Runtime Environment

CSE 340 FALL 2021

Rida Bazzi

NOTE I covered only a small part of these notes in class

Runtime Environment

So far, we have looked at how to analyze the syntax (parsing) and some semantics (type checking) of a program.

Now, we will consider how to generate a lower-level representation of the program that we can execute on a target machine. In particular, we will look at

1. What memory allocation means (required)
2. Generating code for complex expressions (not covered)
3. What is involved in function calls (required)
    1. call setup (required)
    2. call return (required)
    3. who can do what: caller, callee or either (required)
    4. allocation of memory for local variables (required)
    5. accessing local and global variables (required)
    6. calling conventions (required)
    7. saving and restoring registers (required)
    8. accessing non-local variables (not covered)

NOTE I covered only a small part of these notes in class

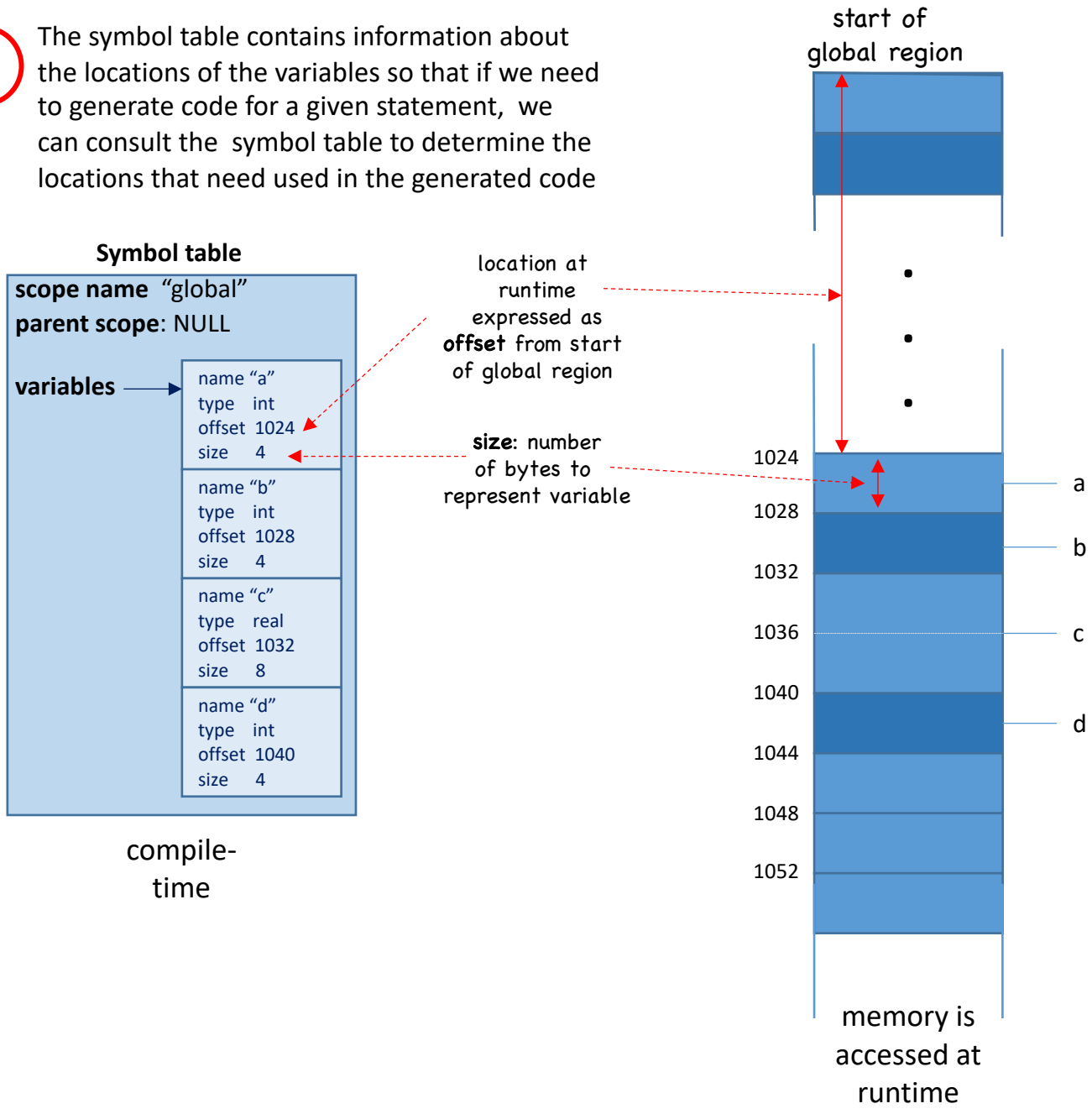
Memory Allocation for Global Variables

Consider a program with global variable declarations

```
a : int;
b : int;
c : real;
d : int;
```

In my exposition, I will assume all variables have space allocated in memory, but in practice, some variable might have registers allocated to them for the duration of their lifetime. The distinction is not really fundamental

1 The symbol table contains information about the locations of the variables so that if we need to generate code for a given statement, we can consult the symbol table to determine the locations that need used in the generated code



Memory Allocation for Global Variables

2 the information in the symbol table is used to generate the code that will be execute at runtime for the program

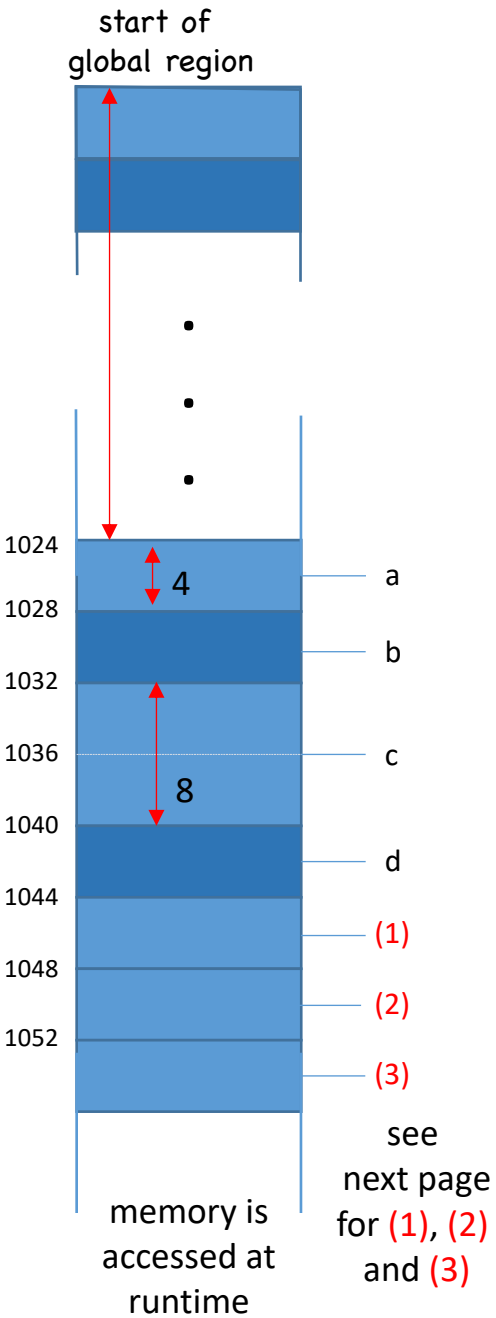
Symbol table

scope name "global"
parent scope: NULL

variables →

name "a"	
type	int
size	4
offset	1024
name "b"	
type	int
size	4
offset	1028
name "c"	
type	real
size	8
offset	1032
name "d"	
type	int
size	4
offset	1040

compile-time



For example

```
a = 5
b = a
print b
```

→

```
mem[1024] = 5
mem[1028] = mem[1024]
system_print mem[1028]
```

Note 1024 (address of a) and 1028 (address of b) are constants that are known at compile time. We call the address an offset (in reality the actual address is determined when the program is linked) . I will use a_{offset} and b_{offset} for to refer to the addresses of the global variables a and b.

Note if we are generating assembly code, values need to be typically moved to registers before they are manipulated. Also, accessing floating point or integer values might require different instructions. In what follows, I assume that all variables are integers

Accessing global variables: examples

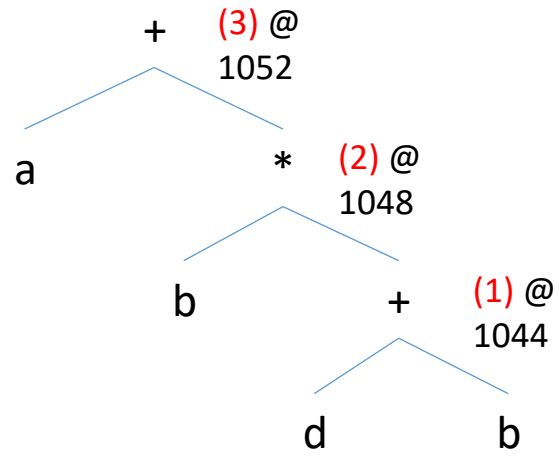
As another example, consider

a = a + b ⟹ $mem[a_{offset}] = mem[a_{offset}] + mem[b_{offset}]$

Finally consider a more involved expression:

a = a + b * (d + b)

For this we draw the expression tree:



we break down the computation into smaller computations of the tree node values as follows

(1) = d + b

$mem[1044] = mem[d_{offset}] + mem[b_{offset}]$

(2) = b * value of node (1)

$mem[1048] = mem[b_{offset}] * mem[1044]$

(3) = a + value of node (2)

$mem[1052] = mem[a_{offset}] + mem[1048]$

a = value of node (3)

$mem[a_{offset}] = mem[1052]$

location 1052 contains value of expression a + b * (d + b)

