Lec 2: Abstraction

CS220: Programming Principles

Sang Kil Cha

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don't accept late responses.

Submit Your GitHub ID

Programs

Programming

- Programming is the process of creating a program.

Programming

- Programming is the process of creating a program.
- We represent a program with a ***language*** (or programming language).

Programming Languages

High-level language: F#, Haskell, Scala, etc.

Low-level language: Assembly language

Low-level Programming Languages

Low-level programming languages provide a way to ***directly*** manipulate the computer hardware.

Low-level Programming Languages

Low-level programming languages provide a way to ***directly*** manipulate the computer hardware.

What are the pros and cons of low-level programming languages?

High-level Programming Languages

High-level programming languages provide a way to ***abstract*** our ideas about computations.

High-level Programming Languages

High-level programming languages provide a way to ***abstract*** our ideas about computations.

Why abstraction matters?

Abstraction in Math

We see them all the time.

Let x be a "complex formula". Then, $y = x + \ldots$.

Why Abstraction?

1. Abstraction hides unnecessary details, making things easier to understand.
2. Abstraction allows us to focus on the important parts of a problem.
3. Abstraction allows us to build complex systems by combining simple ideas (because abstracted ideas can be easily combined to form more complex ideas).

Abstraction in Programming Language

Q: Which programming language construct helps in abstracting things away?

Recall from CS101.

Programming is All About Abstraction

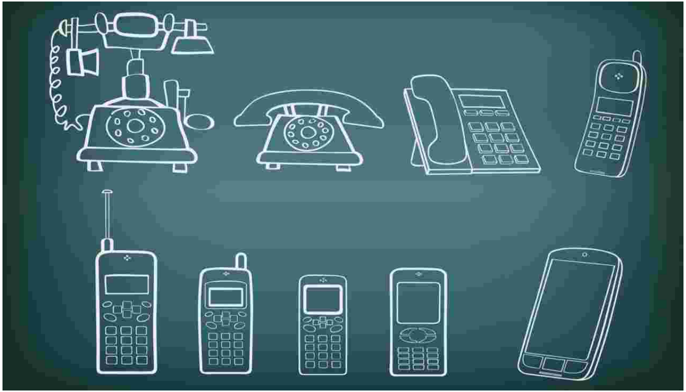


Image from https://techsuplex.com/2017/02/01/evolution-mobile-phone/

Why Abstraction?

Abstraction is what our brains do.

- We can only understand simple information at a time.
- We can only understand abstracted information.
- We need abstraction to build large and complex systems.

Good Programming Language?

A good programming language should help developers abstract things away thereby making the code readable. And we will learn how to do it with F#.

F#

F# is a “functional-first” language.

- F# is a multi-paradigm language that prioritizes functional programming.
- F# code is concise and easy to read.
- F# code is likely to be correct.
- F# code requires less maintenance cost.
- But may require skills and effort to write code.

Ingredients of Programs

Key Ingredient: Value

A program is a series of computations (***functions***) that eventually result in a value.

Key Ingredient: Value

A program is a series of computations (***functions***) that eventually result in a value.

In programming, we use functions to represent computations. We build abstractions with functions, and manipulate data with functions. We also combine functions to make another abstraction, i.e., a function.

In the end, programming is all about creating functions (or writing down values).

Functional Programming

Imperative programming is all about firing up commands: “do this, do that, etc.”, but functional programming is all about writing down functions denoting values. Therefore, we often call functional programming as value-oriented programming.

Expression

An ***expression*** is an abstraction that can be ***evaluated*** by the programming language interpreter to produce a ***value***.

Expressions are the basic building block for programs.

Expression

An ***expression*** is an abstraction that can be ***evaluated*** by the programming language interpreter to produce a ***value***.

Expressions are the basic building block for programs.

Every expression has its own ***semantics***, which describes what kind of ***computation*** the expression represents.

Types

Each expression has its own ***type***, which poses constraints on the expression. We say a program is ***well-typed*** when all the expressions in the program satisfy the type constraints.

Evaluation of an expression simply fails when the type checking fails.

Values

The simplest form of an expression, which does not need further evaluation.

Simple number (`int`).

```
42
```

Simple string (`string`).

```
"hello"
```

Floating-point number (`float`).

```
4.2
```

Boolean (`bool`).

```
true
```

Compound Expressions

Compound expression.

```
42 + 10 * 2
```

?

```
4.2 + 1
```

Compound Expressions

Compound expression.

```
42 + 10 * 2
```

?

```
4.2 + 1
```

A type error raised while ***evaluating*** the expression!

