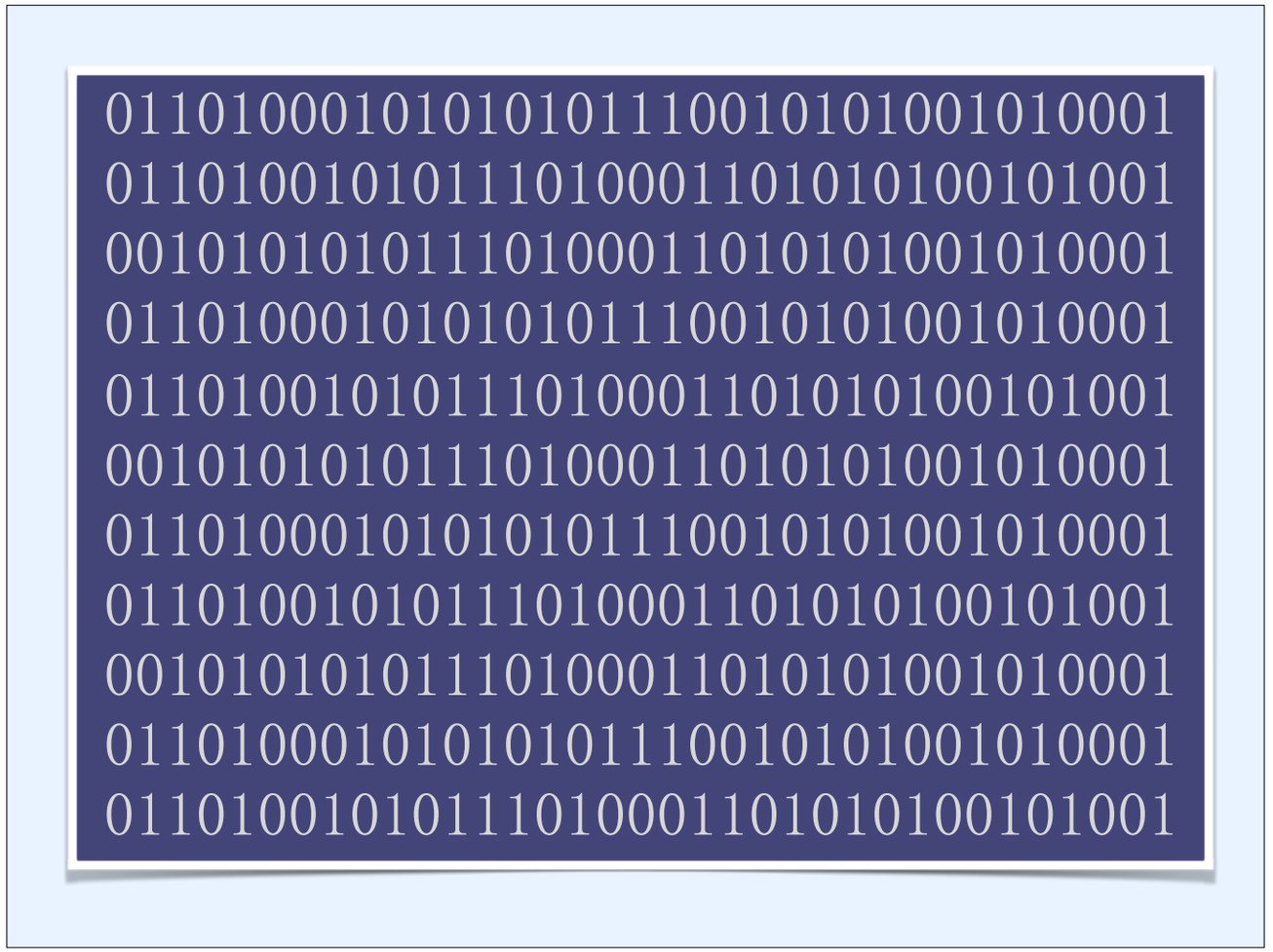
Programming Languages

COMSC343

Fall 2018

How do we talk about
programming languages?

What's the most basic programming language?



Why is it basic?

Right, so it’s basic because there’s no abstraction, right?

What is abstraction?
Why is it useful?

How we describe a programming language is largely based on what kind of abstractions it provides.