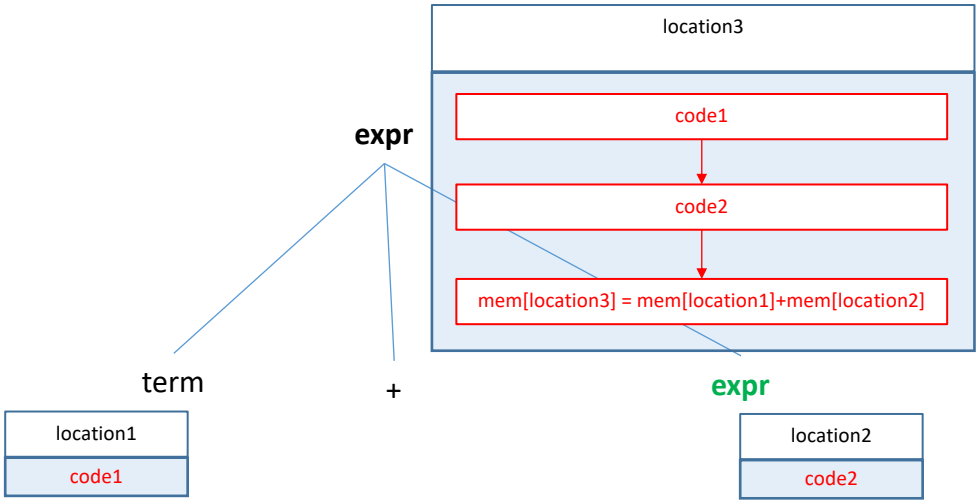
Generating Code for Complex Expressions (not covered)

```
struct code_node * parse_factor()
{
        tok = lexer.getToken();

        switch tok.token_type
        case ID :
        {       n = make_code_node();
                n->code = NULL;
                n->location = find_location(ID);
                return n;
        }
        case NUM :
        {       n = make_code_node();
                n->location = new_location();
                n->location->value = string_to_int(tok.lexeme);
                n->code =  NULL:
                return n;
        }
        case LPAREN:
        {       n = parse_expr();
                expect(RPAREN);
                return n;
        }
}
```

Generating Code for Complex Expressions (not covered)

```
struct code_node * parse_expr()
{
        n1 = parse_term();
        tok = lexer.getToken();
        if (lexer.tok_type = PLUS)
        {       n = make_code_node()
                n2 = parse_expr();

                // append the code of of n1 to the end of
                // the code of n2
                n->code = append(n1->code,n2-code);

                // reserve new location for the result
                // of the expression
                n->location = new_location();

                // make an assignment statement that does the
                // computation and stores
                // the result in the location of the expression
                s = make_assign_stmt( n->location, n1->location,
                                                n2-location, PLUS);

                // append the assignment statement to
                // the end of the code.
                n->code = append(n->code, s);
                return n;

                // the next page illustrates the
                // relationship between n1, n2 and n
        }
        ...

}
```

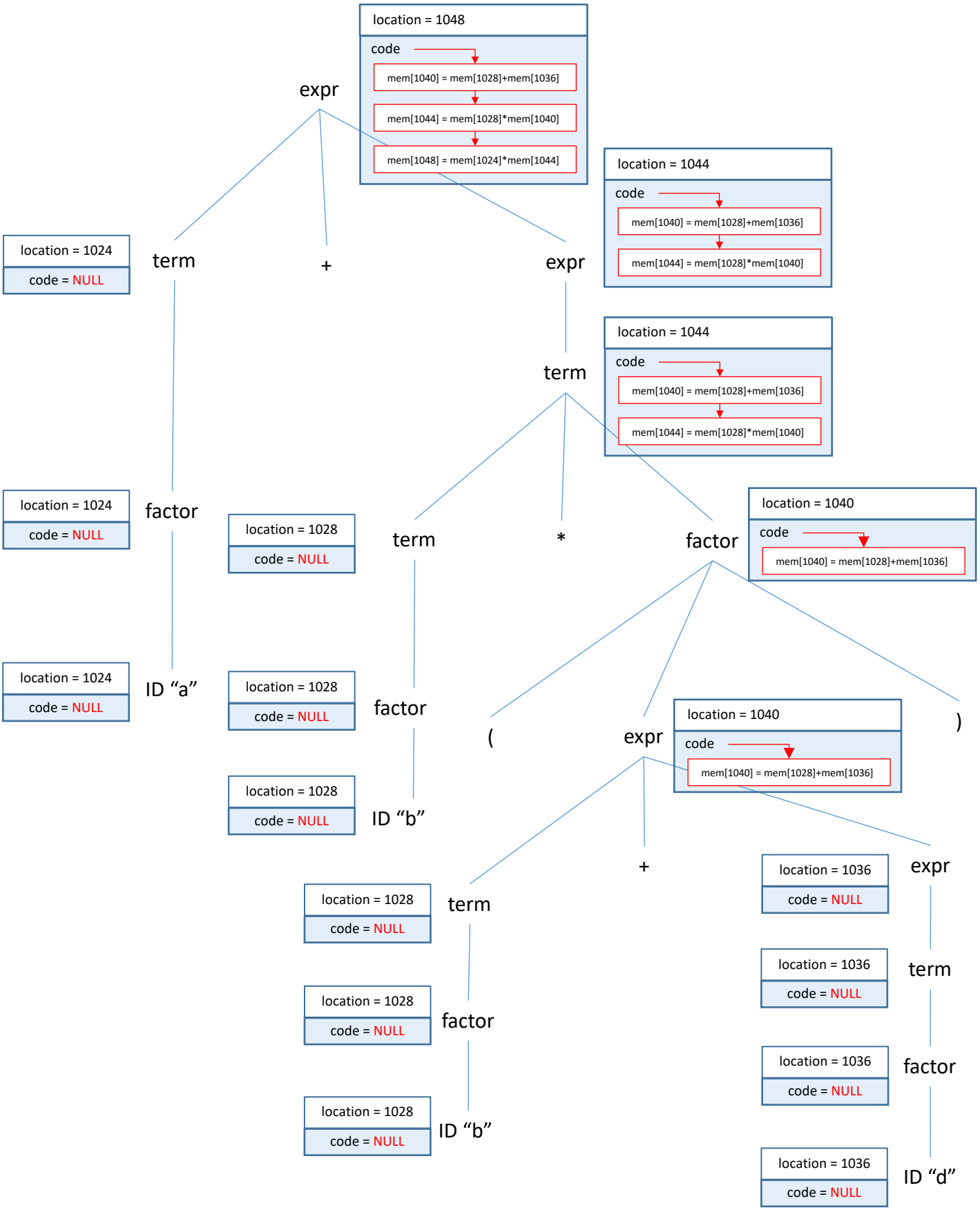
Illustration of how to generate code for expressions recursively



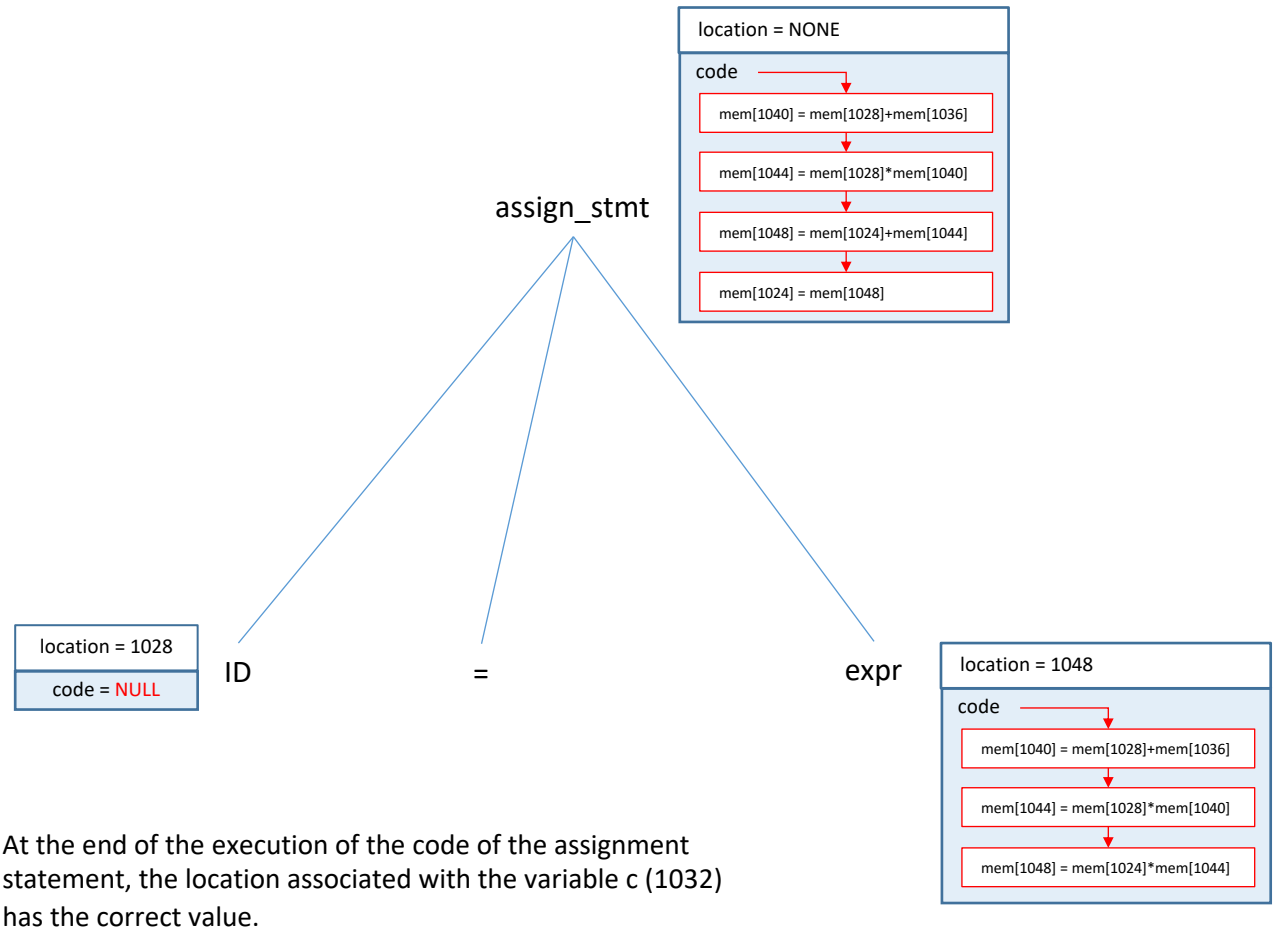
To evaluate the **expr** at the root, we need to evaluate the term (code 1) and the **expr** in term+expr (code2), then we need to add the two values which are at location1 and location2 respectively and store the result in the location for **expr**.