Giving a Name to a Value

We often see statements in Math that look as below:

Let $\mathcal{N}$ be XYZ...

"let" Binding.

```
let price = 100
let numCars = 2 * 3 + 4
// You don't need to remember the exact numbers.
let total = price * numCars
```

Functions

A function is an expression that takes in an expression and returns an expression.

A simple function that takes in an integer and returns an integer.

```
function x -> x + 1
```

In math, the above function would be written as $f(x) = x + 1$.

Named Functions and Applications

We usually give a name to a function in order to call it. We say we ***apply*** a function f to an argument a, when we call f with a as a parameter. Function applications typically do ***not*** require parentheses in F#.

Example: a simple increment function.

```
let increment = function x -> x + 1

increment 1  // this will return 2
increment(1) // you can use parentheses
```

REPL

- Read-Evaluate-Print Loop.
- You first type in an expression to REPL.
- It will ***read*** it, ***evaluate*** it, and ***print*** out the result of evaluation.
- A REPL for F# is `fsi` (or `fsharpi`).
- The result of evaluation is stored in a special identifier, called "`it`".

Type Annotation

We can use ‘colon’ to indicate type of an expression.

Example: type annotation.

```
function (x: int) -> x + 1 // specify argument type

let increment: (int -> int) = // specify func. type
  function x -> x + 1
```

“`int -> int`” is a function type that takes input of type `int` and returns output of type `int`.

Indentation Matters

Valid.

```
let f =
  let y = 42
  function x -> x + y
```

Invalid.

```
let f =
  let y = 42
    function x -> x + y // Wrong indentation.
```

Further reading: `https://fsharpforfunandprofit.com/posts/fsharp-syntax/`.

In-Class Activity #0

- Create a `.fs` file.
- Write a function, named `square`, which takes an integer n as input, and returns $n \times n$ as output.
- Test your function with the REPL.

In-Class Activity #1

- Open up a terminal (console/shell).
- Make sure you can fire the following commands: `git` and `dotnet`.
- Clone the repository from `https://github.com/KAIST-CS220/CS220-Main` by typing

  ```
  $ git clone https://github.com/KAIST-CS220/CS220-Main.git
  ```
- Move to the `Activities` directory:

  ```
  $ cd Activities
  ```
- Type `dotnet run --project Act01` and make sure you can see the output 0 from your console.
- Follow the instruction in the comment of the `Program.fs` file and modify the file to implement the `square` function.

Functions with Two or More Arguments

Q: We have only learned a way to write a function that takes a single argument. How can we write a function that takes in two or more arguments without introducing a new syntax?

(Hint)

- Function is an expression.
- Therefore, we can return a function from a function.

Currying

The technique of translating the evaluation of a function takes multiple arguments into evaluating a sequence of functions, each with a single argument[1].

A simple add function with currying.

```
let add =
  function x -> function y -> x + y

add 1 2 // returns 3
```

[1] Definition taken from Wikipedia: https://en.wikipedia.org/wiki/Currying

Partial Function Application

What just happened when we evaluated the expression: `add 1 2`?

Partial Application Examples

Example: define a new function.

```
let addByTwo = add 2
addByTwo 3 // returns 5
```

Example: define a function that adds three integers.

```
let addThreeInts = // exercise
```

We Love Simplicity!

The Problem: the "`function`" keyword is too strict: it forces that the body, i.e., the expression after the arrow (`->`), of the function should be in one line, and it only allows a single parameter at a time.

We Love Simplicity!

The Problem: the “`function`” keyword is too strict: it forces that the body, i.e., the expression after the arrow (`->`), of the function should be in one line, and it only allows a single parameter at a time.

Use “`fun`” instead of “`function`”!

Have Fun with `fun`

(1) Rewrite add function with `fun`.

```
let add = fun x -> fun y -> x + y
```

(2) Rewrite add function with `fun`.

```
let add = fun x y -> x + y
```

Note: `fun` is usually a better choice due to its simplicity, but there are cases where you should use `function`, especially when you do "pattern matching", which will be covered later.

Make it Even Simpler

We will mostly use this form throughout the course.

