Stacks and Translating High Level Code to Machine Instructions

Representation of computational expressions

- Common algebraic approach (infix)
  - Operators occur between inputs to operator, example algebraic expressions
    - $(a+b)\times c-d\text{^}3$
- Prefix Form:
  - Operators precede inputs, Similar to function calls, example
    - *Subtract(Multiply(Add(a,b),c),Power(d,3))*
- Postfix Form:
  - Operators follow inputs (Also known as Reverse Polish Notation), example
    - $a\ b+c\times d\ 3\ \text{^}\ -$
    - No parenthesis needed
    - Any infix expression can be converted to postfix form
    - Most natural for implementation with machine language

Stack

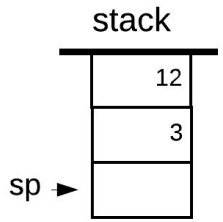
Stack operations:

- `push`: add an element at the stack's top
- `pop`: remove the top element
- `empty`: stack contains elements

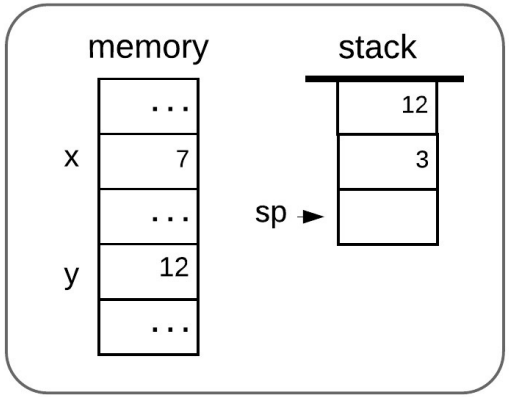
About Stacks:

- Also known as a LIFO buffer (Last In, First Out)
- Derived operation can include top which returns top of stack without pop operation
- Can be implemented easily with pointers and an array of memory

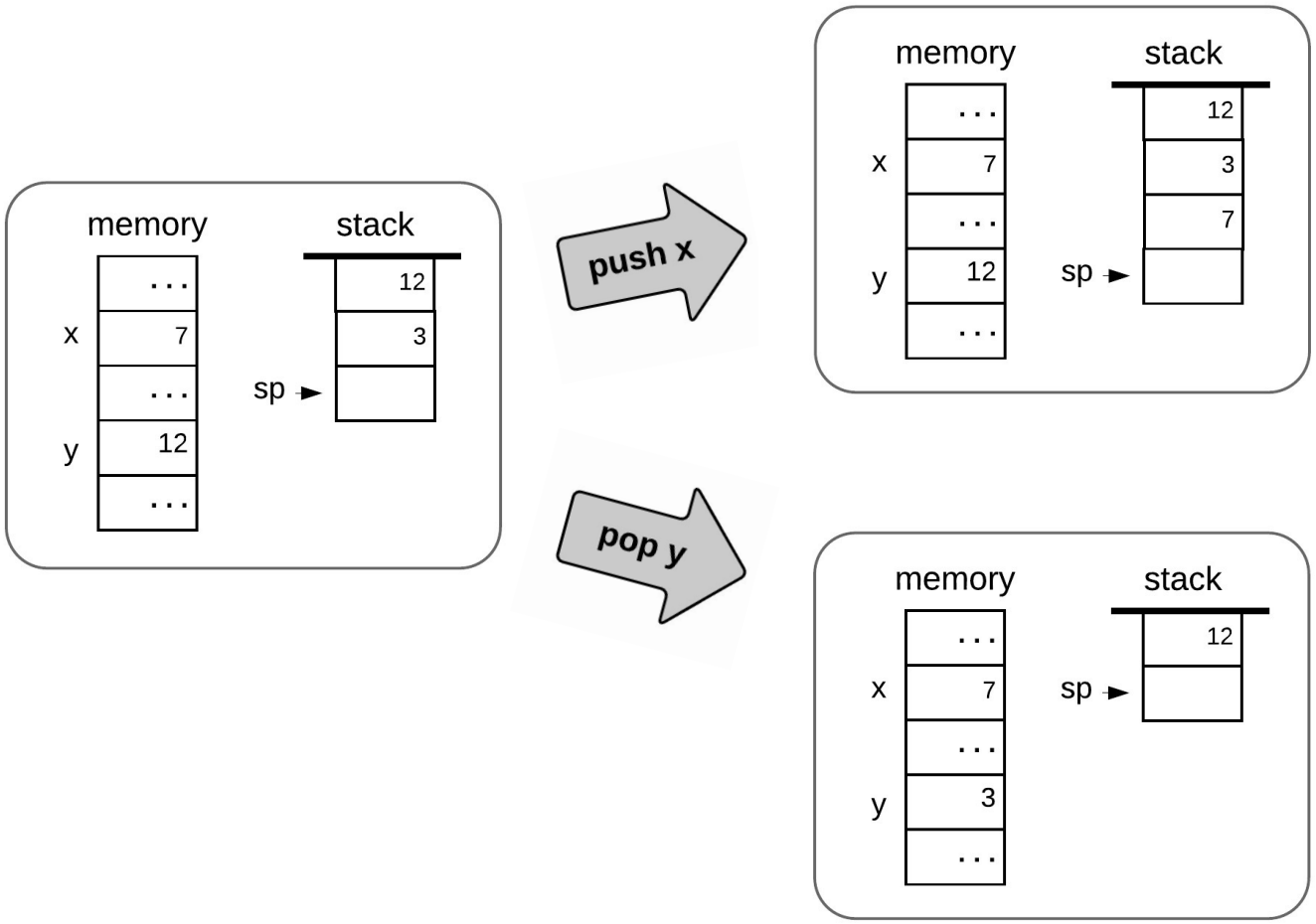
Stack



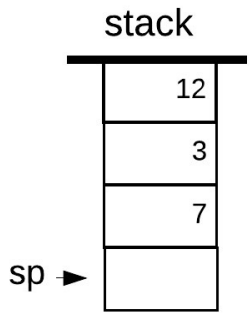
Stack



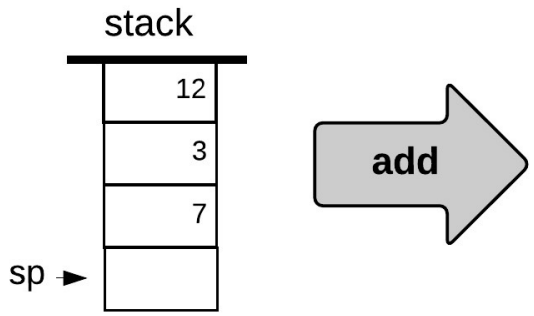
Stack


memory
stack
...
x
7
...
y
12
...
12
3
sp
push x
pop y
memory
stack
...
x
7
...
y
12
...
12
3
7
sp
memory
stack
...
x
7
...
y
3
...
12
sp

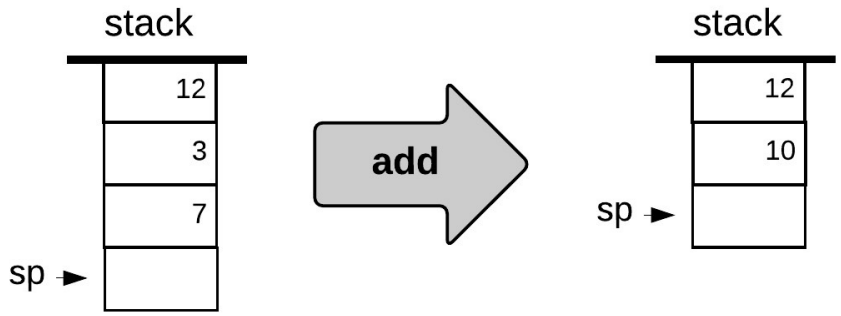
Stack arithmetic



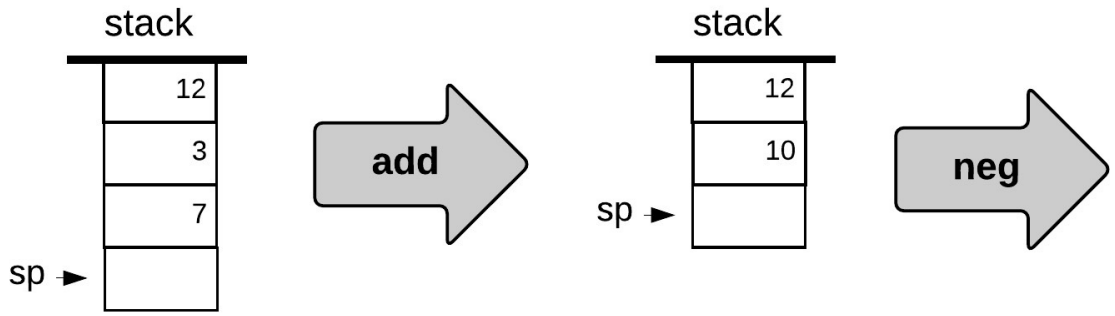
Stack arithmetic



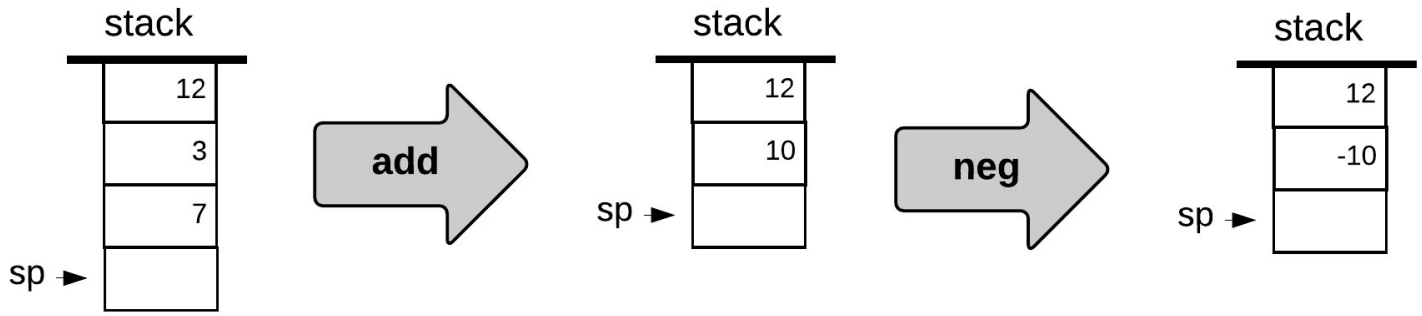
Stack arithmetic



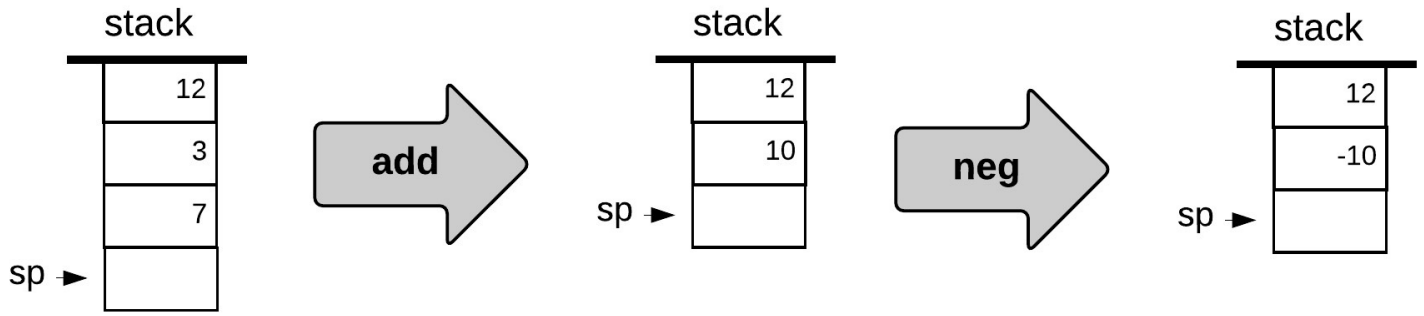
Stack arithmetic



Stack arithmetic



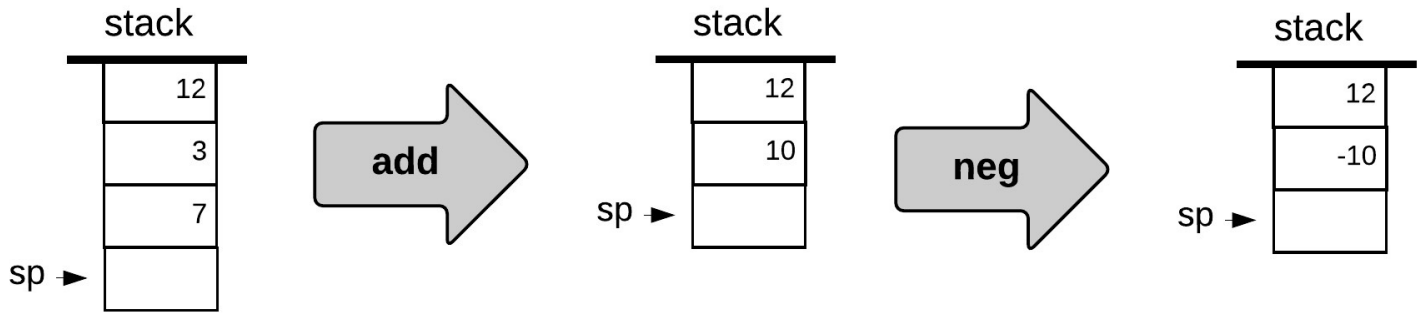
Stack arithmetic



Applying a function *f* on the stack:

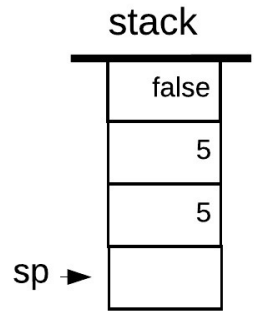
- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

Stack arithmetic

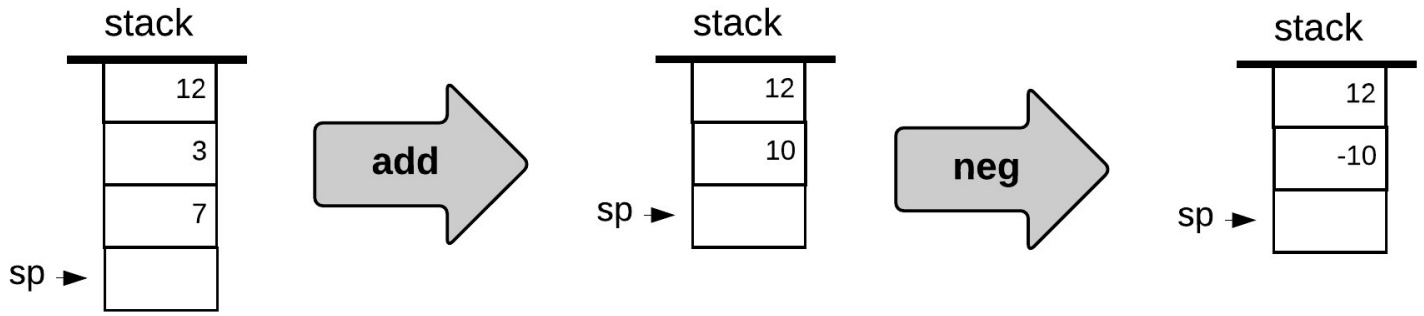


Applying a function *f* on the stack:

- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

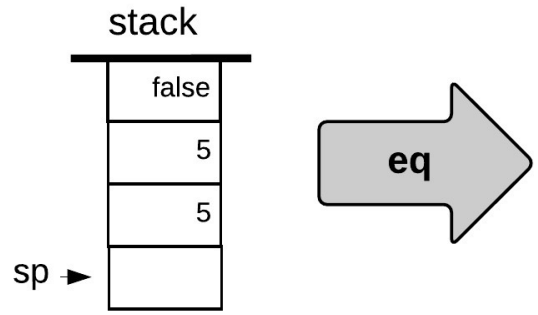


Stack arithmetic

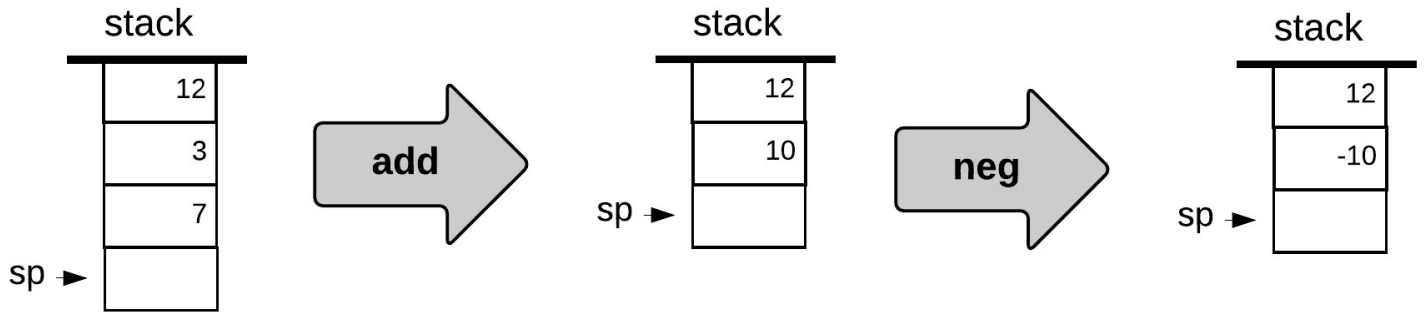


Applying a function *f* on the stack:

- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

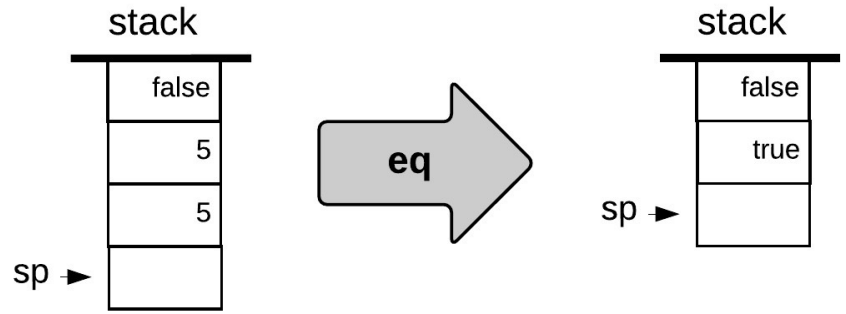


Stack arithmetic

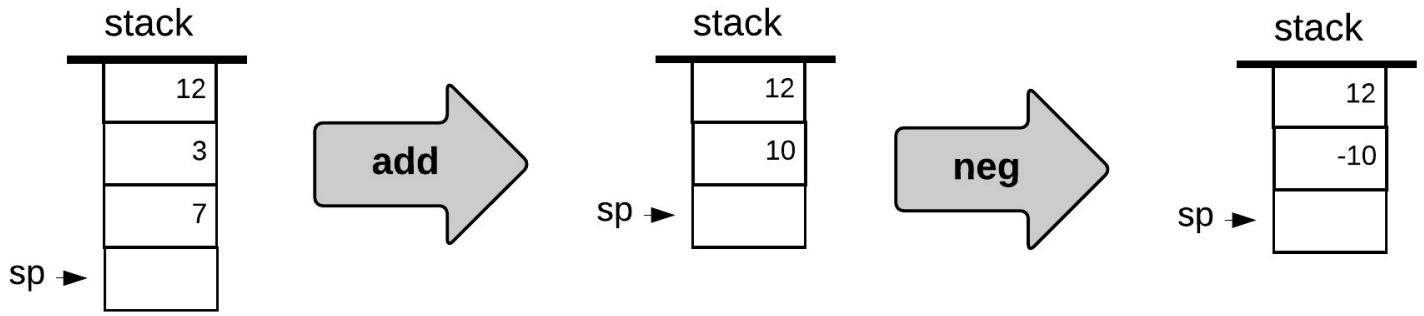


Applying a function *f* on the stack:

- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

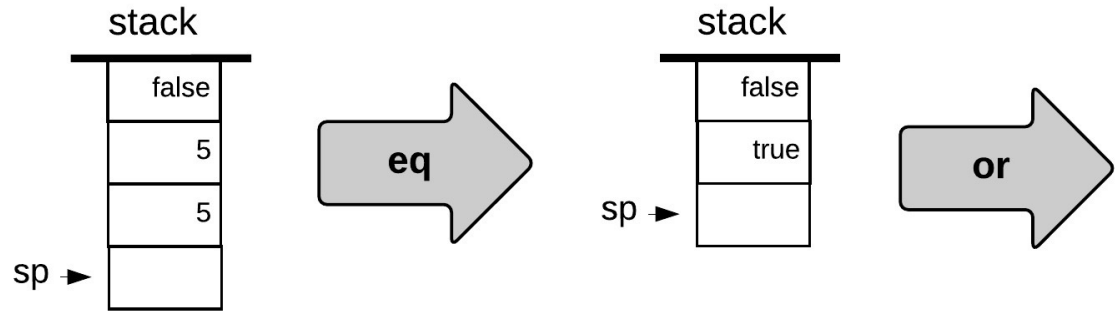


Stack arithmetic

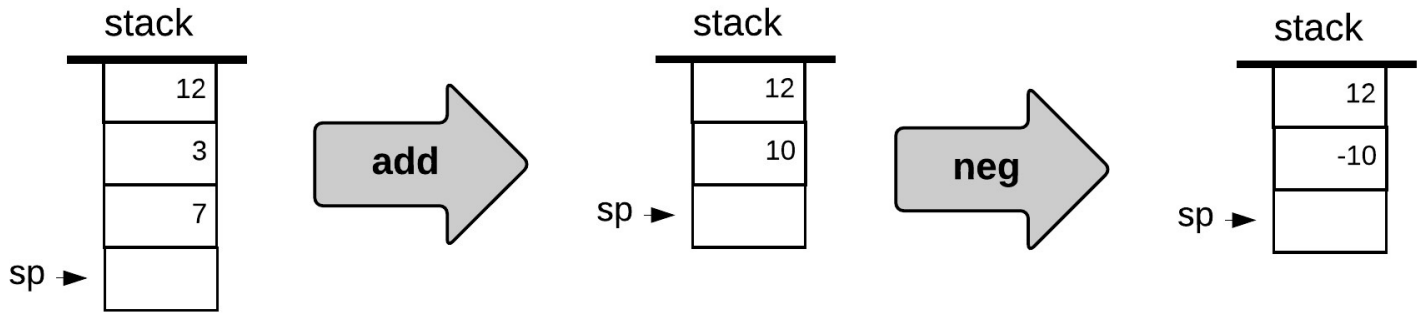


Applying a function *f* on the stack:

- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.

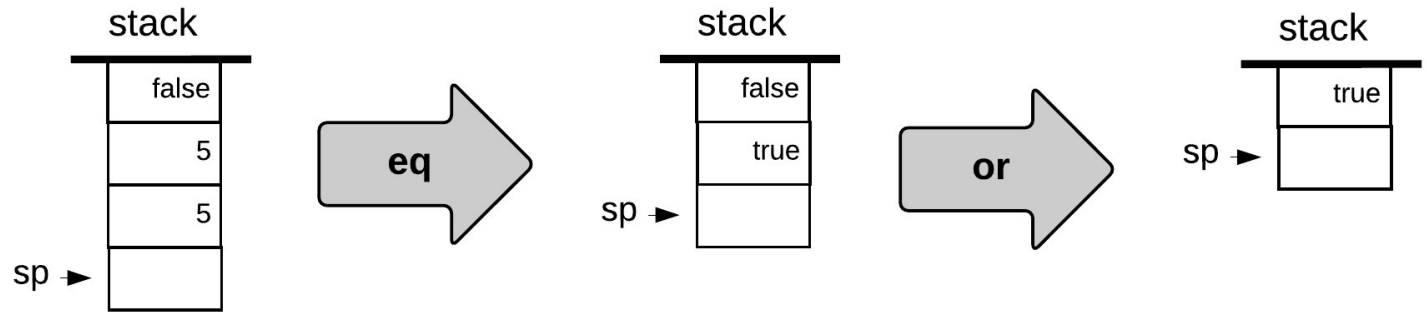


Stack arithmetic



Applying a function *f* on the stack:

- pops the argument(s) from the stack
- Computes *f* on the arguments
- Pushes the result onto the stack.