The next page shows how this applies to a larger example

Example code generation for complex expressions: a+b*(b+d)



Example code generation for complex expressions: b = a+b*(b+d)



At the end of the execution of the code of the assignment statement, the location associated with the variable c (1032) has the correct value.

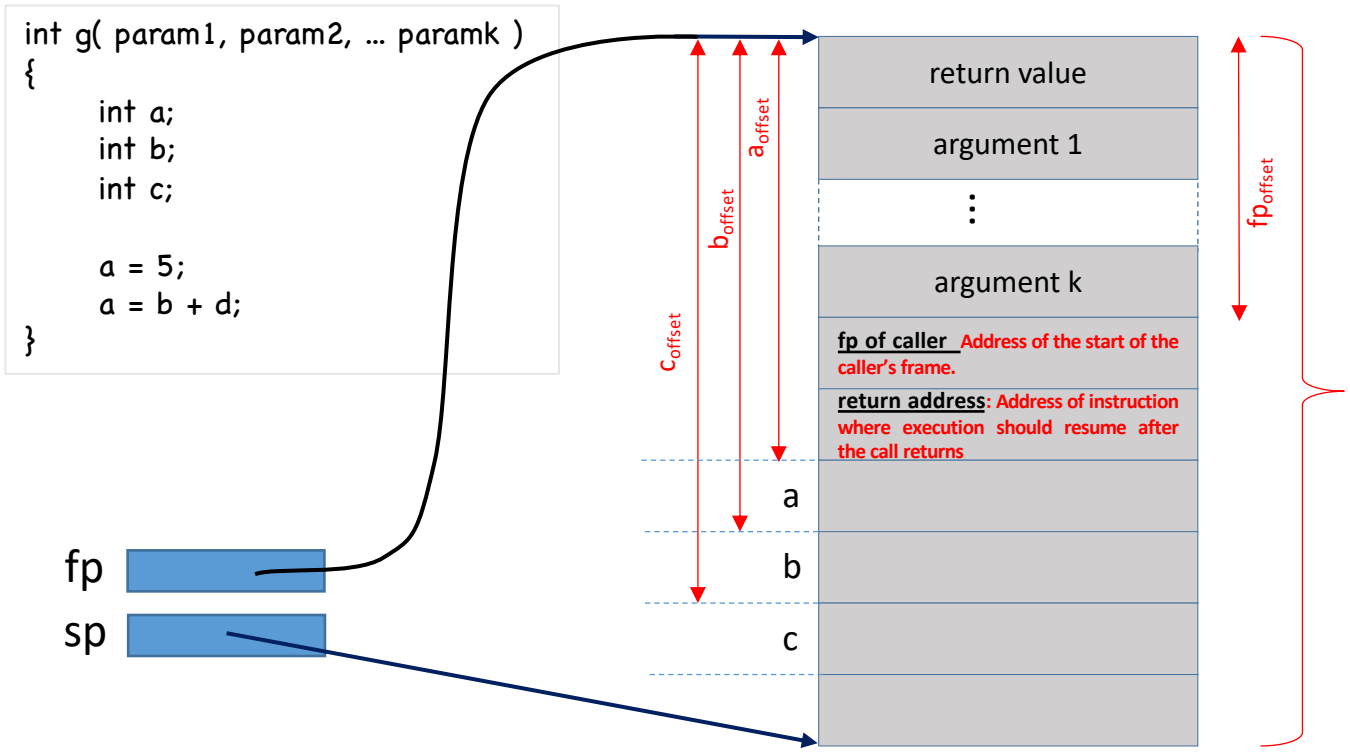
Function Calls: Activation Records

When a function is called, an activation record (also called frame) is created to allocate space for (1) the local variables that are active for the duration of the call (the time during which the called function is active), (2) the arguments passed to the function, and (3) the return value of the function.

The allocated space also contains **information** that is needed to restore the caller's state when the call returns

The activation record of an executing function is typically on the stack*. The start address of the frame is typically stored in a hardware register call **fp**: **frame pointer**. The frame pointer is also called the environment pointer. The address of the memory just beyond the end of the frame is typically stored in a hardware register called **sp**: **stack pointer**.

The **frame** extends from the memory location pointed to by fp to the memory location pointed to by sp.



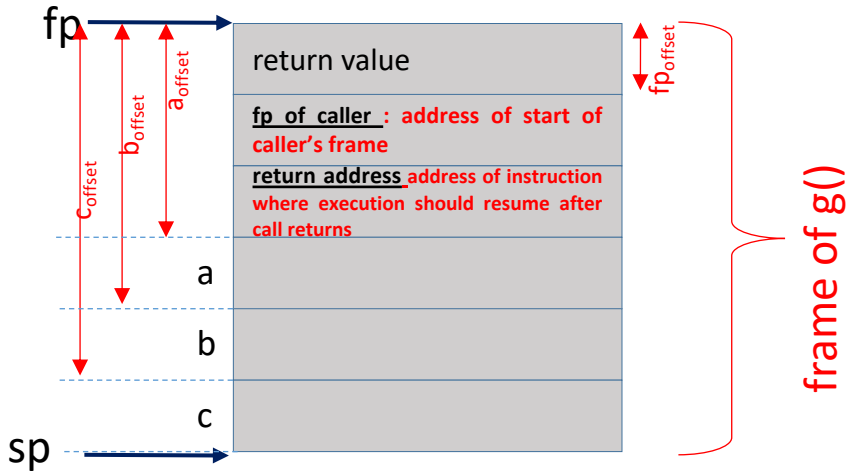
Note. There is one hardware fp register and one hardware sp register. To support multiple function calls, the values of fp and sp are updated when a function is called and restored when a call returns.

For a given function call, the values of fp and sp are not known at compile time

* Some languages that support dynamically created functions and allow function state to persist after the function exits would require the frames to be allocated on the heap

Activation Record: Example

```
int g( )
{
    int a;
    int b;
    int c;

    a = 5;
    a = b + d;
    return a;
}
```



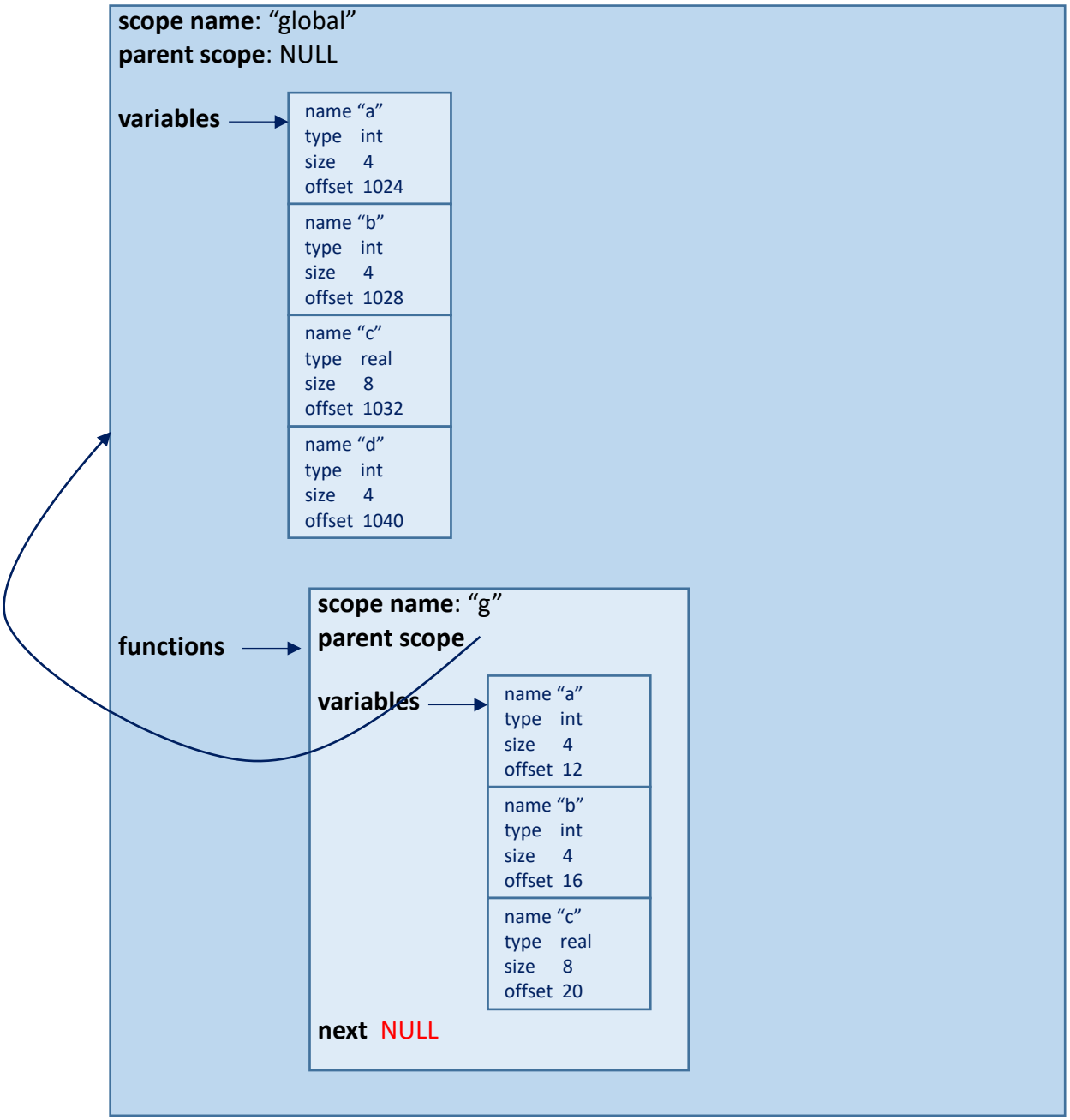
In this example, there are no parameters and the local variables, a, b , and c are given offsets 12, 16, and 20. This assumes that the return value, the fp of the caller and the return address are 4 bytes each (32-bit)

For local variables, the offset is the address of the variable within the frame (the frame starts at offset 0) . The address of the local variables would then be

address of local variable a = fp + a_{offset}

Note how the starting address of the caller's frame is stored in the callee's frame. This value is needed to restore fp when the callee returns. I talk about that some more on page 14.

