

The Daily PL - 5/25/2023

Imperative Programming Languages

Any language that is an abstraction of the von Neumann Architecture can be considered an imperative programming language.

There are 5 calling cards of imperative programming languages:

1. *state, assignment statements, and expressions*: Imperative programs have state. Assignment statements are used to modify the program state with computed values from expressions
 1. *state*: The contents of the computer's memory as a program executes.
 2. *expression*: The fundamental means of specifying a computation in a programming language. As a computation, they produce a value.
 3. *assignment statement*: A statement with the semantic effect of destroying a previous value contained in memory and replacing it with a new value. The primary purpose of the assignment statement is to have a side effect of changing values in memory. As Sebesta says, "The essence of the imperative programming languages is the dominant role of the assignment statement."
2. *variables*: The abstraction of the memory cell.
3. *loops*: Iterative form of repetition (for, while, do ... while, foreach, etc)
4. *selection statements*: Conditional statements (if/then, switch, when)
5. *procedural abstraction*: A way to specify a process without providing details of how the process is performed. The primary means of procedural abstraction is through definition of *subprograms* (functions, procedures, methods).

To get our eyes used to seeing each of these components in the wild, let's look at an annotated piece of quintessentially imperative code:

```
"""
A quintessentially imperative program.
"""

# The defining form of abstraction in an imperative programming
# language is procedural (a.k.a. process) abstraction: a way to
# specify a process without providing the details of how the
# process is performed.
def sum_evens(low, high):

    # Variable + assignment statement: The imperative way of
    # updating state in an imperative program.
    sum = 0

    # Loops are the imperative way of defining repeated execution
    # of a series of statements.
    for i in range(low, high):
        # The typical form of selection statement (Sebesta: A
```

```

# selection statement provides the means of choosing
# between two or more execution paths in a program). The
# if statement is a two-way selection statement (per
# Sebesta).
if i % 2 == 0:
    # sum + i is an expression - the fundamental means
    # of specifying computations in a programming language
    # (Sebesta).
    sum = sum + i
return sum

if __name__ == "__main__":
    # 2 + 4
    sum_of_evens_bw_one_five = sum_evens(1, 5)
    print(f"{sum_of_evens_bw_one_five}")

```

(as with all examples, you can find this source code online in the class' git repository:


<https://github.com/hawkinsw/cs3003/tree/main/paradigms>  (<https://github.com/hawkinsw/cs3003/tree/main/paradigms>)

Variables

There are 6 *attributes* of *variables*. Remember, though, that a variable is an abstraction of a memory cell.

1. *type*: Collection of a variable's valid data values and the collection of valid operations on those values.
2. *name*: String of characters used to identify the variable in the program's source code.
3. *scope*: The range of statements in a program in which a variable is visible.
 - Using the yet-to-be-defined concept of binding, there is an alternative definition: The range of statements where the name's binding to the variable is active.
4. *lifetime*: The period of time during program execution when a variable is associated with computer memory.
5. *address*: The place in memory where a variable's contents (value) are stored. This is sometimes called the variable's *l-value* because only a variable associated with an address can be placed on the *left* side of an assignment operator.
6. *value*: The contents of the variable. The value is sometimes call the variable's *r-value* because a variable with a value can be used on the *right* side of an assignment operator.

Looking forward to Binding (New Material Alert)

A *binding* is an association between an attribute and an entity in a programming language. For example, you can *bind* an operation to a symbol: the  symbol can be bound to the addition operation.

Binding can happen at various times:

1. Language design (when the language's syntax and semantics are defined or standardized)

2. Language implementation (when the language's compiler or interpreter is implemented)
3. Compilation
4. Loading (when a program [either compiled or interpreted] is loaded into memory)
5. Execution

A *static binding* occurs before runtime and does not change throughout program execution. A *dynamic binding* occurs at runtime and/or changes during program execution.

Notice that the six "things" we talked about that characterize variables *are actually attributes*!! In other words, those attributes have to be bound to variables at some point. When these bindings occur is important for users of a programming language to understand. We will discuss this further next class.

To get you thinking about the next class, think about the following:

We talked above about the + operator being an entity and, therefore, having a set of characteristics. In the example, we said that the operation (the characteristic) could be mathematical addition. However, that's not always the case, is it? In Python, sometimes the + can be used to execute a string concatenation. Here is a C++ and a Python snippet that both use the + operator. Consider the difference in the binding times of the operation to the symbol for the two languages!

C++

```
#include <iostream>

auto operate_on(int left, int right) {
    return left + right;
}
```

Python

```
def operate_on(left, right):
    return left + right
```

The Daily PL - 5/30/2023

Binding

Recall the definition of *binding*: A binding is an association between an *attribute* and an *entity* in a programming language. We have talked extensively about variables as entities in a programming languages and their attributes. So, it makes sense to say that you can bind those attributes to the entity. However, variables are not the only entities in a programming language that have attributes. For example, you can bind an operation to a symbol: the + symbol can be bound to the addition operation.

Binding can happen at various times:

1. Language design (when the language's syntax and semantics are defined or standardized)
2. Language implementation (when the language's compiler or interpreter is implemented)
3. Compilation
4. Loading (when a program [either compiled or interpreted] is loaded into memory)
5. Execution (i.e., runtime)

A *static binding* occurs before runtime and does not change throughout program execution. A *dynamic binding* occurs at runtime and/or changes during program execution.

Putting Binders On

Binding Example: C++

Consider the following function written in C++:

```
1 auto do_assignment(int x, int y) {  
2     int result = x + y;  
3     return result;  
4 }
```

What are some of the entities and their attributes that are bound in this snippet of code and at what time are they associated?

1. The types of the variables `x`, `y`, and `result` are bound at compile time. C++ knows enough about the types of a program's variables in order to associate the type with a variable when the program is compiled.
2. The value of the variables are bound at runtime. There is a mechanism in C++ to bind values to variables at the time a program is compiled, but that is not in use here.
3. In C++, `auto` is a keyword that can be used in certain places to replace a type. The utility of `auto` is that it saves programmers the hassle of writing out C++'s notoriously verbose type

names (think `std::vector<std::string>`). Just because we declare a variable to have an `auto` type does not suddenly mean that the variable's type can change at runtime. It just means that we will let the compiler determine the actual type for us (what C++ calls the *deduced type*). As a result, the deduced type that the `auto` represents can be determined at the time the program is compiled. Therefore, a binding between the `auto` identifier in this code and its deduced type attribute can be done at the time the program is compiled.

4. The meaning of the `+` operation is bound to "mathematical addition" at compile time. C++ gives the programmer the ability to change the meaning of operators based on the types of their operands. However, because the type(s) of every operand are known at the time the program is compiled, the proper overloaded meaning of an operator can be determined when the program is compiled.
5. The form of the internal representation (the way an entity is represented in the computer's memory) is determined for all three `int` variables at the time the language is designed.

Note that this list is not exhaustive but rather representative of the type of analysis you can perform on code using the lens of binding time.

Binding Example: Python

Consider a Python statement like this:

```
vrp = arb + 5
```

1. The symbol `+` (entity) must be bound to an operation (attribute). In a language like Python, that binding can only be done at runtime. In order to determine whether the operation is a mathematical addition, a string concatenation or some other behavior, the interpreter needs to know the type of `arb` which is only possible at runtime.
2. The numerical literal 5 (entity) must be bound to some in-memory representation (attribute). For Python, it appears that the interpreter chooses the format for representing numbers in memory (https://docs.python.org/3/library/sys.html#sys.int_info <https://docs.python.org/3/library/sys.html#sys.int> [_info](https://docs.python.org/3/library/sys.html#sys.int), https://docs.python.org/3/library/sys.html#sys.float_info <https://docs.python.org/3/library/sys.html#sys.float> [_info](https://docs.python.org/3/library/sys.html#sys.float)) which means that this binding is done at the time of language implementation.
3. The value (attribute) of the variables `vrp` and `arb` (entities) are bound at runtime. Remember that the value of a variable is just another binding.

This is not an exhaustive list of the bindings that are active for this statement. In particular, the variables `vrp` and `arb` must be bound to some address, lifetime and scope. Discussing those bindings requires more information about the statement's place in the source code.

Variables' Type Bindings

Broadly speaking, a variables type is bound either *statically* or *dynamically*. In a language where types are bound statically, the variables are bound to types when the program is compiled (or loaded). In a dynamically typed language, the variables are bound to types when the value is assigned to a variable. Examples of the former include C++, Rust, Go, Fortran, Java, C. Examples of the latter include Python, JavaScript, Ruby, PHP.

Static

In statically typed languages, the variables' types must be known at the time the program is compiled or loaded. In order to learn that information, the compiler (or loader) relies on the programmer to mark the types *explicitly* or the compiler learns the types by *implication*.

C++ variables' types can be explicitly typed. E.g.,

```
1 int main() {  
2     int v{0};  
3     int x{0};  
4     double result{0};  
5 }
```

`v`, `x` and `result` are all explicitly marked with their types (`int`, `int`, and `double`, respectively).

A compiler (or loader) can learn the variables' types through implication in one of two ways. First, it can learn by convention. Variables whose names include a certain prefix or symbol are given a specific type. Languages that implement this type of implicit type binding are uncommon and we won't spend time on them here.

However, there are many new languages that rely on *inference* in order to implicitly learn the type of a variable at the time a program is compiled or loaded. In an implicitly dynamically typed language, the compiler uses information about the types of literals or other variables to determine the type of a variable. You can see this in action in Go:

```
type Internet {  
    url string  
}  
func main() {  
    internet := Internet{url: "http://www.cnn.com"  "(http://www.cnn.com%22)"}  
}
```

In Go, the compiler just looks at the type of the expression generating the value to be assigned to a variable initially and infers that will be the type of the variable throughout the program. Pretty cool!

