Classical Programming Topics with Functional Programming

VISNOVITZ Márton

Abstract. In traditional Hungarian programming education programming theorems are the foundation of learning programming. These generic algorithm patterns are traditionally introduced using the imperative programming paradigm with sequences, loops, and conditions. After learning about programming theorems and basic algorithms the next step is usually to go in the direction of object-oriented programming. One possible way to approach object-oriented programming is through enumerators: the implementation of enumerable data types and the application of programming theorems on them. To prove that these classical programming topics, programming theorems and enumerators can also be implemented with the principles of functional programming we present implementations using the functional programming paradigm along with concepts from object-oriented programming. With these we aim to prove that it is possible to introduce programming theorems and enumerators in introductory programming with functional programming as well. This paper also presents the possibility of a *higher-order-first* approach to programming education and the possible educational advantages this method.

Keywords: introductory programming, programming theorems, functional programming, object-oriented programming, enumerators, higher-order-first approach

1. Introduction and state of the art

In introductory programming the choice of first programming language and the corresponding first programming paradigm has a very important role. Based on the foundation of programming paradigms several strategies for teaching introductory programming exist, such as the algorithm-first, imperative-first, functional-first, hardware-first, etc. approaches [1]. In Hungarian literature a slightly different naming convention is present [2] but the core concepts are basically the same.

Based on our investigation of the curricula of the most renowned universities in Hungary that have some form of Computer Science (CS) of Computer Engineering (CE) program[1], in Hungarian practice an algorithm-first approach is used in most cases starting with basic input-output operations and basic control structures and algorithms. This is usually followed up by the learning about the object-oriented paradigm. Some universities also introduce the functional paradigm early, along with the imperative and the object-oriented paradigms. The training programmes of the top US and European universities with CS or CE programs[2] follow a similar pattern, however courses on functional programming are more prevalent. This shows that alongside the usual imperative-first and algorithms-first approaches [3] the functional-first approach is also relevant [4,5] in current practice.

The goal of this paper is to show that it is possible to use the tools of functional programming to introduce various topics of the classical algorithm-first and object-oriented-second methodologies, namely programming theorems and enumerators, and to show the analogies between the classical

[1] Eötvös Loránd University, University of Debrecen, Budapest University of Technology and Economics, University of Szeged, University of Pécs

[2] Stanford University, MIT, Carnegie Mellon University, Harvard University, University of Oxford, ETH Zurich, University of Cambridge, Imperial College London

and functional implementations. Based on the findings of this investigation we also present a possible direction for creating a new methodology for early programming education.

2. Programming theorems

Classic Hungarian programming education primarily uses the algorithm-first, object-oriented-second approach based on the determinative works of Szlávi and Zsakó [6] that were also published in the booklet series Mikrológia [3,7,8]. One of the core concepts of this so called "Systematic programming" methodology is the usage of programming theorems. Programming theorems are a group of basic algorithms that provide solutions to generic group of algorithmic problems [7,9,10,11]. They also provide a framework for problem-solving by making the specification, planning and implementation of algorithms straightforward by using an analogy-based approach [10]. Szlávi and Zsakó identify eleven such programming theorems [7,11], the system of Gregorics has seven [9,10]. While there is a large intersection between the two lists, the names of theorems can vary. There is also a third source of naming conventions to consider: the vocabulary (keywords) of programming languages. In this paper we follow the naming conventions used by most programming languages to refer to the programming theorems.

All programming theorems can be defined as a function that takes one or more collections of values as input. The type of the values in these collections can be arbitrary. Hereinafter we will use the generic types `X` (and `Y`) to refer these types. Szlávi and Zsakó group the programming theorems into two categories based on the number of their inputs and outputs [7,11] Based on this categorization they identify the following groups:

In the first category there are programming theorems that take a single collection of values as an input and output a single value:

- **Reduction (also called Folding):** Returns a single value by applying a combining function systematically to a cumulative value and each element in the input collection. The combining function takes two values of type `X` and produces a single value of type `X`.
- **Maximum Selection**: Returns a single element (or its index) from the input collection that is the maximum (or minimum) based on some comparison between elements.
- **Counting:** Returns a single integer value that is the number of elements in the input collection that meet a certain condition.
- **Decision:** Returns a single logical value that shows whether the input collection has any values that meet a certain condition.
- **Selection:** Returns the first element (or its index) from the input collection that meets a certain condition (existence of such item is guaranteed by the precondition).
- **Search:** Returns the first element (or its index) from the collection that meets a certain condition if there is any such element.