Stack arithmetic (big picture)

high-level

```
x = 17 + 19
```

lower-level

```
push 17
push 19
add
pop x
```

Abstraction / implementation

- The high-level language is an abstraction;
- It can be implemented by a stack machine.
- The stack machine is also an abstraction;
- It can be implemented by... Instructions/macros/or intermediate code/virtual machine

Arithmetic commands

VM code

```
// d=(2-x)+(y+9)
```

Arithmetic commands

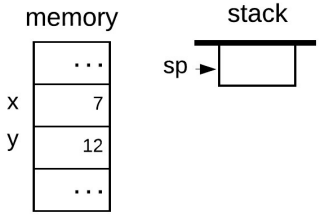
intermediate code

```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```

Arithmetic commands

intermediate code

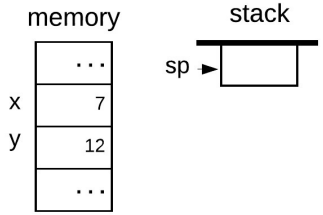
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

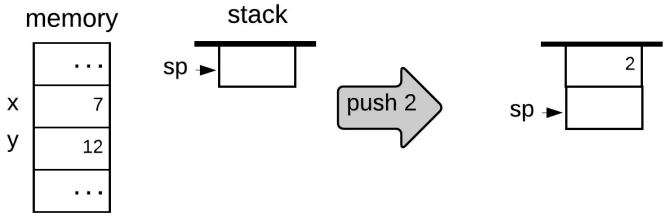
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

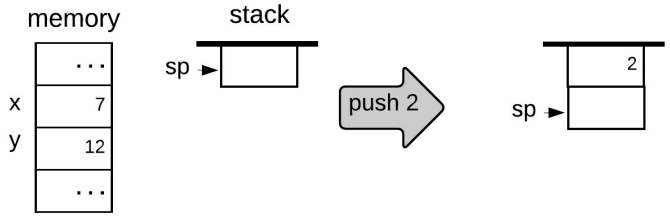
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

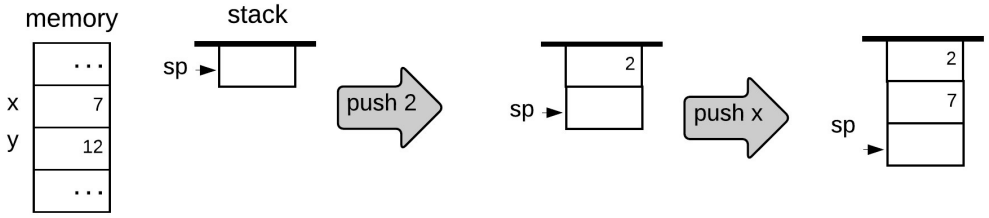
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

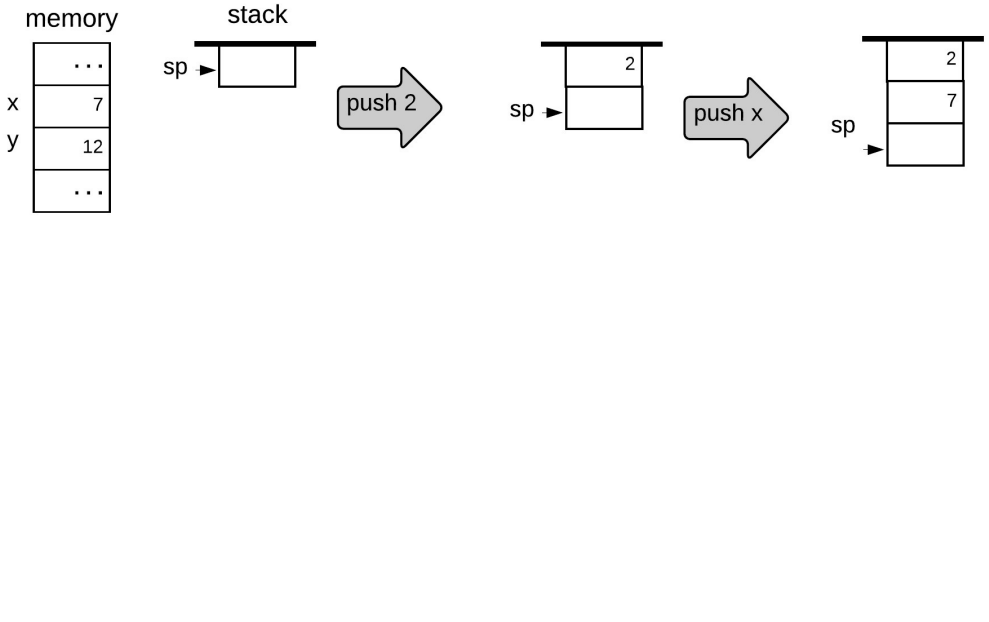
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

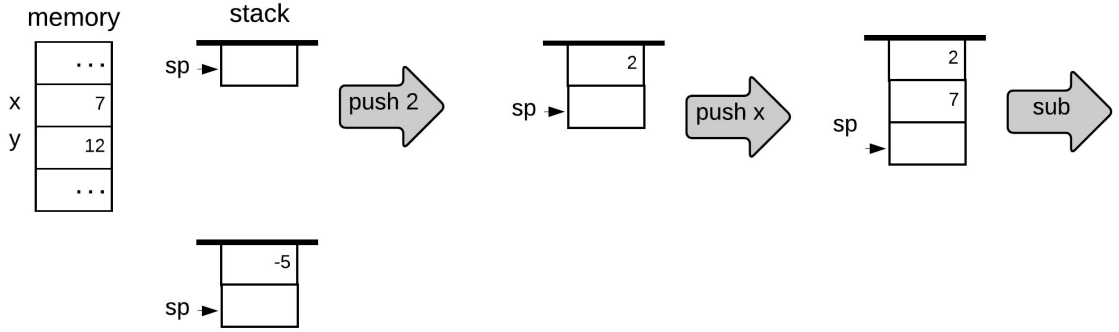
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

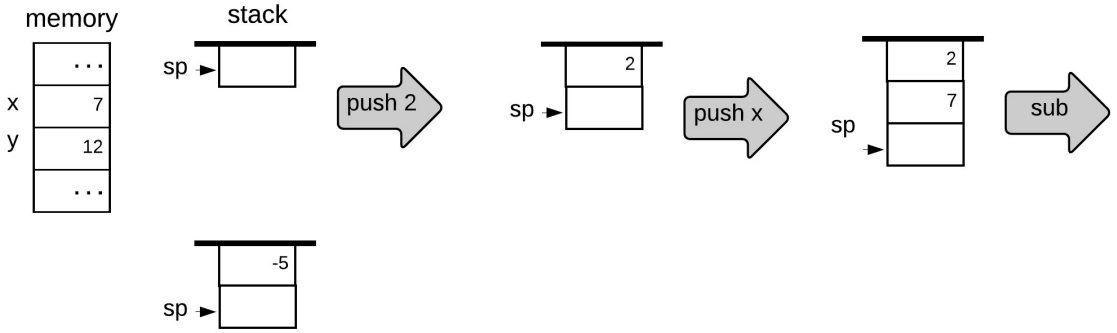
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intemediate code

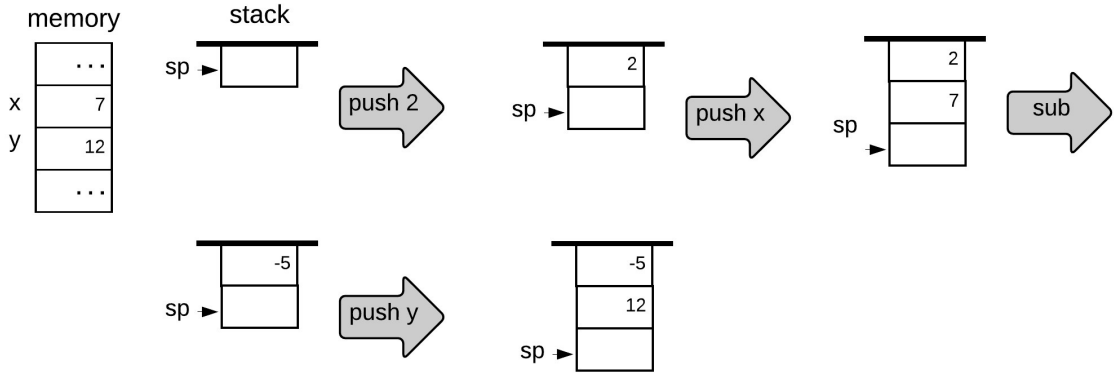
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

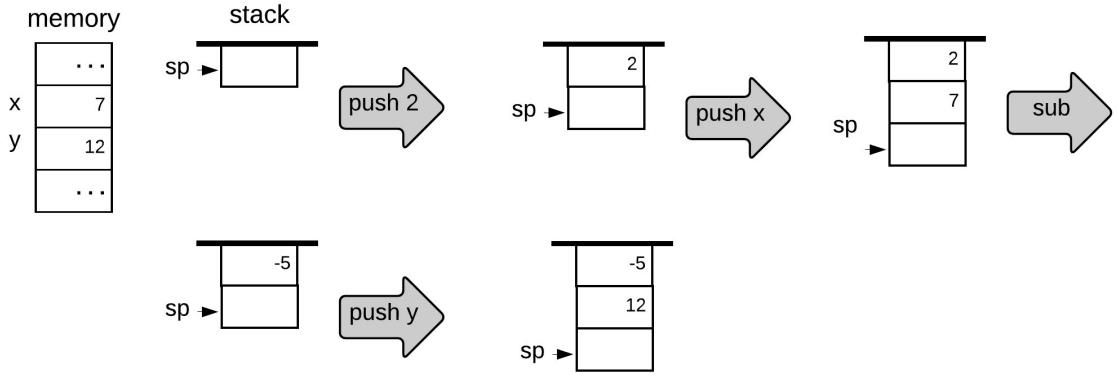
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

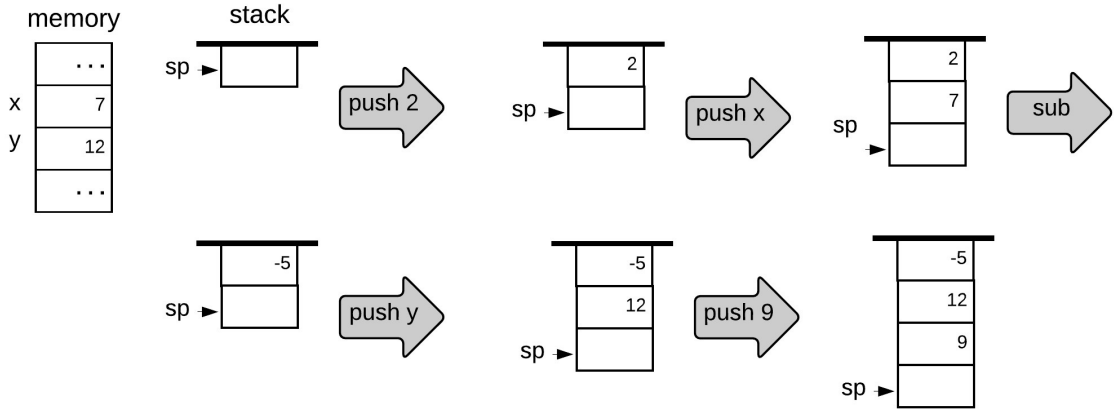
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

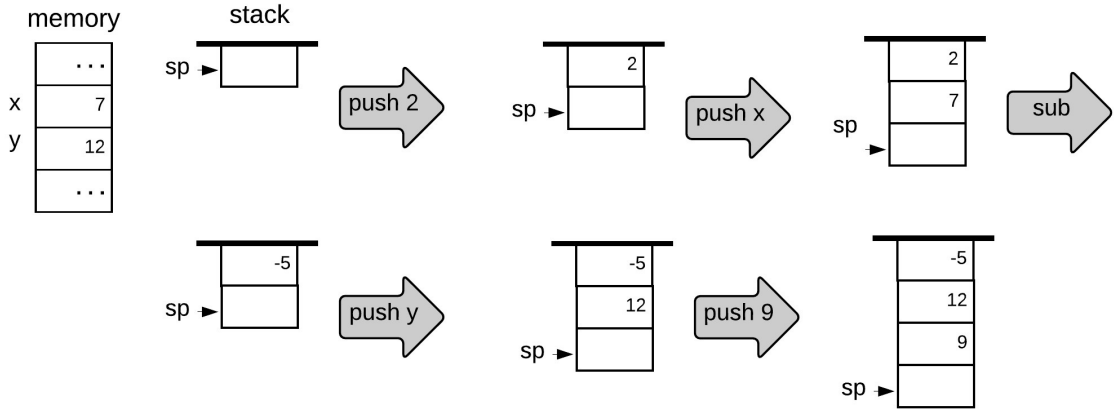
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

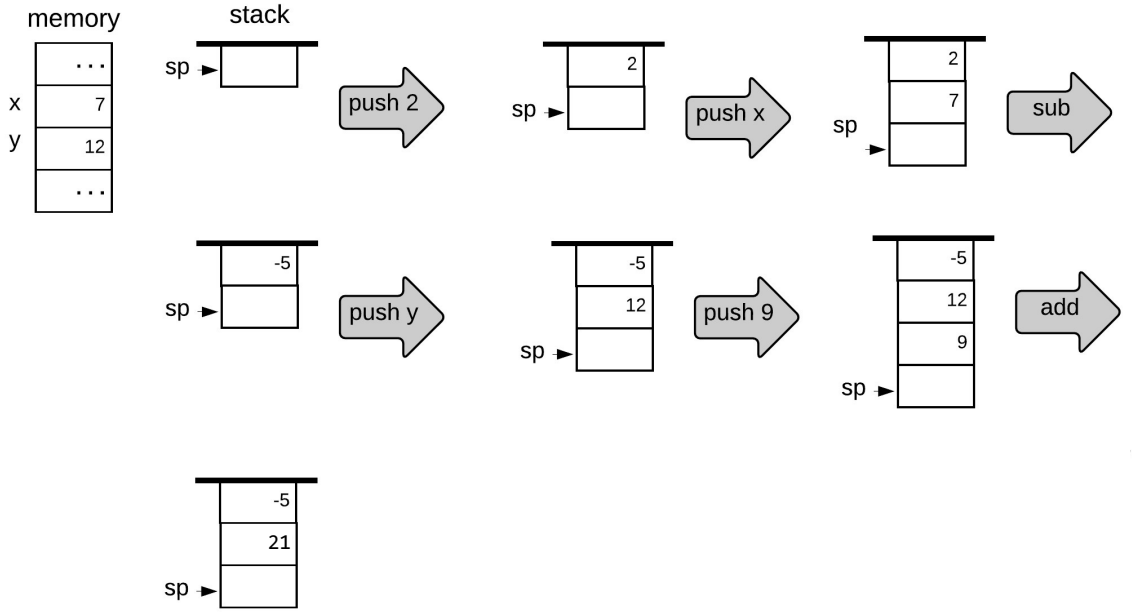
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

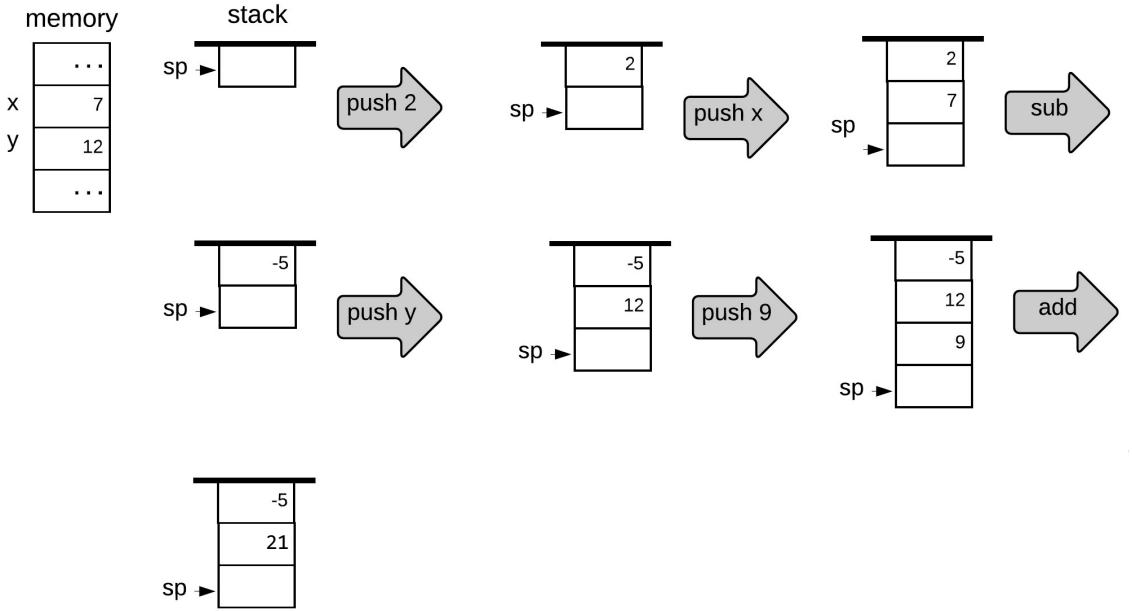
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

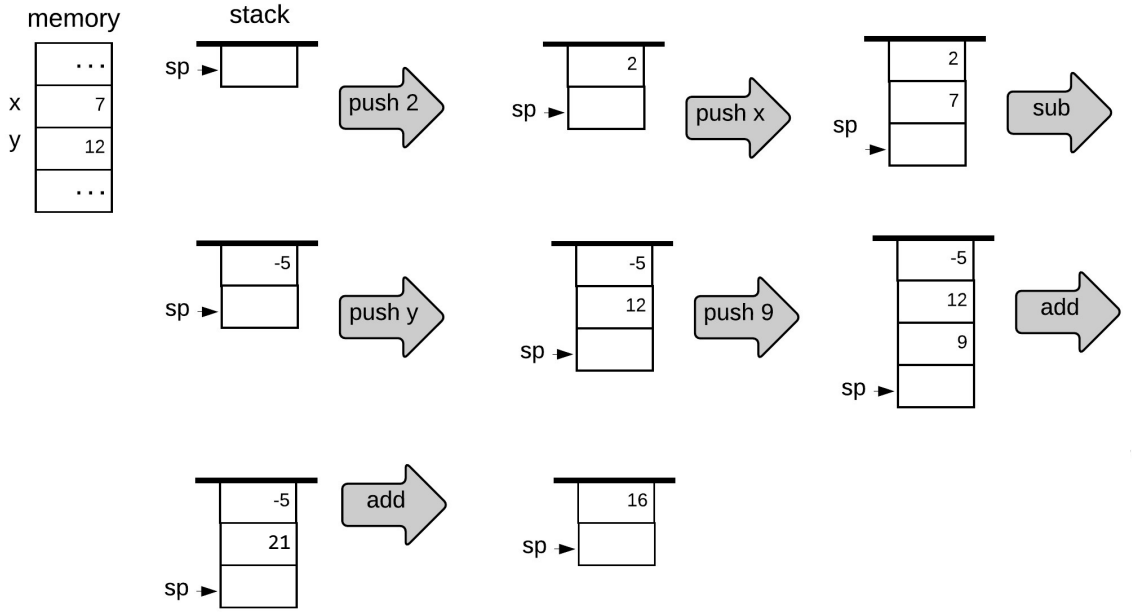
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

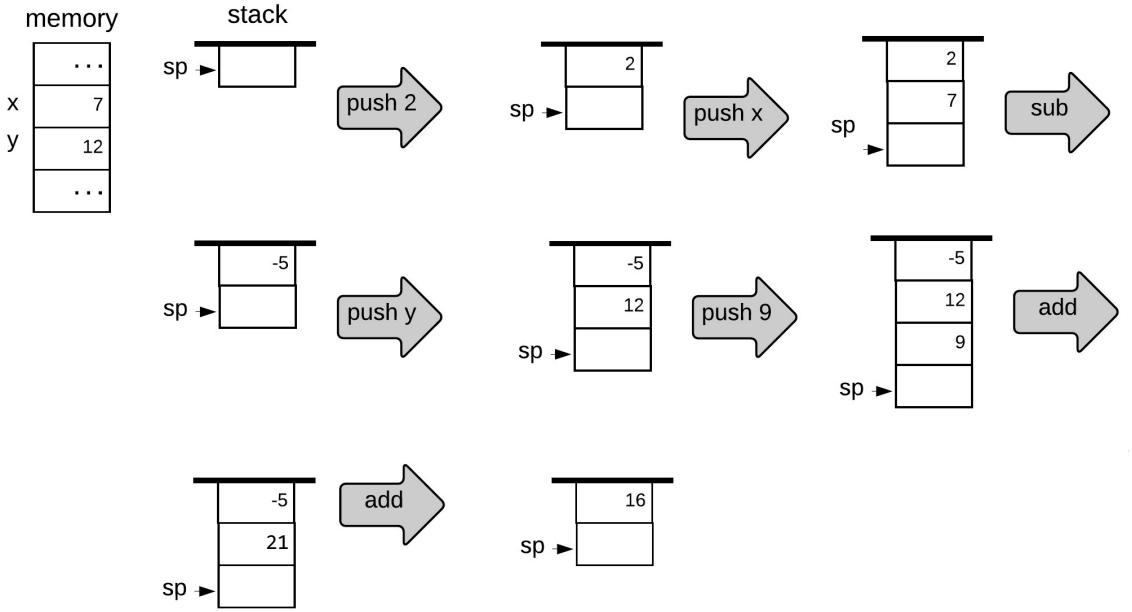
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

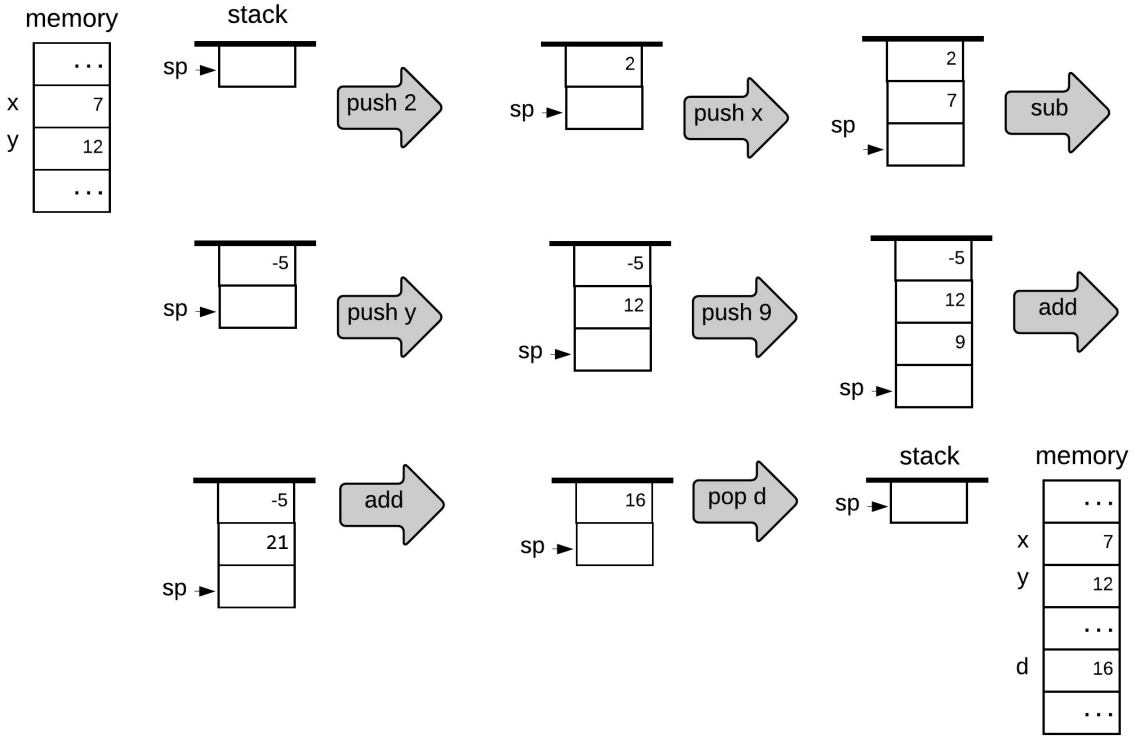
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

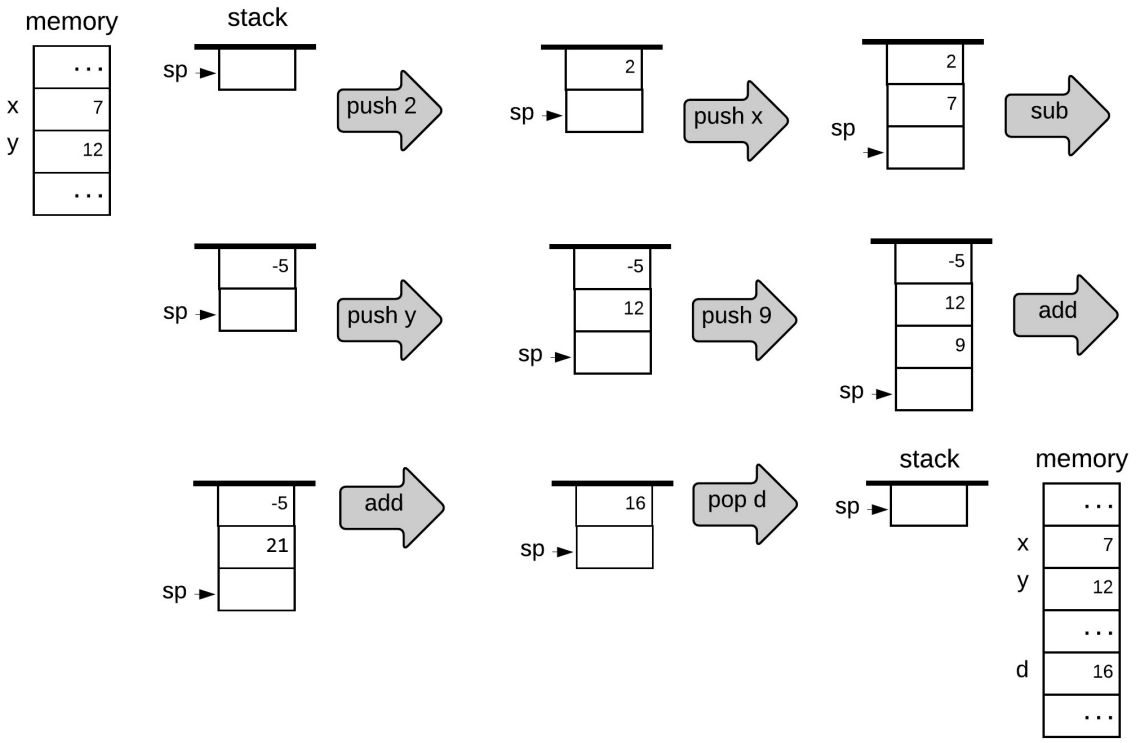
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Arithmetic commands

intermediate code

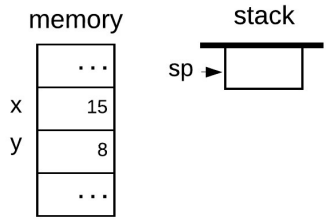
```
// d=(2-x)+(y+9)
push 2
push x
sub
push y
push 9
add
add
pop d
```



Logical commands

intermediate code

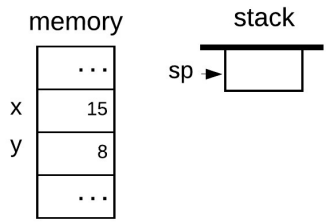
```
// (x<7) or (y==8)
```



Logical commands

intermediate code

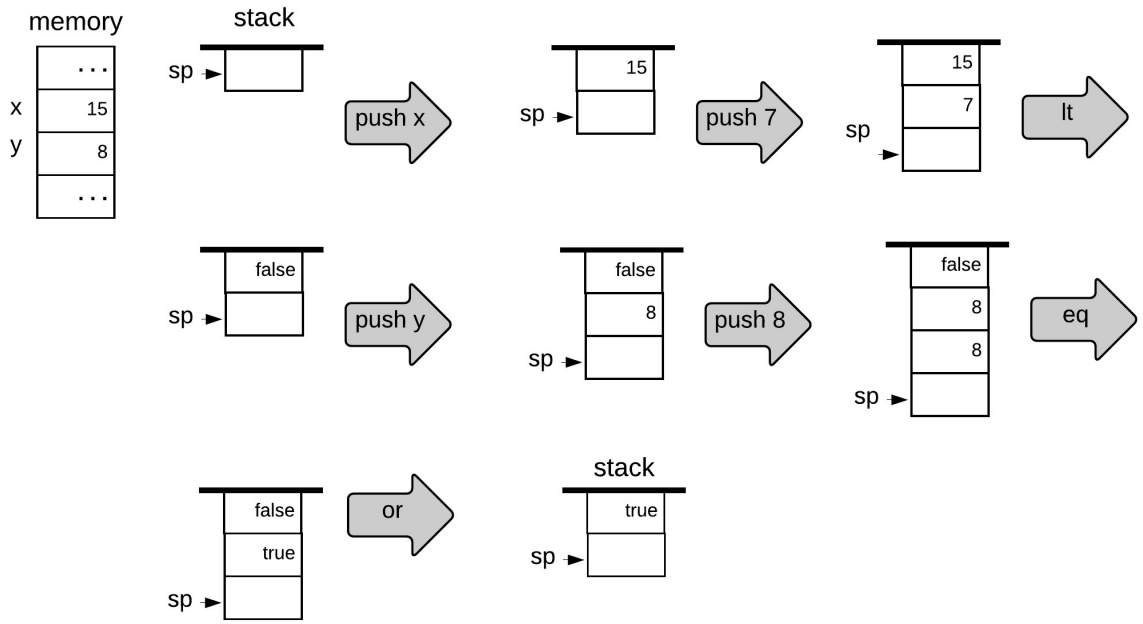
```
// (x<7) or (y==8)
push x
push 7
lt
push y
push 8
eq
or
```



Logical commands

intermediate code

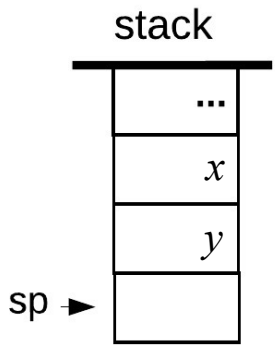
```
// (x<7) or (y==8)
push x
push 7
lt
push y
push 8
eq
or
```



Arithmetic / Logical commands

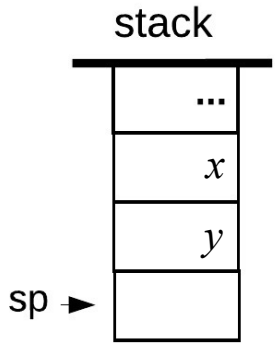
Arithmetic / Logical commands

Command	Return value	Return value
add	$x + y$	integer
sub	$x - y$	integer
neg	$-y$	integer



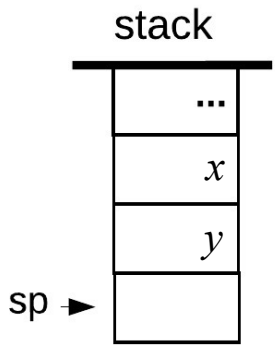
Arithmetic / Logical commands

Command	Return value	Return value
add	$x + y$	integer
sub	$x - y$	integer
neg	$-y$	integer
eq	$x==0$	boolean
gt	$x > y$	boolean
lt	$x < y$	boolean



Arithmetic / Logical commands

Command	Return value	Return value
add	$x + y$	integer
sub	$x - y$	integer
neg	$-y$	integer
eq	$x==0$	boolean
gt	$x > y$	boolean
lt	$x < y$	boolean
and	x and y	boolean
or	x or y	boolean