Symbol Table



In the symbol table, the local variables, a, b , and c are given offsets 12, 16, and 20. This assumes that that the return value, the fp of the caller and the return address are 4 bytes each (32-bit)

For local variables, the offset is the address of the variable within the frame (the frame starts at offset 0) . The address of the local variables would then be

address of local variable a = fp + a_{offset}

Accessing local variables

```
int g( )
{
   int a;
   int b;
   int c;

   a = 5;
   a = b + d;
}
```

global

local

consider the statement in g( )

a = b + d

where d is the global variable d and a and b are the local variables of g( )

The statement will be translated to

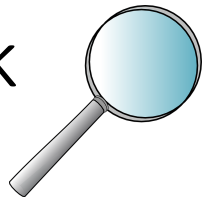
$mem[\ fp + a_{offset}\ ] = mem[\ fp + b_{offset}] + mem[\ d_{offset}\ ]$

known at runtime

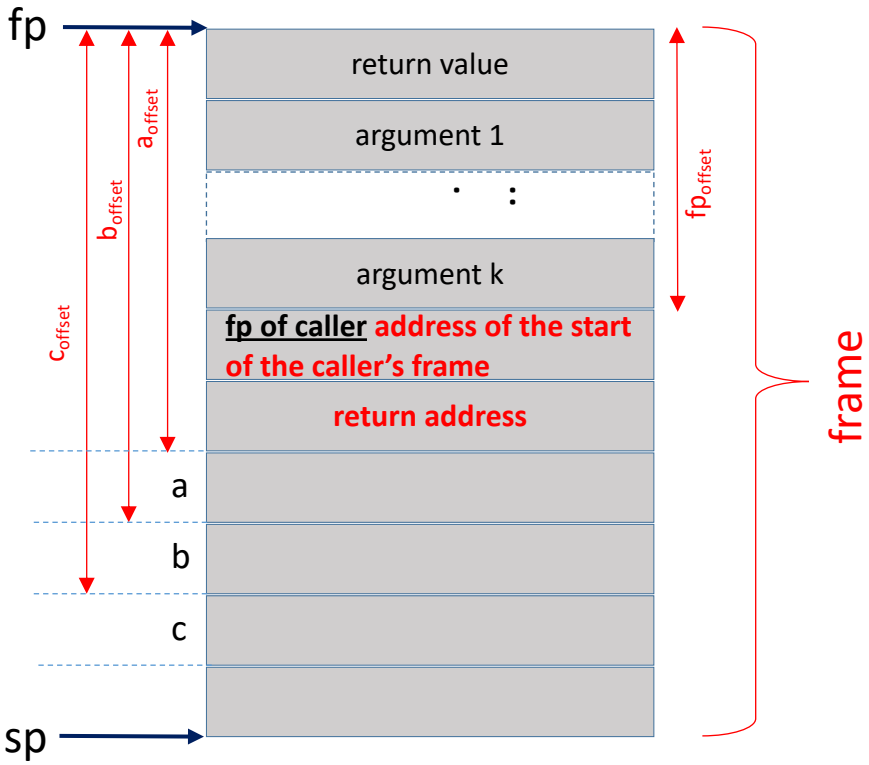
known at compile-time

It is important to understand that the compiler has the information needed at compile time to generate code like the one above. This is achieved by looking up the variables in the symbol tables. Accesses to local variable x is translated to $mem[fp+xo_{ffset}]$. At compile time the value of fp is not known but the "code" to access the variable does not depend on the value of fp. The compiler would also need to generate some other code to update the value of fp. We will talk about that later

Activation Records: A Closer Look



We have already described the local variables and their offsets



return value and arguments

the activation record will also contain the return value and the parameters. The return value is typically at the start of the frame (if it is not in a register). If these quantities are on the stack, they also have offsets relative to fp. So, we will have $return\_value_{offset}$ $argument1_{offset}$, $argument2_{offset}$, ...

An alternative organization, which is commonly used, stores the arguments in the activation record of the caller (see next page).

fp of caller and return address

In addition to the return value, the arguments, and the local variables, we need to store the address of the start of the frame of the caller and the code address where execution should resume after a call returns.

The starting address of the caller's frame is needed so that the fp register can be restored after the call returns. We call the value of the starting address of the caller's frame the frame pointer of the caller. It is needed so that the caller can access its local variables after the call returns

The return address is the address of the next instruction that should be executed when the caller resumes execution after the call returns.

fp of caller and return address are stored in the frame of the callee.

- fp_{offset} is the offset from the start of the frame of the callee where the frame pointer value for the caller is stored

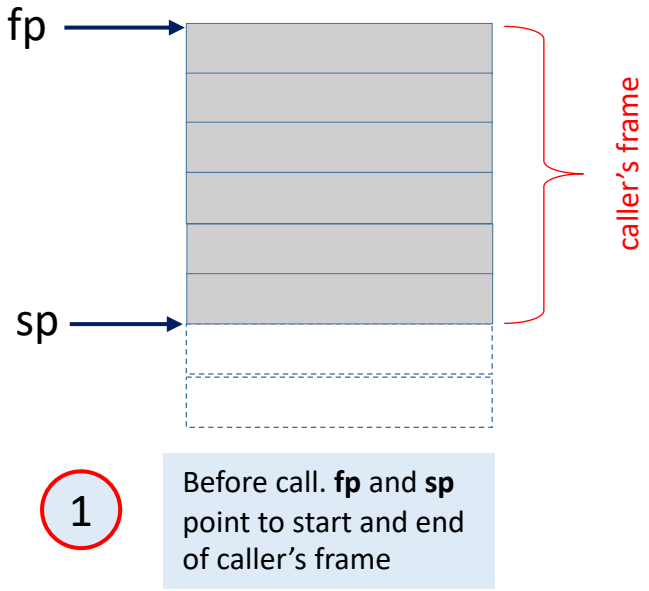
Activation Records: alternative organization

The activation record we have discussed so far stores the arguments in the frame of the callee.

This makes fp_{offset} , $return_{offset}$, and AL_{offset} (see later) dependent on the number of arguments (assuming the arguments are stored before them). More importantly, this makes the offsets of local variables dependent on the number of arguments.

The activation record we have seen so far does not support a variable number of parameters because in that case some offsets cannot be determined at compile time as they depend on the actual number of parameters

An alternative organization would store the arguments in the frame of the caller. Before the call, the caller increases the frame size (by changing sp) to accommodate the arguments. The callee would then access the arguments in the frame of the caller by using negative offsets (assuming stack grows from low to high). This organization is illustrates below



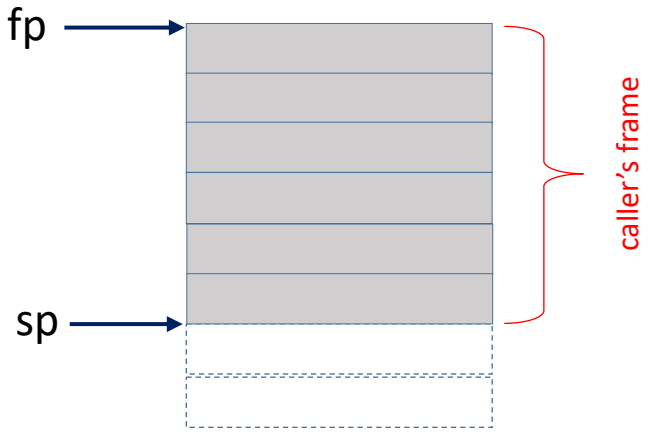
Activation Records: alternative organization

The activation record we have discussed so far stores the arguments in the frame of the callee.

This makes fp_{offset} , $return_{offset}$, and AL_{offset} (see later) dependent on the number of arguments (assuming the arguments are stored before them). More importantly, this makes the offsets of local variables dependent on the number of arguments.