The second category has programming theorems that take one or more collections of values as an input and output one or more collections of values:

- **Mapping:** Returns a new collection in which each element is a transformed version of the value of the element with the same index in the input array. The transformation is done by some transforming function that takes a value of type `X` and returns a value of type `Y`.
- **Filtering:** Returns a new collection that contains those and only those elements in the input collection that meet a certain condition.

- **Partitioning:** Returns two collections. The first contains elements from the input collection that meet a certain condition, the second contains those elements that do not.
- **Intersection:** Takes two or more sets with elements of type `X` and returns a new set that contains those elements of the input sets that are present in all of them.
- **Union:** Takes two or more sets with elements of type `X` and returns a new set that contains all values that are present in any of the input sets.

2.1. Imperative and functional implementations

In classical algorithm-based programming we use the core concepts of imperative programming (i.e. statements, loops, and conditions) to implement programming theorems. In the following sections we provide the formal signature for each programming theorem and two implementations for each: the classic implementation using imperative programming and an implementation using functional programming with the concept of folding. The goal of this comparison is to prove that the programming theorems can be introduced through not only imperative but functional programming as well.

Notes for the implementations

For the following code examples a programming language or algorithm/function-description method had to be chosen [1,2,12,13]. For the purpose of the demonstration we chose the TypeScript programming language[3] as it is a multi-purpose programming language that supports many programming paradigms (including imperative, functional, and object-oriented); thus, we can use it in all examples. This helps with the transition between the concepts that we present in the following sections.

The syntax of TypeScript follows the conventions of C-style programming languages making the code easier to read. TypeScript also has support for static type annotations and template functions and classes. This makes the following code snippets generic for any input and output value types. All the implementations below are written as template functions and classes (using generic types `X` and `Y`). Another reasoning for choosing the TypeScript programming language for the demonstration is that TypeScript (and the JavaScript language that it is a superset of) has a lot of properties that are beneficial from an educational point of view. [14,15]

The TypeScript programming language has a special value called `undefined` that represents a missing on uninitialized value. We used this `undefined` value for the Search theorem to represent "no output". Implementations of some programming theorems assume a non-empty array as an input. Checking for this precondition is not present for more concise code. We also used some aliases for built-in types and values for the same reason. The list of these aliases can be seen in Table 1. For the functional implementations we use the array destructuring[4] (`[x0, ...xs]`) and conditional statement (`a ? b : c`) features of TypeScript. All functional code snippets with the exception of the Selection theorem are based on pattern matching with conditional statements: *if the input collection has exactly one element/has no elements then return some value, otherwise use a recursive formula to calculate the result.* If a programming theorem cannot accept an empty input, then the first check

[3] https://typescriptlang.org

[4] https://www.typescriptlang.org/docs/handbook/variable-declarations.html#array-destructuring

is whether there is only one element left in the collection. If an empty input collection is accepted, then we check whether the first element is `undefined` (i.e. the input is empty).

Alias (symbol)	Meaning	TypeScript value/type
Z	integer type	`number` (type)
R	real type	`number` (type)
B	logical type	`boolean` (type)
T	true logical value	`true` (value)
F	false logical value	`false` (value)
U	undefined type and value	`undefined` (type, value)

Table 1: Aliases used in the code examples

Reduction, Counting, Maximum Selection

```
type ReduceTheorem  = <X> (x: X[], f: (s: X, e: X) => X) => X
type MaximumTheorem = <X> (x: X[], m: (a: X, b: X) => X) => X
type CountTheorem   = <X> (x: X[], p: (e: X) => B)       => Z
```

Function signatures of the Reduction, Counting and Maximum Selection theorems

The Reduction (a.k.a. Folding, Sequential computing [11] or Summation [10]) programming theorem has two variants. The first takes the first value of the input collection and uses it as the starting value for the cumulation. The second variant uses a starting value (often named `f0`) for the reduction. This second version is more general but, in most cases, we use a value that is neutral for the `f` function so that the starting value does not change the result (e.g. 0 for addition, 1 for multiplication). This means that most of the time this starting value can be omitted, and we can start the reduction with the first element of the input.

