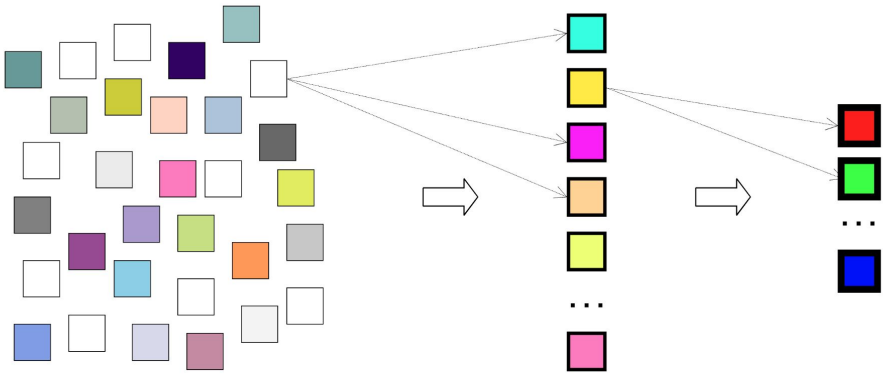
Introduction to OCaml

Programming Paradigm

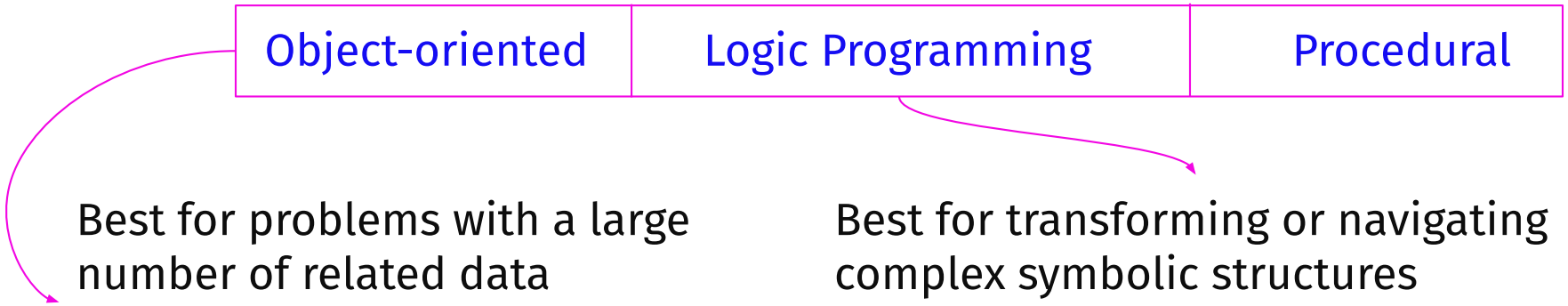
A programming paradigm is an approach to programming a computer based on a mathematical theory or a coherent set of principles.



Ref: Programming Paradigms by Peter Van Roy

Programming Paradigm

Each paradigm supports a set of concepts that makes it the best for a certain kind of problem.



Best for problems with a large number of related data abstractions organized in a hierarchy.

Best for transforming or navigating complex symbolic structures according to logical rules.

Functional Programming

- Functional Programming (FP) is a programming paradigm where computation is expressed as the evaluation of mathematical functions, emphasizing immutability and minimizing side effects and changing state.
- I have two functions $f(x)$ and $g(x)$ where the $f(x)$ uses the results computed by $g(x)$. How do I compose?
  Mathematically:
  $result = f(g(x))$
- Functional programming is about building such pipelines.

Functional Style in C

```c
#include <stdio.h>

int g(int x)
{
   return x * 2;
}

int f(int x)
{
   return x + 5;
}
```

```c
typedef int (*int_func)(int);

int pipeline(int_func outer_function, int_func inner_function, int x)
{
   int inner_result = inner_function(x);
   int final_result = outer_function(inner_result);
   return final_result;
}
```

Functional Style in C

```c
int main()
{
    int input = 10;
    printf("Input: %d\n", input);

    int result_f_g = pipeline(f, g, input);
    printf("Result: %d\n", result_f_g);

    int result_g_f = pipeline(g, f, input);
    printf("Result: %d\n", result_g_f);

    return 0;
}
```

Benefit: Can compose the function sequence at will

Side-Effects

```c
#include <stdio.h>

int count = 0;

void add_to_acc(int value) {
   count += value;
}
```

Modifies external state

Why learn OCaml?