Does syntax matter?

```
-module(hello).
-export([hello_world/0]).
hello_world() -> io:fwrite("hello, world\n").
```

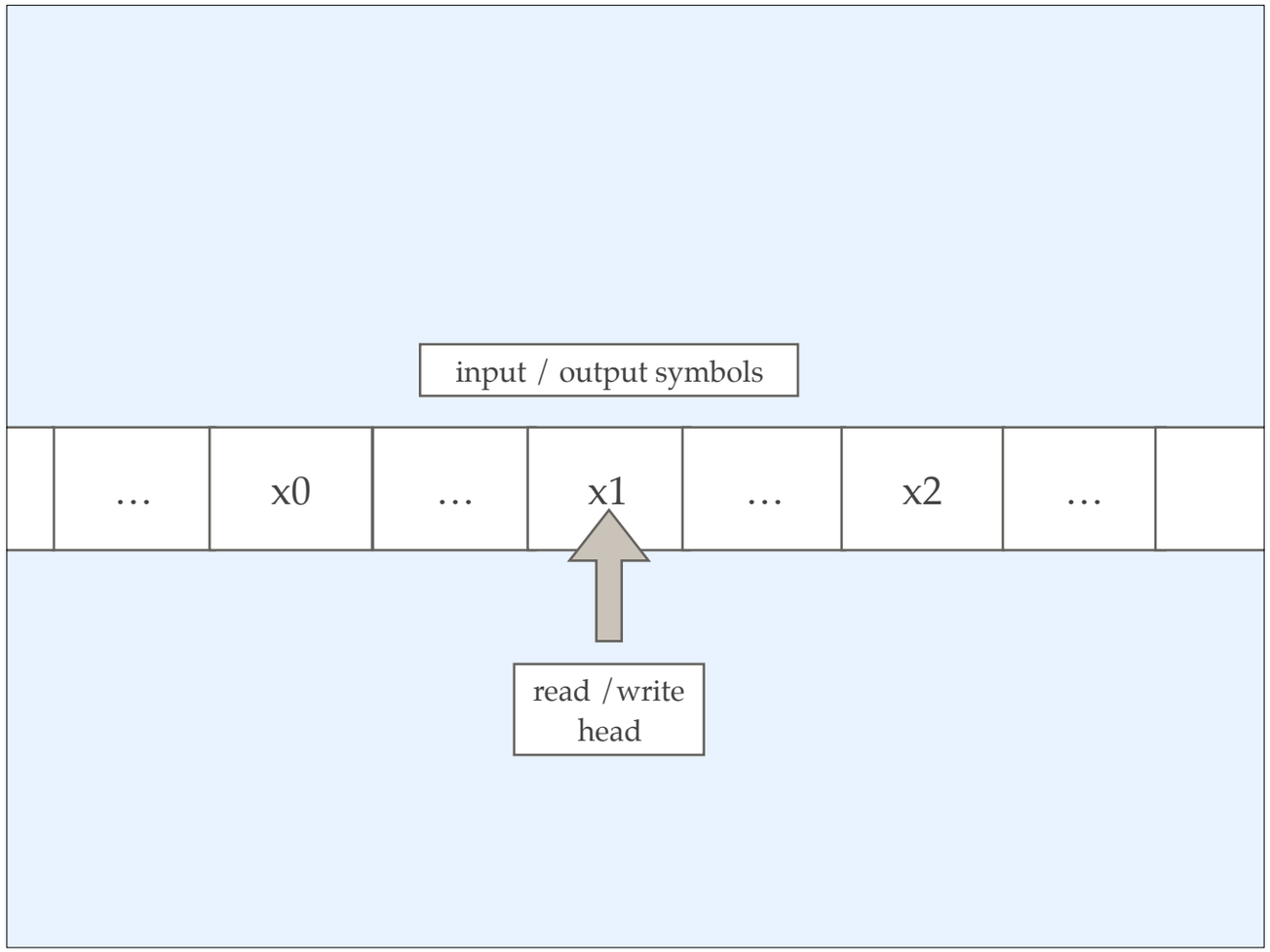
```
mov edx,len;
mov ebx,1;
mov eax,4;
int 0x80 ;
mov eax,1;
```

```
(define (hello-world )
  (printf “Hello world!”))
```

```
def HelloWorld():
  print(“Hello world!”)
  return
```

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

You can describe programming languages based on their syntax (“curly brackets”, “white space”), but that’s pretty superficial.
What we really care about is what the programming language does, right? That is the semantics of the language.
So we decided that binary is the most basic syntax. But what we really care about is what the program does, not what it looks like.
What about semantics? Can we also talk about a fundamental semantics?