In the following example we show the implementation for the first variant of the Reduction theorem that does not use the starting value. This means that only non-empty inputs can be accepted for the theorem. The Reduction theorem starts with the first element of the input then applies the `f` function to the current result and the next element of the collection until all elements of the input have been processed.

```
reduce: ReduceTheorem = (x, f) => {
  let s = x[0];

  for (let i = 1; i < x.length; i++) {
    s = f(s, x[i]);
  }

  return s;
}
```

```
reduce: ReduceTheorem = ([x0, ...xs], f) =>
  xs.length === 0 ? x0
  :                 f(reduce(xs, f), x0)
```

Imperative and functional implementations of the Reduction theorem

The Maximum Selection theorem [10,11] takes the function `m` as an input which is a function that returns the maximum of two values based on some ordering. The theorem then uses this function to calculate the "larger" value of the current maximum and the next element in the input. This concept is the same as the Reduction theorem where the `f` function of the reduction is the `m` “maximum” function.

```
maximum: MaximumTheorem = (x, m) => {
  let s = x[0];

  for (let i = 1; i < x.length; i++) {
    s = m(s, x[i]);
  }

  return s;
};
```

```
maximum: MaximumTheorem = ([x0, ...xs], m) =>
  xs.length === 0 ? x0
  :                   m(maximum(xs, m), x0)
```

Imperative and functional implementations of the Maximum Selection theorem

The Counting theorem’s [10,11] implementation is based on the concept of adding ones and zeroes for each element in the input based on whether they meet the `p` condition.

```
count: CountTheorem = (x, p) => {
  let s = 0;

  for (let i = 0; i < x.length; i++) {
    if (p(x[i])) {
      s += 1;
    }
  }

  return s;
}
```

```
count: CountTheorem = ([x0, ...xs], p) =>
  x0 === U ? 0
  :          (p(x0) ? 1 : 0) + count(xs, p)
```

Imperative and functional implementations of the Counting theorem

Decision, Selection, Linear Search

```
type DecideTheorem = <X> (x: X[], p: (e: X) => B) => B
type SelectTheorem = <X> (x: X[], p: (e: X) => B) => X
type SearchTheorem = <X> (x: X[], p: (e: X) => B) => X | U
```

Function signatures of the Decision, Selection and Linear Search theorems

The Decision theorem [11] is one of the theorems that is missing from Gregorics’ list [10]. It takes the predicate `p` as an input and uses it to decide whether there is any element in the collection that satisfies `p`. Both the imperative and the functional (due to lazy evaluation) implementations stop the evaluation if a “good” value is found.

```
decide: DecideTheorem = (x, p) => {
  let i = 0;

  while (i < x.length && !p(x[i])) {
    i++;
  }

  return i < x.length;
};
```

```
decide: DecideTheorem = ([x0, ...xs], p) =>
  x0 === U ? false
  :          p(x0) || decide(xs, p);
```

Imperative and functional implementations of the Decision theorem

The Selection [10,11] and the Search [11], (aka. Linear Search [10]) theorems are very similar. Both take a predicate `p` as an input to find an element in the input that satisfies `p`. The main difference is that for the Selection theorem we have an extra precondition: there is at least one element for which `p` is true. That is why we do not have to check if we reached the end of the collection.

```
select: SelectTheorem = (x, p) => {
  let i = 0;

  while (!p(x[i])) {
    i++;
  }

  return x[i];
};
```

```
select: SelectTheorem = ([x0, ...xs], p) =>
  p(x0) ? x0 : select(xs, p);
```

Imperative and functional implementations of the Select theorem

The Search theorem does not have this precondition. Some implementations return a logical value that indicates whether we found any element for which `p` is true. Other implementations return a special value for inputs without a “good” value. In our implementation we return `undefined` if there are no element in the input for which `p` is true.

```
search: SearchTheorem = (x, p) => {
  let i = 0;

  while (i < x.length && !p(x[i])) {
    i++;
  }

  return i < x.length ? x[i] : U;
};
```

```
search: SearchTheorem = ([x0, ...xs], p) =>
  x0 === U ? U
  : p(x0)  ? x0
  :          search(xs, p);
```

Imperative and functional implementations of the Search theorem

Mapping, Filtering, Partitioning

```
type MapTheorem        = <X, Y> (x: X[], f: (e: X) => Y) => Y[]
type FilterTheorem     = <X>    (x: X[], p: (e: X) => B) => X[]
type ParitionTheorem   = <X>    (x: X[], p: (e: X) => B) => [X[], X[]]
```