What's even cooler is that new(er) versions of C++ has this power, too:

```
1 int main() { auto name{"Teresa"}; }
```

Dynamic

And what about dynamic typing. How does that work? It's best to learn this by looking at an example from an interactive Python session:



```
Python 3.10.4 (main, Apr 8 2022, 17:35:13) [GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from bintrees import BinaryTree
>>> tree = BinaryTree()
>>> type(tree)
<class 'bintrees.bintree.BinaryTree'>
>>> tree = "Elm"
>>> type(tree)
<class 'str'>
```

From this interactive session output you can see how the type of the `tree` variable changes every time that it is given a new value (based on the type of the expression that generates that new value). Be careful to remember that just because a language is dynamically typed does not mean that its variables do not have types. They do, in fact! It's just that those types change throughout the program's execution.

Variables' Storage Bindings

The storage binding is related to the variable's *lifetime* (the time during which a variable is bound to memory). There are four common lifetimes:

1. static: Variable is bound to storage before execution and remains bound to the same storage throughout program execution.
 1. Variables with static storage binding cannot share memory with other variables (they need their storage throughout execution).
 2. Variables with static storage binding can be accessed directly (in other words, their access does not require redirection through a pointer) because the address of their storage is constant throughout execution. Direct addressing means that accesses are faster.
 3. Storage for variables with static binding does not need to be repeatedly allocated and deallocated throughout execution -- this will make program execution faster.
 4. In C++, variables with static storage binding are declared using the `static` keyword inside functions and classes.
 5. Variables with static storage binding are sometimes referred to as *history sensitive* because they retain their value throughout execution.
2. stack dynamic: Variable is bound to storage when its declaration statements are *elaborated* (the time when a declaration statement is executed).
 1. Variables with stack dynamic storage bindings make recursion possible because their storage is allocated anew every time that their declaration is elaborated. To fully understand this point it is necessary to understand the way that function invocation is

- handled using a runtime stack. We will cover this topic next week. Stay tuned!
2. Variables with stack dynamic storage bindings cannot be directly accessed. Accesses must be made through an intermediary which makes them slower. Again, this will make more sense when we discuss the typical mechanism for function invocation.
 3. The storage for variables with stack dynamic storage bindings are constantly allocated and deallocated which adds to runtime overhead.
 4. Variables with stack dynamic storage bindings are *not* history sensitive.
3. Explicit heap dynamic: Variable is bound to storage by explicit instruction from the programmer. E.g., `new`/`malloc` in C/C++.
1. The binding to storage is done at runtime when these explicit instructions are executed.
 2. The storage sizes can be customized for the use.
 3. The storage is hard to manage and requires careful attention from the programmer.
 4. The storage for variables with explicit heap dynamic storage bindings are constantly allocated and deallocated which adds to runtime overhead.
4. Implicit heap dynamic: Variable is bound to storage when it is assigned a value at runtime.
1. All storage bindings for variables in Python are handled in this way. 
(<https://docs.python.org/3/c-api/memory.html>) <https://docs.python.org/3/c-api/memory.html>  (<https://docs.python.org/3/c-api/memory.html>)
 2. When a variable with implicit heap dynamic storage bindings is assigned a value, storage for that variable is dynamically allocated.
 3. Allocation and deallocation of storage for variables with implicit heap dynamic storage bindings is handled automatically by the language compiler/interpreter. (More on this when we discuss memory management techniques in Module 3).

A Brain Teaser:

Here's something that I hope will make you think ... What does the following program print?

```
1 #include <iostream>
2
3 bool *function1() {
4     bool return_value = true;
5     return &return_value;
6 }
7
8 int function2() {
9     int return_value = 0;
10    return return_value;
11 }
12
13 int main() {
14     bool *function1_result = function1();
15     int function2_result = function2();
16
17     if (*function1_result) {
18         std::cout << "True!\n";
19     } else {
20         std::cout << "False!\n";
21     }
```



```
22  return 0;  
23 }
```

Read the program carefully and put it in a compiler if you like! We'll return to it on Thursday!

The Daily PL - 6/01/2023

Scope

Scope is the range of statements in which a certain name can be used to access (either through reference or assignment) a variable. Using the vocabulary of bindings, *scope* can also be defined as the collection of statements which can access a name binding. In other words, scope plays a role in determining the binding of a name to a variable.

It is easy to get fooled into thinking that a variable's name attribute is somehow intrinsic to the variable. However, a variable's name is just another attribute bound to a variable (like address, storage, value, etc) that may change throughout program execution. So, then, how is the binding between name and variable done? When is it done?

Before we get started answering those questions, it helps to think about two different types of scopes, *local* and *non-local*.

- **local**: A variable is locally scoped to a unit or block of a program if it is declared there. Though not entirely synonymous, because the phrase "unit or block of a program" is so cumbersome we will play a little fast and loose with the definition of scope and use the terms interchangeably. To be clear, however, when we are using the phrase scope in the context of a variable being local, we are talking about the parts of a program (in whatever language you are working in) where name-to-variable bindings must be unique. In Python, such a region is defined by function and class definitions. In Python, a variable that is the subject of an assignment is local to the block of code defining that function. For instance, in

```
def add(a, b):  
    total = a + b  
    return total
```

`total` is a local variable. In C++, a variable is local to the part of the source code beginning at the variable's declaration and continuing through the the `}` that matches the most recent `{`. Yes, I know that's way more complicated than it needs to be, but it's accurate. It is easier to simplify slightly and say that a variable is local to the part of the source code between the most closely nested `{` and its matching `}`. For instance, in

```
1 #include <iostream>
2
3 int main() {
4     int a{0};
5
6     if (a == 0) {
7         int b{0};
8     }
9 }
```

b is local to the source code between the **{** in line 6 and the **}** on line 8. **a** is local to the scope created at the **{** in line 3 and ended at the **}** on line 9.

- non-local: A variable is non-local if it can be accessed (again, through reference or assignment) but is not local.

Now, let's return to the issue of determining how the language determines the variable to which a name refers. In other words, let's talk about the ways a language can determine the binding between a name and a variable.

First, by definition, if the programmer references the name of a local variable, the binding can be determined relatively easily. However, it gets interesting if the programmer refers to the name of a non-local variable.

Here is *the* most common algorithm for performing the resolution of the binding between a variable and a variable. The process described by this algorithm is known as *scope resolution*. The algorithm is described with respect to a statement, *s*. Think of *s* as the statement that uses the name to be resolved (which we will refer to as *x*):

1. If *x* names a local variable, *x* is bound to that variable.
2. If *x* does not name a local variable, consider the parent scope of *s*, *p*.
3. If the parent scope *p* contains a local variable named *x*, *x* is bound to that variable.
4. If the name *x* is still not bound to a variable, consider the parent scope of *p*. Continue searching parent scopes until there are no more ancestor scopes.

The algorithm is relatively straightforward, right? All that formalism reflects our intuition that we resolve name to variable bindings by looking for the declaration of the name beginning as close to the point of its use as possible and then continuing to work outward.

Well, yes, but we haven't yet discussed the most important part(s) of that algorithm, have we? What, exactly, is *parent scope* and what are *ancestor scopes*?

Static Scoping

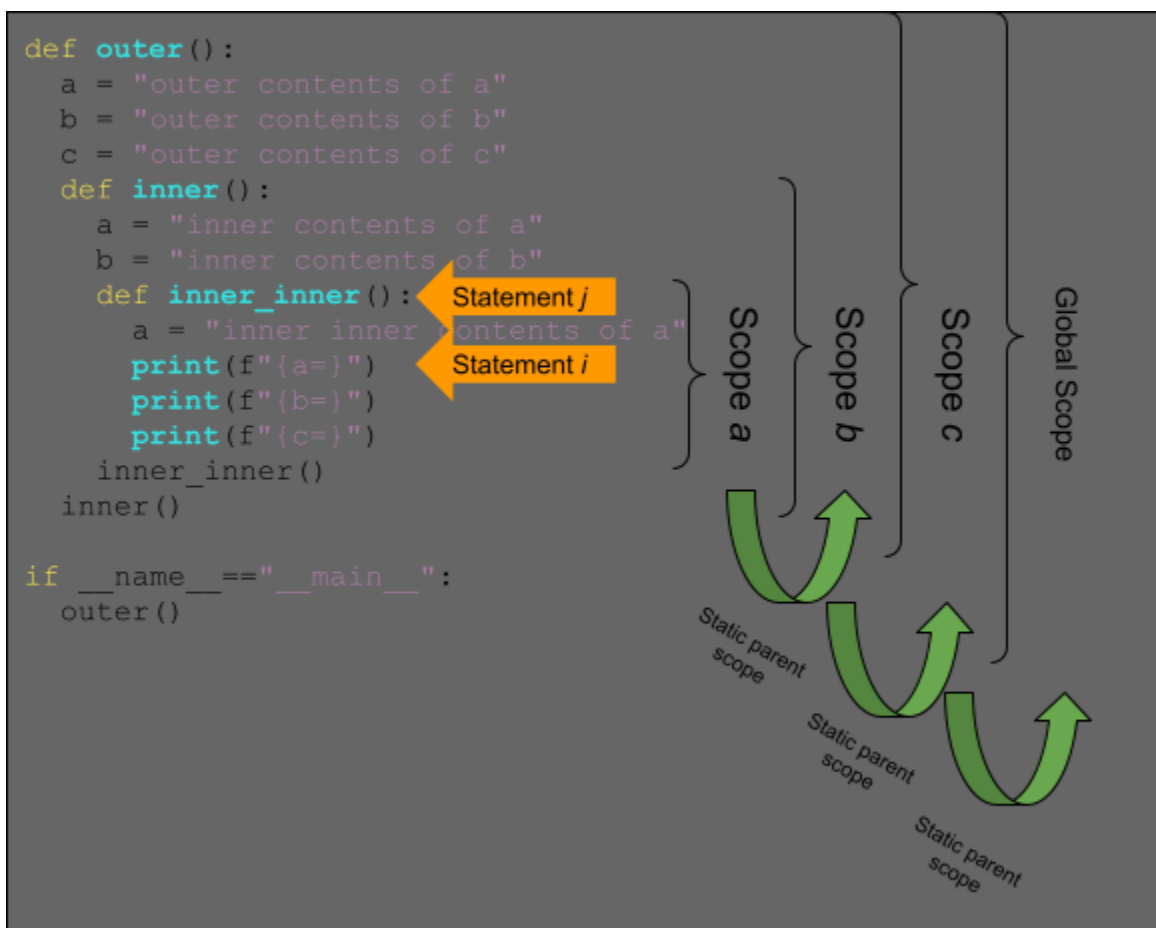
Static scoping, also known as *lexical scoping* is a scope resolution policy where the binding of a name to a variable can be determined using only the program's source code. The parent scope