not	not x	boolean



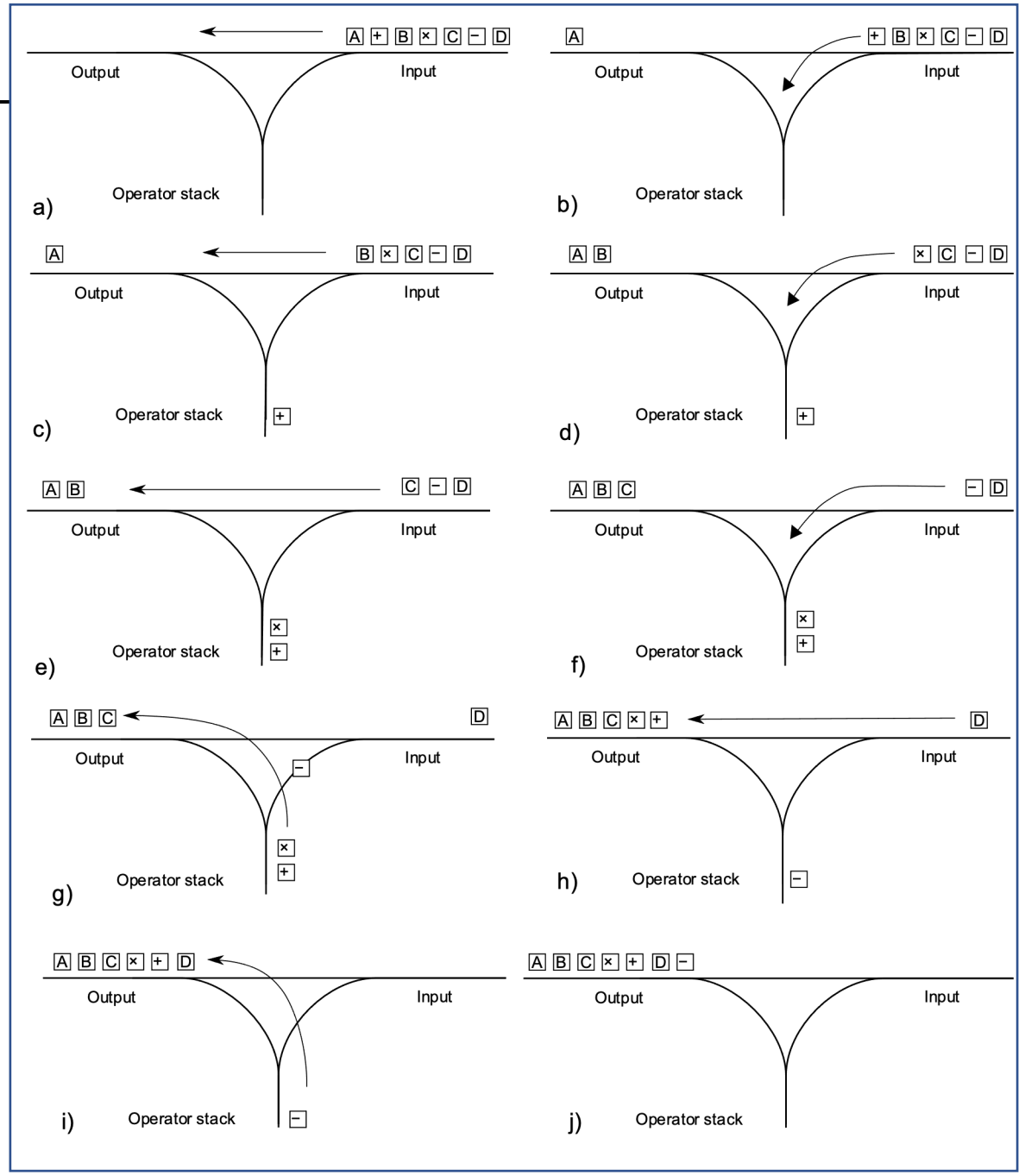
Observation: Any arithmetic or logical expression can be expressed and evaluated by applying some sequence of the above operations on a stack.

Converting infix to postfix

- Shunting-yard algorithm can convert infix expressions to postfix form
  - Stack based algorithm
  - If a literal (variable or number) is next, then output a push operation
  - While the current operator is lower or equal precedence to top of stack, pop operator from stack and output operator to postfix notation
  - When at the end of the processing the infix expression pop remaining operators from stack and put add them t the end of the postfix expression.
  - Similar logic can be used with functions or parenthesis

Shunting Yard

- Analogous to sorting railroad cars
- To change the order of cars some cars are shunted to a second line
- If done cleverly, this approach can be used to restructure infix expressions to postfix expressions



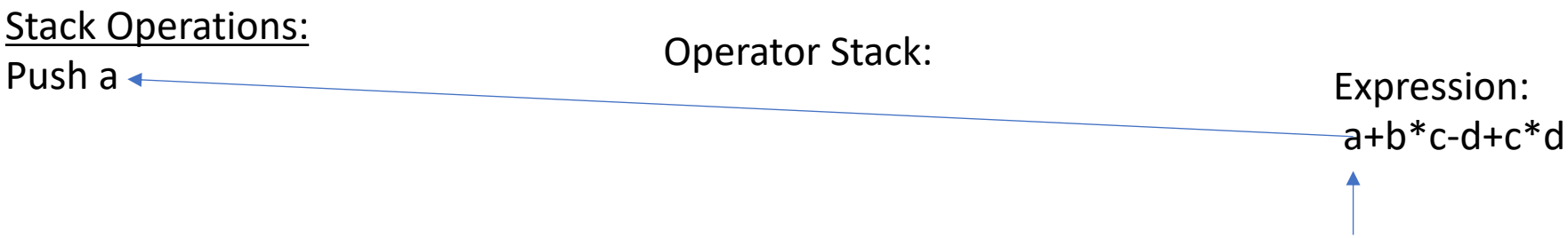
Shunting-Yard: Simple Example

Stack Operations:

Operator Stack:

Expression:
a+b*c-d+c*d

Shunting-Yard: Simple Example

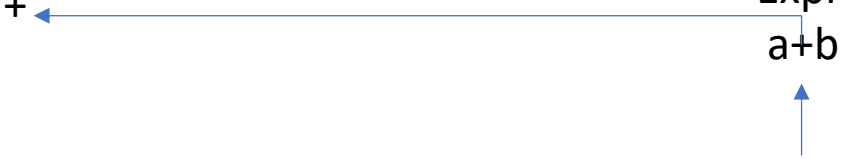


Shunting-Yard: Simple Example

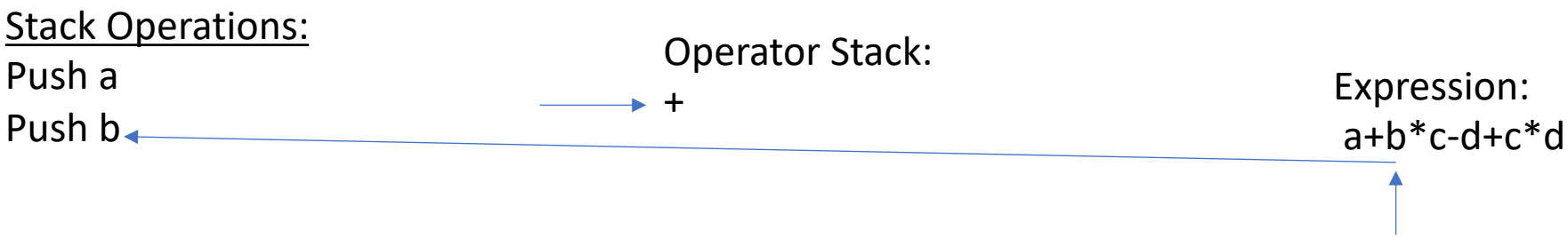
Stack Operations:
Push a

Operator Stack:
+

Expression:
a+b*c-d+c*d

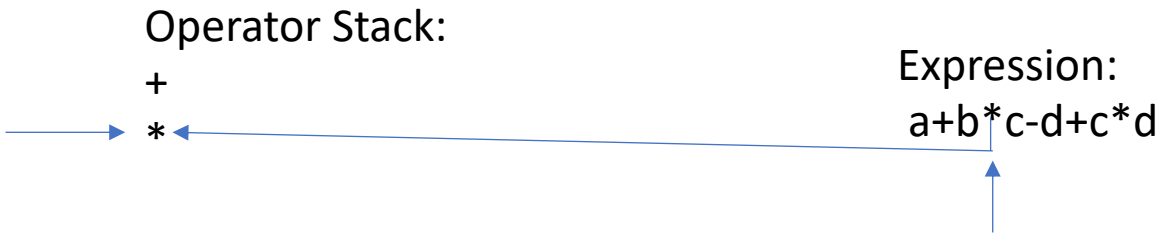


Shunting-Yard: Simple Example

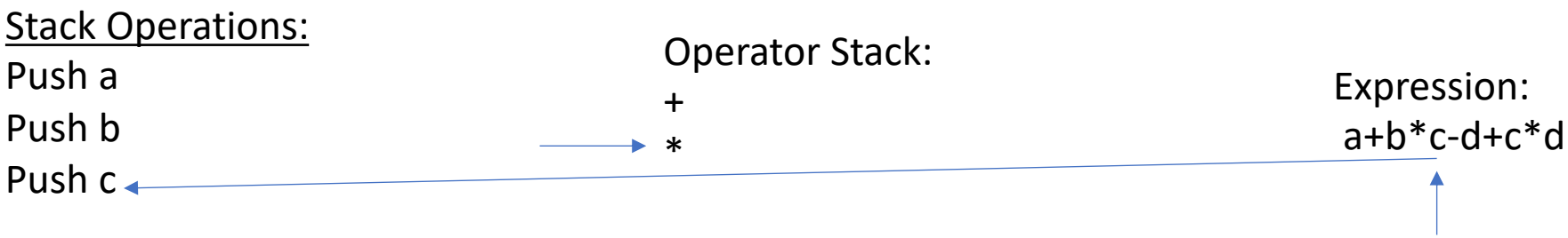


Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b



Shunting-Yard: Simple Example

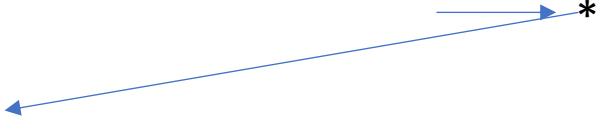


Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b
Push c
Multiply

Operator Stack:
+
*

Expression:
a+b*c-d+c*d



Shunting-Yard: Simple Example

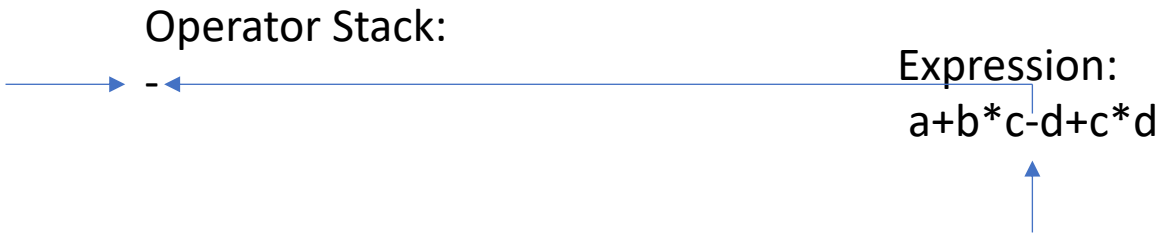
Stack Operations:
Push a
Push b
Push c
Multiply
Add

Operator Stack:
+

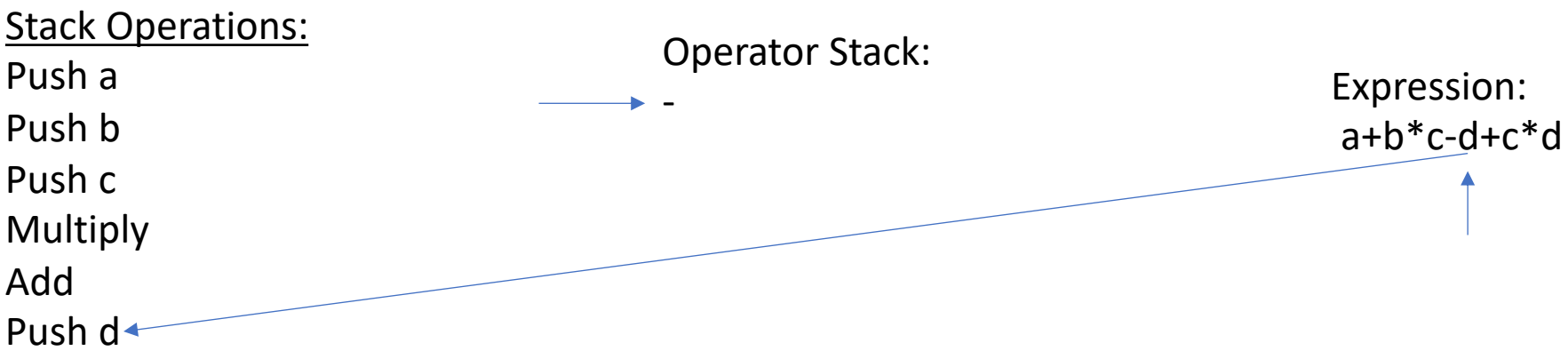
Expression:
a+b*c-d+c*d

Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b
Push c
Multiply
Add



Shunting-Yard: Simple Example

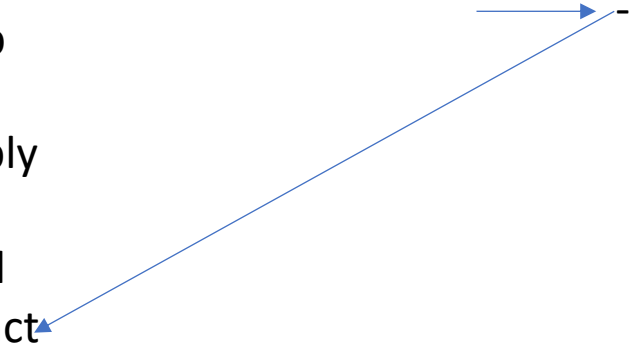


Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b
Push c
Multiply
Add
Push d
Subtract

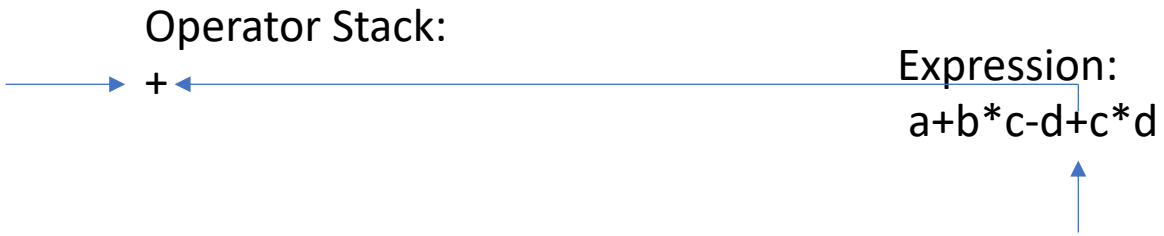
Operator Stack:
-

Expression:
a+b*c-d+c*d

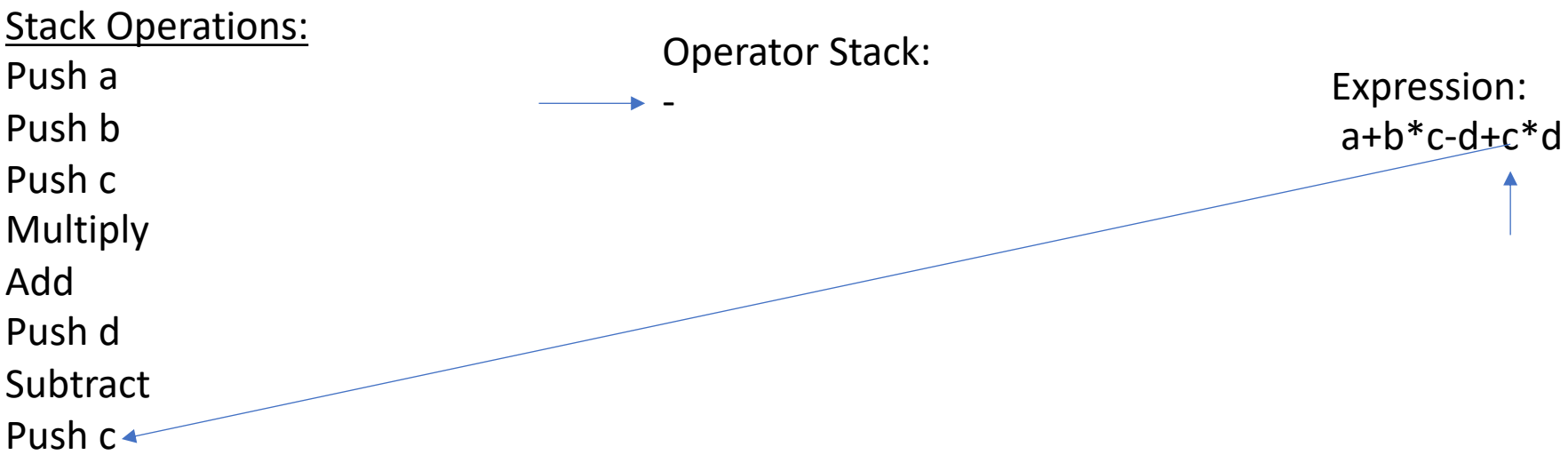


Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b
Push c
Multiply
Add
Push d
Subtract

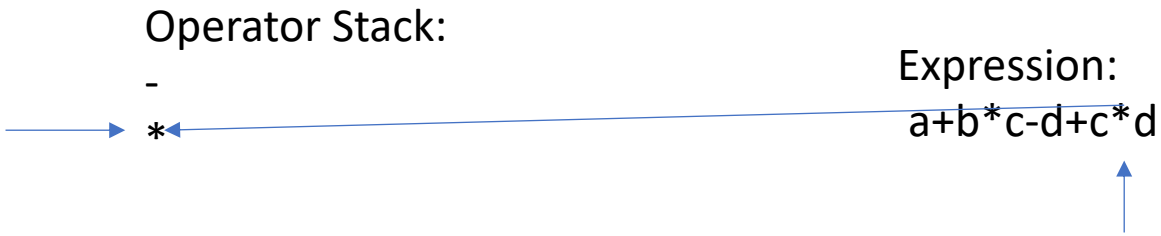


Shunting-Yard: Simple Example



Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b
Push c
Multiply
Add
Push d
Subtract
Push c

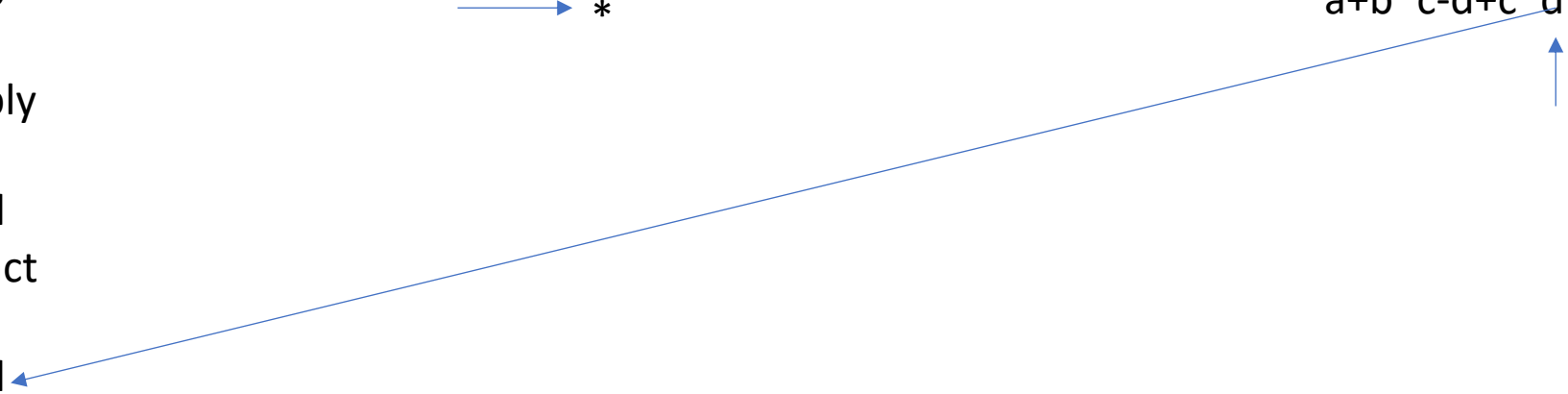


Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b
Push c
Multiply
Add
Push d
Subtract
Push c
Push d

Operator Stack:
-
*

Expression:
a+b*c-d+c*d



Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b
Push c
Multiply
Add
Push d
Subtract
Push c
Push d
Multiply

Operator Stack:
-
*

Expression:
a+b*c-d+c*d

Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b
Push c
Multiply
Add
Push d
Subtract
Push c
Push d
Multiply
Subtract

Operator Stack:
-

Expression:
a+b*c-d+c*d

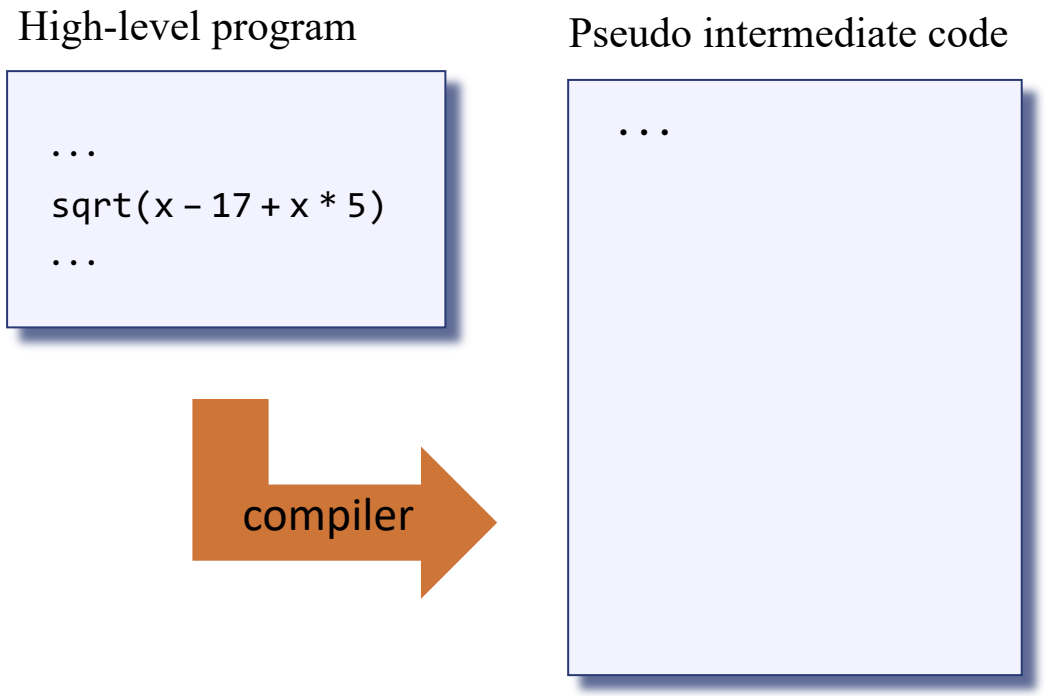
Shunting-Yard: Simple Example

Stack Operations:
Push a
Push b
Push c
Multiply
Add
Push d
Subtract
Push c
Push d
Multiply
Subtract

Operator Stack:

Expression:
a+b*c-d+c*d

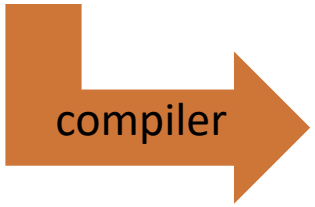
Functions in the intermediate language



Functions in the intermediate language

High-level program