- OCaml is an industrial-strength, statically-typed functional language.
- New programming paradigm.
- Used by industry
  - Jane Street
  - Meta
  - Bloomberg
  - ...
  - Many more

Hello World!

hello_world.ml

```
let () = print_endline "Hello, World!";;
```

```
(* single line comment *)
(* multiple line comment,
commenting out part
of a program *)
```

A built-in function

Bind values

```
> ocaml
OCaml version 5.3.0
> ocamlc -o hello_world hello_world.ml
> ./hello_world
Hello, World!
```

Expressions

- Everything has a value & every value has a type.
- OCaml has type inference. For most parts, it can automatically determine type of an expression.

```
10 * 5;;

let student = "Ashoka";;
```

```
- : int = 50
val student : string = "Ashoka"
```

value declaration

if then else

- This is an expression unlike if-else in C/C++ where it is a statement.
- It always produces a value.
- Both branches must return the same type.
- It can be used anywhere an expression is allowed.

```
2 * if "one" = "two" then 3 else 4;;
```

```
- : int = 8
```

🤔Any similar expression in C/C++?

- Specifies where to go next or what action to perform?
- It does not evaluate to a value and an if block can't be assigned to a variable.

Data Types

```
let house_number = 71;;
let lakh = 1_00_000;;

let pi = 4.0 *. atan 1.0;;

(1 < 2) = false;;

'a';;

"Hello" ^ " " ^ "world";;
```

int → 64-bit platforms range is -2^{62} to 2^{62} -1. Why 63 bits? One bit is used by OCaml runtime implementation.

float → approximate representation. What does adding 0.1 and 0.2 produce?

bool → IEEE 754 standard. 53 bits of mantissa & exponent ranges -1022 to 1023

char → represented as 8-bit integers between 0 and 255

string → ^ is string concat operator

Try Out

- Evaluate `max_int`
- Evaluate `min_int`
- Evaluate `max_int + 1`

Local Definitions

```
let y = 50 in y * y;;
y;;
```

```
Error: Unbound value y
```

- This creates a binding for `y` only in the expression following the `in` keyword. It throws an error in the global scope as can be seen from the last line of the above code block.
- `let __ = __ in __` is an expression, and it can be used within another expression in order to have several values with their own names.

```
let a = 1 in
let b = 2 in
   a + b;;
```

Type Conversions

```
int_of_char '\n';;

float_of_int 2;;

"Hello".[1];;
```

```
- : int = 10
- : float = 2.
- : char = 'e'
```

- float_of_int
- int_of_float
- int_of_char
- char_of_int
- string_of_int
- string_of_float
- string_of_bool
- bool_of_string
- int_of_string
- float_of_string

Data Types

- Predefined data structures include `tuples`, `arrays`, `lists`
- `list`
  - empty list are called *nil* and represented by [ ]
  - the : : operator is used to add elements in front of the list
  - all elements must be of same type
  - lists are immutable

```
let lst = ["one"; "two"; "three"];;
"zero" :: lst;;
let e = [];;
```

type variable, meaning it can be a list of any type

```
val lst : string list = ["one"; "two"; "three"]
- : string list = ["zero"; "one"; "two"; "three"]
val e : 'a list = []
```

Data Types

- `tuple`
  - Tuples are fixed-length collections of elements of any type. Written as `(1, 2, 3)` or `(1, "two", 'T')`
- `array`
  - Arrays are mutable, fixed-size block of memory.
  - all elements must be of same type.

```
let nums = [| 1; 2; 3 |];;
nums.(1) <- 200;;
nums;;
```

```
val nums : int array = [|1; 2; 3|]
- : unit = ()
- : int array = [|1; 200; 3|]
```