Lec 2: Abstraction

CS220: Programming Principles

Sang Kil Cha
Programs
Program and Computation

• A **program** is a set of commands to instruct a computer: we make the computer perform **computations**, or computational processes, by means of a program.
Program and Computation

• A program is a set of commands to instruct a computer: we make the computer perform computations, or computational processes, by means of a program.

• A computation typically involves manipulation of data.
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**.
Programming Languages

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

Low-level language: Assembly language
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.

Can we express *simple* ideas in a *simple* manner?
High-level Programming Languages

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

Can we express simple ideas in a simple manner?

Can we combine simple ideas to form more complex ideas?
Programming is All About Abstraction

Image from https://techsuplex.com/2017/02/01(evolution-mobile-phone/
Abstraction
Q: What does this map represent?

Image from https://goo.gl/CYbmFX
Q: Same person?

Image from https://helpx.adobe.com/lightroom/help/face-recognition.html
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 to make the code readable. And we will learn how to do it with F#.
F#

- 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

<table>
<thead>
<tr>
<th>Compound expression.</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>42 + 10 * 2</td>
<td>4.2 + 1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Compound expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>42 + 10 * 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 + 1</td>
</tr>
</tbody>
</table>

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 $N$ be the number of XYZ...

“let” Binding.

```javascript
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.

```fsharp
let increment = function x -> x + 1
increment 1  // this will return 2
increment(1)  // you can use parentheses
```
• 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.

```plaintext
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.
Valid.

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

Invalid.

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

In-Class Activity #0

- Create a .fs file.
- Write a function, named `double`, which takes an integer \( n \) as input, and returns \( n \times n \) as output.
- Test your function with the REPL.
In-Class Activity #1

• git clone https://github.com/KAIST-CS220/2020s-activities

• Change your directory to 01, and type dotnet test.

• Fix the implementation of myfunc in MyFunc.fs, and change the module name so that it becomes M*, where * is your student ID.

• Now, submit the MyFunc.fs file to the activity server\(^1\).

• See if you can get your score.

• If not, you will see an error message, and that means you need to debug your program according to the message.

• To get the full score, you can always re-submit your file during the activity.

\(^1\)The URL will be given during the class.
How Things Work?

See https://github.com/KAIST-CS220/FsClassRoom for more details.

- The server runs with two pre-compiled FS files: (1) library, and (2) test driver.
  - The library defines types and functions. Current it is empty.
  - The test driver defines several test functions.
- The client sends an FS file $f$ to the server.
- The server scans the contents of the file, and compiles it with its library.
- The server then runs the test driver to test your implementation in $f$.
- To get the full score, you need to pass all the tests defined in the test driver.
Notes for In-Class Activities

- You cannot hack the server. Any hacking attempts result in an ‘F’ grade.
- You cannot use System module.
- Do not rename the myfunc function.
- Always remember to fix the module name with your student ID.
Notes for In-Class Activities

- You cannot hack the server. Any hacking attempts result in an ‘F’ grade.
- You cannot use System module.
- Do not rename the myfunc function.
- Always remember to fix the module name with your student ID.
- We will do in-class activities in replacement of attendance check.
Notes for In-Class Activities

• You cannot hack the server. Any hacking attempts result in an ‘F’ grade.
• You cannot use System module.
• Do not rename the myfunc function.
• Always remember to fix the module name with your student ID.
• We will do in-class activities in replacement of attendance check.
• Even if you cannot finish coding in time, you must submit the current code before the class finishes in order to show your attendance.
Notes for In-Class Activities

- You cannot hack the server. Any hacking attempts result in an ‘F’ grade.
- You cannot use System module.
- Do not rename the myfunc function.
- Always remember to fix the module name with your student ID.
- We will do in-class activities in replacement of attendance check.
- Even if you cannot finish coding in time, you must submit the current code before the class finishes in order to show your attendance.
- I will be happy to give extra points to students who can create a valid Pull Request (PR) to the FsClassRoom project. There are several TODOs in the CONTRIBUTING.md file.
Notes for In-Class Activities (cont’d)

• For simplicity, we only ask you to fix a single function, i.e., `myfunc` function, during an activity.
Notes for In-Class Activities (cont’d)

• For simplicity, we only ask you to fix a single function, i.e., myfunc function, during an activity.

• You may do multiple activities within a lecture.
Notes for In-Class Activities (cont’d)

• For simplicity, we only ask you to fix a single function, i.e., myfunc function, during an activity.
• You may do multiple activities within a lecture.
• There will be no extension: any activity should be done before the class finishes.
Notes for In-Class Activities (cont’d)

• For simplicity, we only ask you to fix a single function, i.e., `myfunc` function, during an activity.
• You may do multiple activities within a lecture.
• There will be no extension: any activity should be done before the class finishes.
• If you did not submit anything during a lecture, it will be counted as “absent”. Therefore, you must submit something before a class terminates.
Notes for In-Class Activities (cont’d)

• For simplicity, we only ask you to fix a single function, i.e., `myfunc` function, during an activity.
• You may do multiple activities within a lecture.
• There will be no extension: any activity should be done before the class finishes.
• If you did not submit anything during a lecture, it will be counted as “absent”. Therefore, you must submit something before a class terminates.
• Even if your solution was entirely correct except that your module name had a typo, you will get zero point. But, note, you get instant feedback from the system, so this cannot be an excuse.
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\(^2\).

A simple add function with currying.

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

add 1 2 // returns 3
```

\(^2\)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.

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

#### Example: define a function that adds three integers.

```javascript
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.

Use "fun" instead of "function"!
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`.

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

(2) Rewrite add function with `fun`.

```ocaml
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”.

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 ($\lambda$). We will see throughout this course why lambda expressions matter.

Anonymous function vs. named function.

```plaintext
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 square 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 $f(x, y) = x^2 + y^2$.

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

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

```plaintext
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?

```plaintext
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.

```fsharp
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 (\(\mid\rangle\)). This operator allows function pipelining, which links a sequence of functions in an intuitive manner.

\[ f \ 10 \ // \ normal \ application \]
\[ 10 \mid\rangle \ f \ // \ reverse \ application \]
\[ \text{inc} \ (\text{abs} \ 10) \ // \ normal \ function \ chaining \]
\[ 10 \mid\rangle \ \text{abs} \mid\rangle \ \text{inc} \ // \ pipelining \ without \ parentheses \]
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\(^3\) (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.

\(^3\)We often call this as **applicative-order** evaluation.
Evaluation Example

Consider a compound expression: \((2 + 4 \times 6) \times (3 + 5)\).
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/