Function signatures of the Mapping and Filtering theorems

The Mapping theorem (aka. Copying [11]) has an additional generic type `Y` present in its function signature. The reason for this is that the `f` function may transform values to a different type than the type of the input values (e.g. mapping strings to their lengths). This theorem's implementation is based on either the option to add a new element to an array (`push`) or the ability to construct a new array by listing the elements of another array using the spread operator (`...`).

```
map: MapTheorem = (x, f) => {
  let s = [];

  for (let i = 0; i < x.length; i++) {
    s.push(f(x[i]));
  }

  return s;
};
```

```
map: MapTheorem = ([x0, ...xs], f) =>
  x0 === U ? []
  :          [f(x0), ...map(xs, f)];
```

Imperative and functional implementations of the Map theorem

The Filtering theorem (aka. Multiple Item Selection [11]) uses a similar concept as the Mapping theorem. The difference is that instead of changing the values of the input collection when we copy it to the output collection, we may omit some elements based on the predicate `p`. It is also similar to the Search theorem with the exception that it does not only search for the first value that satisfies the `p` condition but creates a collection of all of such elements in the input.

```
filter: FilterTheorem = (x, p) => {
  let s = [];

  for (let i = 0; i < x.length; i++) {
    if (p(x[i])) {
      s.push(x[i]);
    }
  }

  return s;
};
```

```
filter: FilterTheorem = ([x0, ...xs], p) =>
  x0 === U ? []
  : p(x0)  ? [x0, ...filter(xs, p)]
  :          [...filter(xs, p)];
```

Imperative and functional implementations of the Filter theorem

The classic imperative implementation of the Partitioning theorem [7,11] is just a more efficient algorithm for applying the Filtering theorem twice. It uses a single loop for both filters thus making the algorithm faster.

Unlike other theorems Partitioning returns two values (two collections) as an output. In the implementations below we used a TypeScript array to return both collections at the same time.

```
partition: ParitionTheorem = (x, p) => {
  let s1 = [], s2 = [];

  for (let i = 0; i < x.length; i++) {
    if (p(x[i])) {
      s1.push(x[i]);
    } else {
      s2.push(x[i]);
    }
  }

  return [s1, s2];
};
```

```
const partition: ParitionTheorem = ([x0, ...xs], p) =>
  x0 === U ? [[], []]
  : p(x0)  ? [[x0, ...partition(xs, p)[0]], [...partition(xs, p)[1]]]
           : [[...partition(xs, p)[0]], [x0, ...partition(xs, p)[1]]]
```

Imperative and functional implementations of the Filter theorem

Intersection, Union

```
type UnionTheorem     = <X> (x: X[], y: X[]) => X[]
type IntersectTheorem = <X> (x: X[], y: X[]) => X[]
```

Function signatures of the Partitioning, Intersection and Union theorems

Even classically the Intersection and Union theorems are just combination of other theorems [7,11]. The Intersection theorem can be viewed as the combination of the Filtering and the Decision theorems, while the Union theorem is basically a Mapping theorem plus the combination of again Filtering and Decision. For this reason, we do not detail the implementations for these theorems.

2.2. Programming theorems with higher-order functions

In addition to using *folding* and *recursion*, another way of implementing programming theorems with functional programming would be by using basic *higher-order functions*. The higher-order functions `reduce`, `map`, and `filter` are present in practically every programming language that supports functional programming (names may vary). Using only these three higher-order functions, it is possible to create easy "one-liner" solutions for all programming theorems (see figure below). This means that introducing only the concept of higher-order functions and these three basic functions are enough to easily solve problems that require the usage of programming theorems.

```
reduce       = (x, f) => x.reduce(f)
count        = (x, p) => x.filter(p).length
maximum      = (x, m) => x.reduce(m)
decide       = (x, p) => x.filter(p).length > 0
select       = (x, p) => x.filter(p)[0]
search       = (x, p) => x.filter(p)[0]
map          = (x, f) => x.map(f)
filter       = (x, p) => x.filter(p)
partition    = (x, p) => [x.filter(p), x.filter(e => !p(e))]
```

Functional implementation of the theorems with higher-order functions

As seen in the code snippet above, `reduce`, `map`, and `filter` are exactly equivalent with the corresponding Reduction, Mapping and Filtering programming theorems. All the other theorems can be implemented using only these three higher-order functions. Many functional programming languages provide built-in functions for more than only these three theorems (e.g. in TypeScript the `some` method for the Decision and the `find` method for the Selection and Search theorems). However, `reduce`, `map`, and `filter` are very common in real-life programming and are enough to provide an "easy-enough" solution to the rest of the theorems.

Szlávi and Zsakó group programming theorems based on whether their output is a single value or one or more collections of values [11]. Another way to group them could be based on their form in functional programming. It is possible to group the theorems into three distinct categories based on which higher-order function can be used for their implementations.