```
...
sqrt(x – 17 + x * 5)
...
```



Pseudo intermediate code

```
...
push x
push 17
sub
push x
push 5
call Math.multiply
add
call Math.sqrt
...
```

The intermediate language features:

- primitive operations (fixed): `add`, `sub`, ...
- abstract operations (extensible): `multiply`, `sqrt`, ...

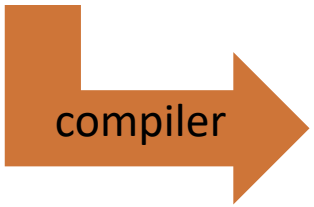
Programming style:

- Applying a primitive operator or calling a function have the same look-and-feel.

Functions in the intermediate language: defining

High-level program

```
// Returns x * y
int mult(int x, int y) {
   int sum = 0;
   int n = 1;
   // sum = sum + x, y times
   while !(n > y) {
      sum += x;
      n++;
   }
   return sum;
}
```



Pseudo i-code

```
function mult(x,y)
   push 0
   pop sum
   push 1
   pop n
label LOOP
   push n
   push y
   gt
   if-goto END
   push sum
   push x
   add
   pop sum
   push n
   push 1
   add
   pop n
   goto LOOP
label END
   push sum
   return
```

Functions in the i-code: executing

```
// Computes 3+5*8
 0 function main 0
 1   push constant 3
 2   push constant 8
 3   push constant 5
 4   call mult 2
 5   add
 6   return
```

caller

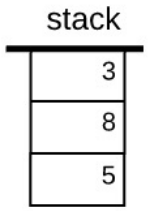
```
// Computes the product of two given arguments
 0 function mult 2
 1   push constant 0
 2   pop local 0
 3   push constant 1
 4   pop local 1
 5 label LOOP
 6   push local 1
 7   push argument 1
     //... computes the product into local 0
19 label END
20   push local 0
21   return
```

(same code as previous slide, with line numbers, for easy reference)

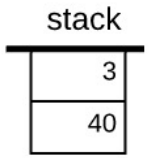
callee

main view:

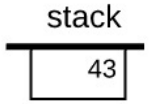
after line 3 is executed:



after line 4 is executed:



after line 5 is executed:



Magic!

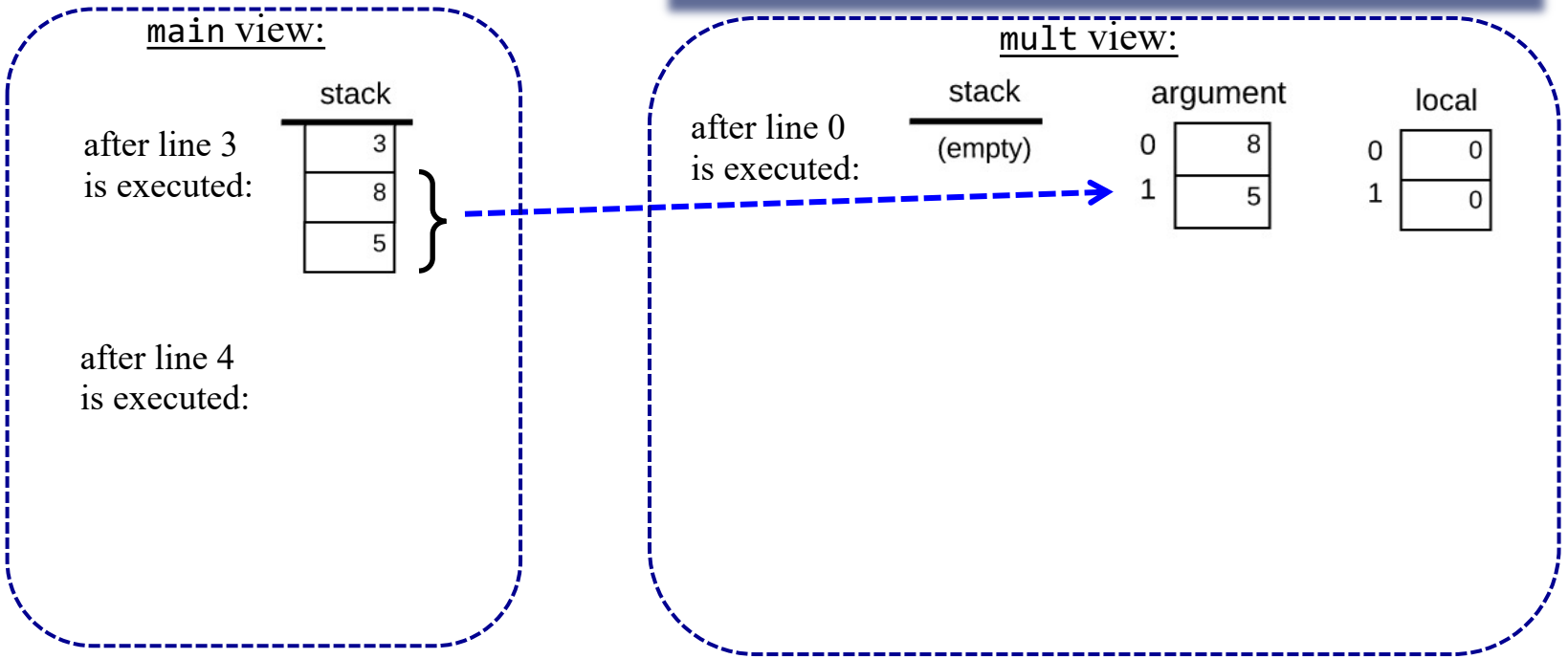
Functions in the i-code language: executing

```
// Computes 3+5*8
 0 function main 0
 1   push 3
 2   push 8
 3   push 5
 4   call mult 2
 5   add
 6   return
```

caller

```
// Computes the product of two given arguments
 0 function mult 2
 1   push 0
 2   pop local sum
 3   push constant 1
 4   pop local 1
 5 label LOOP
 6   push local 1
 7   push argument 1
     //... computes the product into local 0
19 label END
20   push local 0
21   return
```

callee



Functions in the i-code language: executing

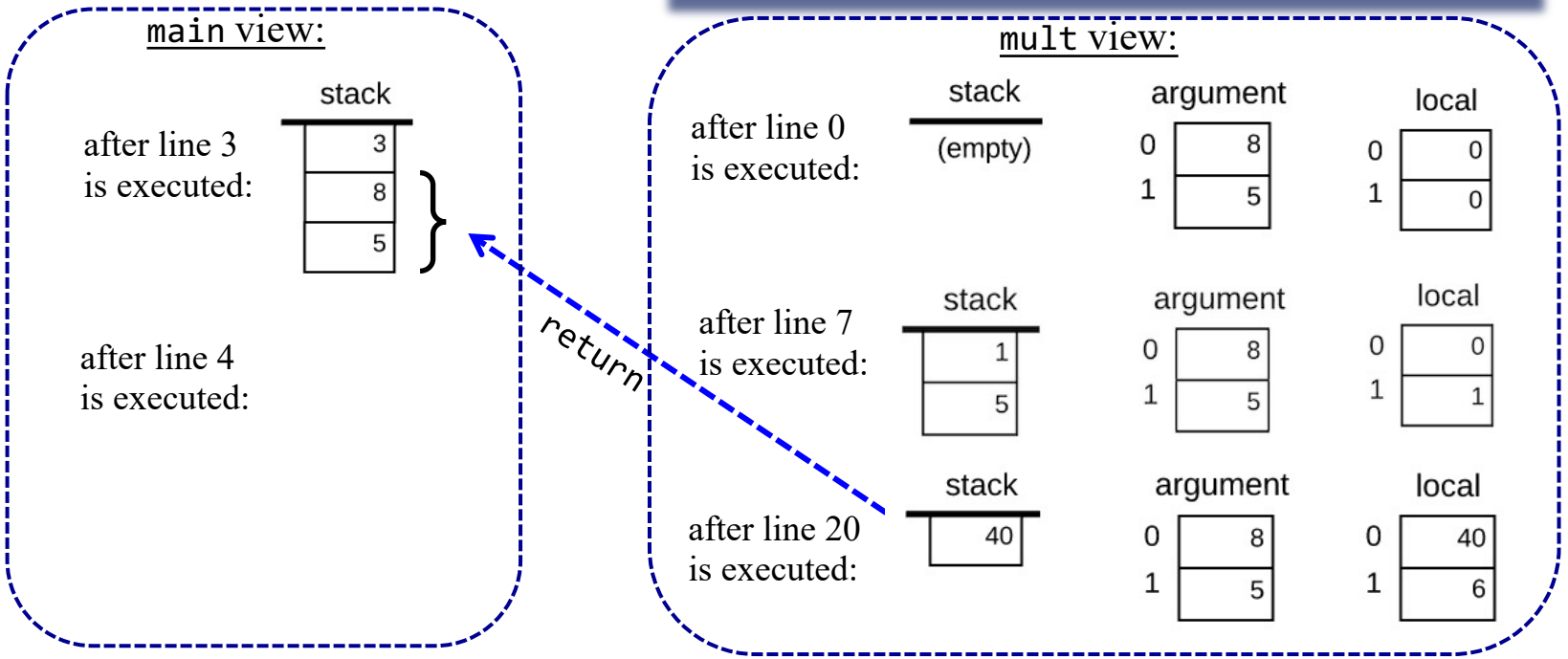
```
// Computes 3+5*8
 0 function main 0
 1   push constant 3
 2   push constant 8
 3   push constant 5
 4   call mult 2
 5   add
 6   return
```

caller

```
// Computes the product of two given arguments
 0 function mult 2
 1   push constant 0
 2   pop local 0
 3   push constant 1
 4   pop local 1
 5 label LOOP
 6   push local 1
 7   push argument 1
     //... computes the product into local 0
19 label END
20   push local 0
21   return
```

(same code as previous slide, with line numbers, for easy reference)

callee



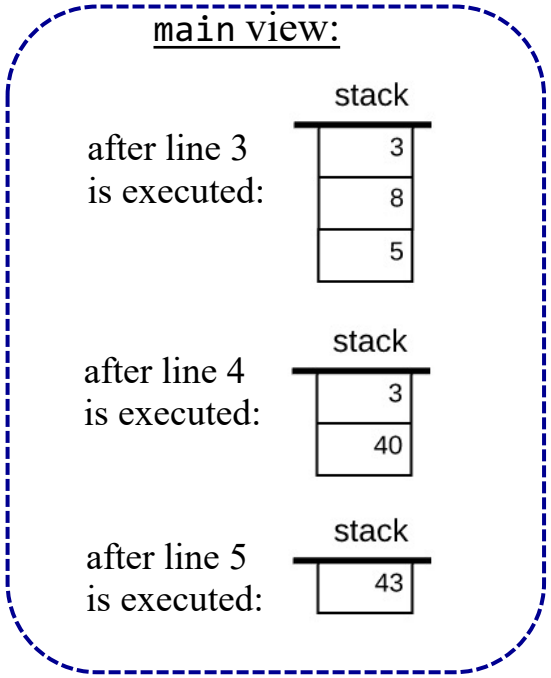
Functions in the i-code language: executing

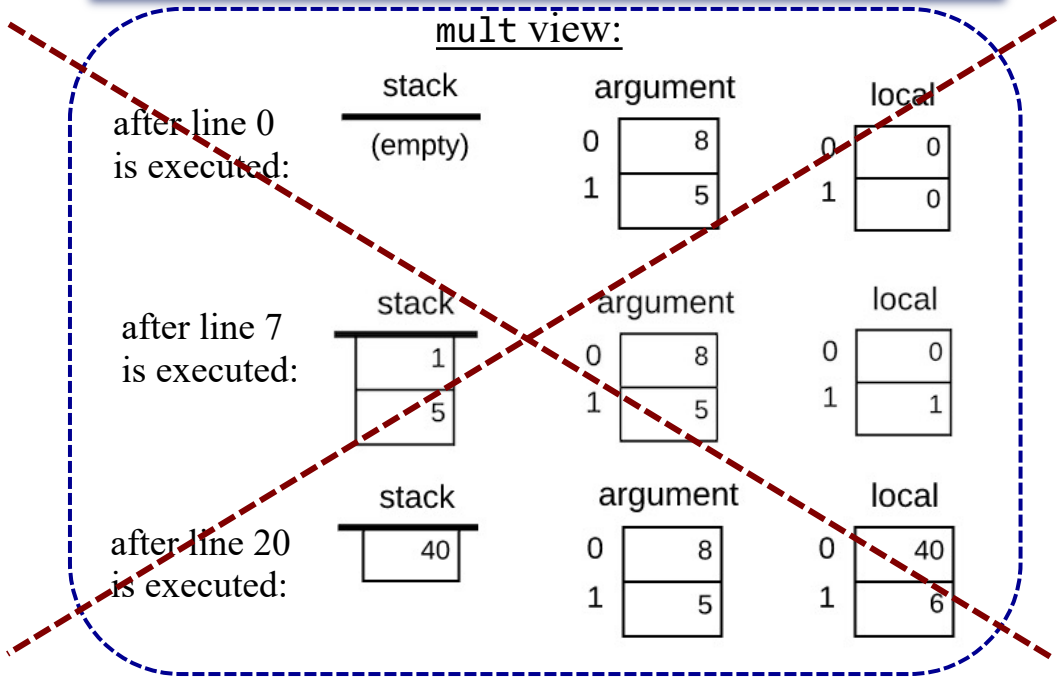
```
// Computes 3+5*8
 0 function main 0
 1   push constant 3
 2   push constant 8
 3   push constant 5
 4   call mult 2
 5   add
 6   return
```

caller

```
// Computes the product of two given arguments
 0 function mult 2
 1   push constant 0
 2   pop local 0
 3   push constant 1
 4   pop local 1
 5 label LOOP
 6   push local 1
 7   push argument 1
     //... computes the product into local 0
19 label END
20   push local 0
22   return
```

callee





Functions in the i-code: implementation

```
// Computes 3+5*8
function main 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  return
```

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Implementation

How to orchestrate the drama just described?

We can write low-level code that manages the parameter passing, the saving and re-instating of function states, etc.

This task can be realized by writing code that...

- Handles the command `call`
- Handles the command `function`
- Handles the command `return`.

Functions in the i-code: implementation

```
// Computes 3+5*8
function main 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  return
```

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling `call`:

Functions in the i-code: implementation

```
// Computes 3+5*8
function main 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  return
```

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling `call`:

- *Note: Responsibilities are divided between the caller and callee, this division is not unique and is a matter of convention*
- Determine the return address within the caller's code;
- Push the return address onto the stack;
- Assume arguments have been pushed on the stack before call;
- Jump to execute the callee.

Functions in the i-code: implementation

```
// Computes 3+5*8
function main 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  return
```

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling `function`:

- Allocate space on the stack for the local variables of the callee;
- Save the state that will overlap with caller
  - This will be handled by stack frames (more to be discussed later)

Functions in the i-code: implementation

```
// Computes 3+5*8
function main 0
  push constant 3
  push constant 8
  push constant 5
  call mult 2
  add
  return
```

caller

```
// Computes the product of two given arguments
function mult 2
  push constant 0
  pop local 0
  push constant 1
  pop local 1
label LOOP
  push local 1
  push argument 1
  //... computes the product into local 0
label END
  push local 0
  return
```

callee

Handling `return`:

(a function always ends by pushing a return value on the stack)

- Return the *return value* to the caller;
- Recycle the memory resources used by the callee;
- Reinstate the caller's stack and memory segments;
- Jump to the return address in the caller's code.

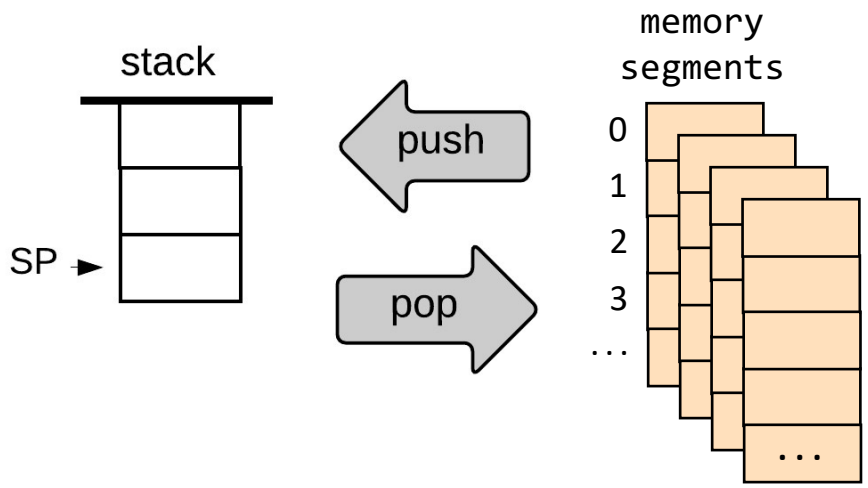
STOP HERE

Function execution

- A computer program typically consists of many functions
- At any given point of time, only a few functions are executing
- Calling chain: `foo > bar > sqrt > ...`

For each function in the calling chain during run-time, we must maintain the function's *state*

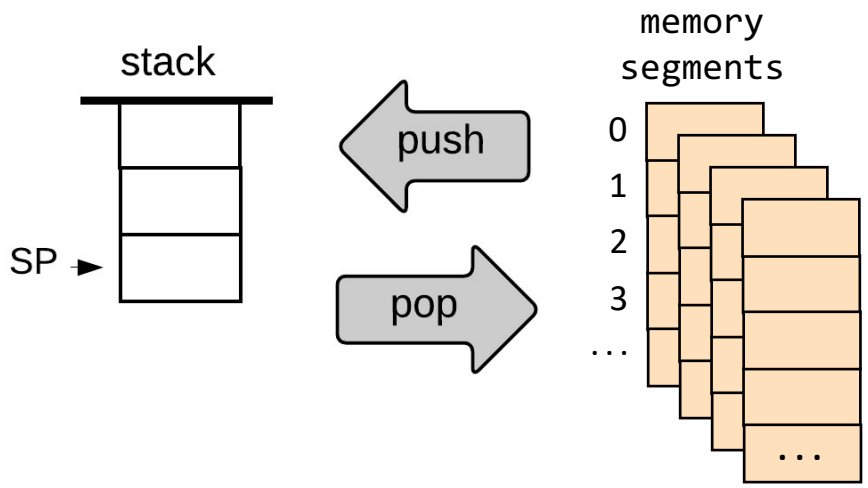
The function's state



<u>During run-time:</u>

- Each function uses a working stack + memory segments

The function's state



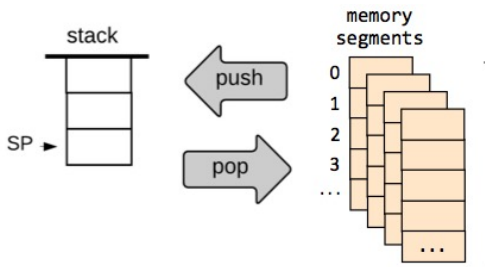
During run-time:

- Each function uses a working stack + memory segments
- The working stack and some of the segments should be:
  - Created when the function starts running,
  - Maintained as long as the function is executing,
  - Recycled when the function returns.

The function's state

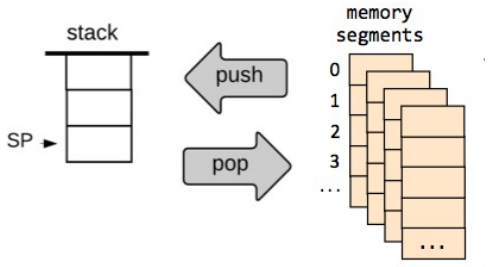
```
function main 0
   push constant 3
   push constant 8
   push constant 5
   call mult 2
   add
   return
```

caller