(using our slightly loose definition of scope here [see above]) of a statement s is the unit/block of a program that declared the unit/block of the program that contains s .

Statically scoped languages are the kind of languages that we are used to! The way that they perform scope resolution is so natural it almost seems uninteresting to discuss the algorithm!

However, here is an example to consider just to make sure that we are on the same page.

```
def outer():
    a = "outer contents of a"
    b = "outer contents of b"
    c = "outer contents of c"
    def inner():
        a = "inner contents of a"
        b = "inner contents of b"
        def inner_inner():
            a = "inner inner contents of a"
            print(f"{a}")
            print(f"{b}")
            print(f"{c}")
        inner_inner()
    inner()
if __name__=="main":
    outer()
```



When *Statement i* is encountered statically, the algorithm for resolving the binding between name a and a variable proceeds as follows:

1. ***a*** is a local variable. The name is bound to the local variable.

Pretty easy, right?

Now, what happens when the statement after *Statement i* is encountered statically and the binding between ***b*** and a variable needs to be resolved?

1. ***b*** is not a local variable.
2. Resolution proceeds to search the static parent of *Scope a* for a local variable named ***b***.
3. *Scope b* does contain a local variable named ***b***. The name is bound to that local variable.

Just what we expected!

And, finally, what happens when the binding between the name ***c*** and a variable needs to be resolved?

1. ***c*** is not a local variable.
2. Resolution proceeds to search the static parent of *Scope a* for a local variable named ***c***.
3. *Scope b* does not contain a local variable named ***c***.
4. Resolution proceeds to search the static parent of *Scope b* for a local variable named ***c***.
5. *Scope c* does contain a local variable named ***c***. The name is bound to that local variable.

Lots of work, but largely as expected! Given that, the program prints what we would expect:

```
a='inner inner contents of a'
b='inner contents of b'
c='outer contents of c'
```

Consider this ...

Python and C++ have different ways of creating scopes. In Python and C++ a new scope is created at the beginning of a function definition (and that scope contains the function's parameters automatically). However, Python and C++ differ in the way that scopes are declared (or not!) for variables used in loops. Consider the following Python and C++ code (also available at https://github.com/hawkinsw/cs3003/blob/main/scope_lifetime/loop_scope.cpp and https://github.com/hawkinsw/cs3003/blob/main/scope_lifetime/loop_scope.py):


```
def f():
    for i in range(1, 10):
        print(f"i (in loop body): {i}")
    print(f"i (outside loop body): {i}")
```

```
void f() {
    for (int i = 0; i<10; i++) {
        std::cout << "i: " << i << "\n";
    }
}
```

```

}
// The following statement will cause a compilation error
// because i is local to the code in the body of the for
// loop.
// std::cout << "i: " << i << "\n";
}

```

In the C++ code, the `for` loop introduces a new scope and `i` is in that scope. In the Python code, the `for` loop does not introduce a new scope and `i` is in the scope of `f`. Try to run the following Python code (also available at https://github.com/hawkinsw/cs3003/blob/main/scope_lifetime/loop_scope_error.py  https://github.com/hawkinsw/cs3003/blob/main/scope_lifetime/loop_scope_error.py) to see why this distinction is important:

```

def f():
    print(f"i (outside loop body): {i}")
    for i in range(1, 10):
        print(f"i (in loop body): {i}")

```

Dynamic Scoping

Static scoping is what we have grown up with, so to speak, as programmers. We are so used to it that it almost seems like the *only* way to do scope resolution! And yet, there is another way!

Dynamic scoping is the type of scope that can be determined only during program execution. In a dynamically scoped programming language, determining the name/value binding is done iteratively by searching through a block's nesting *dynamic parents*. The *dynamic parent* of a block is the block from which the current block was *executed*. Very few programming languages use dynamic scoping (BASH, PERL [optionally]) because it makes checking the types of variables difficult for the programmer (and impossible for the compiler/interpreter) and because it increases the "distance" between name/variable binding and use during program execution. However, dynamic binding makes it possible for functions to require fewer parameters because dynamically scoped non-local variables can be used in their place. We will learn more about this latter advantage when we discuss closures during our exploration of functional programming languages.

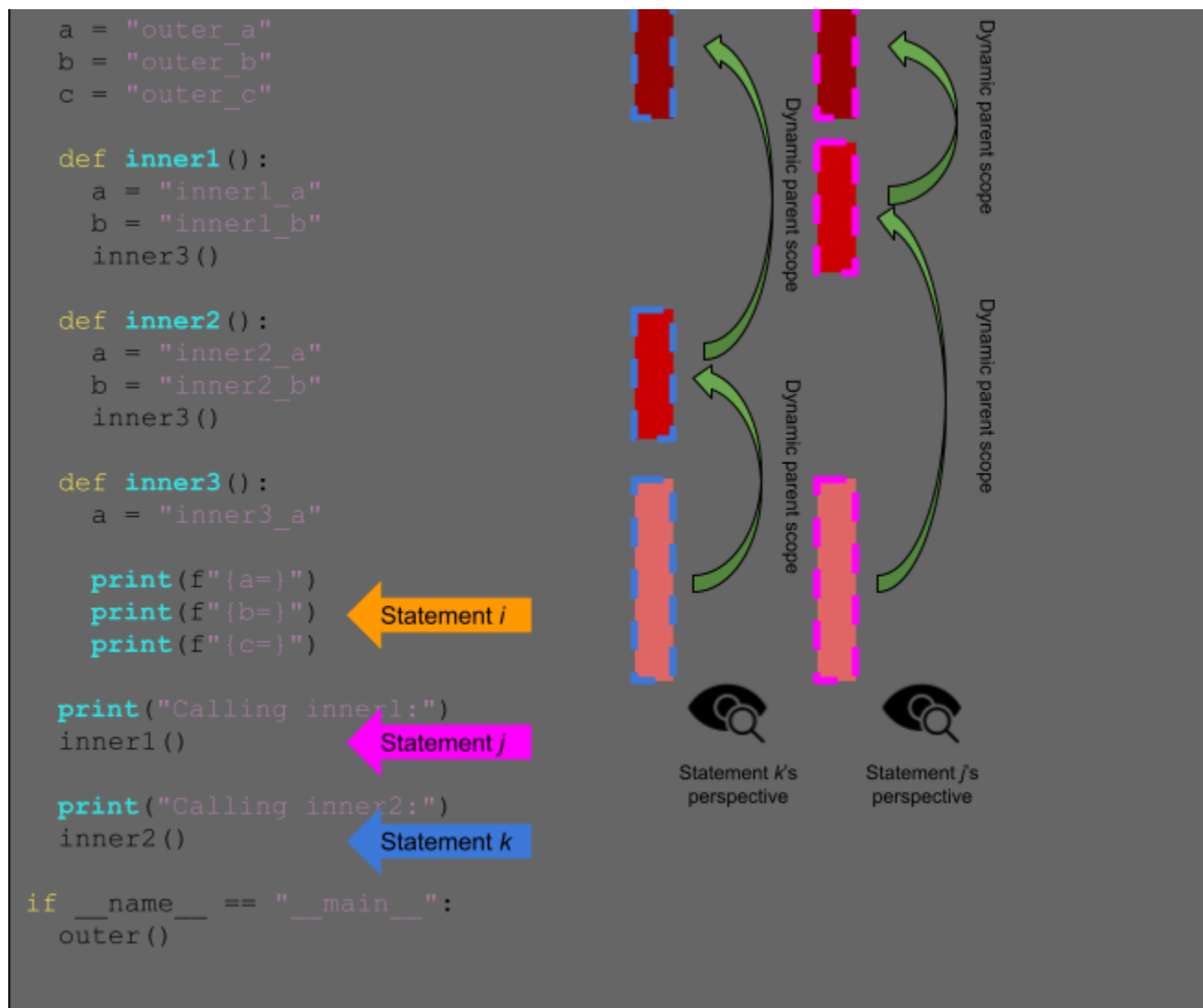
The concept of *dynamic parents* has a close connection with *activation records* (a.k.a. *stack frames*) and the *runtime stack* which we will study in a few lessons. In the meantime, let's assume that Python is a dynamically scoped language and decipher the output of the same program as above:

```

Calling inner1:
a='inner3_a'
b='inner1_b'
c='outer_c'
Calling inner2:
a='inner3_a'
b='inner2_b'
c='outer_c'

```

```
def outer():
```




Eye by Vectorstall from NounProject.com

How does this work? What happens when *Statement i* is encountered at runtime and the binding between `b` and a variable needs to be resolved?