Well, kind of. You've heard of Turing-complete languages, right? What does that mean?
But wait, isn't assembly also Turing complete? I mean, our Java programs compile down to assembly, right? So talking about the expressive power of a language also isn't particularly helpful, since all the languages we regularly work with are Turing complete.

Application domains

Statistical modeling

Web programming

Mobile devices

Networks

Scripting

Scientific computing

Machine learning

You can also describe programming languages based on their domain of use (“web programming languages”, “scripting languages”). That’s slightly more informative. But the domain of use is really a reflection of what abstractions the language makes available. For instance, in many web programming languages, input is obtained through event handlers/listeners, whereas in languages like Python and Java, you have abstractions that give access to the file system. That difference is because it’s not safe to give web programs access to the files on the user’s computer.

Abstractions are the core of a programming language

I’d argue that talking about what abstractions a language makes available is the best way of describing a language. It tells us what patterns of thought, what modes of communication, are easier or harder in that language. It tells us something about the style of the language, but more importantly, it tells us about the substance of the language, its meaning.

What are functional programming languages?

- They provide abstractions over functions

In this class, we are going to focus on a type of programming language called a functional program language. What kinds of abstractions do these languages make available? Well, the clue is in the name: they provide rich abstractions over functions.
Most of the abstractions that we work with in languages like Java and Python are abstractions over data: for instance, a Java array is an abstraction over an atomic data type like strings or ints. You cannot, in Java, have an array of functions. But in Racket, the language we're going to learn, you can!

What are functional programming languages?

- They provide abstractions over functions
- They treat functions like other values in the language

The key property of so-called functional programming languages is that they treat functions as values and give them equal status as other more familiar datatypes like integers and strings. These languages provide abstractions over functions. In this class, we're going to combine functions, write functions that produce and consume other functions, and abstract over functions themselves, rather than just writing functions that abstract over data. Functions are going to be our data.

What are functional programming languages?

- They provide abstractions over functions
- They treat functions like other values in the language
- They emphasize recursion over iteration

Because we're focusing on these new abstractions, we're going to restrict ourselves from using some of the more familiar data abstractions, In particular, in this class, we are going to use a subset of Racket that does not allow mutation.

Mutation

```
x = 5
x = 7
```

```
for (int x = 0; x < 10; x++){
      println(x);
}
```

What is mutation? Mutation is when we overwrite the value of a variable. So in the subset of Racket that we will use, we won't be able to write code like this:

Are we drastically limiting our expressive power by doing this? It turns out that we aren't.

Lambda Calculus

$$\lambda y.\lambda x.y+x$$

lambdas bind variables (they demarcate scope)
lambda calculus encodes computation using the concepts of function application, substitution, binding, and scope.
There's no mutation in lambda calculus, unlike with Turing machines.
BUT anything that can be computed by a Turing machine can also be computed in lambda calculus.

> It was stated ... that "a function is effectively calculable if its values can be found by some purely mechanical process". We may take this literally, understanding that by a purely mechanical process one which could be carried out by a machine. The development ... leads to ... an identification of computability with effective calculability.
> Alan Turing, *Systems of Logic Based on Ordinals*

the Church–Turing thesis (computability thesis, Turing–Church thesis, Church–Turing conjecture, Church's thesis, Church's conjecture, and Turing's thesis)
In a proof-sketch added as an "Appendix" to his 1936–37 paper, Turing showed that the classes of functions defined by λ-calculus and Turing machines coincided

Racket is Turing-complete

(even without mutation)

We just have to learn to think in a functional style

That is, it has equal expressive power: it just reframes the way that Turing machines handle data in terms of function application, substitution, and binding.
That's the reason we're diving into functional programming: if we can grasp the core concepts of computer science from this different perspective, then we'll understand them even more deeply.