(3) Rewrite add function.

```
let add x y = x + y
```

In the same vein, we prefer a simpler form of function application.

```
add 1 2       // Good
add (1) (2) // Bad
add (1, 2)  // Wrong
```

Anonymous Functions

We call a function without its name as an anonymous function, a.k.a., lambda expression (λ). We will see throughout this course why lambda expressions matter.

Anonymous function vs. named function.

```
function x -> x + 1 // anonymous function
let inc x = x + 1   // named function
```

Infix Operators

A function taking two arguments can often be more readable if we use infix notation.

"`add 1 2`" vs. "`1 + 2`"?

In fact, infix operators are a function. For example, the `(+)` operator was also a function.

Example: infix operator.

```
let (+) x y = x + y
```

Nested Functions

Consider a function $f(x, y) = x^2 + y^2$.

```
let sumOfSquares x y = x * x + y * y // (1)
```

But, we can refactor the sqaure function, and use it in `sumOfSquares`.

```
let square x  = x * x
let sumOfSquares x y = square x + square y // (2)
```

Nested Functions

Consider a function $f(x, y) = x^2 + y^2$.

```
let sumOfSquares x y = x * x + y * y // (1)
```

But, we can refactor the sqaure function, and use it in `sumOfSquares`.

```
let square x   = x * x
let sumOfSquares x y = square x + square y // (2)
```

What's the difference between the two functions? Think about the ***abstraction***!

Conditional Expressions

How can we make tests and perform different operations depending on the results of a test?

```
if (boolean expression) then (expression 1) else (expression 2)
```

- The boolean expression part is called ***predicate***, which is an expression that can be evaluated to either `true` or `false`.
- The expression 1 and 2 should have the same type, otherwise, type checking will fail.

Conditional Expression Example

A `abs` function that returns an absolute integer value.

```
let abs x = if x < 0 then - x else x
```

Logical Composition

```
not a   // boolean negation
a || b // boolean or
a && b // boolean and

if (a && b) || (c && d) then 100 else 200
```

Function Pipelining

We can apply function in a reverse order by using the infix operator (|>). This operator allows function pipelining, which links a sequence of functions in an intuitive manner.

```
f 10     // normal application
10 |> f // reverse application
inc (abs 10) // normal function chaining
10 |> abs |> inc // pipelining without parentheses
```

Evaluation

Evaluating an Expression

Evaluation is a procedure, i.e., a function, that takes in an expression as input, and returns a value as output.

The evaluation function `eval`.

```
let eval expr = // How do we implement this ... ?
```

Evaluation Rule

So far we have learned three kinds of expressions: values, let-bindings, and functions. The `eval` function of the expressions should run as follows.

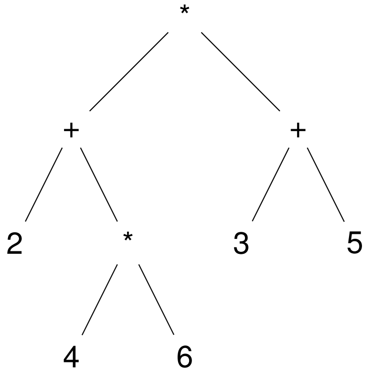
- If the current expression is a value, we simply return the value.
- If the current expression is a let-binding, we first evaluate the ***subexpression***[2] (i.e., the expression body), and memorize the relationship between the identifier and the evaluated value.
- If the current expression is a function, we first evaluate ***subexpressions*** (i.e., parameters) of it, and apply the function to the evaluated parameter values.

---

[2] We often call this as ***applicative-order*** evaluation.

Evaluation Example

Consider a compound expression: `(2 + 4 * 6) * (3 + 5)`.



Conclusion

- A computation is the process of ***evaluating*** a series of ***expressions*** on a given user input.
- Expressions in programming languages provide a way to represent data and ***combine*** computations.
- Functions provide nice ***abstraction*** about computations. In fact, programming is all about abstraction.
- Currying provides a way to deal with multiple parameters with single-parameter functions.
- F# is beautiful, because it is ***concise*** while providing powerful abstraction mechanisms, which will be well covered throughout the course.
- The expression evaluation process naturally introduces recursion, which is indeed the next topic ☺.

Question?

Further Reading

- The wizard book: Chapter 1.1.
- F# Indentation:
  `https://fsharpforfunandprofit.com/posts/fsharp-syntax/`