1. `b` is not a local variable.
2. Resolution proceeds to search the dynamic parent of *Statement i* for a local variable named `b`. Depending on whether `inner3` was called from `inner2` or `inner1`, the next scope to be searched will be different.
3. If `inner3` was called from `inner1` (*Statement j*), then the scope of that function is searched for a local variable named `b`. Success! `b` is bound to a local variable whose contents are `"inner1_b"`.
4. If `inner3` was called from `inner2` (*Statement k*), then the scope of that function is searched for a local variable named `b`. Success! `b` is bound to a local variable whose contents are `"inner2_b"`.

Referencing Environment

The referencing environment of a statement contains all the name/variable bindings visible at that statement. NOTE: In Python, you can use the built-in function `locals()` to view the current referencing environment.

You can read all about it in the official documentation of the Python execution model and see how the language relies on the concept of referencing environments: <https://docs.python.org/3/reference/executionmodel.html#naming-and-binding>  [_ \(https://docs.python.org/3/reference/executionmodel.html#naming-and-binding\)](https://docs.python.org/3/reference/executionmodel.html#naming-and-binding). Pretty cool!!

Algebraic Data Types

Computer scientists love to name things. Too bad that they aren't very good at it. As the [adage](https://martinfowler.com/bliki/TwoHardThings.html)  [_ \(https://martinfowler.com/bliki/TwoHardThings.html\)](https://martinfowler.com/bliki/TwoHardThings.html) goes  [_ \(https://twitter.com/timbray/status/817025379109990402?cn=cmVwbHk%3D\)](https://twitter.com/timbray/status/817025379109990402?cn=cmVwbHk%3D), there's only two things that are really hard about computer science:

1. Cache invalidation;
2. Naming things;
3. Off-by-1 errors.

Get the joke? Yeah, it's bad. Algebraic Data Types (ADTs, but do *not* confuse these ADTs with the ADTs of abstract data types fame) are one of the concepts in computer science that, I think, have an ungainly name that makes the concept harder to grasp than it needs to be. There are two kinds of ADTs: the *product type* and the *sum type*.

Product Types

Imagine a scenario where you want to keep track of cars! If you only had primitive data types (int, char, bool, etc), you would have to maintain a lot of variables to store the different attributes of the car. A String type variable to store the name, another String type to store color, an int type to store year, etc!

Now imagine keeping track of 2 cars, even more variables, imagine a 100 cars! Yikes!!

It would be really convenient to have some sort of custom datatype to store the information related to a car, and the product type will let us do just that!

Product types are much, much closer to a type of variable that is near and dear to our hearts: records. (Remember that records are just the fancy name for `struct`s in C/C++, Go, etc or data classes in Python or object literals (kinda) in JavaScript ...) In our pseudo language, a product type might be declared like this:

```
productType Car {
```



```
make: string
model: string
year: int
doors: int
}
```

And we could construct such a product type like this:

```
Car my_car = Car{ make: "Nissan", model: "Altima" ... }
```

To access the elements contained in an instance of a product-typed variable, we usually use the `.`:

```
print(my_car.make + " launched a new car " + my_car.model + " which I bought in " + my_car.year)
```

Product types can also have other product types as their attributes.

For example,

```
productType Person {
  name: string
  age: int
  car: Car
}
```

To access the elements contained in an instance of a nested product-typed variable, we do the following:

```
print(person.name + " drives a " + person.car.model + " which was bought in " + person.car.year)
```

Product types are very useful and are pretty easy to comprehend!

Sum Types

Sum types look like a union. To be more precise, sum types look like discriminated unions. Like a union, the value of a sum-typed variable is one of ***n*** different types (the valid possible types are listed at the time the sum type is declared -- it's not a free for all!). At runtime, the value of a sum-typed variable is either type ***a*** , type ***b*** , type ***c*** , etc. It is never more than one type at the same time and the type that it holds can change as the program executes. In C++, sum types are known as **variants** [↗\(https://en.cppreference.com/w/cpp/utility/variant\)](https://en.cppreference.com/w/cpp/utility/variant). In Rust, sum types are known as **enums** [↗\(https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html\)](https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html).

Like discriminated unions, we programmers can always ask about the type of value that is *active* in a sum-typed variable. We can write code that looks like this:

```
Vehicle = { Car | Plane }
function handle_vehicle(Vehicle v) {
  if (IsCar(v)) {
```

```
// make it run.
} else if (IsPlane(v)) {
    // make it fly.
} else {
    // There's no way for this case to ever happen!
}
}
```

What differentiates sum types from discriminated unions (and one of the reasons that computer scientists decided to give them a unique name), is *usually* the way that they interact with other parts of a programming language. Values for sum-typed variables are usually instantiated with a *constructor*. Yes, you know the term constructor from object-oriented languages like Java and C++ and the concept here is very similar. Returning to the `ActionSumType` example from above, we would instantiate a variable with the `ActionSumType` type by using a constructor like this:

```
Vehicle v = Car()
```

Later, I could reassign it to a Plane:

```
v = Plane()
```

The second unique power of sum-typed variables is their ability to work with *pattern matching*. Pattern matching is a really amazing language construct that came from functional programming languages and is now making its way in to imperative programming languages (e.g., Rust, Python, C++). Pattern matching against a sum type is akin to a `switch` statement. In a switch statement, we programmers define a series of actions to execute depending on the *value* of a variable. However, in pattern matching, we programmers define a series of actions depending on the *type* of a variable. Now *that* is neat! Let's replace the `if/else-if` in the code from above using pattern matching:

```
Vehicle = { Car | Truck }
function handle_vehicle(Vehicle v) {
    match v: {
        case Car: {
            // make it run.
        }
        case Plane: {
            // make it fly.
        }
    }
}
```

So far in our example, the `Car` and the `Plane` are missing a crucial piece of information: the action to *run* or the action to *fly*. So, let's add a piece of data that a sum-typed variable of type `Vehicle` of value `Car` or of value `Plane` carries to indicate the action to "run" and a flag to indicate whether to "fly" on auto-pilot or manual:

```
Vehicle = { Car(speed: int) | Plane(auto_pilot: bool)}
```

(Note: Very importantly, the data carried by two different types of variables that belong to the same sum type do *not* have to have the same name. They don't even have to have the same type. They don't even have to be the same size!)

In our hypothetical language, our constructors for `Car` and `Plane` will need syntax that allows us programmers the ability to set the value of `action`. Let's say that it looks something like this:

```
Vehicle c = Car(speed = 50)
Vehicle p = Plane(auto_pilot = true)
```

Awesome. Now an instance of our `Vehicle` sum-typed variable carries around with it an `action`. But, how can we access that action?

Wait for it.

Pattern matching!

```
Vehicle = { Car | Truck }
function handle_vehicle(Vehicle v) {
  match v: {
    case Car(speed = x): {
      print("I am going at " + x + " mph!!")
    }
    case Plane(auto_pilot = x): {
      if (x) {
        print("I am flying auto_pilot!")
      } else {
        print("I am not flying auto!")
      }
    }
  }
}
```

The pattern matching statement reaches into the variable being matched and can pull out the value of the things that it contains! That's really cool!

The problem is that I am terribly averse to `if` statements and I still see one in the code! What can I do?

Are you sick of this yet?

Pattern Matching!

I can add a condition in the pattern to make the requirements for matching even more explicit:

```
Vehicle = { Car | Truck }
function handle_vehicle(Vehicle v) {
  match v: {
    case Car(speed = x): {
      print("I am going at " + x + " mph!!")
    }
    case Plane(auto_pilot = true): {
      print("I am flying auto_pilot!")
    }
  }
}
```

```
}
case Plane(auto_pilot = false): {
    print("I am not flying auto_pilot!")
}
}
```

Now, if you don't think that's cool, well, ...

We have been coding our examples in a pseudo language and each language that supports pattern matching has a slightly different syntax: **C++** [↗\(https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r2.pdf\)](https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1371r2.pdf), **Rust** [↗\(https://doc.rust-lang.org/book/ch06-02-match.html\)](https://doc.rust-lang.org/book/ch06-02-match.html), **Haskell** [↗\(https://www.haskell.org/tutorial/patterns.html\)](https://www.haskell.org/tutorial/patterns.html), **Python** [↗\(https://peps.python.org/pep-0634/\)](https://peps.python.org/pep-0634/).

For a tutorial/introduction to pattern matching in Python, check out the special edition of the Daily PL on the subject!

Why the Odd (Even?) Names?

The names are a product (pun intended) of the number of unique values that a variable whose type is sum or product can hold. I think that the concept is best defined using examples. Let's assume that we have a sum type defined like this:

```
SomeSumType = bool | uint8
```

where `bool` is a Boolean type (can hold either true or false) and `uint8` is a integral type that can hold all the values greater than or equal to zero that can fit in 8 bits. A variable with the type `SomeSumType` can be either a `bool` or an `uint8` at any given time during the execution of a program. If that variable is a `bool` then its value could be either true or false (there's two (2) values!). If that variable is a `uint8`, then its value could be either 0, or 1, or 2, or ... $2^8 - 1$ (there's 2^8 more values). So, in total, the number of possible values of that variable at any given time during program execution is $2 + 2^8$. What's another term for *plus*? That's right, *sum*. Boom.

