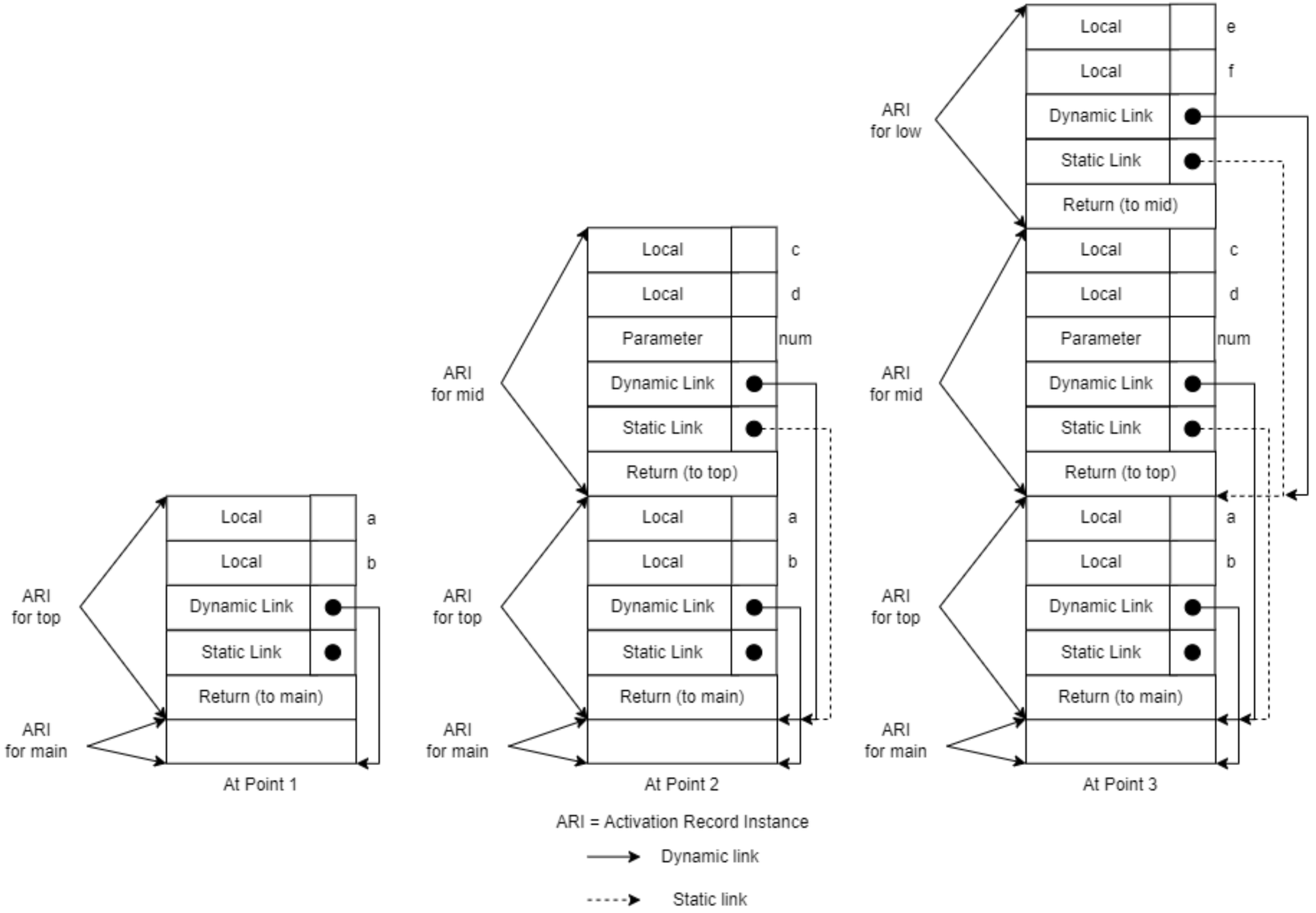
- **(0, 3)**



At Point 3

```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

Stack Contents at all Positions



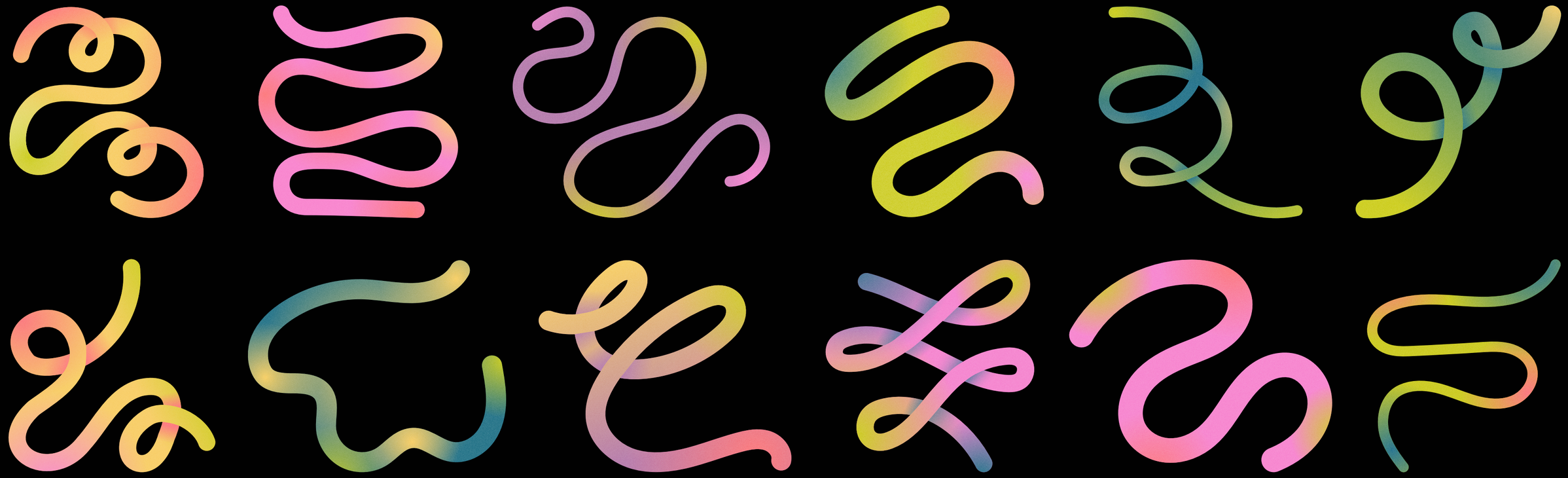
RESULT

```
def top():
 a = 1
 b = 2
 def mid(num):
  c = 3
  d = 4
  print(f"a = {a} b = {b}")
  def low():
   e = 5
   f = 6
   print(f"num = {num}")
   print(f"a = {a} b = {b}")
   print(f"c = {c} d = {d}")
  low()
 mid(10)
top()
```

a = 1 b = 2
num = 10
a = 1 b = 2
c = 3 d = 4

Resource Page

Use these design resources in your Canva Presentation.
Happy designing! Delete or hide this page before presenting.



Static Chain Maintenance

- **Static chain must be modified for each *subprogram call* and *return***
- **At the return,**
  - trivial
  - subprogram ends => ARI is removed from stack
  - new top ARI => caller of the above subprogram

Static Chain Maintenance

- **At the call,**
  - ARI of static parent must be found
  - Static link points to the most recent ARI of the static parent