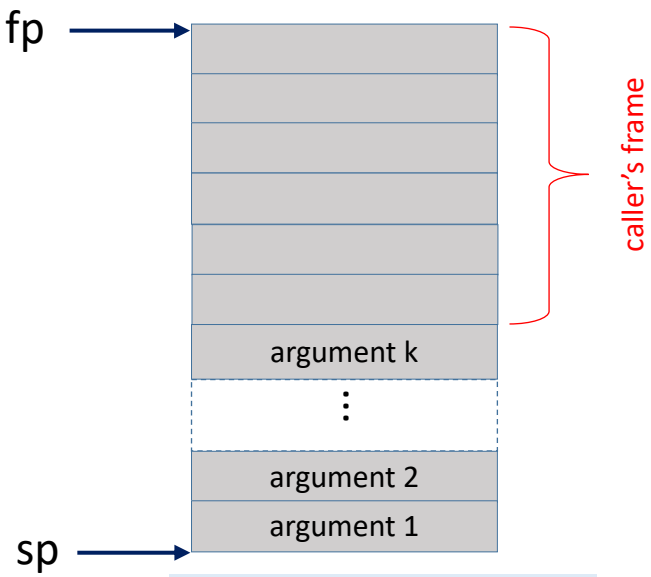
The activation record we have seen so far does not support a variable number of parameters because in that case some offsets cannot be determined at compile time as they depend on the actual number of parameters

An alternative organization would store the arguments in the frame of the caller. Before the call, the caller increases the frame size (by changing sp) to accommodate the arguments. The callee would then access the arguments in the frame of the caller by using negative offsets (assuming stack grows from low to high). This organization is illustrates below



1

Before call. **fp** and **sp** point to start and end of caller's frame



2

callers adjusts its frame size by changing **sp** and stores the arguments in its own frame

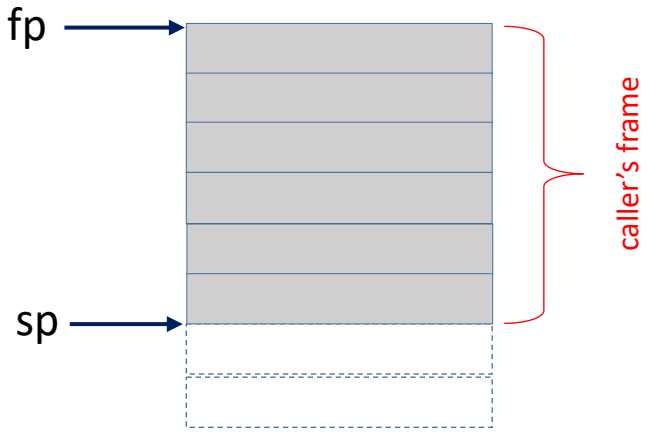
Activation Records: alternative organization

The activation record we have discussed so far stores the arguments in the frame of the callee.

This makes fp_{offset} , $return_{offset}$, and AL_{offset} (see later) dependent on the number of arguments (assuming the arguments are stored before them). More importantly, this makes the offsets of local variables dependent on the number of arguments.

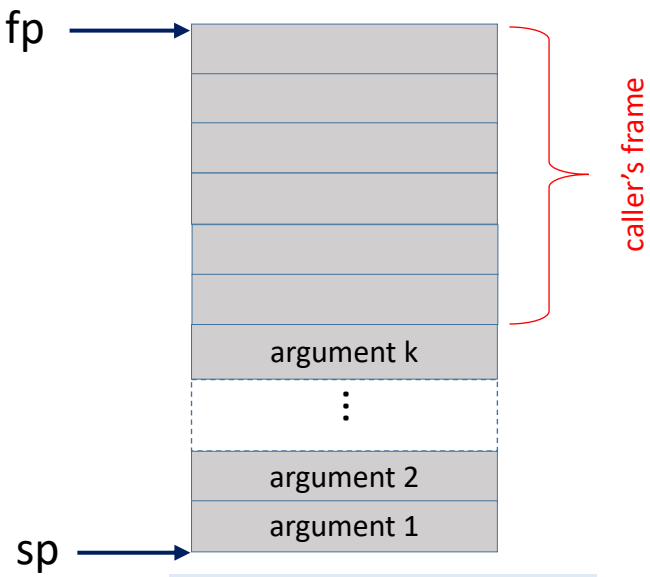
The activation record we have seen so far does not support a variable number of parameters because in that case some offsets cannot be determined at compile time as they depend on the actual number of parameters

An alternative organization would store the arguments in the frame of the caller. Before the call, the caller increases the frame size (by changing sp) to accommodate the arguments. The callee would then access the arguments in the frame of the caller by using negative offsets (assuming stack grows from low to high). This organization is illustrates below



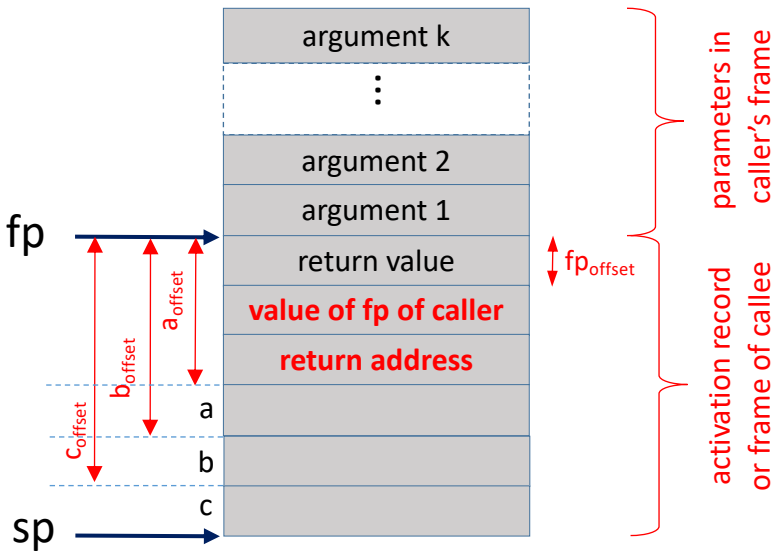
1

Before call. **fp** and **sp** point to start and end of caller's frame



2

callers adjusts its frame size by changing **sp** and stores the arguments in its own frame



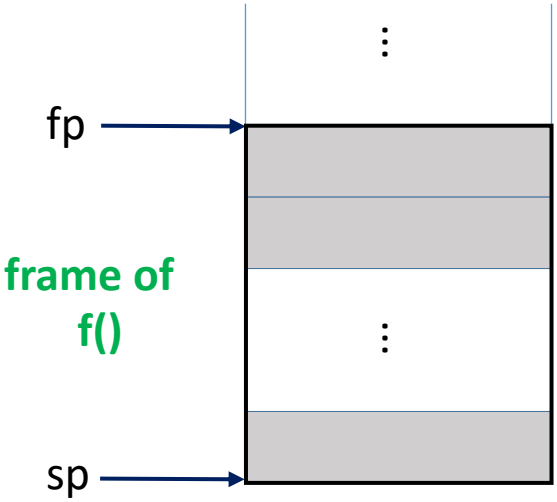
3

callee is now active. **fp** and **sp** point to start and end of callee's frame. Callee can access the parameters from caller's frame

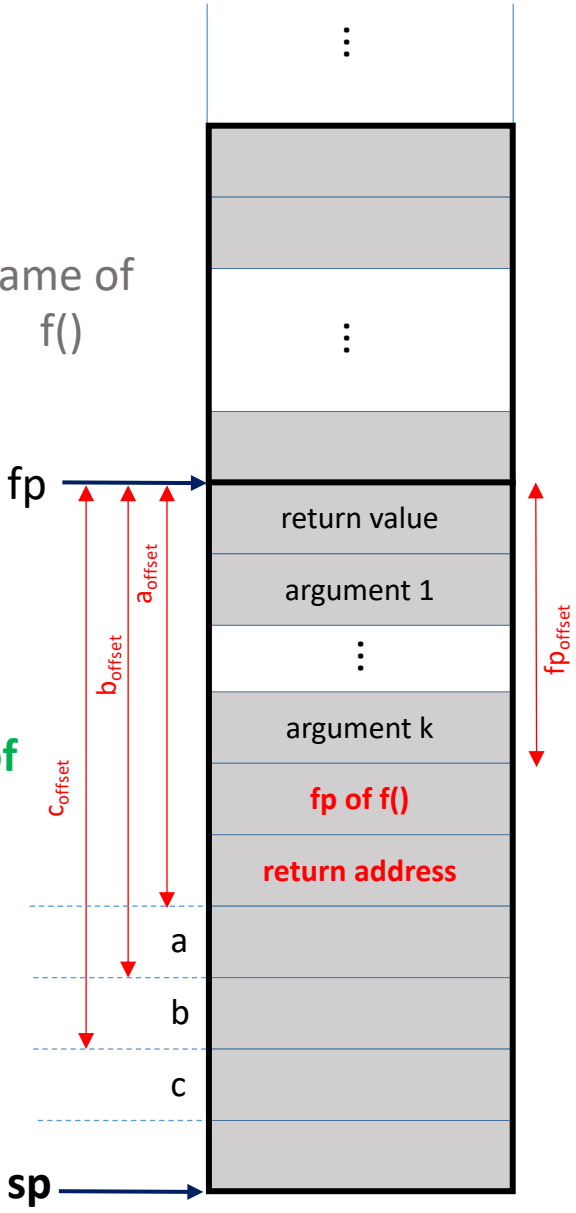
Function call: a closer look

We consider a function f() that calls a function g().

Before the call, **f() is active** and we have the following stack configuration:



when call is made, f() is not active and **g() becomes active**



When a call is made, we need to:

1. calculate new values for fp and sp so that they point to the beginning and end of the callee's frame
2. store information about the caller's environment to be able to recover the caller's environment when the call returns

We will look at each one separately

Calculating new values for fp and sp

The new values for fp and sp are given by the following formula

$$fp_{new} = sp_{old}$$
$$sp_{new} = fp_{new} + \text{frame\_size of callee}$$

Note frame size is given in unit of addressable space. It can be in bytes or words depending on the architecture. Some architectures allow addressing individual bytes even if the bus size is multiple bytes. Some architecture only allow addressing at the word granularity

Note In the equations above, we are saying how new values of fp and sp relate to old values of fp and sp. On a given architecture, there is one fp and one sp register, so the equation above say how those registers should be updated.