Active Type Value		
1	<code>bool</code>	True
2	<code>bool</code>	False
3	<code>uint8</code>	0
4	<code>uint8</code>	1
5	<code>uint8</code>	2
		...
$2 + 2^8$	<code>uint8</code>	$2^8 - 1$

I bet that you can see where this is going.

Let's assume that we have a product type defined like this:

```
SomeProductType {
  bool b
  uint8 u
}
```

A variable with the type `SomeProductType` contains a `bool` and an `uint8` at any given time during the execution of a program. That variable could have a `b` that is either True or False. For each of those values of `b`, the variable could have a `u` that is any of the possible values between `0` and $2^8 - 1$.

<code>b</code>	Value	<code>u</code>	Value
1	True		0
2	True		1
3	True		2
			...
...	True		$2^8 - 1$
	False		0
	False		1
	False		2
			...
...	False		$2^8 - 1$

So, in total, the number of possible values of a variable whose type is `SomeProductType` is $2 * 2^8$. What's another term for *multiply*? That's right, *product*. Boom, Boom.

The Daily PL - 6/06/2023

Expressions

An *expression* is the means of specifying computations in a programming language. Informally, it is anything that yields a value. For example,

- `5` is an expression (value 5)
- `5 + 2` is an expression (value 7)
- Assuming `fun` is a function that returns a value, `fun()` is an expression (value is the return value)
- Assuming `f` is a variable, `f` is an expression (the value is the value of the variable)

Certain languages allow more *exotic* statements to be expressions. For example, in C/C++, the `=` operator yields a value (there are technical differences between how it works in C and C++, but you can generally think of it as evaluating to the value of the expression of the right operand). It is this choice by the language designers that allows a C/C++ programmer to write

```
1 #include <iostream>
2
3 int main() {
4     int a{0}, b{0}, c{0}, d{0};
5     a = b = c = d = 5;
6 }
```

to set all four variables to 5 without having to make a special rule for such a construction in the language's syntax (the way that, e.g., **Python does** https://docs.python.org/3/reference/simple_stmts.html#assignment-statements).

When we discuss functional programming languages, we will see how many more things are expressions that programmers typically think are simply statements.

Order of Evaluation

Programmers learn the associativity and precedence of operations in their languages. That knowledge enables them to mentally calculate the value of statements like `5 + 4 * 3 / 2`.

What programmers often forget to learn about their language, is the order of evaluation of operands. Take several of those constants from the previous expression and replace them with variables and function calls:

```
1 #include <iostream>
```

```
2
3 int c{5};
4
5 int a() { return 1; }
6
7 int b() { return 2; }
8
9 int main() {
10     int value{0};
11     value = 5 + a() * c / b();
12 }
```

The questions about line 9 abound:

- Is `a()` executed before variable `c` is evaluated?
- Is `b()` executed before `c` is evaluated?
- Is `b()` executed at all?

In a language with *side effects*, the answer to these questions matter. A side effect occurs when the process of evaluating an expression (e.g., retrieving the value of a variable, executing a function to determine its return value, etc) has an effect on the state of the program in addition to generating a value. That's great, but only makes us ask the next question: Just *what* is the program's *state*? We've alluded to the state of a program before, but now it's time for a reasonably precise definition: The collection of values in storage at a particular point in program execution.

A language without side effects has *referential transparency*. In a language with referential transparency, any two expressions that evaluate to the same value can be replaced with one another without changing the behavior of a program's execution. A different, but compatible description of referential transparency, defines it as a property of functions: A function has referential transparency if its value (its output) depends only on the value of its parameter(s). In other words, if given the same inputs, a referentially transparent function always gives the same output. We will discuss side effects and referential transparency more when we return to functional programming languages.

C++ is a language without referential transparency. In a language like C++ why are the answers to the questions posed above so important? Consider that `a` could have a side effect that changes `c`. If the value of `c` is retrieved *before* the execution of `a` then the expression will evaluate to a certain value. But if the value of `c` is retrieved *after* execution of `a` then the expression will evaluate to a different value. What a mess!

Certain languages define the order of evaluation of operands (Python, Java) and others do not (C/C++). There are reasons why defining the order is a good thing:

- The programmer can depend on that order and benefit from the consistency
- The program's readability is improved.

- The program's reliability is improved.

But there is at least one really good reason for not defining that order: optimizations. If the compiler/interpreter can move around the order of evaluation of those operands, it may be able to find a way to generate faster code!

Short-circuit Evaluation

Languages with *short-circuit evaluation* take these potential optimizations one step further. In a language with short-circuit evaluation, the code that the compiler generates (or the interpreter executes) will do only as much work as is strictly necessary to determine the value of a Boolean expression. For instance, in `a() && b()`, if `a()` is `false`, then the entire statement will always be false, no matter the value of `b()`. In this case, the compiler/interpreter will simply not execute `b()`. On the other hand, in `a() || b()` if `a()` is true, then the entire statement will always be true, no matter the value of `b()`. In this case, the compiler/interpreter will simply not execute `b()`.

A programmer's reliance on this type of behavior in a programming language is very common. For instance, this is a common idiom in C/C++:

```
1 #include <iostream>
2
3 int main() {
4     int *value = nullptr;
5     ...
6
7     if (value && *value == 5) {
8         ...
9     }
10    ...
11 }
```

In this code, the programmer is checking to see whether there is memory allocated to `value` before they attempt to read that memory. This is defensive programming thanks to short-circuit evaluation.

Subprograms

A *subprogram* is a type of abstraction. It is *process abstraction* where the *how* of a process is hidden from the user who concerns themselves only with the *what*. A *subprogram* provides process abstraction by naming a collection of statements that define parameterized computations. Again, the collection of statements determines how the process is implemented. *Subprogram parameters* give the user the ability to control the way that the process executes. There are three types of subprograms:

1. Procedure: A subprogram that does *not* return a value.

2. Function: A subprogram that *does* return a value.
3. Method: A subprogram that operates with an implicit association to an object; a method may or may not return a value.

Pay close attention to the book's explanation and definitions of terms like *parameter*, *parameter profile*, *argument*, *protocol*, *definition*, and *declaration*.

Subprograms are characterized by three facts:

1. A subprogram has only one entry point
2. Only one subprogram is active at any time
3. Program execution returns to the caller upon completion

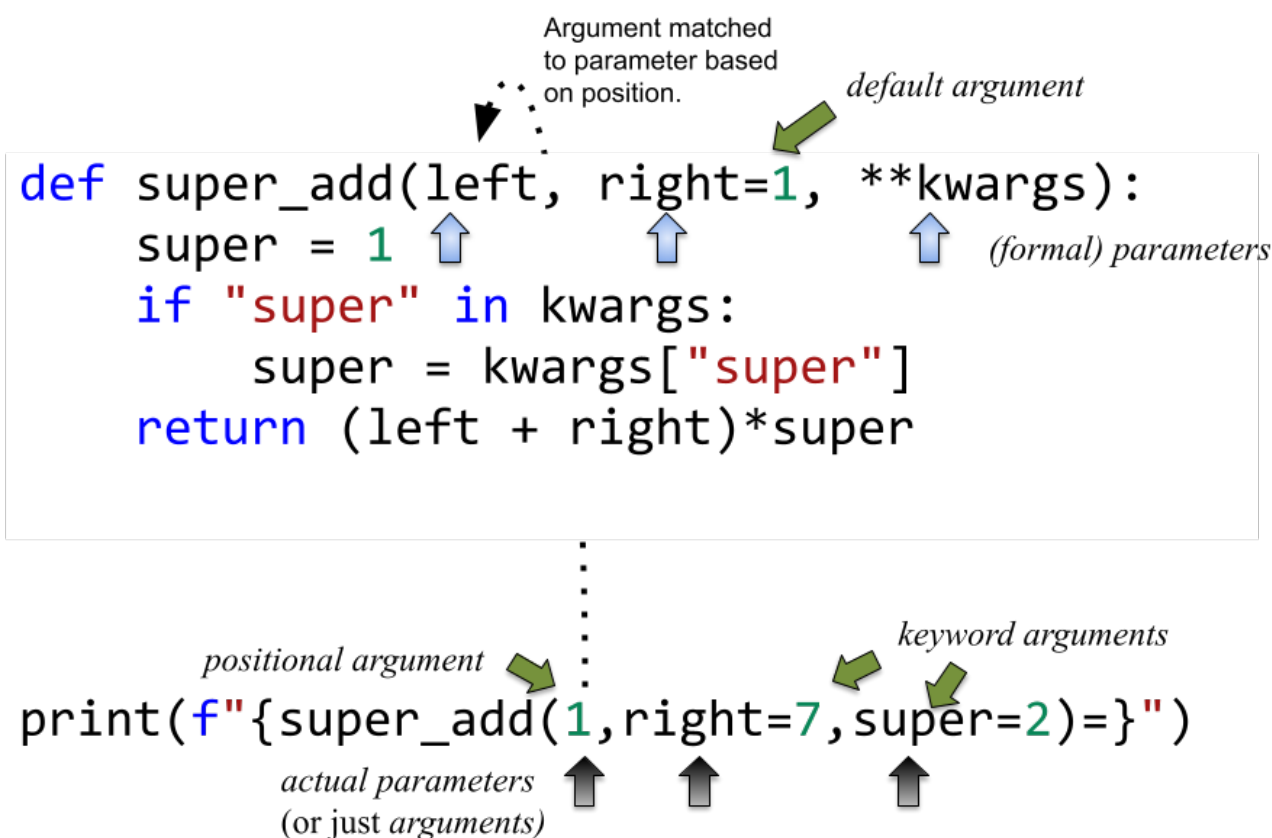
Calling a Subprogram: Matching Parameters with Arguments

There is a difference between (formal) parameters and actual parameters (also know as arguments). (Formal) parameters belong with the function definition (or the declaration, protocol). They give names to variables that the function implementation can use to refer to values that the user specifies at the time that the function is called. Those values (specified by the user) are called the arguments. The number, type and order of arguments must (sometimes) match the number, type and order of the parameters. In most cases, order is used to match the arguments to the parameters. The first parameter is matched to the first argument, and so on. Once the arguments are matched to the parameters, their types are confirmed to match. Matching arguments to parameters positionally implies that there are the same number of arguments and parameters.