  - Dynamic link points to the preceding ARI
- **Two methods:**
  - Search the dynamic chain
  - Treat subprogram calls and definitions like variable references and definitions

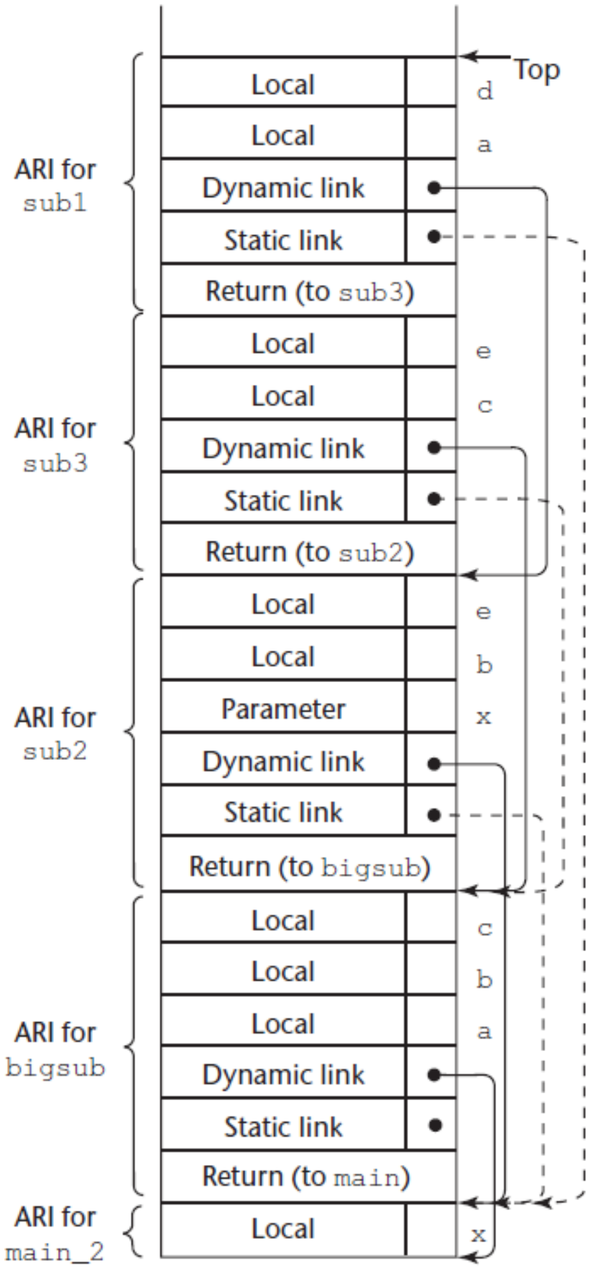
Static Chain Maintenance

- **Treat subprogram calls like variable declaration**
  - compiler determines the subprogram that declared the called subprogram
  - nesting_depth - number of enclosing scopes between the caller and the subprogram that declared the called subprogram
  - static link of called subprogram is found by moving down nesting_depth times the static chain of caller

Static Chain Maintenance

- Find static link of sub1()
- nesting_depth = 2

```
function main(){
  function bigsub() {
    function sub1 {
    } // end of sub1
    function sub2(x) {
      function sub3() {
        sub1();
      } // end of sub3
      sub3();
    } // end of sub2
    sub2();
  } // end of bigsub
  bigsub();
} // end of main
```