```
function mult 2
   push constant 0
   pop local 0
   ...
label LOOP
   push local 1
   //... computes the product
label END
   push local 0
   return
```

callee

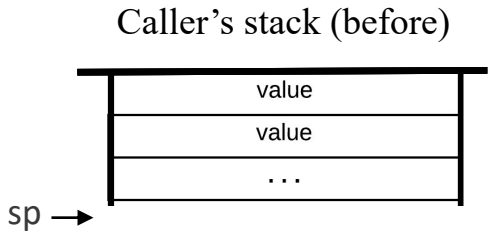


Challenge:

- Maintain the states of all the functions up the calling chain
- Can be done by using a single *global stack*.

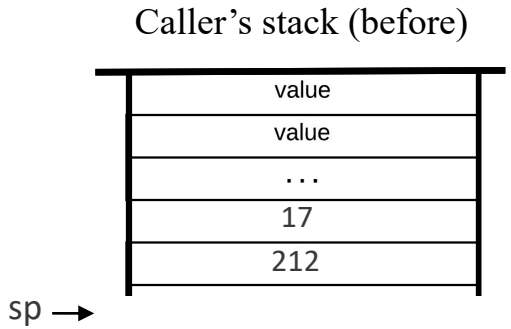
Function call and return: abstraction

Example: computing `mult(17,212)`



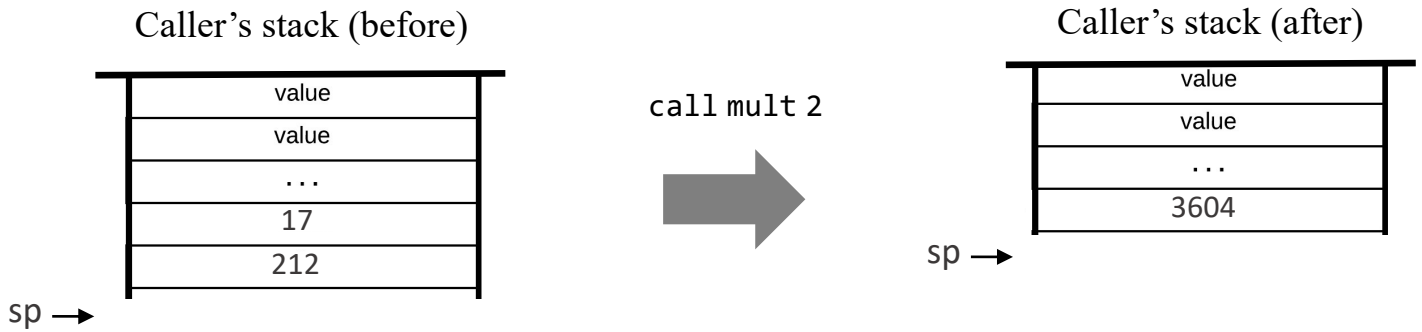
Function call and return: abstraction

Example: computing `mult(17,212)`



Function call and return: abstraction

Example: computing `mult(17,212)`

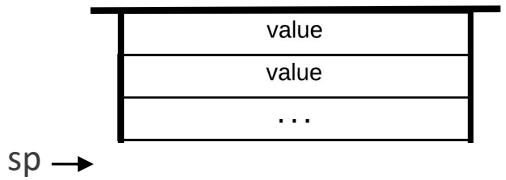


Net effect:

The function's arguments were replaced by the function's value

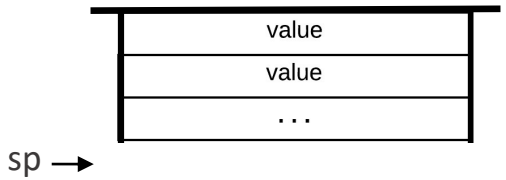
Function call and return: implementation

The function is running,
doing something



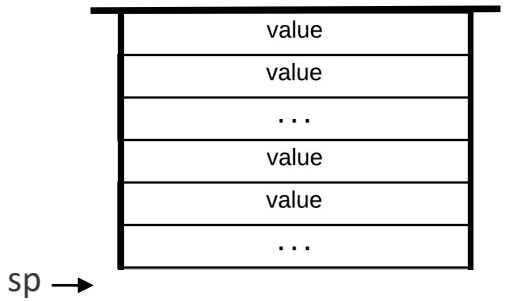
Function call and return: implementation

The function prepares
to call another function:



Function call and return: implementation

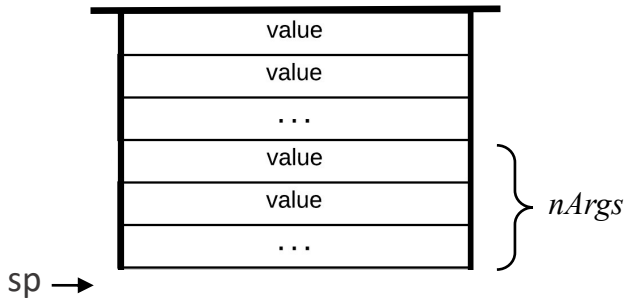
The function prepares
to call another function:



Function call and return: implementation

The function says:

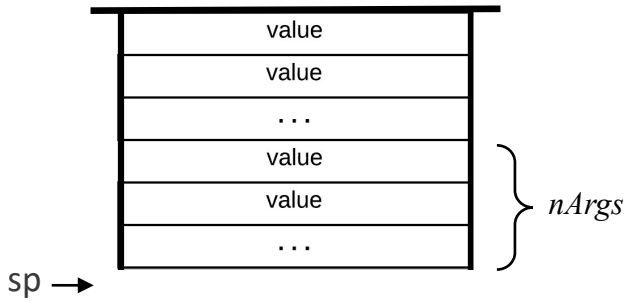
`call` *foo nArgs*



Function call and return: implementation

The function says:

`call` *foo nArgs*

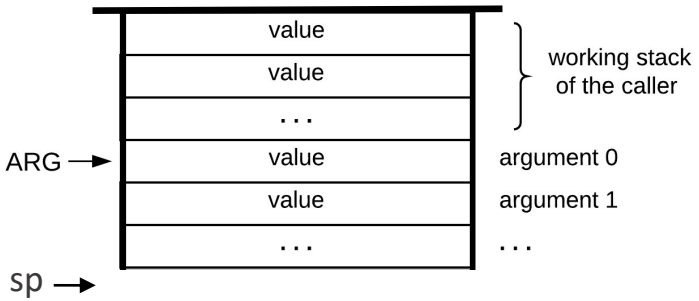


implementation (handling `call`):

Function call and return: implementation

The function says:

`call` *foo nArgs*



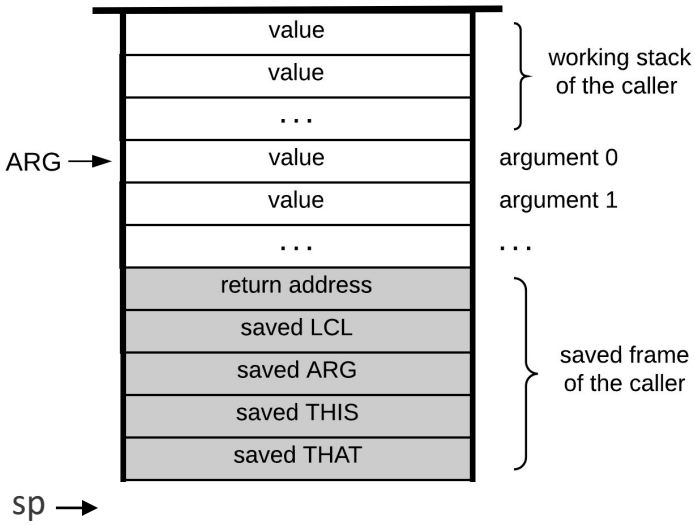
implementation (handling `call`):

1. Saves the caller's frame

Function call and return: implementation

The function says:

`call` *foo nArgs*



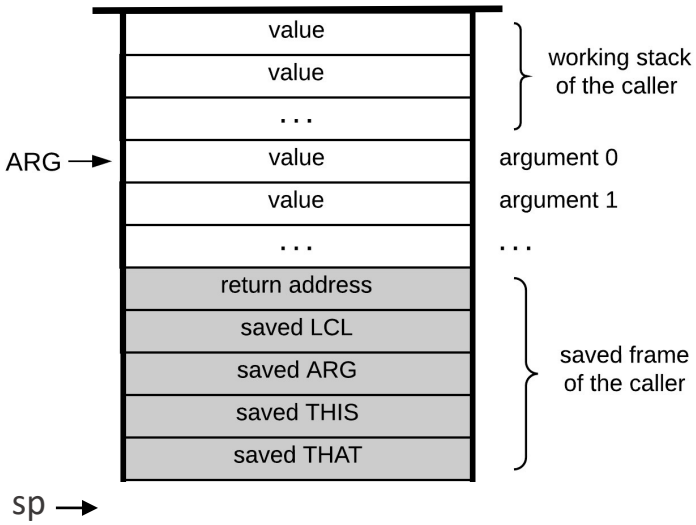
implementation (handling `call`):

1. Saves the caller's frame

Function call and return: implementation

The function says:

`call` *foo nArgs*



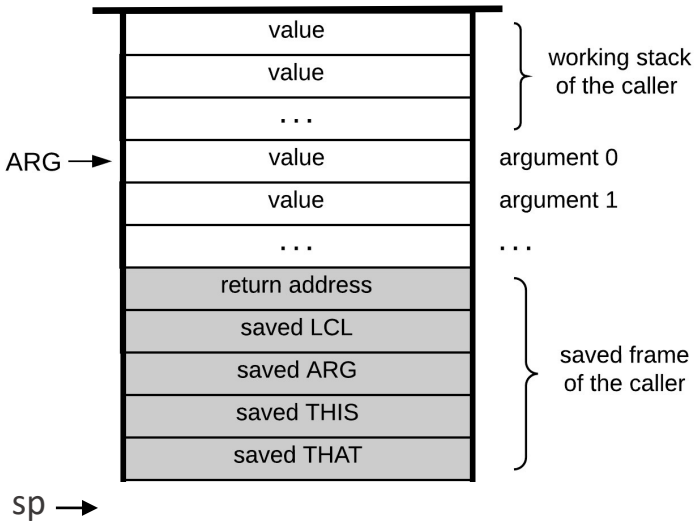
implementation (handling `call`):

1. Saves the caller's frame
2. Jumps to execute *foo*

Function call and return: implementation

The called function is entered:

`function` *foo nVars*



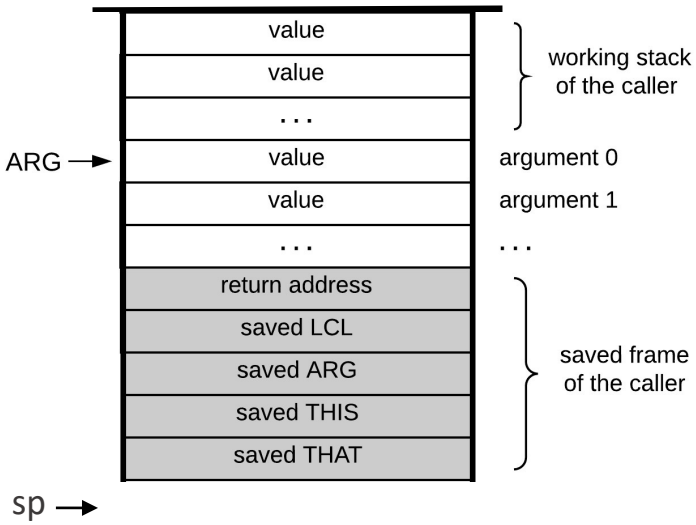
implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

Function call and return: implementation

The called function is entered:

`function` *foo nVars*



implementation (handling `call`):

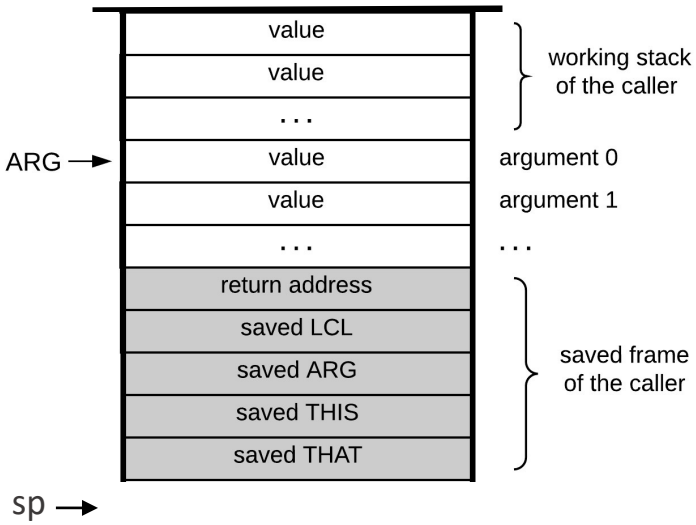
1. Saves the caller's frame
2. Jumps to execute *foo*
3. Sets ARG

implementation (handling `function`):

Function call and return: implementation

The called function is entered:

`function` *foo nVars*



implementation (handling `call`):

1. Saves the caller's frame
2. Jumps to execute *foo*
3. Sets ARG

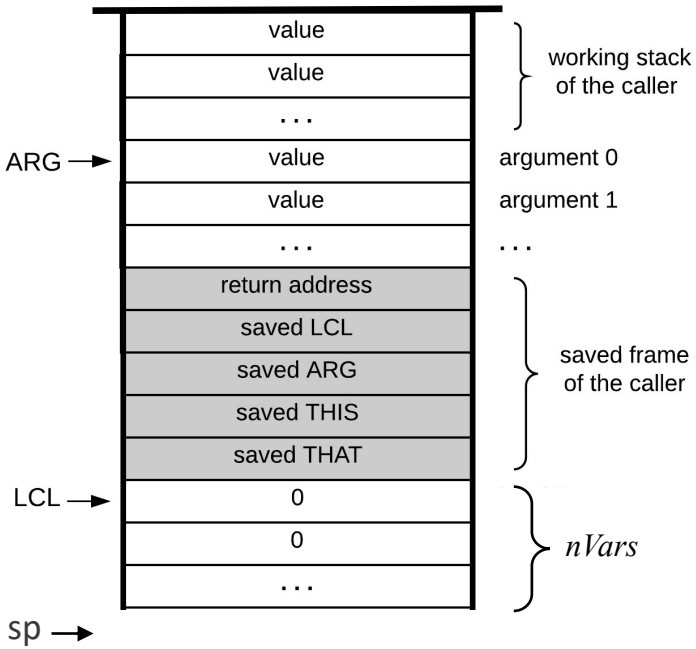
implementation (handling `function`):

Sets up the `local` segment of the called function

Function call and return: implementation

The called function is entered:

`function` *foo nVars*



implementation (handling `call`):

1. Saves the caller's frame
2. Jumps to execute *foo*
3. Sets ARG

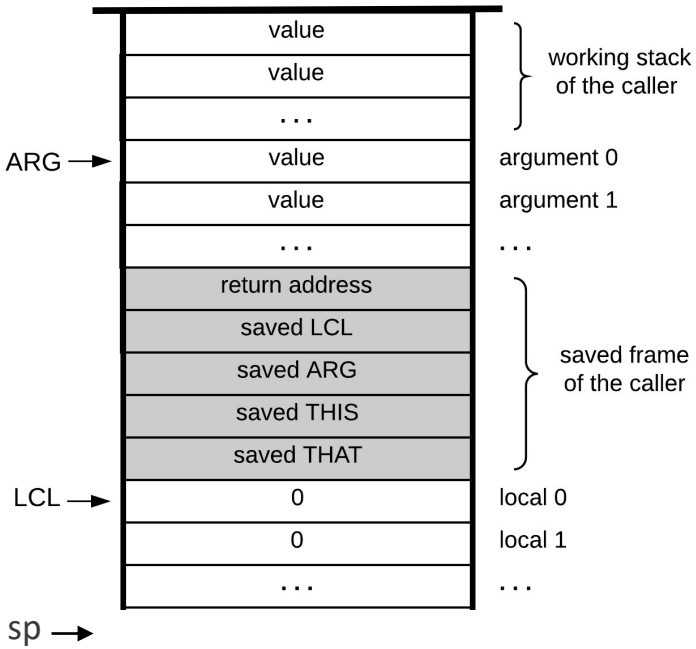
implementation (handling `function`):

Sets up the `local` segment of the called function

Function call and return: implementation

The called function is entered:

`function` *foo nVars*



implementation (handling `call`):

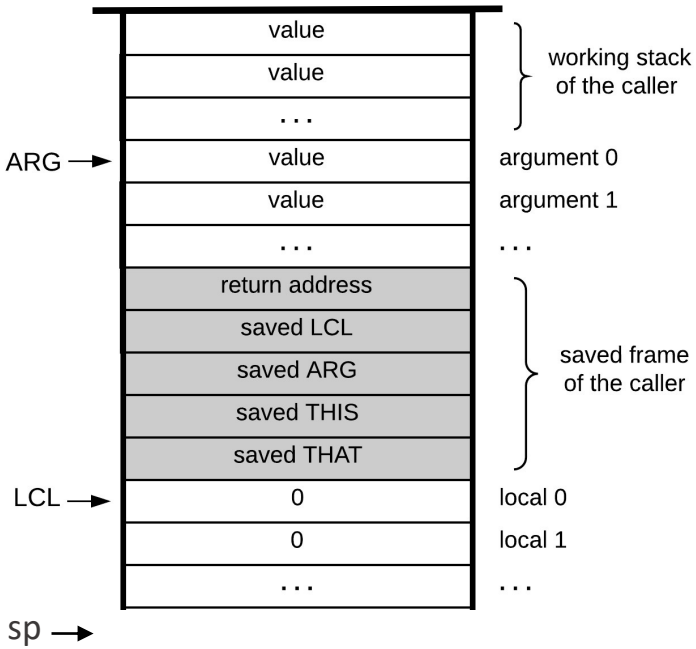
1. Saves the caller's frame
2. Jumps to execute *foo*
3. Sets ARG

implementation (handling `function`):

Sets up the `local` segment of the called function

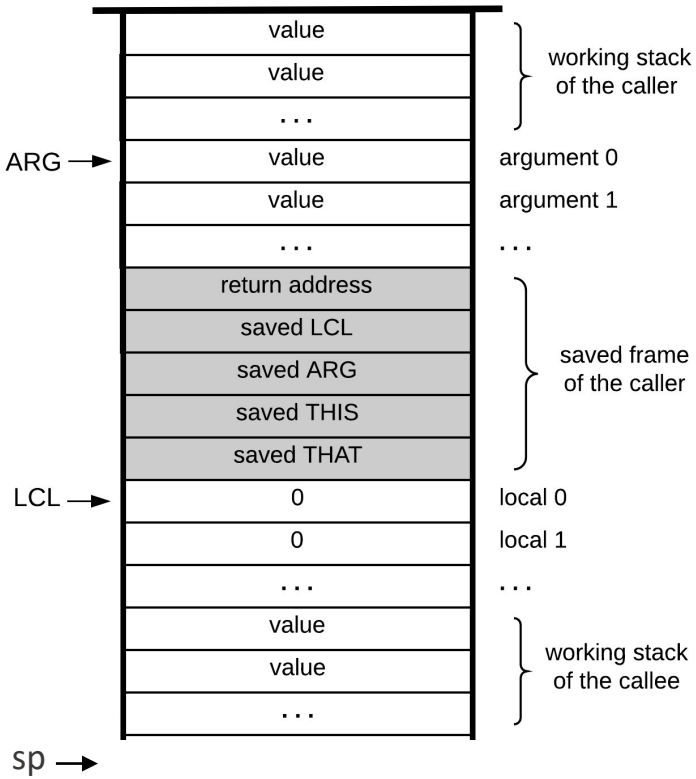
Function call and return: implementation

The called function is running,
doing something



Function call and return: implementation

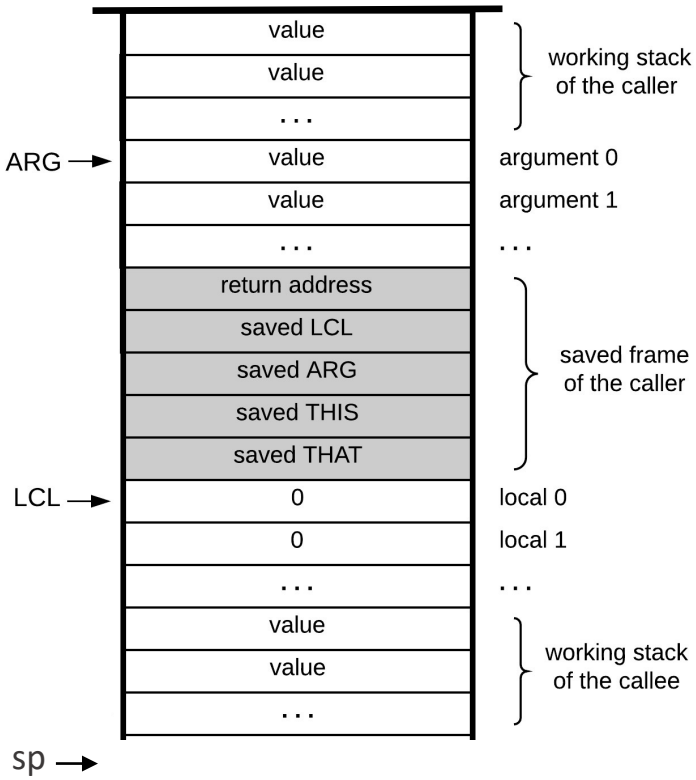
The called function is running,
doing something



Function call and return: implementation

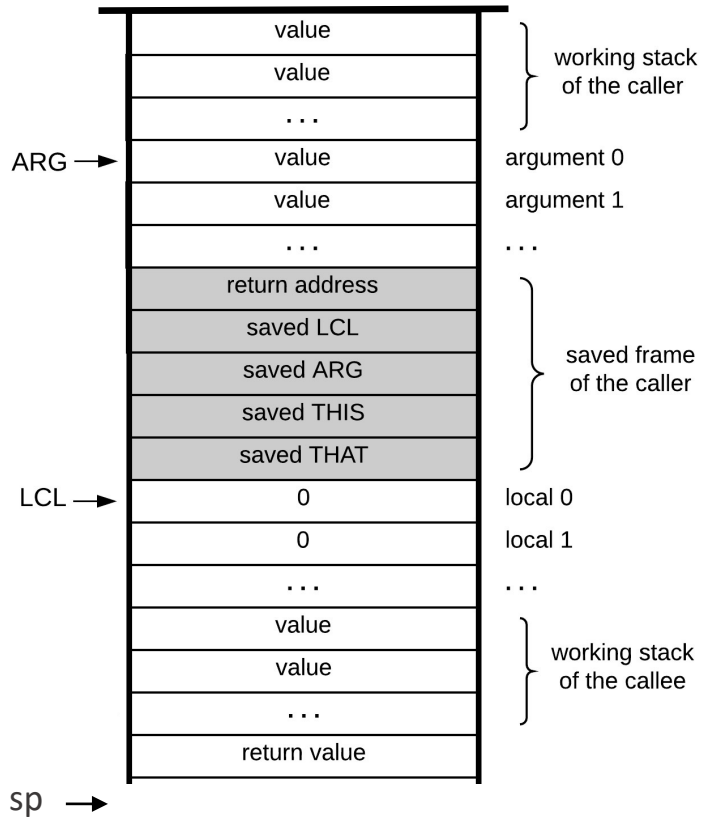
The called function prepares to return:

it pushes a *return value*



Function call and return: implementation

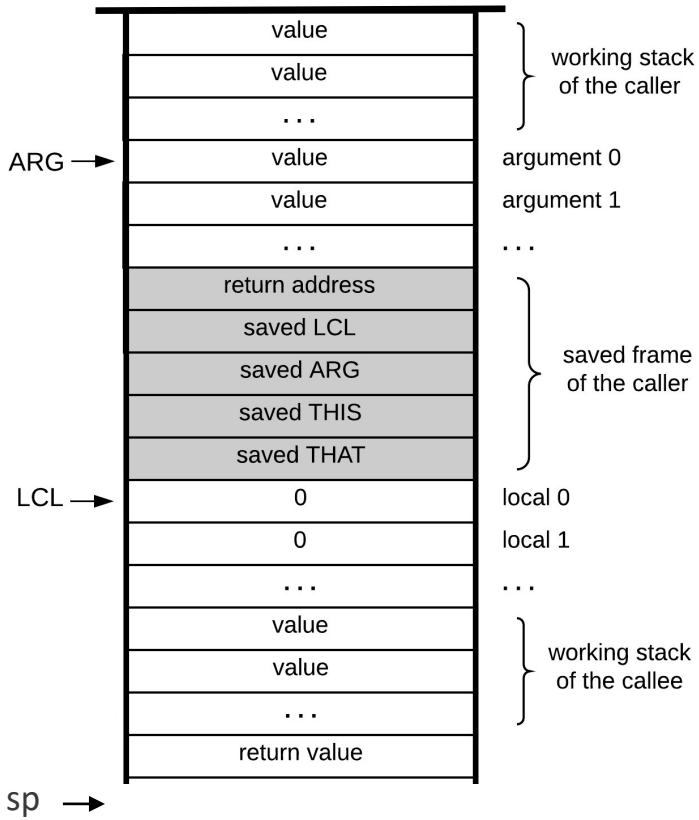
The called function prepares to return:
it pushes a *return value*