So, why the caveat?

Languages offer different mechanisms to make matching parameters with arguments possible. Some languages offer *default arguments*, which allow the caller to leave off an argument by specifying a parameter's value in the absence of a user's choice. Some languages offer *keyword arguments*, which allow the caller to match a particular argument with a particular parameter out of order.

Python is a language that allows all those "features", and more!



The terminology used so far seems to imply that two-way communication between caller and called via arguments/parameters is not possible. There is an implication that communication from the called to the caller is only possible through a *return value*. This is most certainly not the case in every programming languages. Parameters can be:

1. in mode parameters: The traditional type of parameter the caller communicates to the called (one way).
2. out mode parameters: Where the called communicates with the caller (one way).
3. in/out mode parameters: Where the caller and the called can communicate bidirectionally.

Those three definitions specify the semantics of communication between caller and called, but do not describe the mechanics. There are many ways that these modes of communication can be implemented and each have their own benefits and drawbacks:

1. Pass by value: Used to implement in-mode parameters. A local variable (to the function implementation) with the name of the formal parameter is given *the value* of the corresponding actual parameter. That value is usually obtained by copying -- hence the name.
2. Pass by result: Used to implement out-mode parameters. A local variable with the name of the

formal parameter is available to the function implementation. Upon termination of the function definition, the value of that local variable is copied to the value of the corresponding actual parameter.


3. **Pass by reference:** Used to implement in-out-mode parameters. A local variable with the name of the formal parameter contains an *access path* (usually a pointer) to the argument. In the function implementation, all accesses/updates to the variable are done through the access path and are simultaneously visible to the called and the caller.

There are myriad ways that the choice of implementation methods for parameter passing can cause unexpected behavior. The most common issue that arises has to do with *aliasing*. When two (or more) names refer to the same variable, those names are said to be aliases. An access/update using the *different* names changes the value of the *same* variable. Obviously this can be surprising to the programmer who believes that an assignment statement giving a new value to the variable bound to two *different* names is changing the value of two different variables. Aliasing is such a difficult problem to solve that some languages (C++) put the onus on the programmer to ensure that it never happens and make no guarantees about a program's behavior when aliasing is present.


Polymorphism

Polymorphism allows subprograms to take different types of parameters on different invocations. There are two types of polymorphism:

1. *ad-hoc polymorphism:* A type of polymorphism where the semantics of the function may change depending on the parameter types.
2. *parametric polymorphism:* A type of polymorphism where subprograms take an implicit/explicit type parameter used to define the types of their subprogram's parameters; no matter the value of the type parameter, in parametric polymorphism the subprogram's semantics are always the same.

Ad-hoc polymorphism is sometimes call function overloading (C++). Subprograms that participate in ad-hoc polymorphism share the same name but must have different protocols. If the subprograms' protocols and names were the same, how would the compiler/interpreter choose which one to invoke? Although a subprogram's protocol includes its return type, not all languages allow ad-hoc polymorphism to depend on the return type (e.g., C++). See the various definitions of add in the C++ code here: <https://github.com/hawkinsw/cs3003/blob/main/subprograms/subprograms.cpp>  <https://github.com/hawkinsw/cs3003/blob/main/subprograms/subprograms.cpp>. Note how they all have different protocols. Further, note that not all the versions of the function add perform an actual addition! That's the point of *ad-hoc polymorphism* -- the programmer can change the meaning of a function.

Functions that are parametrically polymorphic are sometimes called function templates (C++) or

generics (Java, soon to be in Go, Rust). A parametrically polymorphic function is like the blueprint for a house with a variable number of floors. A home buyer may want a home with three stories -- the architect takes their variably floored house blueprint and "stamps out" a version with three floors. Languages variously call the process *instantiating* a parametrically polymorphic function. Others call it **monomorphization**  (<https://en.wikipedia.org/wiki/Monomorphization>).

The type parameter of a parametrically polymorphic function can be either inferred from the types of the arguments to a particular invocation (or call) of the function or specified explicitly.

Here's an example from C++:

```
1 template <typename T> T minimum(T a, T b) {  
2     if (a <= b) {  
3         return a;  
4     } else {  
5         return b;  
6     }  
7 }  
8
```

Note that there is only *one* definition of the function! In C++, the type parameter referred to in the definition of *parametric polymorphism* is explicit -- in this case `T`. When a programmer uses a parametrically polymorphic function by calling it, a value for `T` is deduced from the types of the arguments in the call. Every time the compiler sees a user make a call with a different version of `T`, the compiler will "stamp out" a new copy of `minimum` with different types that replace `T`.



When the programmer writes,

```
auto m = minimum(5, 4);
```

the compiler will generate

```
1 int minimum(int a, int b) {  
2     if (a <= b) {  
3         return a;  
4     } else {  
5         return b;  
6     }  
7 }
```

behind the scenes.

You can verify that this is, indeed, the case by using an awesome tool known as **C++ Insights**  (<https://cppinsights.io/>). The particular example just explored can be seen **here**  (<https://cppinsights.io/s/1b8aa1e2>).

In Go, the newest language to offer such parametric polymorphism, the same example would look like:

```
import (
    "fmt"
    "golang.org/x/exp/constraints"
)

func minimum[T constraints.Ordered](a T, b T) T {
    if a < b {
        return a
    } else {
        return b
    }
}
```


There is a very interesting difference between the Go version of `minimum` and the C++ version. In addition to requiring that the developer specify that the function `minimum` takes a parametric type (named `T` in this case), Go requires an additional bit of information (in this case `constraints.Ordered`). The additional bit of information is (not surprisingly) known as a *constraint*. Constraints on type parameters are like types on variables -- they limit the range of valid values for the type parameter. In this case, the `constraints.Orderable` constraint specified by the programmer for `minimum` tells the user of the parametrically polymorphic `minimum` function that the only valid types that can be accepted by an instance of the `minimum` function are those that can be *ordered*. The constraint specifies the operations that a type must support to be usable in the context. For instance, for `minimum`, it would make no sense to call the function with two arguments whose values cannot be ordered relative to one another (how could you calculate the minimum, then?). C++ has added similar functionality (known as concepts) and Rust has traits.

Coroutines

Just when you thought that you were getting the hang of subprograms, a new kid steps on the block: coroutines. Sebasta defines *coroutines* as a subprogram that cooperates with a caller. The first time that a programmer uses a coroutine, they *call* it at which point program execution is transferred to the statements of the coroutine. The coroutine executes until it *yields* control. The coroutine may yield control back to its caller or to another coroutine. When the coroutine yields control, it does not cease to exist -- it simply goes dormant. When the coroutine is again invoked -- *resumed* -- the coroutine begins executing where it previously yielded. In other words, coroutines have

1. multiple entry points
2. full control over execution until they yield
3. the property that only one is active at a time (although many may be dormant)

Coroutines could be used to write a card game. Each player is a coroutine that knows about the player to their left (that is, a coroutine). The PlayerA coroutine performs their actions (perhaps drawing a card from the deck, etc) and checks to see if they won. If they did not win, then the PlayerA coroutine yields to the PlayerB coroutine who performs the same set of actions. This

process continues until a player no longer has someone to their left. At that point, everything unwinds back to the point where PlayerA was last resumed -- the signal that a round is complete. The process continues by resuming PlayerA to start another round of the game. Because each player is a coroutine, it never ceased to exist and, therefore, retains information about previous draws from the deck. When a player finally wins, the process completes. To see this in code, check out <https://github.com/hawkinsw/cs3003/blob/main/subprograms/cardgame.py>  (<https://github.com/hawkinsw/cs3003/blob/main/subprograms/cardgame.py>).

The Daily PL - 6/08/2023

Functional Self Awareness


We all aspire to self awareness and subprograms are no different. Subprograms need to know certain information about themselves if they are to be able to operate correctly. In particular, they might need to know

1. Where they can store the values the caller specified as arguments for their parameters (and, perhaps, whether the types of those values need to match the types of the parameters);
2. How values are communicated with the caller (i.e., how they get values for the parameters from the caller and, in the case of functions, how they send the values they generate back to the caller);
3. Where they can store the values of their local variables;
4. Where the program resumes execution after they are complete;

Though self aware, subprograms are not metaphysical -- they need someone else to start them. The way that functions are started (and the coordination between them and the caller during the time that they are active) needs to be well specified or else, at best, one side does all the work while the other sits idle or, at worst, both sides work at cross purposes and the whole thing becomes a mess.



Push It Real Good

The data structure that holds all the information about an active function is known as the *stack frame* (or the *activation record*). "Activation record" makes sense -- after all, the data structure contains a record of important information about *active* functions. Why call it a stack frame?