Evaluation of Static Chains

Problems

1. Nonlocal reference is **slow** if the nesting depth is large (rare in practice, thus not serious)
2. **Time-critical** code is difficult:
   - Costs of nonlocal references are **difficult** to determine
   - Code changes can change the **nesting depth**, and therefore the cost

No alternatives were found to be more superior.

Blocks

C Example

```
{
  int temp;
  temp = list [upper];
  list [upper] = list [lower];
  list [lower] = temp
}
```

- Blocks are user-specified local scopes for variables
- The lifetime of temp in the above example begins when control enters the block
- An advantage of using a local variable like temp is that it cannot interfere with any other variable with the same name

Implementing Blocks

2 Methods

- Treat blocks as parameter-less subprograms that are always called from the same location
  - Every block has an activation record; an instance is created every time the block is executed

- Since the maximum storage required for a block can be statically determined, this amount of space can be allocated after the local variables in the activation record

Implementing Dynamic Scoping

- ***Deep Access***: non-local references are found by searching the activation record instances on the dynamic chain
  - Length of the chain cannot be statically determined
  - Every activation record instance must have variable names

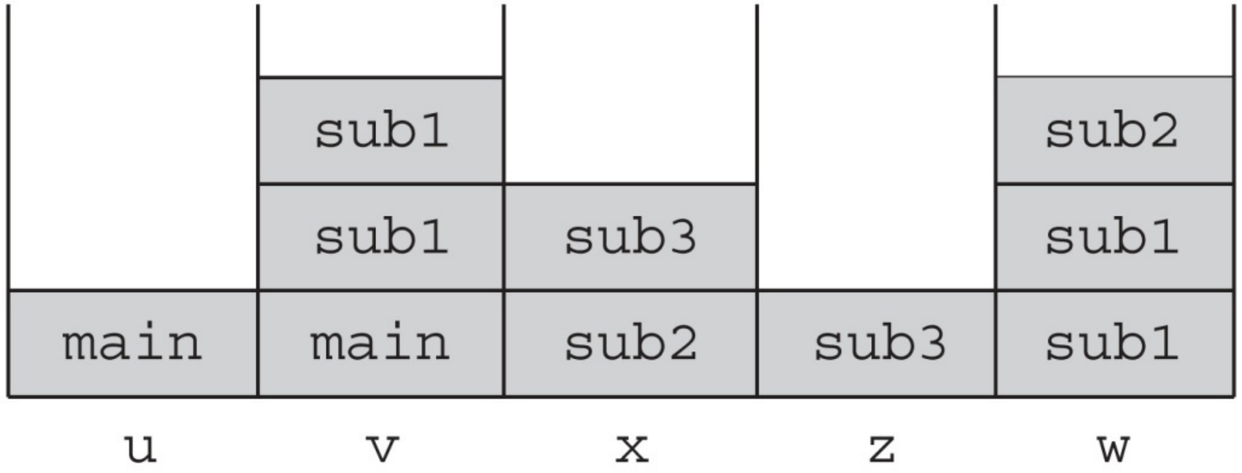
- ***Shallow Access***: put locals in a central place
  - One stack for each variable name
  - Central table with an entry for each variable name

Using Shallow Access to Implement Dynamic Scoping

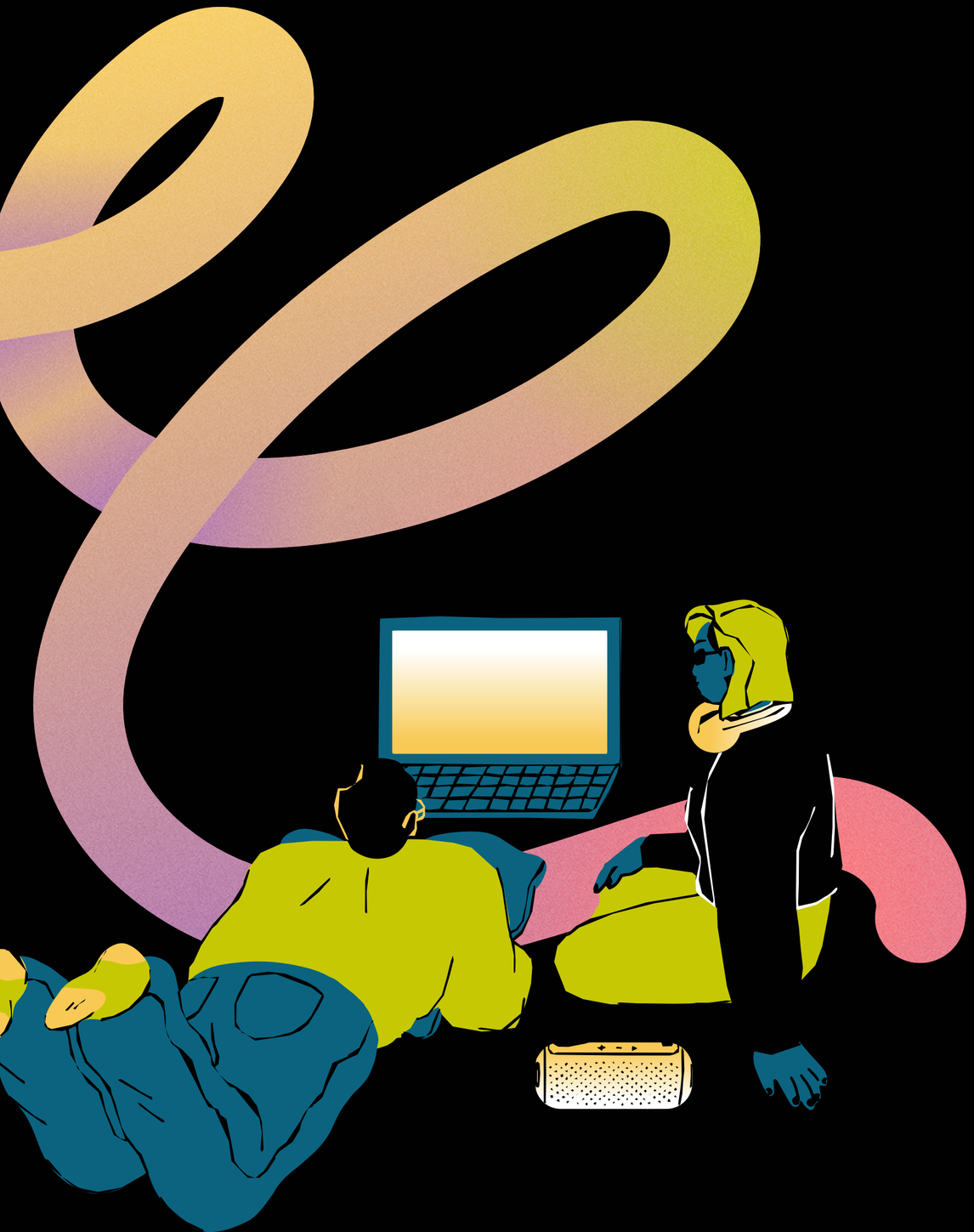
```
void sub3() {
  int x, z;
  x = u + v;
  …
}
void sub2() {
  int w, x;
  …
}
```

```
void sub1() {
  int v, w;
  …
}

void main() {
  int v, u;
  …
}
```



(The names in the stack cells indicate the program units of the variable declaration.)



End!
Mao nato!
Thank you!