Function call and return: implementation

The called function says:

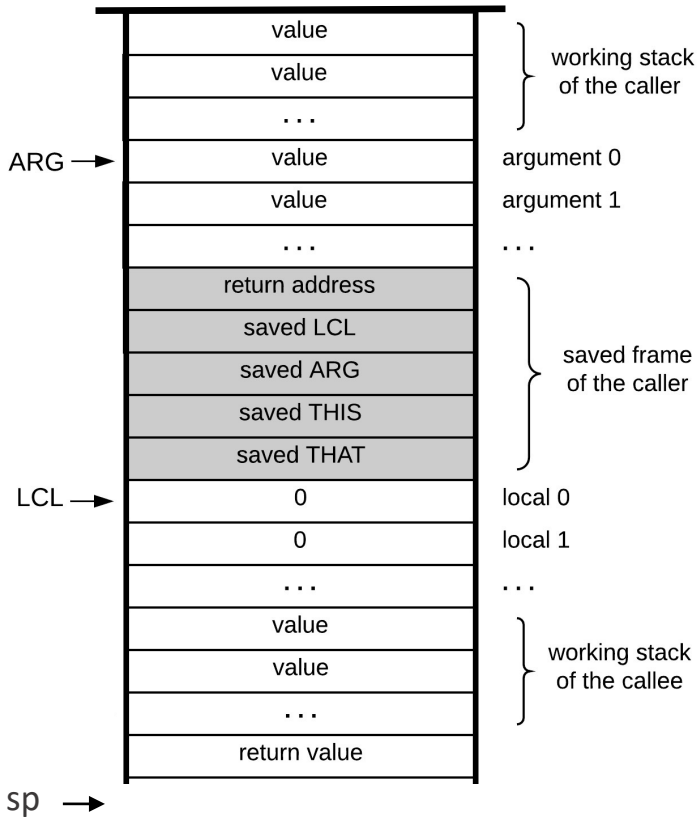
`return`



Function call and return: implementation

The called function says:

`return`



implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

implementation (handling `return`):

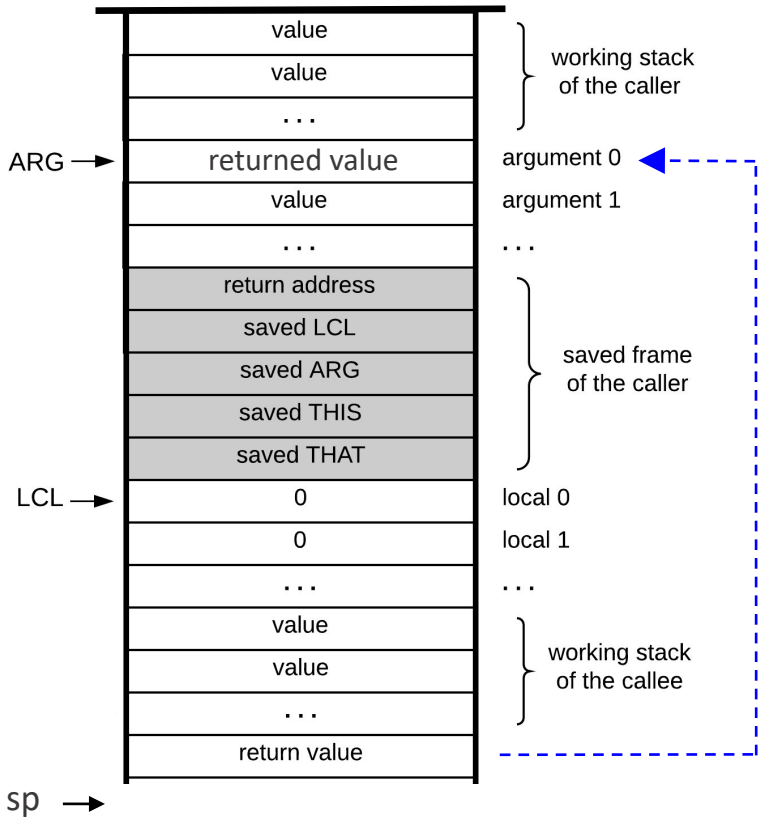
Sets up the `local` segment of the called function

implementation (handling `return`):

Function call and return: implementation

The called function says:

`return`



implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

implementation (handling `return`):

Sets up the `local` segment of the called function

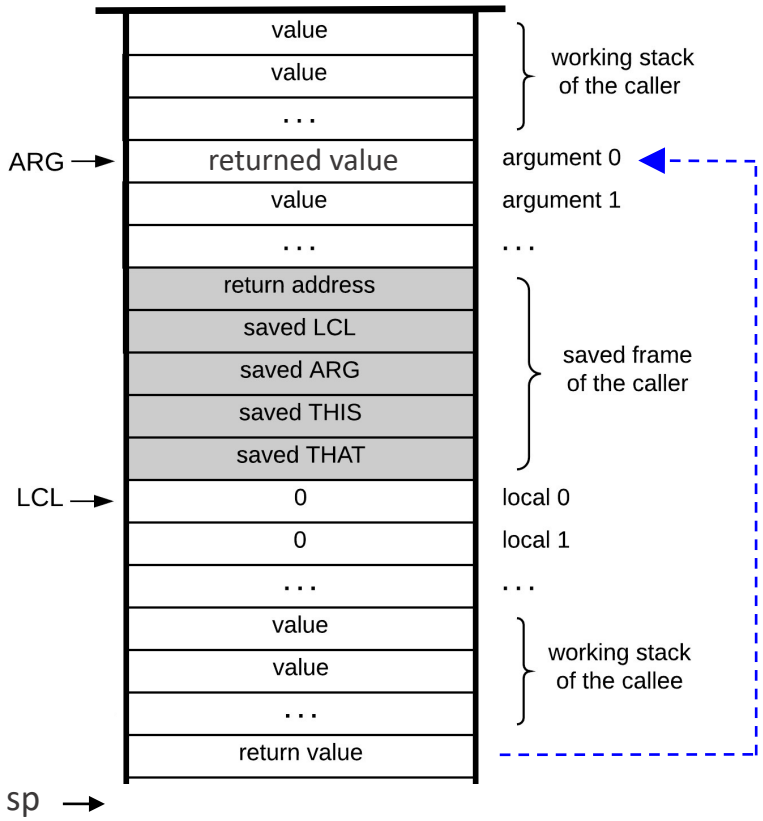
implementation (handling `return`):

1. Copies the return value onto `argument 0`

Function call and return: implementation

The called function says:

`return`



implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

implementation (handling `return`):

Sets up the `local` segment of the called function

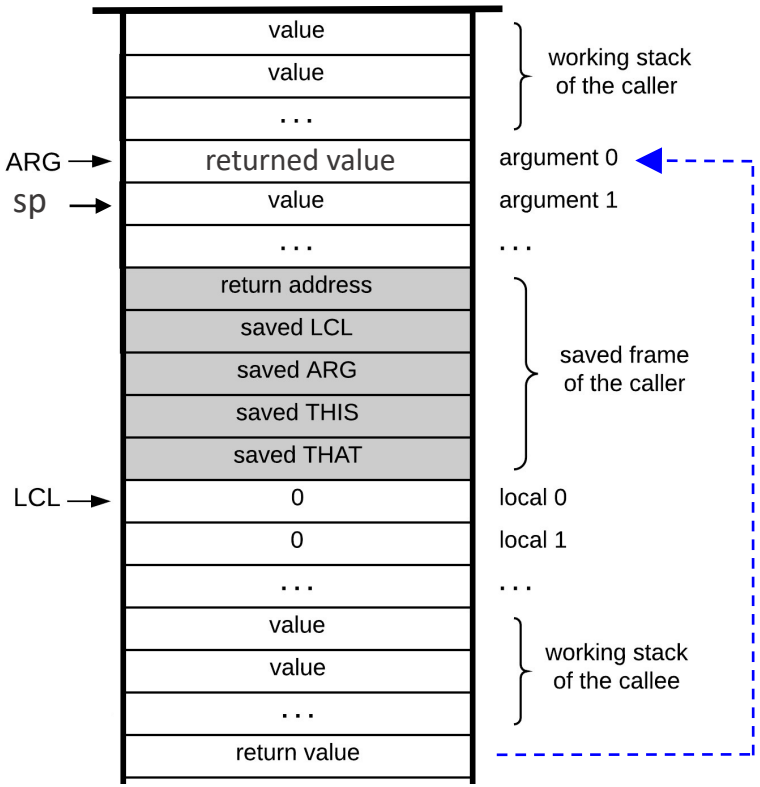
implementation (handling `return`):

1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller

Function call and return: implementation

The called function says:

`return`



implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

implementation (handling `return`):

Sets up the `local` segment of the called function

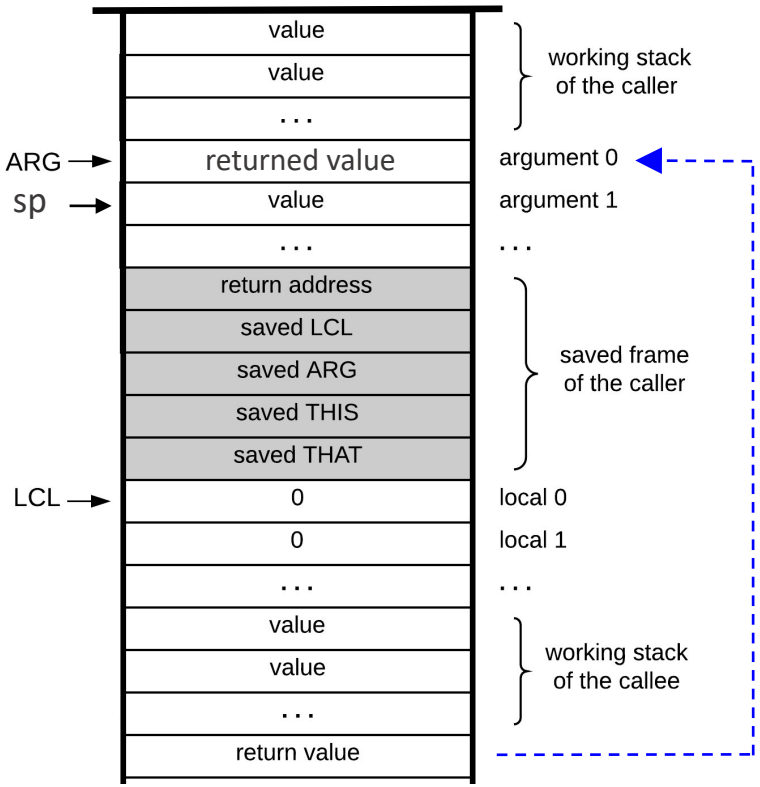
implementation (handling `return`):

1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller

Function call and return: implementation

The called function says:

`return`



implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

implementation (handling `return`):

Sets up the `local` segment of the called function

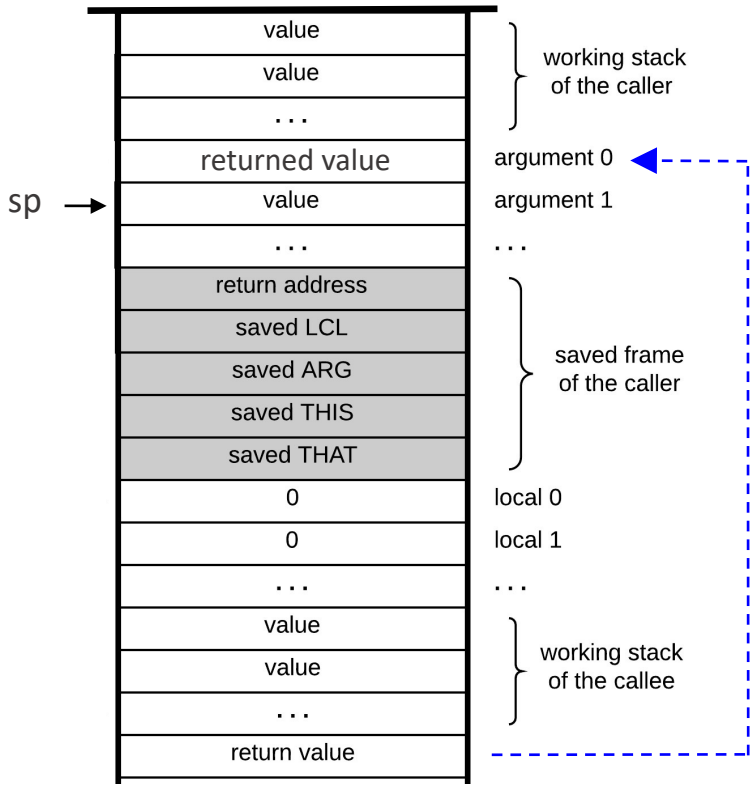
implementation (handling `return`):

1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller
3. Restores the segment pointers of the caller

Function call and return: implementation

The called function says:

`return`



implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

implementation (handling `return`):

Sets up the `local` segment of the called function

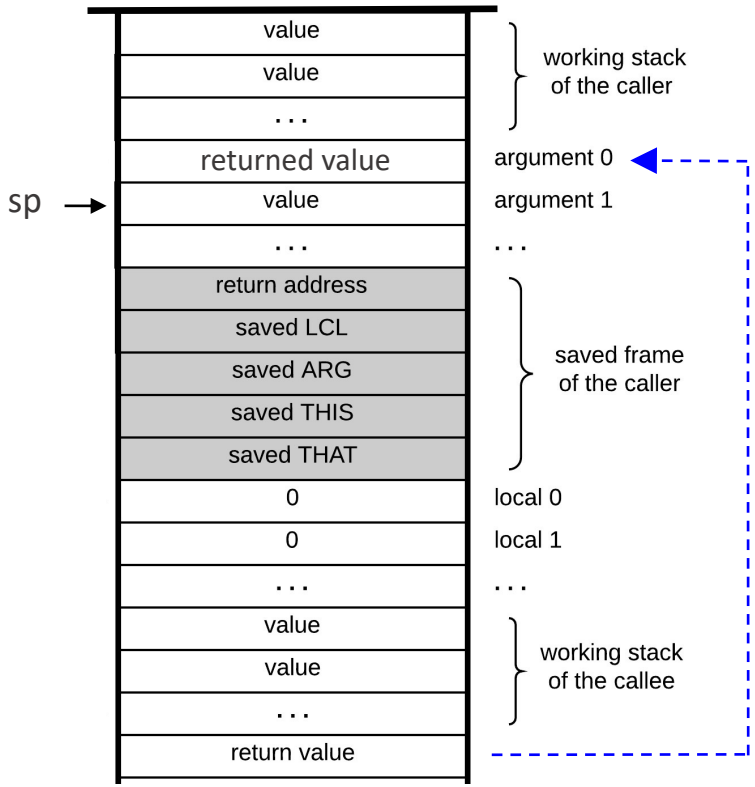
implementation (handling `return`):

1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller
3. Restores the segment pointers of the caller

Function call and return: implementation

The called function says:

`return`



implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

implementation (handling `return`):

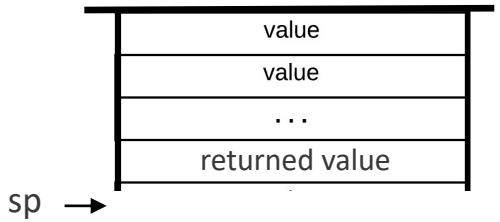
Sets up the `local` segment of the called function

implementation (handling `return`):

1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller
3. Restores the segment pointers of the caller
4. Jumps to the return address within the caller's code (note that the stack space below `sp` is recycled)

Function call and return: implementation

The caller
resumes its execution



implementation (handling `call`):

1. Sets `ARG`
2. Saves the caller's frame
3. Jumps to execute *foo*

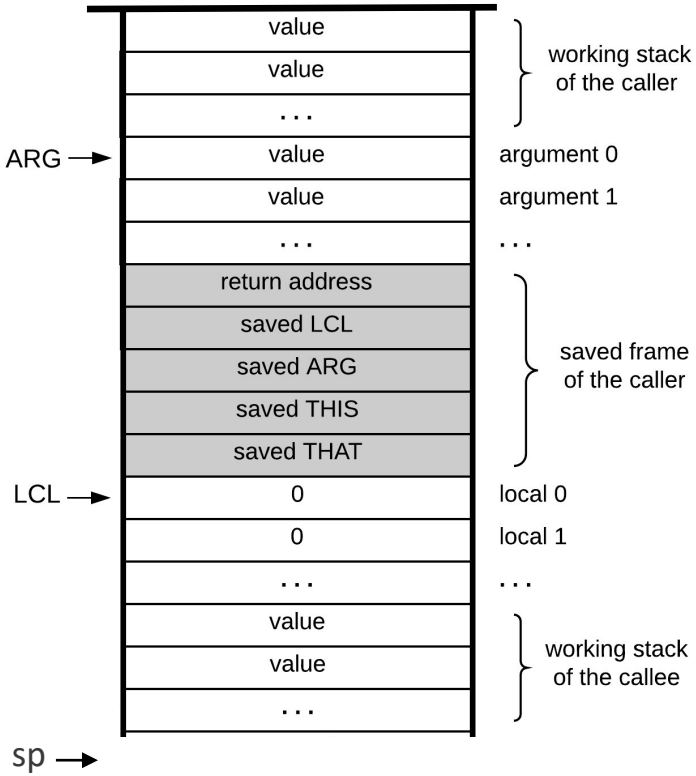
implementation (handling `return`):

Sets up the `local` segment
of the called function

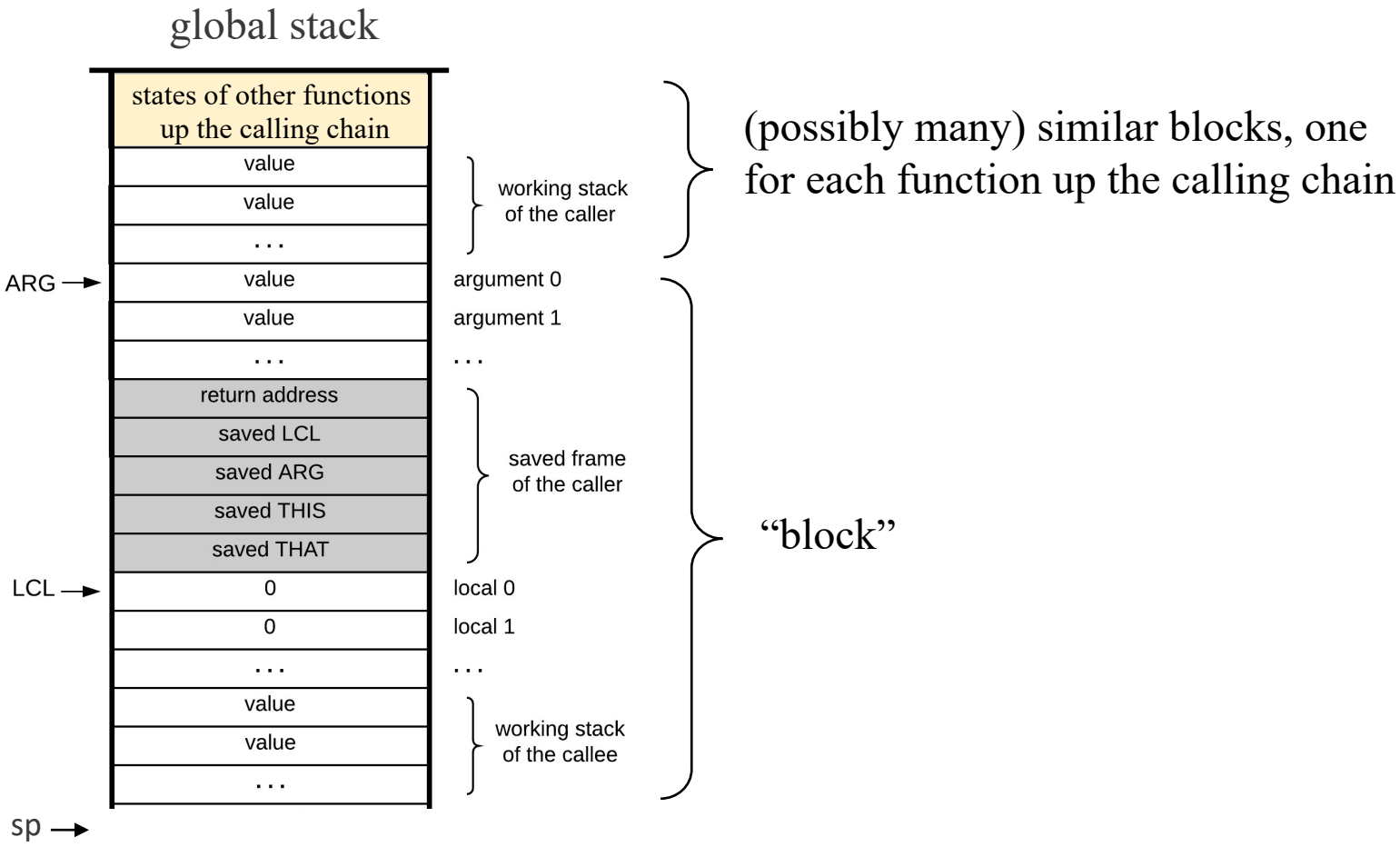
implementation (handling `return`):

1. Copies the return value onto `argument 0`
2. Sets `SP` for the caller
3. Restores the segment pointers of the caller
4. Jumps to the return address within the caller's code
   (note that the stack space below `sp` is recycled)

The global stack



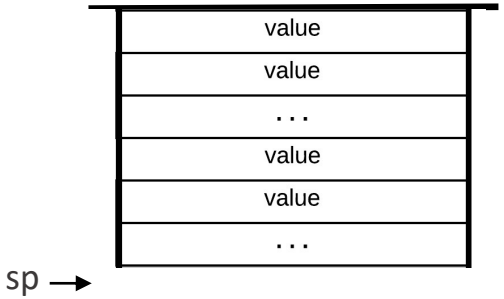
The global stack



Recap: function call and return

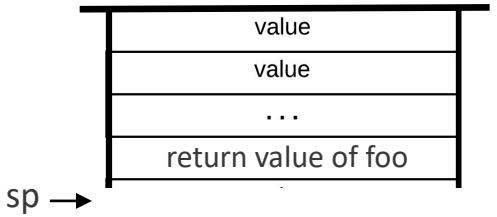
The caller says:

`call` *foo nArgs*

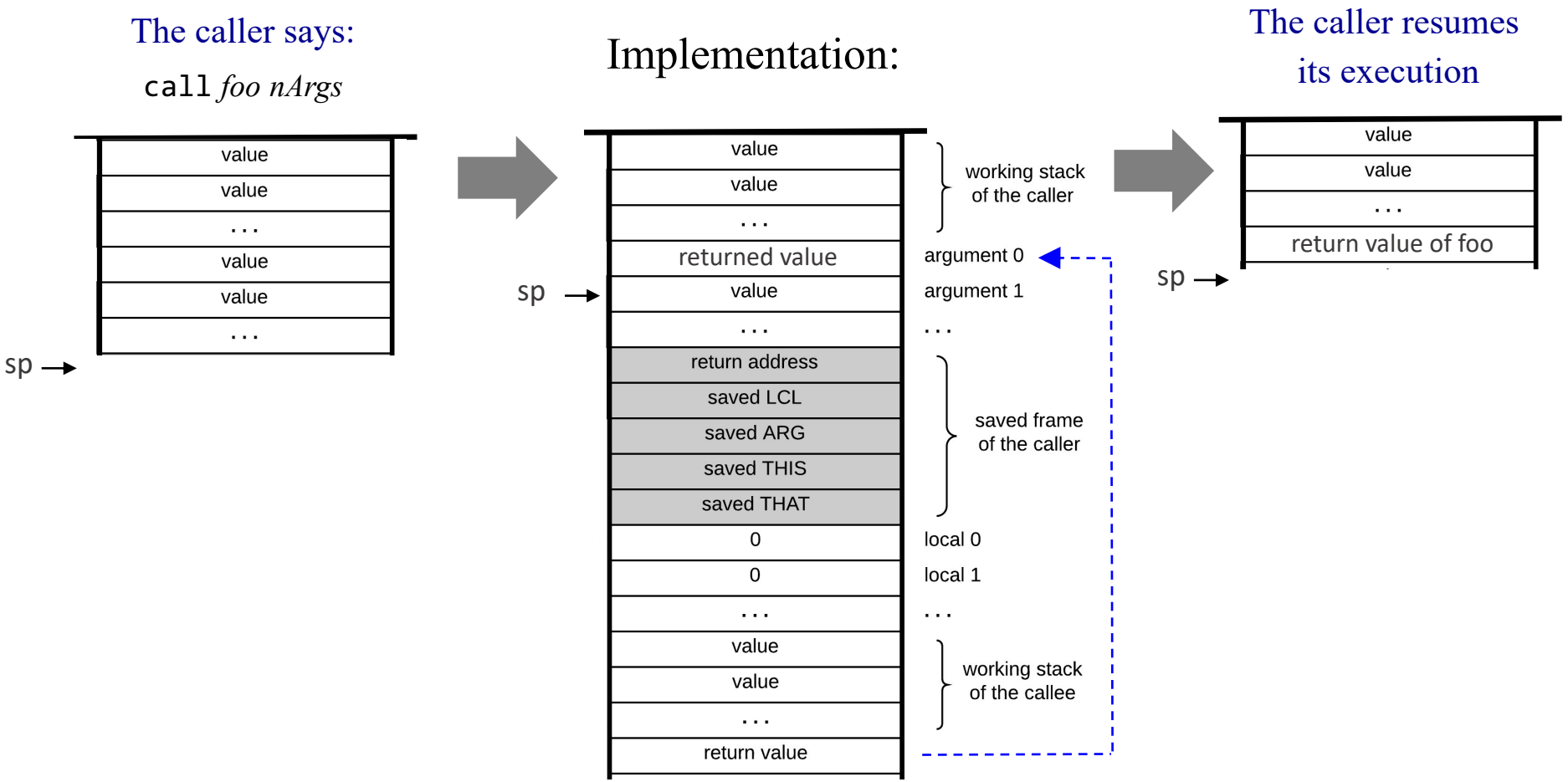


Abstraction:

The caller resumes its execution



Recap: function call and return



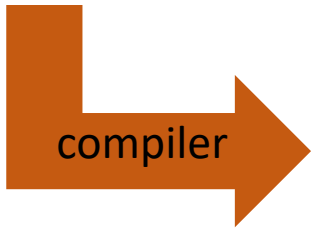
Any sufficiently advanced technology is indistinguishable from magic.

—Arthur C. Clarke (1962)

Example: `factorial`

High-level program

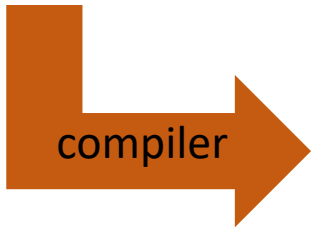
```
// Tests the factorial function
int main() {
   return factorial(3);
}