The term comes from the way that all the instances of the data structure for the active functions are ordered in memory. They form a stack! Okay, now I get it! But, why a stack? Think about the characteristics of a subprogram and you will come to the conclusion that the last subprogram that was started is the first one that is exited. In other words, they form a last-in-first-out order -- perfect for a stack! The stack frames exist in memory while the program is executing. And, for **historical reasons**  (https://tcm.computerhistory.org/ComputerTimeline/Chap37_intel_CS2.pdf), the "stack grows down" -- the addresses of the newest stack frames are smaller than the addresses of the oldest stack frames.

Before talking about just how each individual stack frame is organized and what it holds, let's talk about how the stack frame is maintained throughout program execution. One of the questions for a language designer (as we mentioned above) is *who* (the caller or the called) is responsible for

managing the creation and destruction of the frame. The code that performs the creation of the stack frame and performs other bookkeeping and control-flow operations necessary to invoke the called function is known as the *prologue*. The code that performs the destruction of the stack frame and performs the other operations necessary to transfer control back to the caller is known as the *epilogue*.

Again, the process of calling/terminating a function and managing its stack frame must be coordinated between the caller and the called. The caller could do all the work, the callee could do all the work or they could share the responsibility. Each language (and sometimes the platform -- the combination of hardware and operating system) are responsible for defining who performs what role. These standards are sometimes referred to as calling conventions (See [here](https://www.uclibc.org/docs/psABI-x86_64.pdf)  https://www.uclibc.org/docs/psABI-x86_64.pdf) for an example of the description of the calling conventions that apply to Linux and macOS and [here](https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170)  <https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-170>) for a description of the calling conventions that apply to Windows).

Putting a Rabbit Into a Hat

(*Note*: What is described in this section is a general description of the contents of a stack frame. Each platform and language will create a different format for these data structures that is customized for the target hardware and the features of the language. For instance, Go has their own "internal" definitions which you can read [here](https://go.golangsource.com/go/+refs/heads/dev.regabi/src/cmd/compile/internal-abi.md)  <https://go.golangsource.com/go/+refs/heads/dev.regabi/src/cmd/compile/internal-abi.md>.)

To answer the function's soul-searching questions, each stack frame must *at least* give the function a reference to

1. space for its local variables
2. space for its parameters
3. information about where to resume program execution upon its termination

In most cases, the stack does better than just point to the space for local variables and parameters. Instead, the stack frame *is* the space for the local variables and the parameters. Remember during the earlier discussion of variable lifetimes that one of them was named *stack dynamic*? Well, now it makes sense -- the lifetime of local variables and parameters (the time when the variable is bound to storage) is only as long as the time that the function's activation record exists on the stack.

Because the coordination of the creation and destruction of the stack frame is usually a cooperative endeavor between the caller and the called, there is logical reason for a physical separation of the space in the stack frame for storing the arguments and the local variables. The *caller* knows how many (and how big) the parameters are that the user has given along with the

function invocation. Therefore, the caller is in the best position to allocate the space necessary to hold the arguments that correspond to the parameters of the function. The caller's part of the prologue executes first (obviously) and, therefore, uses memory at a higher address. The function itself knows how much storage it needs to perform its computation (i.e., store local variables) and is, therefore, best positioned for performing the portion of the prologue that allocates space for local variables. That part of the prologue is executed once program control has been transferred to the called function (obviously) so the space for those local variables will exist at lower addresses.

Unstated but important for the process of invoking a function is that the overhead of creation and destruction of a stack frame may play a significant role in decreasing a program's efficiency when functions are executed frequently. Is the overhead for calling two different functions always the same? Or, does it depend on some specific characteristics of the function being called? The latter! If a function named, say, **a**, has more parameters (or parameters that are bigger) and more local stack space, then the time it takes to copy values into those parameters and the time it takes to allocate space for local variables will be longer relative to a function named, say, **b** that has fewer parameters (or parameters that are larger) and needs less local stack space. In other words, the time it takes to invoke a function is a function (pun intended) of the number (or size) of parameters and the size of the space required for storing local parameters.

In between the values for the function's parameters and the space for its local variables exists the address of the program location to which to return upon termination of the function (the *return address*). It is between the storage for the arguments because it is the last value that the caller sets before handing off control to the called. Next to the return address (and between the parameters and locals) is the address of the stored *base pointer*.

Assume that the function to be executed is **add** in

```
#include <iostream>

int add(int a, int b) { return a + b; }

int main() {
    auto a_plus_b = add(5, 4);
    std::cout << "add(5, 4) : " << a_plus_b << "\n";
    return 0;
}
```

To access a variable (either to read it or write it), its value must be read (or written) to (or from) memory. The code generated by the compiler for **add** (or the operation of the interpreter when executing **add**) can directly address **a** and **b** because they know their addresses. Right?

Right?

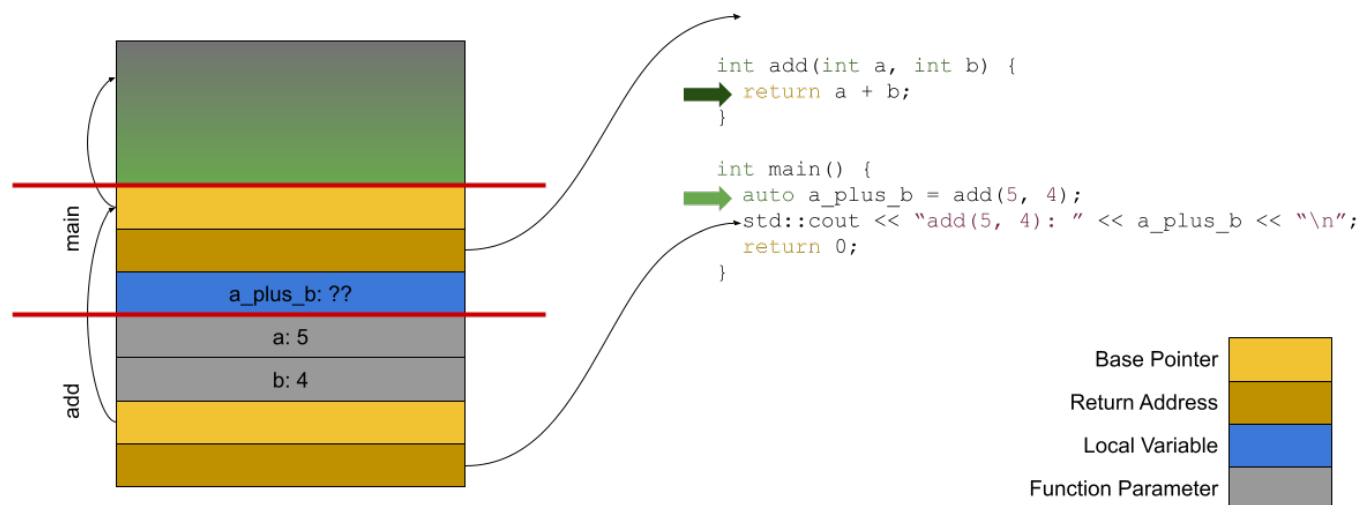
Wrong!

Why? Because it is impossible to know ahead of time just how many functions will be active at the same time as `add` and, depending on that number, the stack frame (and thus the position in memory of `a` and `b`) will vary from execution-to-execution.

Besides trolling StackOverflow, adding print statements and reading Reddit, computer scientists have a goto (pun *definitely* intended) solution for nearly all problems: indirection. To determine the addresses for local variables, the compiler will generate code (or the interpreter will execute instructions) that access `a` and `b` as offsets relative to a known location. That known location is the *base pointer*. At any time during program execution, there is only one base pointer (it is *usually* stored in a register in the hardware but does not have to be).

But, aren't there more than one active functions at a time? Why yes there are! So, what happens to the other base pointers? Great question. As part of the prologue and epilogue, the old value of the base pointer is (re)stored to/from the stack. It is *this* value that is stored next to the return address between the parameters and the local variables.

The figure below shows the layout of the stack for the code above.