Caller's obligations

We consider a function f() that calls a function g().

Caller must do the following

1. save return address

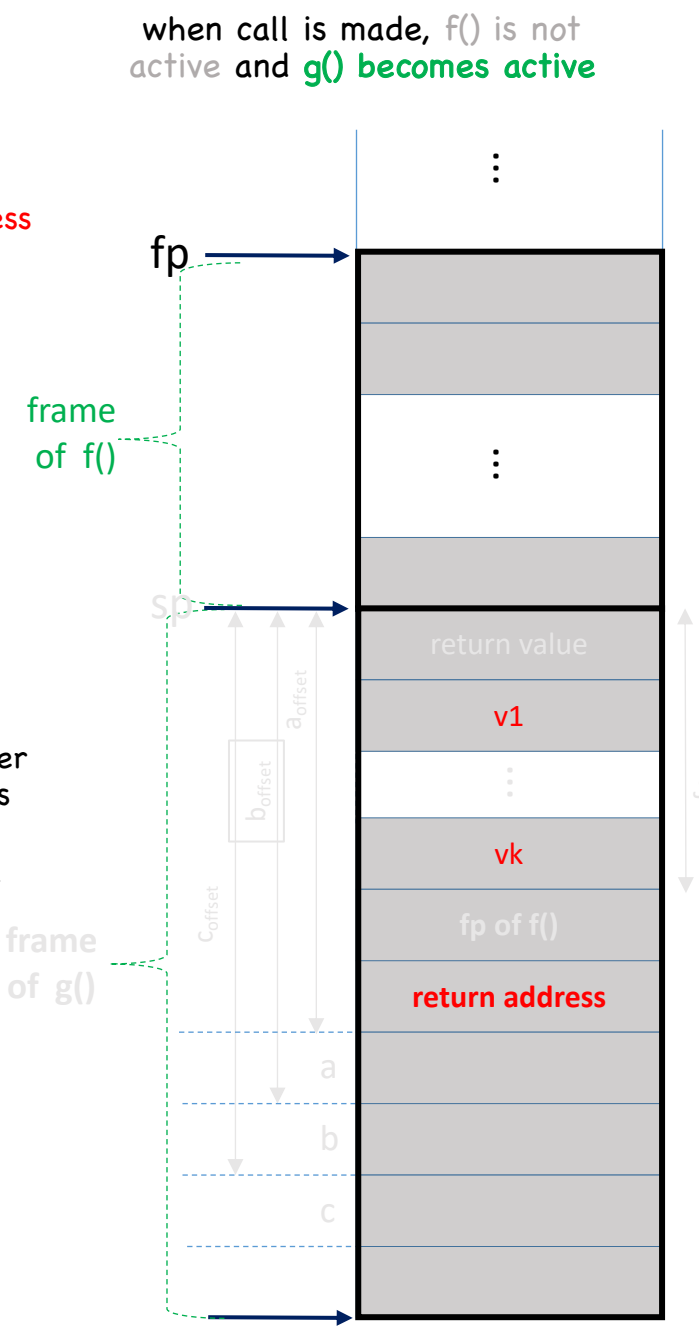
   mem[sp+ret_address$_{offset}$] = return address

2. calculate and copy arguments

   mem[sp+argument1$_{offset}$] = v1
   mem[sp+argument2$_{offset}$] = v2
   …
   mem[sp+argumentk$_{offset}$] = vk

where v1 … vk are the calculated values of the k parameters

Note In the assignments, sp is used as the base of the offset because we assume that these assignments are executed by the caller before fp. The value of sp when the caller is active before the call is the same as the value of fp when the callee becomes active.



Why can't callee save return address and calculate the argument values?

1. callee does not know at what point in the execution the call is made
2. callee does not know the specific arguments that will be used for a particular call

Callee's obligations

We consider a function f() that calls a function g().

Callee must do the following

1. calculate sp for its own frame

   sp = fp + frame size

2. calculate and save return value

   mem[fp+ret_val$_{offset}$] = return value

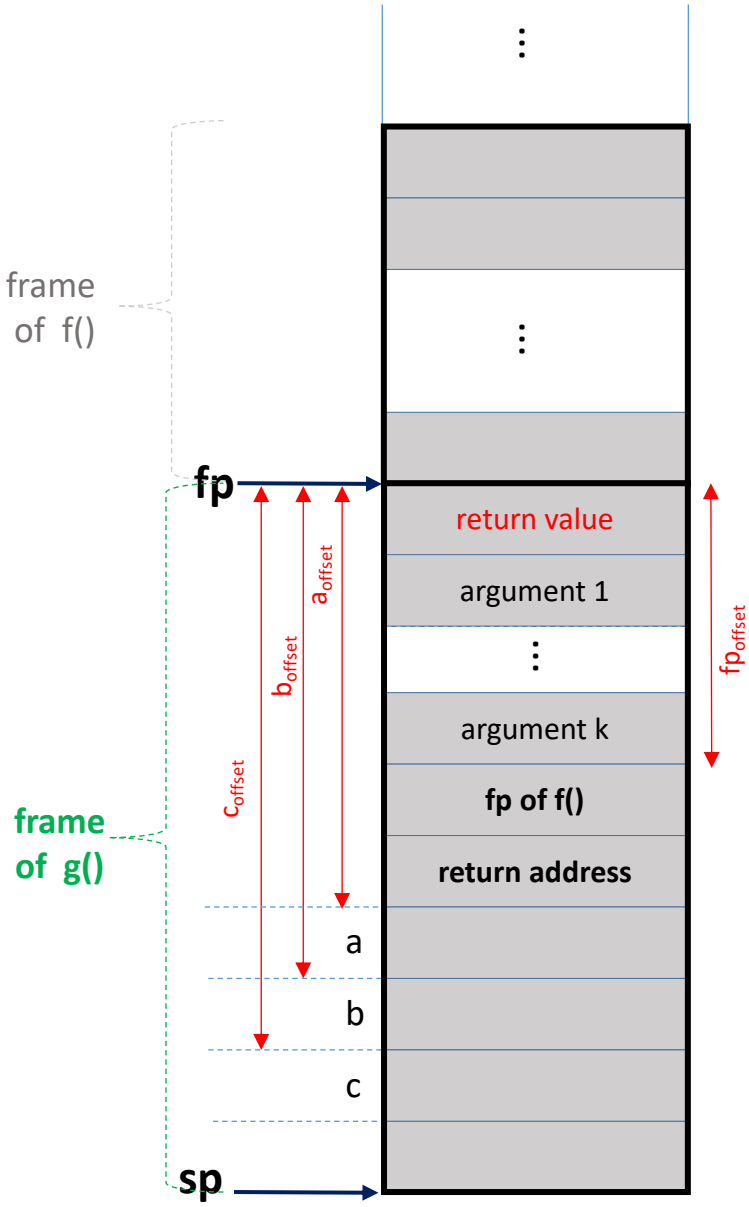
Here fp is used because this is done after fp is updated.

Why can't the caller calculate sp of callee and calculate return value?

1. Calculating sp of callee requires knowing the frame size of the callee. The size of the frame of the callee depends on the number and size of local variables used by the callee. This information is not available to the caller. The caller only needs to know the callee's signature (types of parameters and return value) in order to make the call.

2. The return value is calculated by the callee!

when call is made, f() is not active and g() becomes active



Other obligations by caller or callee

We consider a function f() that calls a function g().

when call is made, f() is not active and g() becomes active

Caller can do the following

1. save the value of its frame pointer

   mem[sp+fp_{offset}] = fp

2. setup fp of callee (after saving its own frame pointer)

   fp = sp

caller cannot setup sp of callee

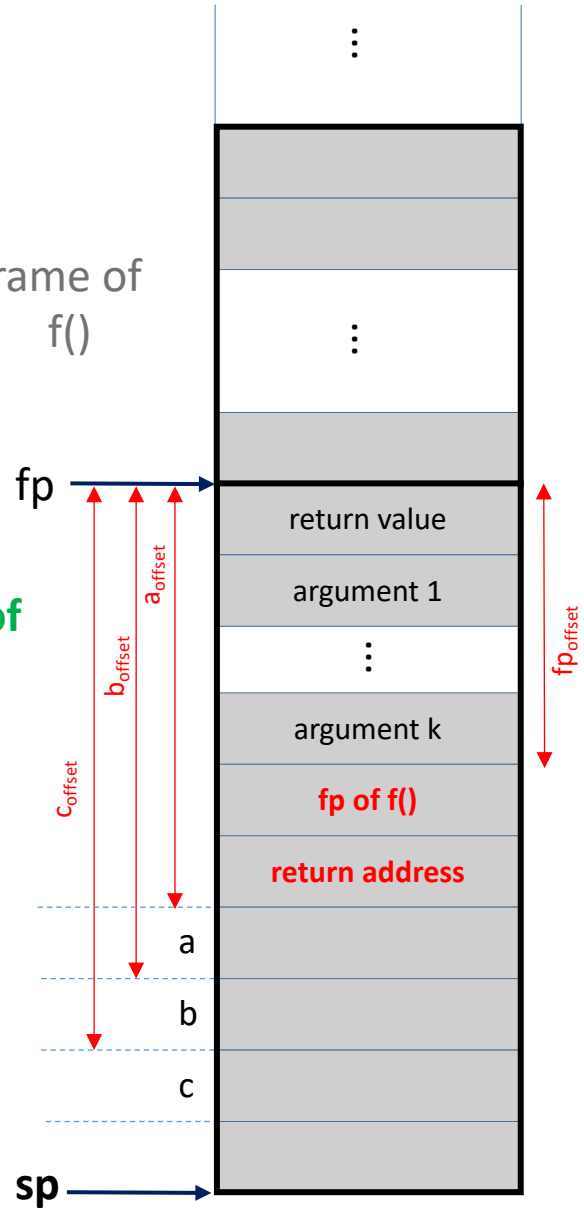
Callee can do the following

1. save fp of caller (before fp and sp are changed)

   mem[sp+fp_{offset}] = fp

2. calculate its own fp and sp (after fp of caller is saved)

   fp = sp
   sp = sp + frame size



Control Link

The control link is a pointer to the activation record of the calling environment

In the example above, f() calls g(), so the value of fp of f() that is saved in the frame of g() is the control link

The address of the control link in a given frame is

$$fp + fp_{offset}$$

where fp is the frame pointer of the current frame and fp_{offset} is the offset relative to the beginning of the frame

Note I use frame and activation record interchangeably

Calling Conventions

It is important to have a calling convention that both caller and callee can follow so that they are in agreement on each other's responsibility regarding the call:

- who does what?
- where are values stored?