// Returns n!
int factorial(int n) {
   if (n==1)
      return 1;
   else
      return n * factorial(n-1);
}
```



Example: `factorial`

High-level program

```
// Tests the factorial function
int main() {
   return factorial(3);
}

// Returns n!
int factorial(int n) {
   if (n==1)
      return 1;
   else
      return n * factorial(n-1);
}
```



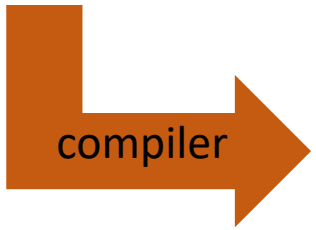
Pseudo VM code

```
function main
   push 3
   call factorial
   return
```

Example: `factorial`

High-level program

```
// Tests the factorial function
int main() {
   return factorial(3);
}

// Returns n!
int factorial(int n) {
   if (n==1)
      return 1;
   else
      return n * factorial(n-1);
}
```



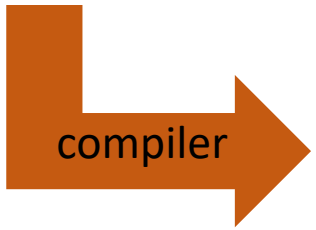
Pseudo VM code

```
function main
   push 3
   call factorial
   return

function factorial(n)
   push n
   push 1
   eq
   if-goto BASECASE

   push n
   push n
   push 1
   sub
   call factorial
   call mult
   return

label BASECASE
   push 1
   return

function mult(a,b)
 // Code omitted
```

Example: `factorial`

High-level program

```
// Tests the factorial function
int main() {
   return factorial(3);
}

// Returns n!
int factorial(int n) {
   if (n==1)
      return 1;
   else
      return n * factorial(n-1);
}
```



Pseudo VM code

```
function main
   push 3
   call factorial
   return

function factorial(n)
   push n
   push 1
   eq
   if-goto BASECASE

   push n
   push n
   push 1
   sub
   call factorial
   call mult
   return

label BASECASE
   push 1
   return

function mult(a,b)
 // Code omitted
```

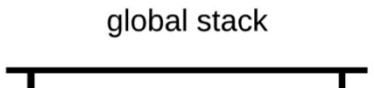
VM program

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE

   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return

label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```

Run-time example

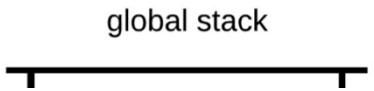
VM program

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



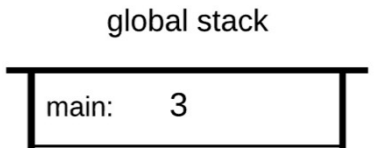
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



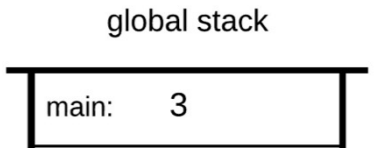
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



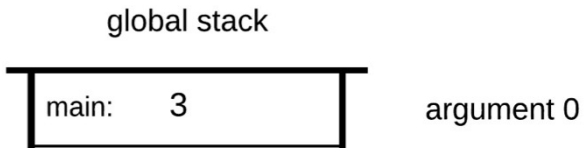
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



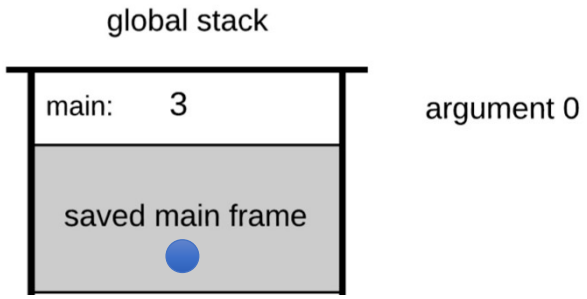
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



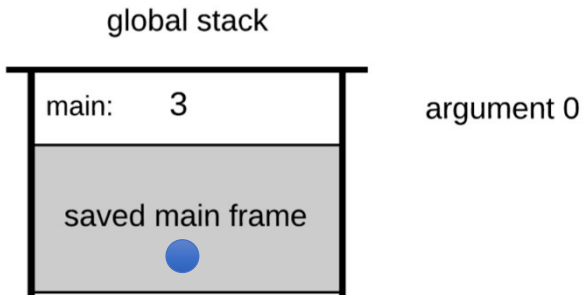
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



Run-time example

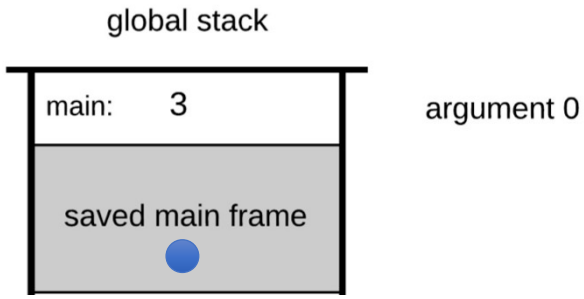
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



Run-time example

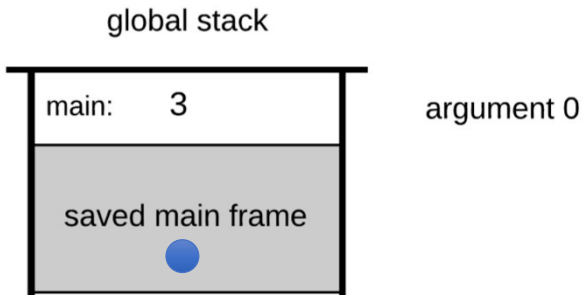
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```

impact on the global stack not shown



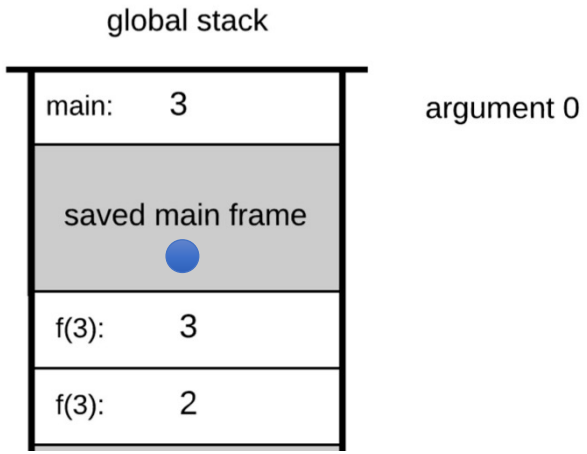
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



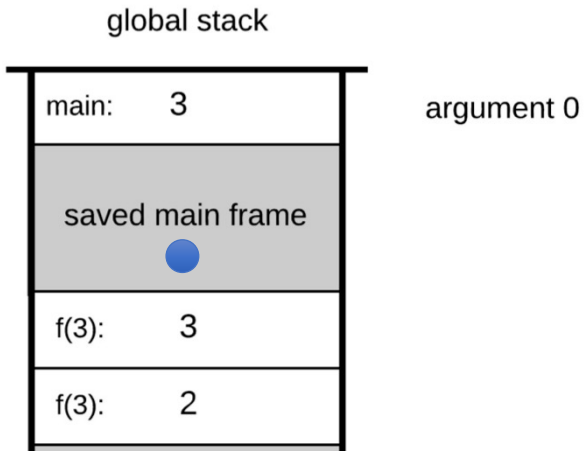
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



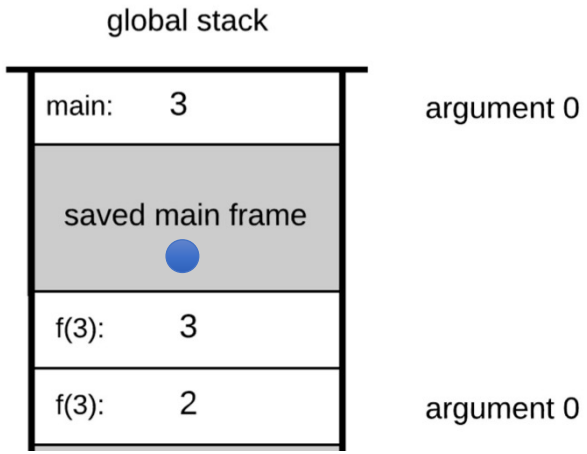
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



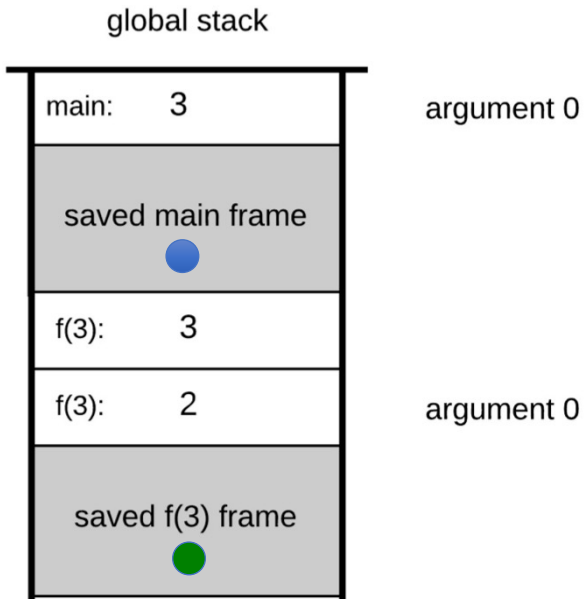
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



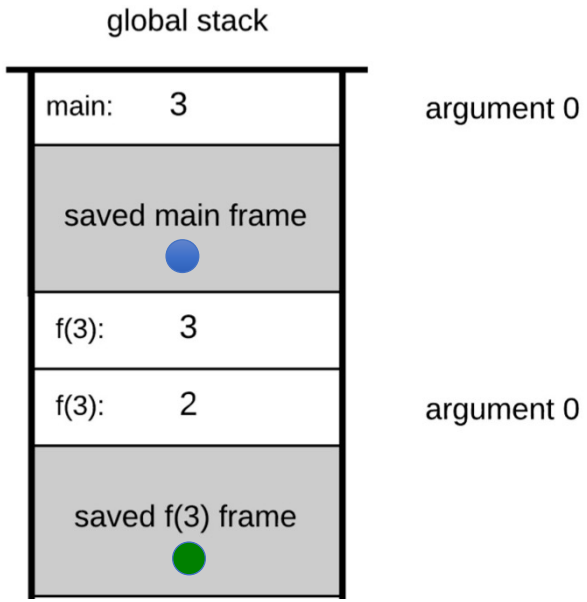
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



Run-time example

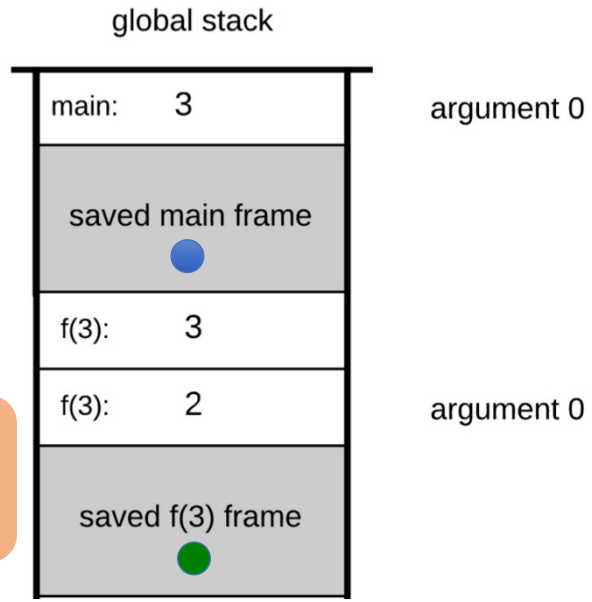
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



Run-time example

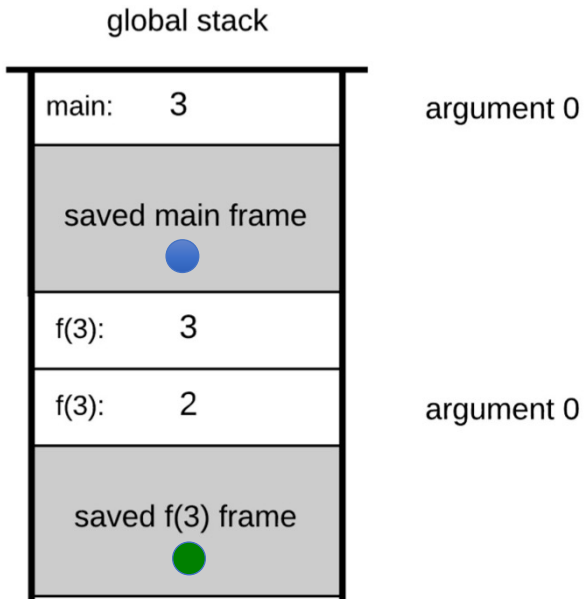
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```

impact on the global stack not shown



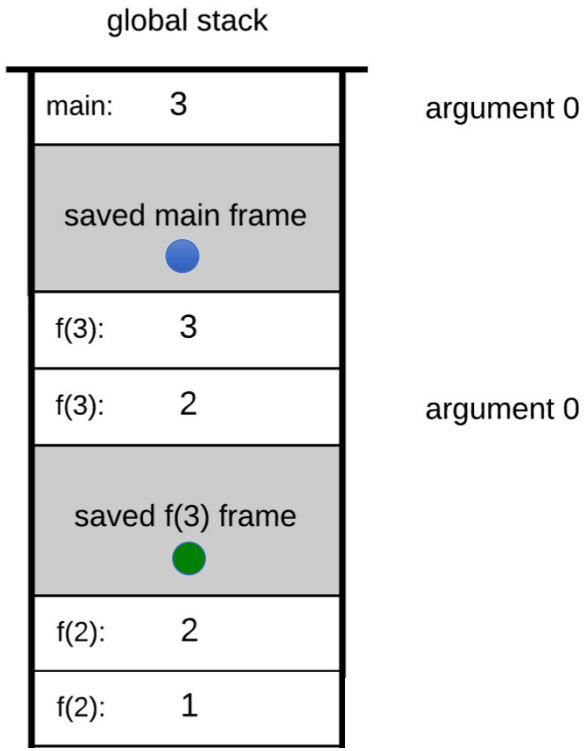
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



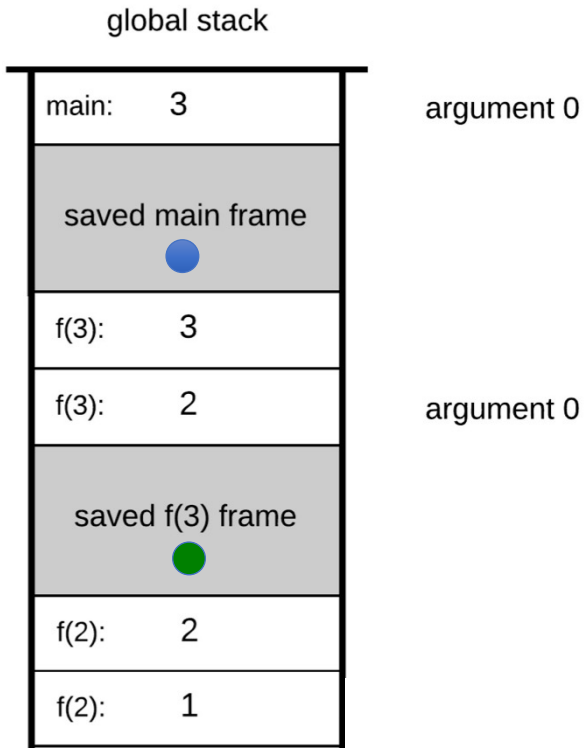
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



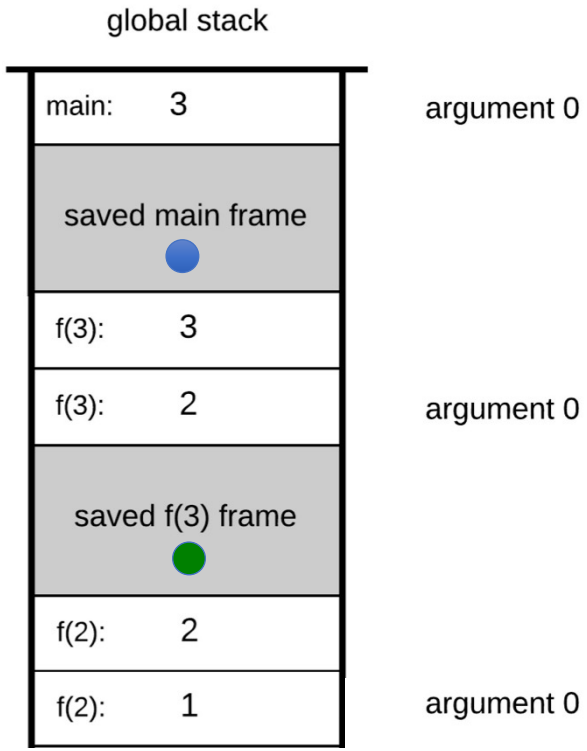
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



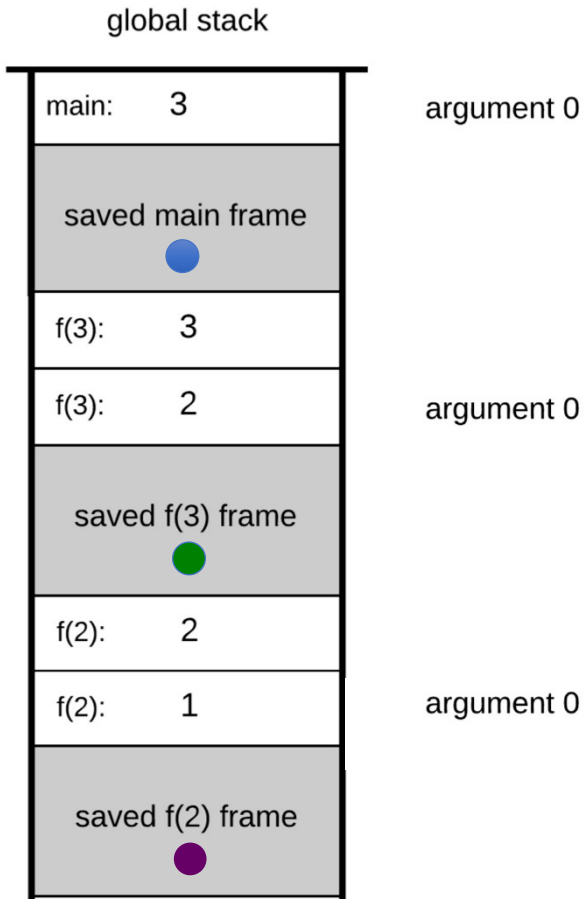
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



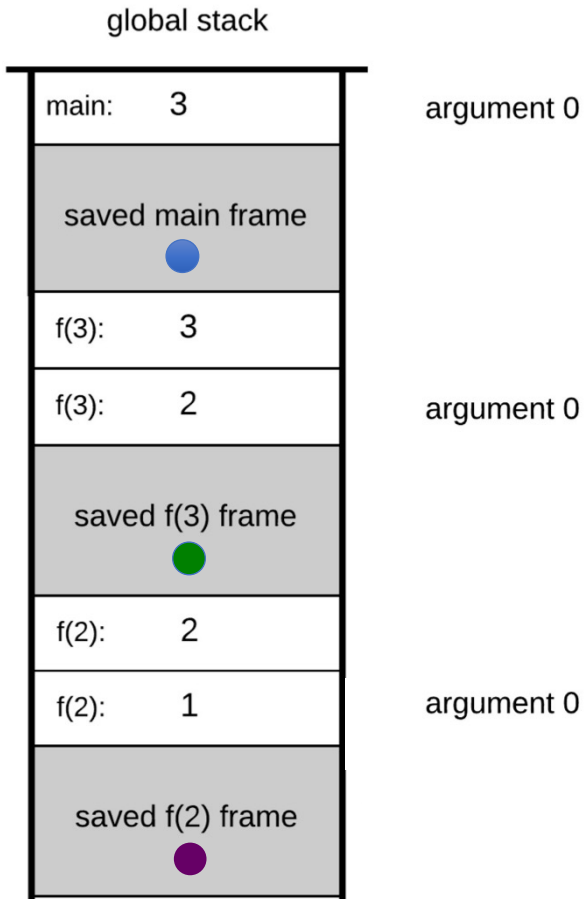
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



Run-time example

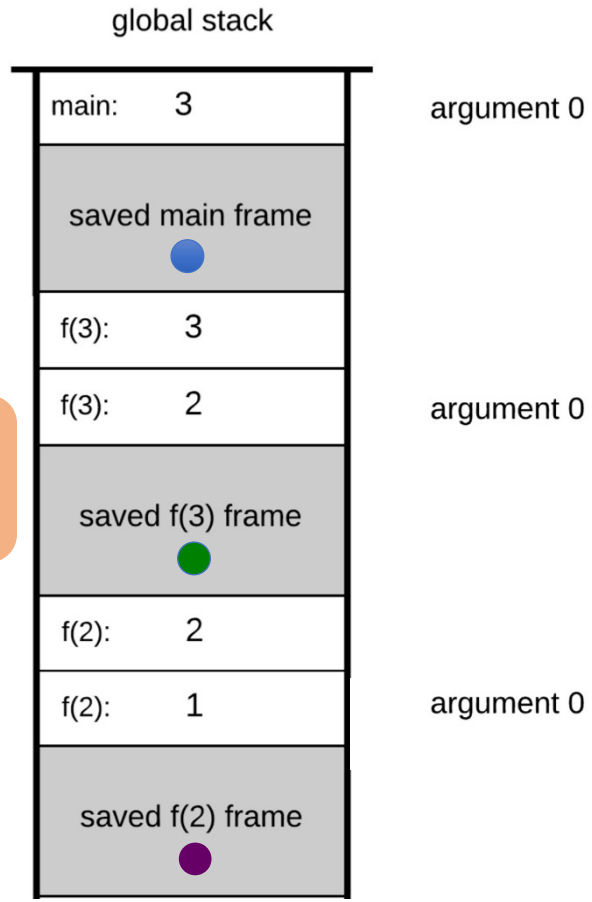
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



Run-time example

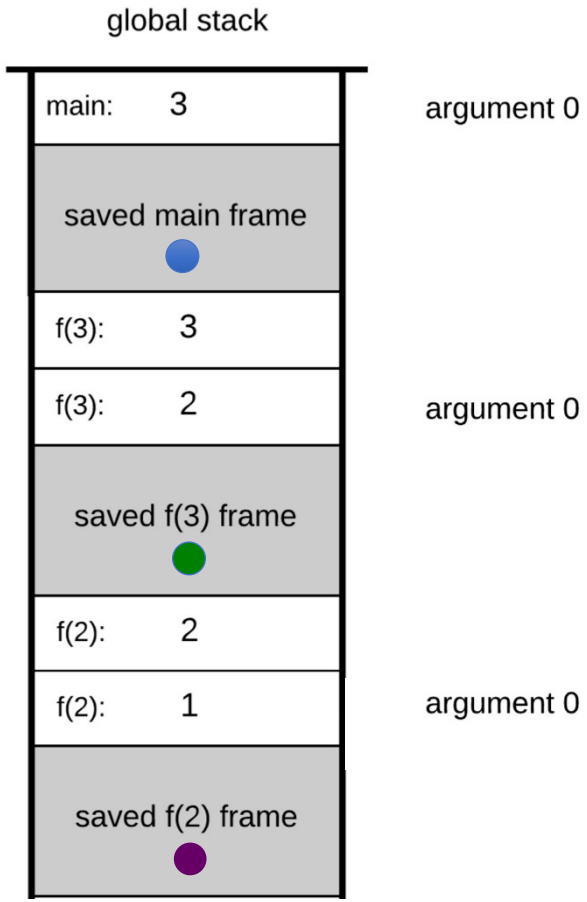
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```

impact on the global stack not shown



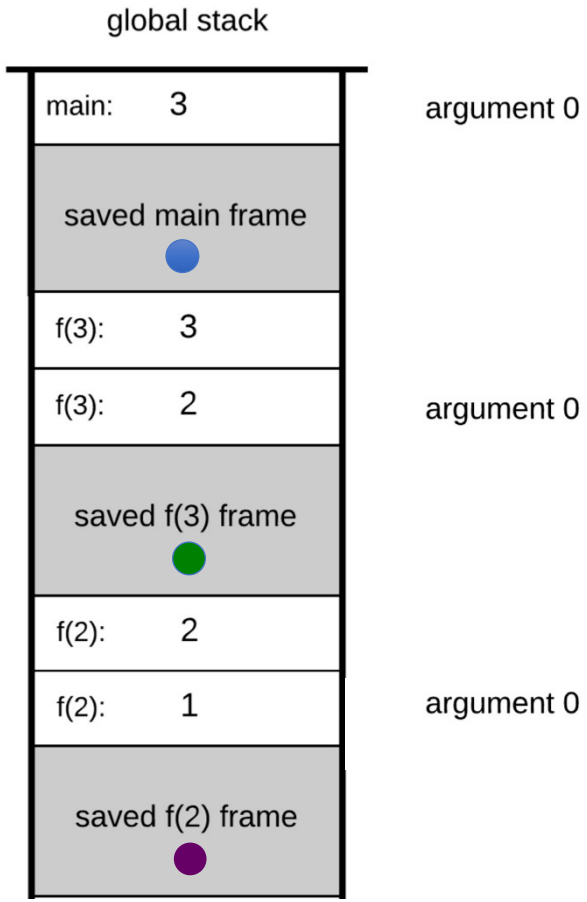
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



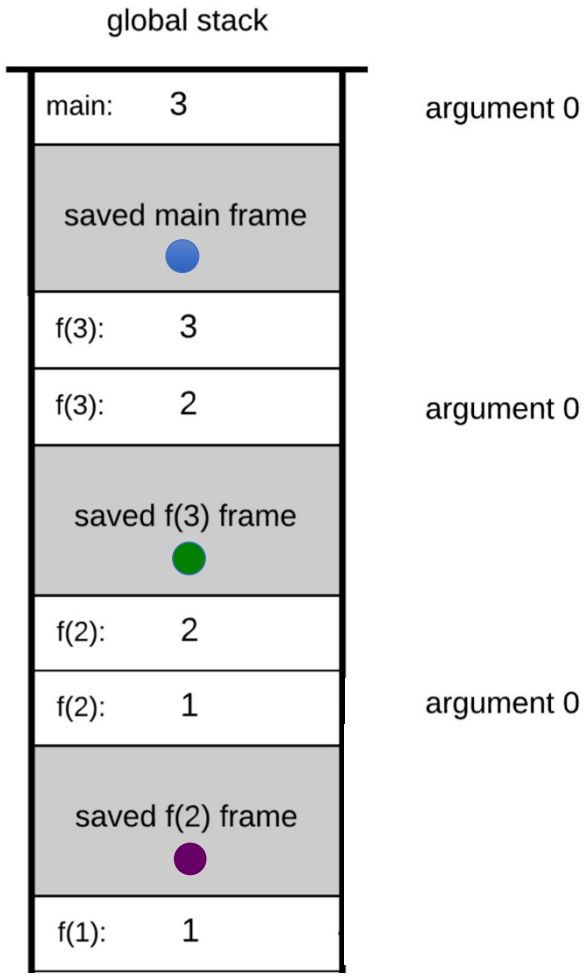
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



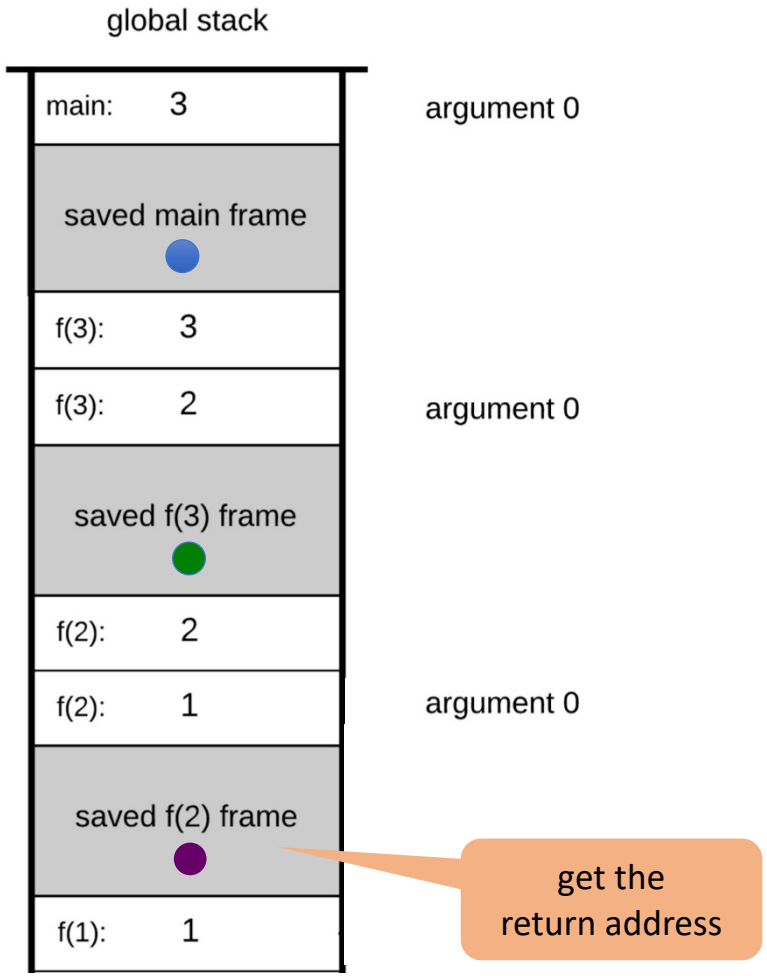
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



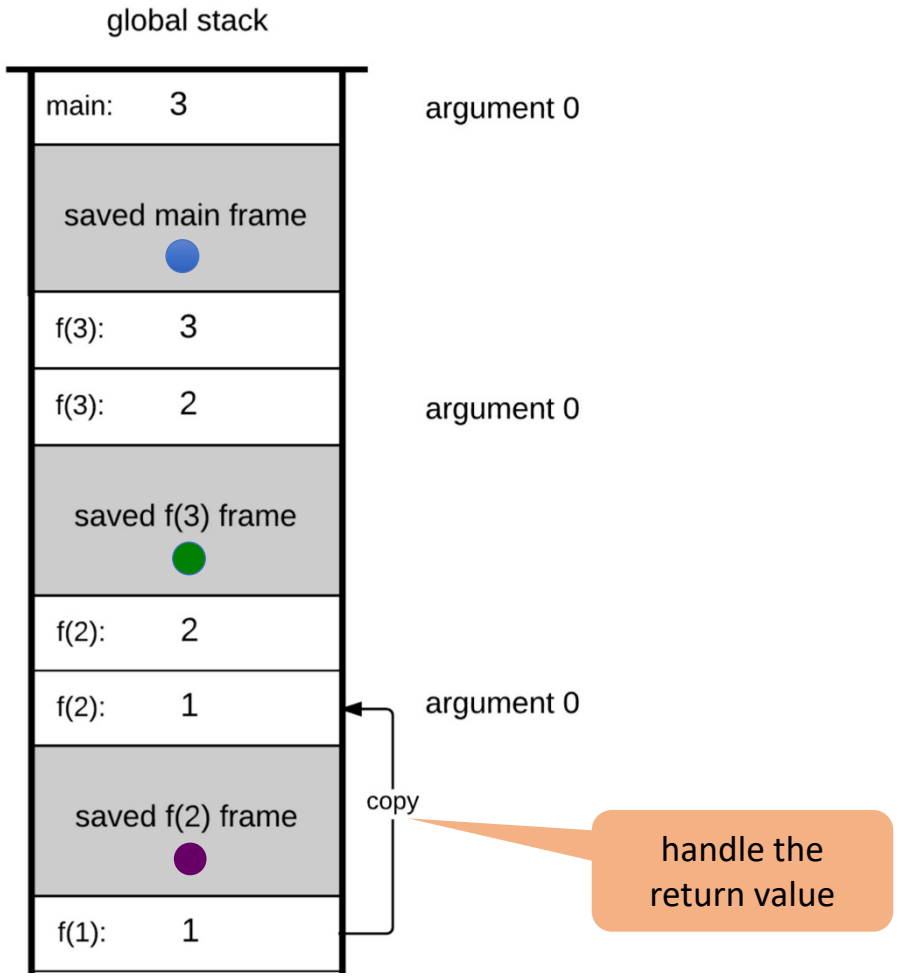
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



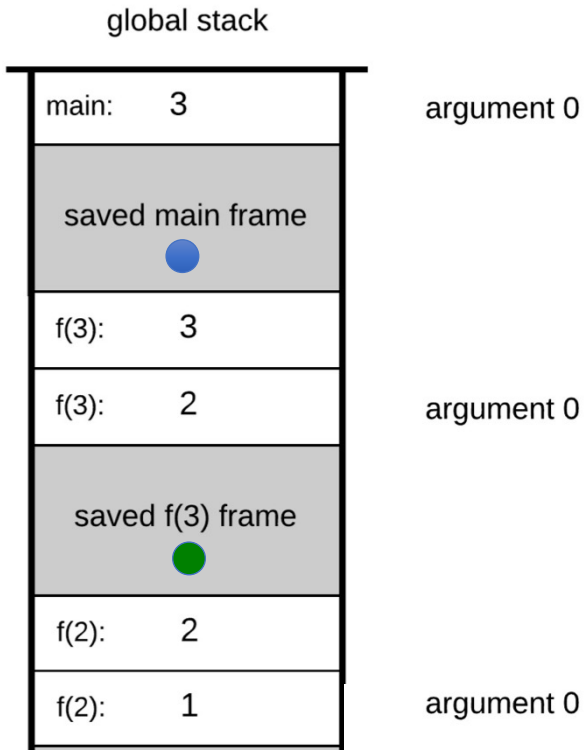
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



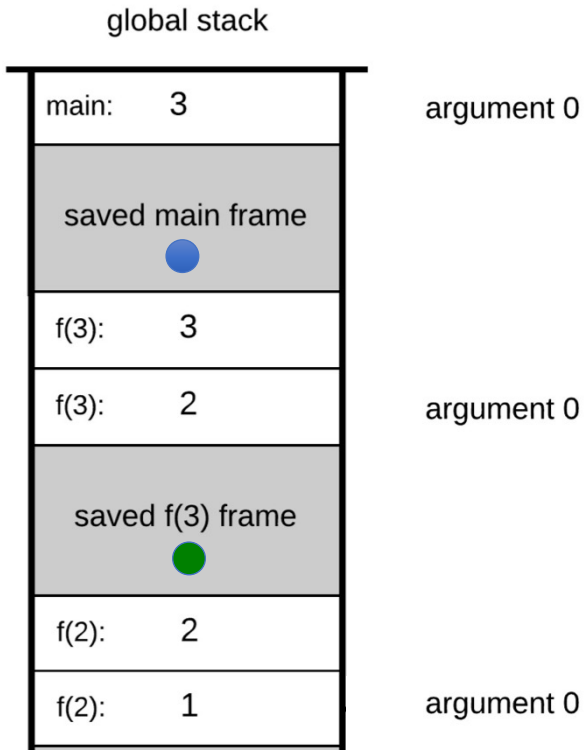
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



Run-time example

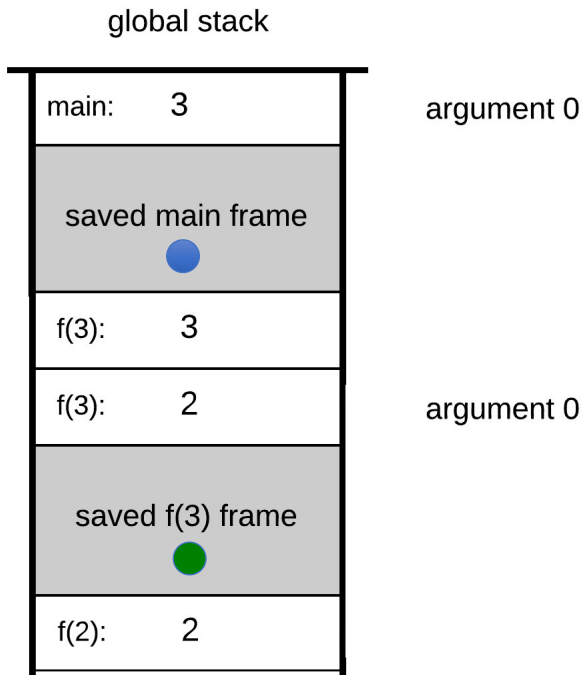
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



impact on the global stack
not shown
(except for end result)

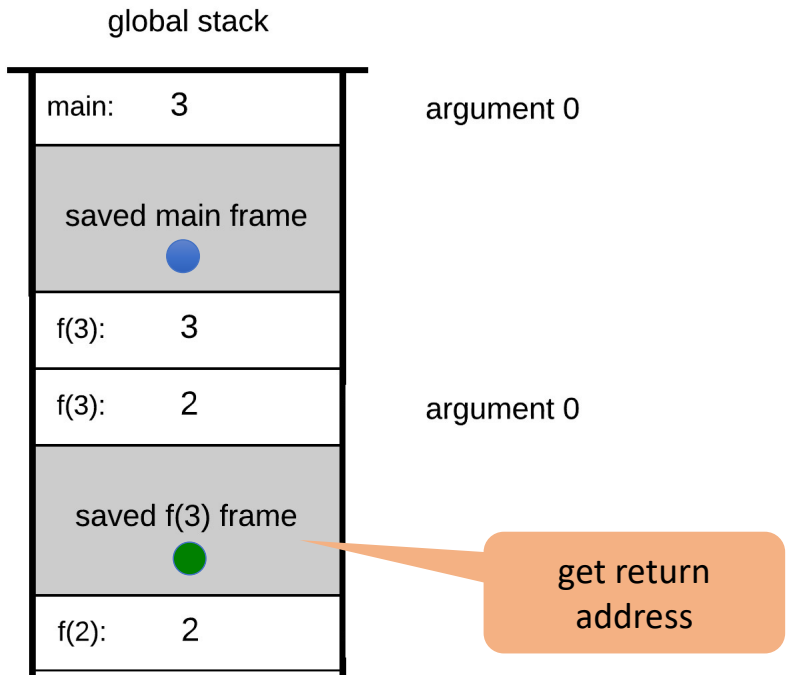
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



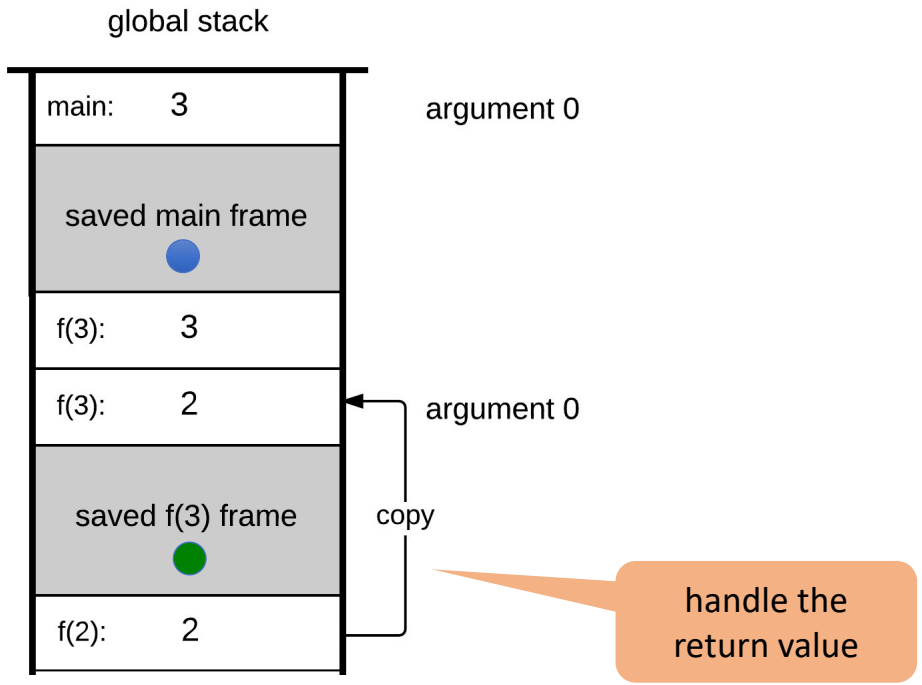
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



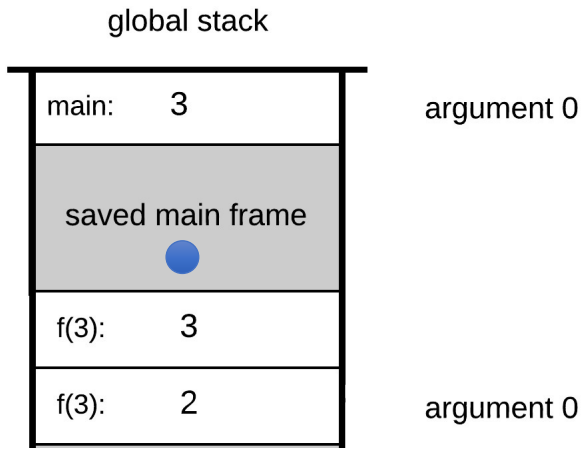
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



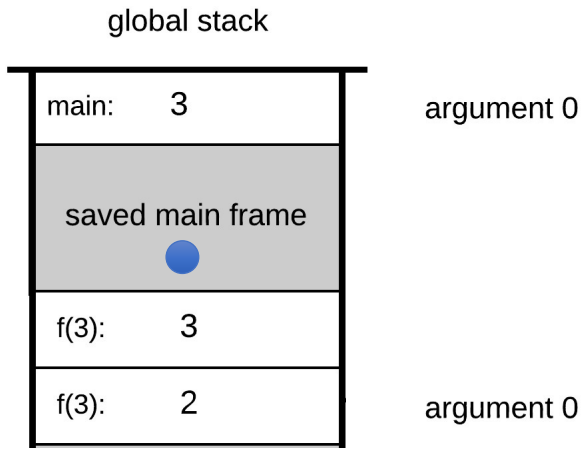
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



Run-time example

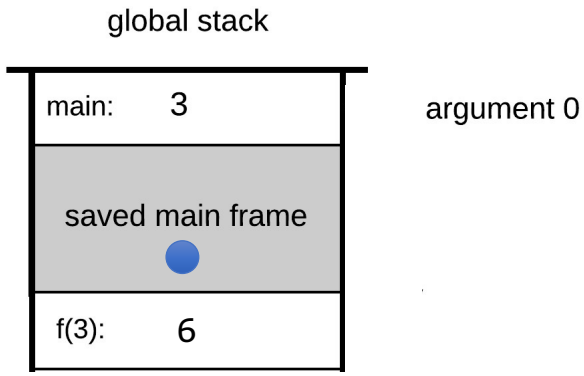
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



impact on the global stack not shown (except for end result)

Run-time example

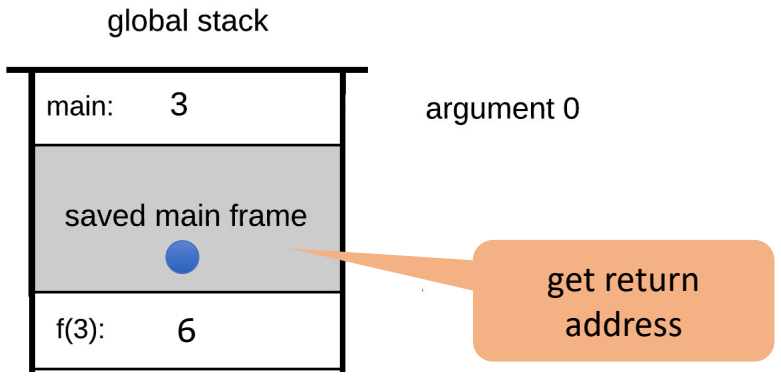
```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