Inception

Recursion is a very powerful problem-solving technique that allows the programmer to specify the solution to a challenge in terms of the solution itself. Having support for recursion in a programming language is a nice feature. Consider a program with a function that calculates the factorial (Note: This function named `factorial` computes something *like* the factorial but not *actually* the factorial -- cut me some slack!).

```

1 int factorial(int n) {
2     int result{1};
3     int even_factor{1};
4     if (!(n % 2)) {
5         even_factor = 2;

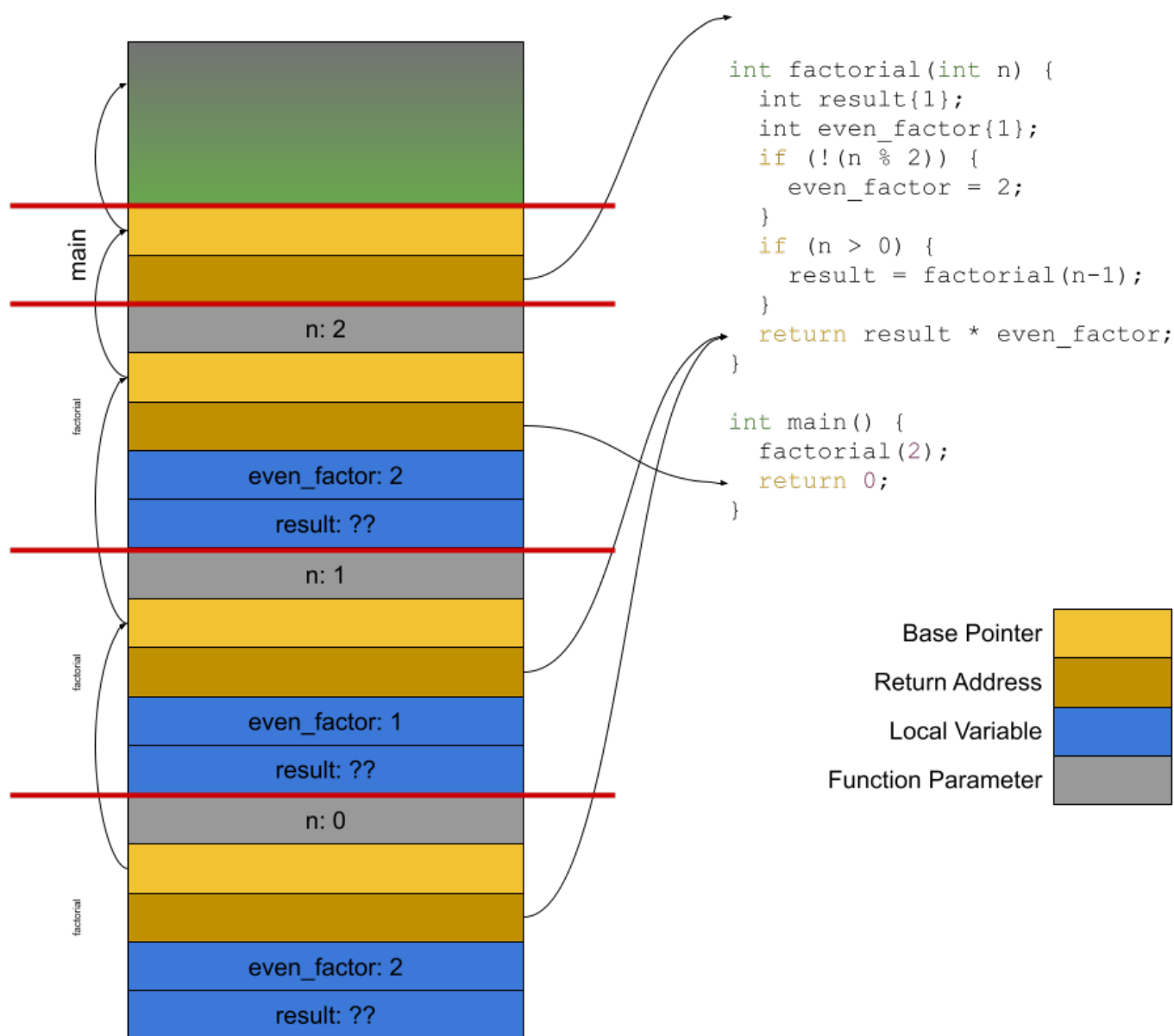
```

```

6  }
7  if (n > 0) {
8      result = factorial(n - 1);
9  }
10 return result * even_factor;
11 }
12
13 int main() {
14     factorial(2);
15     return 0;
16 }

```

The figure below shows the state of the runtime stack at the time when the recursion has "bottomed out" and the results are about to percolate back up.



There is one very important thing to note about this diagram: Every active copy of the `factorial` function has *its own* copies of local variables and parameters and they can each hold different values of those variables. At runtime, the code generated by the compiler (or the behavior of the interpreter) accesses the correct version of the variables by way of indirection through the base pointer. Remember that from above?

Also, note the position of the assignment to and read from `even_factor` (lines 5 and 10) -- they straddle the recursive call to the `factorial` function. The recursive call to `factorial` begins and ends between the time that the local variable `even_factor`'s value is set and the time that it is used.

Let's run the program and see what it computes:

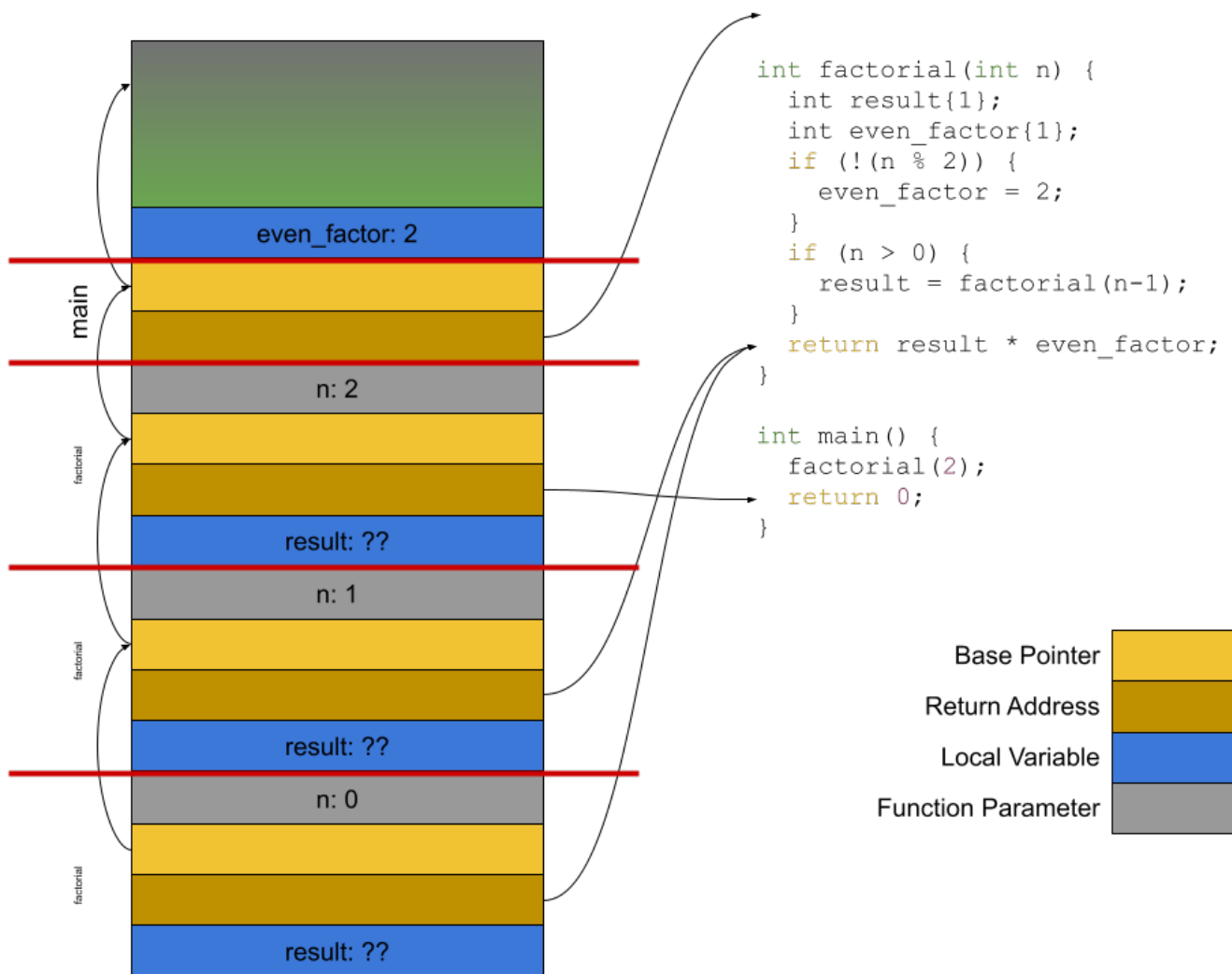
n	Result
2	4
3	4
4	8
5	8

Who's On First

Boy oh boy. Look at all the wasted space there, though. There are ... let me count ... how many copies of `even_factor` and `n` on the stack but there is only one that is really needed at a given time? Not to mention the fact that because there are so many different copies, the code generated by the compiler (or the execution of the interpreter) must use the base pointer to access the right one. So. Slow.

Let's make it easier and faster: There will only be a single copy of each function-local variable (and parameter). That means that the program will save space and will not have to use the base pointer to access values -- the location in memory of each of the local variables and the parameters can be known in advance because they will no longer vary with the depth of the stack. Pretty sweet!

Here's an equivalent snapshot of the stack frame after that optimization:



Where the program is paused, `factorial` is calculating on the basis of the parameter (`n`) with a value of `0`. `0 % 2` is `0`, `!0` is `true` and, therefore, the value of `even_factor` is `2`. Great.

Let's follow the execution to the next step. The version of `factorial` active with the parameter of `n` set to `0` (the one all the way at the bottom) completes and program control returns to the caller and begins at the line just after the recursive call (line 10). To calculate the value that the version of `factorial` with `n` equal to `1` will generate, the value of `even_factor` is multiplied by the result of the version of `factorial` that just completed.

See the problem yet? Slow it down. The version of the `factorial` function that is generating its value now is computing with the parameter `n` as `1`. That means that `even_factor` *should be* `1`. And *it was*! At the time that the version of the `factorial` function computing with `n` equal to `1` invoked itself recursively, `even_factor` was `1`! However, because there is only one slot for a value of `even_factor` for the entire program as a result of our "optimization", its value changed between the time it was written and now when we are about to use it! Geez.

<code>n</code>	Result
----------------	--------

n	Result
2	8
3	16
4	32
5	64

Nice optimization! It changed the behavior. After "Don't build Skynet", a fundamental axiom of computer science is that optimizations should not change the correctness of an algorithm!

A Little of This and a Little of That

So, maybe that wasn't such a good idea after all. Without different copies of stack variables and parameters for each activation of a subprogram, a language cannot support recursion. Maybe that's not such a bad thing. After all, the programmer can always rewrite a recursive algorithm to perform the same calculations iteratively. And, the speedup of not having to perform indirect memory accesses might be worth the programmer's time. That's exactly the tradeoff that early language designers made when they developed languages like Fortran that did not support recursion.