Essentially, everything we considered as content and location of frame is specified by the calling convention

Note that in general, the compiler does not generate the code for both caller and callee. Callee can be a library function that is separately compiled. So, it is not up to a compiler to decide on its own calling convention. A compiler should compile according to some calling convention. The particular convention can depend on the target computer architecture and the programming language itself.

In general the arguments and return value need not be on the stack. They can be in registers depending on the number and size of parameters.

Also, in some languages, a pointer to an argument can be provided to the callee and the actual argument is on the heap.

The details of a particular calling convention is not important to us.

It is important, though, to understand the need for a calling convention, and to understand what a calling convention specifies

Saving and Restoring the Registers

One point that I have omitted so far is a discussion of registers.

On most architecture, the operands of an operation are values that are present in registers.

Functions need to move values from memory to registers in order to do computations on those values and then the results can be moved back to memory to free up registers for other operations

Calling conventions also specify what to do with registers when a function call is made.

The issue is the following. If at the time of a call,

(1) a register contains a value that the caller needs after the call returns and
(2) the same register is overwritten by the callee during the call,

the value needed by the caller needs to be saved (by the caller or callee) and restored when the call returns.

On the next page, we consider various possibilities

Note two specific registers that we already considered are fp and sp

Saving and Restoring the Registers

We consider various possibilities (a given calling convention would follow one of these possibilities)

1. Caller-saved. In this convention, the caller is responsible for saving the registers whose values it needs. The caller is also responsible for restoring the values when the call returns

2. Callee-saved. In this convention, the callee is responsible for saving any registers it uses. The callee is also responsible for restoring the values before the call returns

Advantages/disadvantages

If callee uses many registers, then caller-saved is better because registers that are needed by caller are fewer than those modified by callee and only registers that are needed by caller are saved and restored

If callee uses few registers, then callee-saved is better because registers that are modified by callee are fewer than those that are needed by caller and only registers modified by callee are saved and restored

3. **Caller/Callee-saved**. This option divides registers into two groups

1. One group of registers is caller-saved
2. One group of registers is callee-saved

Caller attempts to use only callee-saved registers and callee attempts to use only caller saved registers. This way no registers need to be saved

Advantages/disadvantages

This work for single level of calls and if there are deeper levels, caller and callee are switched and more registers would need to be saved

Accessing non-local variables

Note I did not cover the material on access links in class.

So far we have talked about accessing global and local variables

In some languages like Javascript, functions can be defined inside the scope of other functions

For such nested functions, an inner function might access a variable that is declared in its parent scope. Such access is not local and is not global and we need a way to achieve it

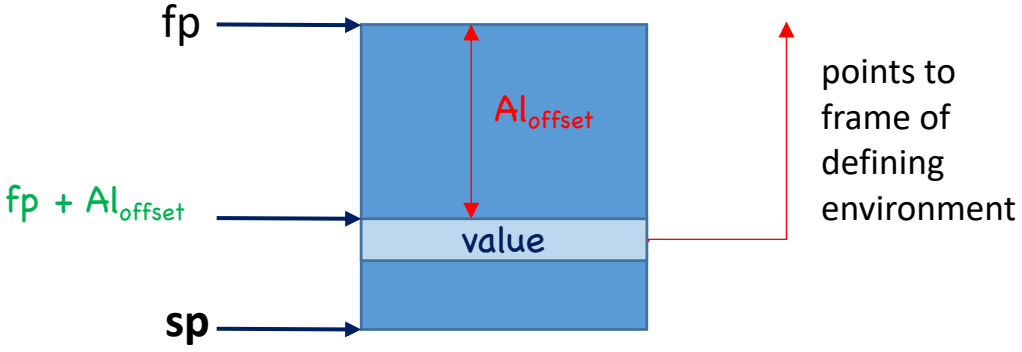
non-local access is made easier using *access links* .

Access link. The access link of an activation record of a function g() points to the activation record of the defining environment of g()

The access link is stored in the activation records.

1. AL_{offset} is the offset of the access link within the activation record
2. The address where the access link is stored in an activation record is

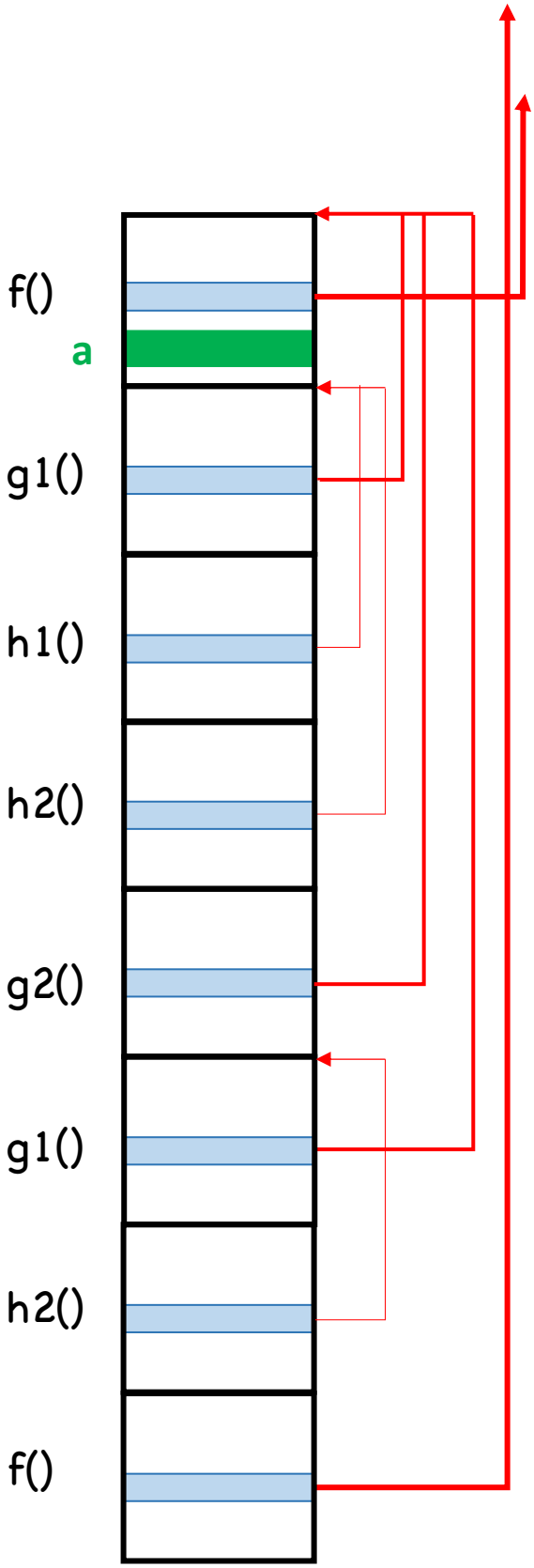
   $fp + AL_{offset}$
3. The value of the access link is the content of the memory whose address is $fp + AL_{offset}$



Accessing non-local variables

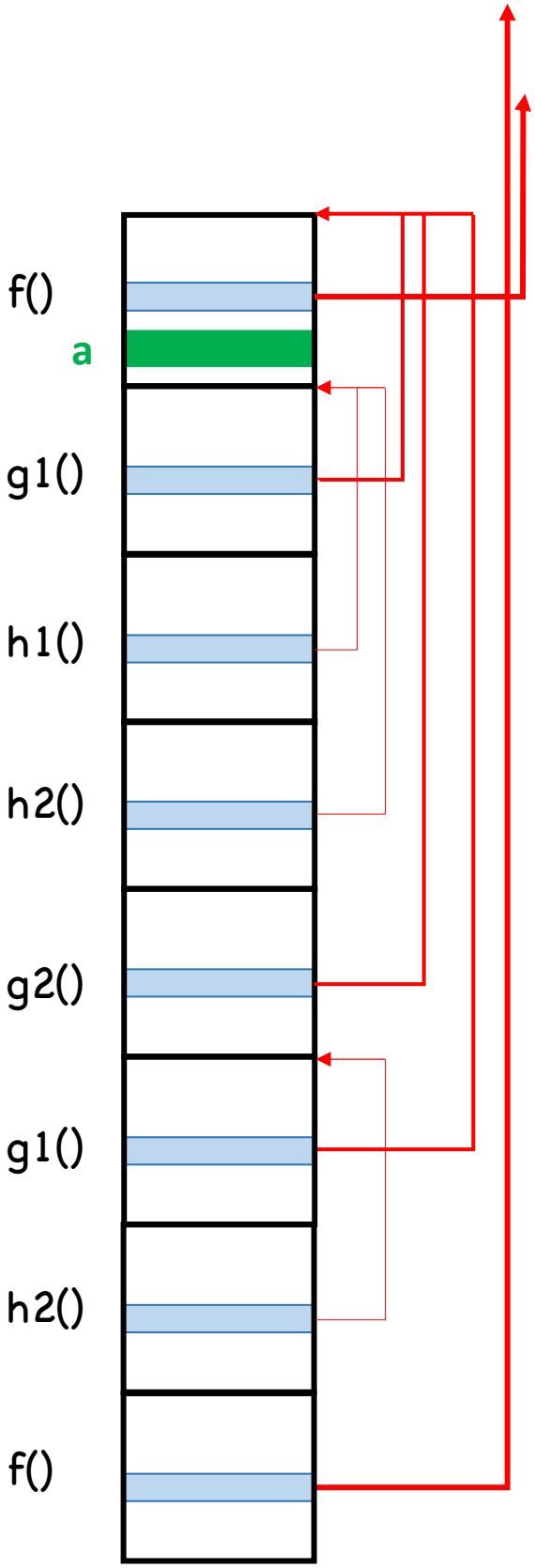
We consider a simple example

```
f()
      a      // local variable in f()

      g1()  // function defined in f()

            h1() // function defined in g1()
            {
                ...              // body of h1()
            }

            h2() // function defined in g1()
            {    ...
                 a = a + 1;  // body of h2()
                 ...
            }

      {
          ...    // body of g1()
      }

      g2() // function defined in f()
      {
          ...    // body of g2()
      }

{
    .... // body of f()
}
```