impact on the global stack
not shown
(except for end result)

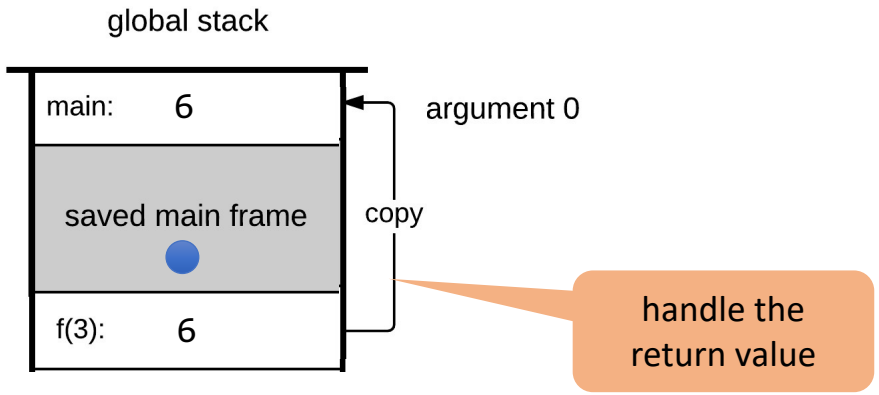
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



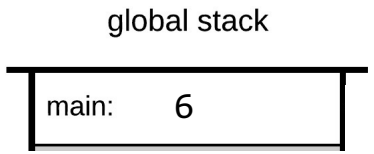
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



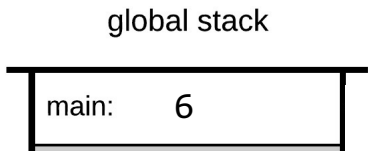
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



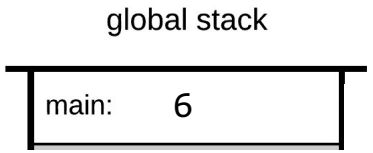
Run-time example

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



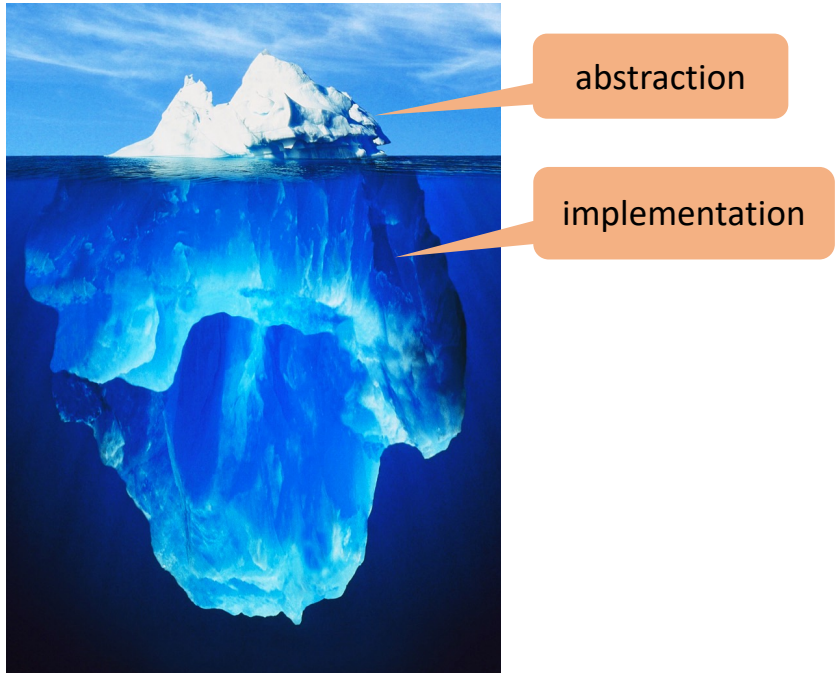
Recap

```
function main 0
   push constant 3
   call factorial 1
   return

function factorial 0
   push argument 0
   push constant 1
   eq
   if-goto BASECASE
   push argument 0
   push argument 0
   push constant 1
   sub
   call factorial 1
   call mult 2
   return
label BASECASE
   push constant 1
   return

function mult 2
   // Code omitted
```



The caller (`main` function) wanted to compute 3!

- it pushed 3, called `factorial`, and got 6
- from the caller's view, nothing exciting happenned...



Other topics

- How do we manage the stack pointer in memory and implement push and pop operations?
- The text builds the stack from low memory to high memory, but this is not how most stacks are implemented.
- Most stacks build down from high memory addresses to lower addresses to allow for heap allocation and the stack to share the free space that lies between them