Accessing non-local variables

- g1() is defined inside f(): the access link of g1() points to the frame of f()
- h1() is defined inside g1(): the access link of h1() points to the frame of g1()
- h2() is defined inside g1(): the access link of h2() points to the frame of g1()
- g2() is defined inside f(): the access link of g2() points to the frame of f()
- the access link of the second call to f() points to the same frame (not shown) that the access link of the first call to f() points to



Accessing non-local variables

We assume that the access link are correctly calculated (we will see how that is done in a couple pages) and we show how access links can be used to calculate the address of non-local variables

We start with the example of h2() in which the local variable a of f() is accessed

Address of a = fp of f() + a_{offset}

So, to calculate the address of a, h2() needs to calculate the address of fp of f()

The calculation is iterative

```
address = fp                                  // points to frame of h2()
address = mem[address + AL_offset ]           // points to frame of g1()
address = mem[address + AL_offset ]           // points to frame of f()
address = address + a_offset                  // points to location of a
```

This can be rewritten as follows

```
address = fp
for i = 1 to 2
     address = mem[address + AL_offset ]
address = address + a_offset
```

Accessing non-local variables

In general, the calculation for the address of variable x declared in f() and accessed in g() which is L level deep within f() can be done as follows

```
address = fp
for i = 1 to L
    address = mem[address + AL_offset ]
address = address + x_offset
```

where L is the difference in nesting levels between the points where x is accessed and the point where x is declared.

In the example above on the previous page, L = 2

The quantities in green are know at compile time. I highlight this to show that the compiler can generate the necessary code

The actual value of the address is calculated at runtime by executing the code

Calculating Access Link

All that remains is to calculate access links

I remind you of the definition of access link and control link

Access link. The access of an activation record of a function g() points to the activation record of the defining environment of g()

The control link of is a pointer to the activation record of the calling environment

Calculating access links

1. f() calls g1()

here the caller is the defining environment, so access link points to the frame of the caller

The caller executes

$$mem[sp+AL_{offset}] = fp$$

---

2. g1() calls h1()

same as previous case

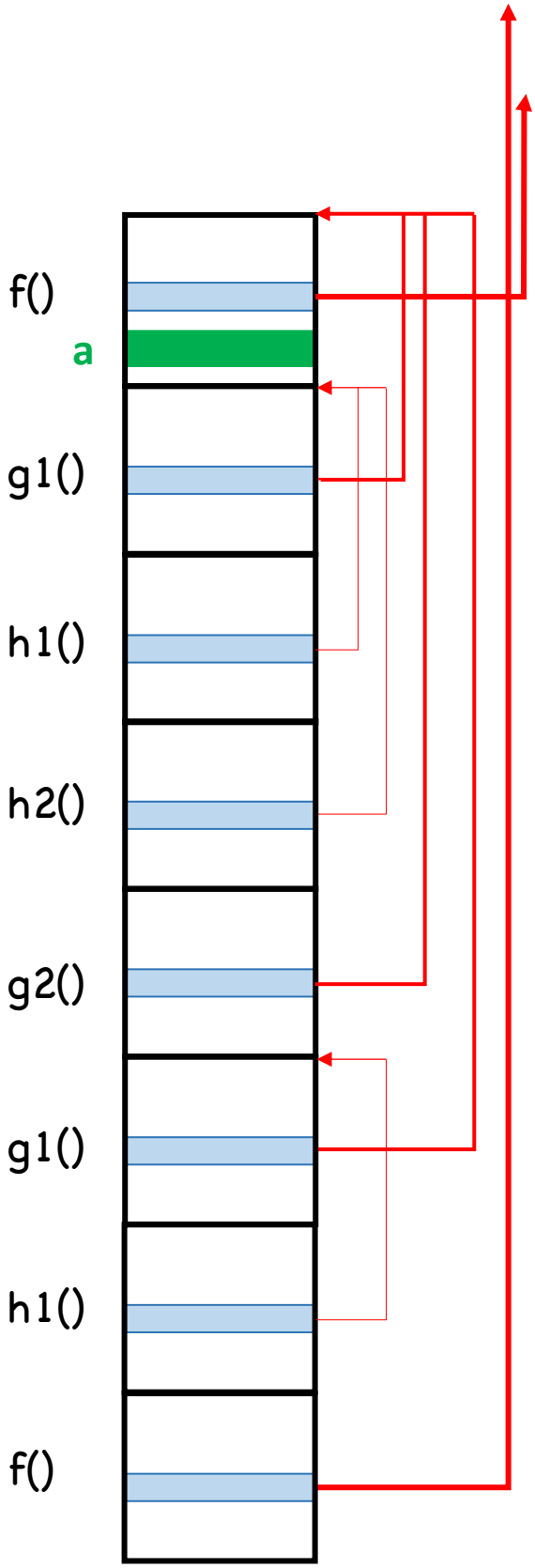
---

3. h1() calls h2()

here h1() and h2() have the same defining environment and their access links have the same value

The caller executes

$$mem[sp+AL_{offset}] = mem[fp+AL_{offset}]$$



Calculating access links

4. h2() calls g2()

this is more interesting

address = fp
// points to current frame

address = mem[address + AL_{offset}]
// points to frame of g1()

address = mem[address + AL_{offset}]
// points to frame of f()

mem[sp + AL_{offset}] = address
// stores access link in frame of callee

---

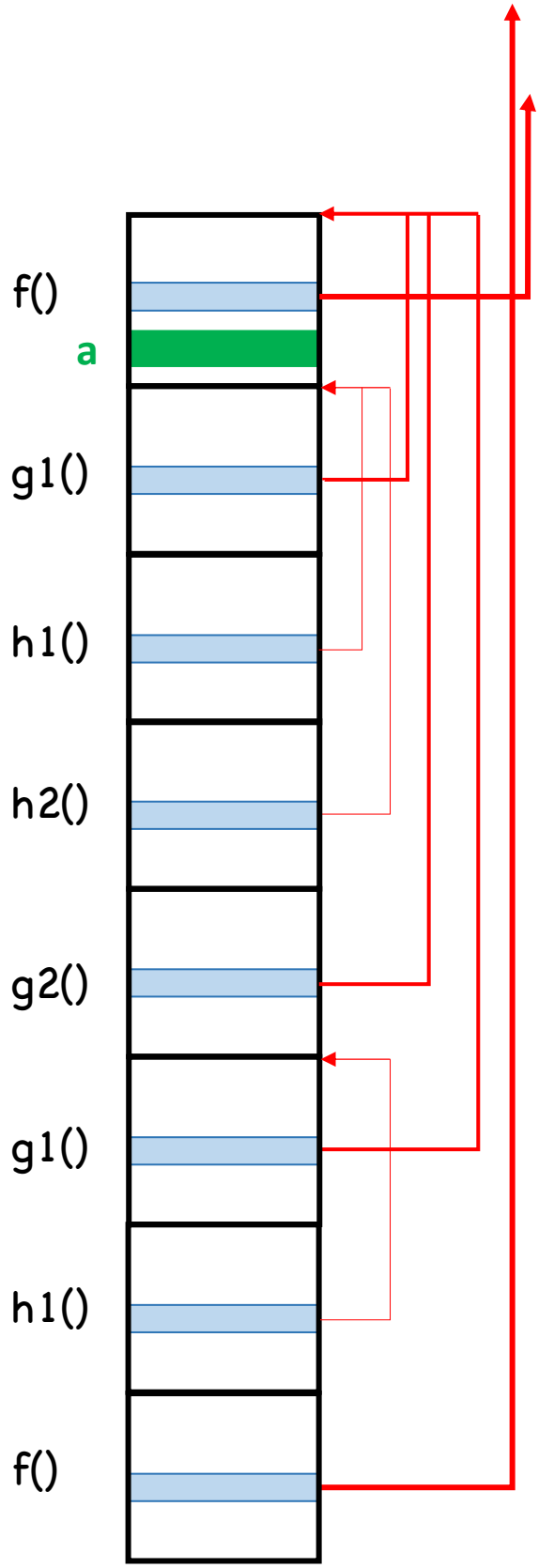
5. g2() calls g1()

same as case (3) on previous page

---

6. g1() calls h1()

same as cases (1) and (2) on previous page



Calculating access links

7. h1() calls f()

This is similar to case (4). We need to follow access links until we get to the frame of the first call to f() where we can copy the access link value

address = fp

address = mem[address + AL_{offset}]
// points to frame of g1()

address = mem[address + AL_{offset}]
// points to frame of f()

address = mem[address + AL_{offset}]
// points to frame of parent scope of f()

mem[sp + AL_{offset}] = address
// stores access link in frame of callee