Reduction	**Filtering**	**Mapping**
Reduction	Filtering	Mapping
Maximum Selection	Decision	
	Searching	
	Counting	
	Partitioning	

Grouping theorems based on their implementations with higher-order functions

It is also possible to implement the functional Mapping and Filtering theorems using the Reduction theorem just like in imperative programming [11,16], however that would result in overly complicated solutions for many theorems.

3. Enumerators

One of the classic ways to follow up an introductory, algorithm-first programming education is to continue with the means of data encapsulation and proceed towards object-oriented programming. This usually leads to the introduction of abstract data structures, classes, and objects. Another approach for proceeding towards object-oriented programming is with *enumerators*. Classic (imperative) enumerators are data types that have the following properties [9,17]:

- It is possible to point to its first element (based on an internal ordering),
- step to the next element,
- ask for the currently pointed element,
- and ask if the enumeration has ended.

The classic, imperative implementations of programming theorems all work with such enumerators. Using the interface of enumerators all the array and indexing specific code in the programming theorem can be easily replaced. [17]

```
interface Enumerable<X> {
  first   : () => void;
  next    : () => void;
  current : () => X;
  end     : () => boolean;
}
```

Interface for “classic” enumerators

```
const reduce = <X>(x: Enumerable<X>, f: (s: X, e: X) => X) => {
  x.first();
  let s = x.current();

  for (x.first(); !x.end(); x.next()) {
    s = f(s, x.current());
  }

  return s;
}
```

Implementation of the Reduction theorem using “classic” enumerators

Collections are a subtype of enumerators that store values of a specific type that can be enumerated [9]. They extend the `Enumerable` interface by a method that allows the addition of an element into the collection. This is required to implement programming theorems that output not only a single value but one or more new collections (e.g. Mapping, Filtering).

```
interface Collection<X> extends Enumerable<X> {
  add : (e: X) => void;
}
```

Interface for collections

3.1. Functional enumerators

It is also possible to create enumerators for the functional implementations of programming theorems as well. This requires creating a new definition for functional enumerators to suit our requirements. The requirements for a functional enumerator are the following:

- It is possible to decide whether it is empty,
- ask for the first (head) element,
- ask for the rest of the elements (tail) – i.e. elements except for the first.

Programming theorems for such enumerators can be implemented as standalone functions or as methods of an enumerator class itself [18]. If we use the latter method, we must use an abstract class instead of an interface to be able to implement the theorems as class methods. As for the methods of the `Enumerable` interface, we leave them as abstract methods, showing that these must be implemented for each specific enumerator (e.g. sequential input file enumerator, range enumerator, etc). The theorems can still use these abstract methods in their implementations.

```
interface Enumerable<X> {
  isEmpty : () => boolean;
  next    : () => X;
  rest    : () => Enumerable<X>;
}
```

Abstract class and methods for "Functional" Enumerators

This functional enumerator interface can also be extended to allow the addition of an element. The signature of this extension to create the `Collection` interface is similar to the method required for classic collections. The main difference is that as in functional programming it is not allowed to change the internal state of an object, we must construct a new object when we add an element to a collection.

```
interface Collection<X> extends Enumerable<X> {
  add : (e: X) => Collection<X>;
}
```

Interface for "functional" collections

3.2. Programming theorems for functional enumerators

Programming theorems for functional enumerators can be implemented either as standalone functions or methods on the enumerator with the folding method that is shown in Section 2.1. The main difference between the two approaches is that if a theorem is implemented as a method then it does not have to take the collection as an input parameter, it is automatically passed via the `this` reference. As shown in Section 2.2, we only need to implement three methods, `reduce`, `map`, and `filter` to have access to all programming theorems.

As with these enumerators we target functional programming it is important that the theorem implementations work in accordance with the main principles of functional programming. This means that none of the theorems can change the internal state of an enumerator (they should be immutable) and must return a new enumerator when necessary.

```
abstract class CollectionWithTheorems<X> extends Collection<X> {
  abstract isEmpty : ()     => boolean;
  abstract first   : ()     => X;
  abstract rest    : ()     => CollectionWithTheorems<X>;
  abstract add     : (e: X) => Collection<X>;

  reduce = (f: (s: X, e: X) => X): X =>
    this.rest().isEmpty() ? this.first()
                          : f(this.rest().reduce(f), this.first());
  map = <Y> (f: (e: X) => Y): MappableCollection<Y> =>
    this.empty()          ? this.constructor()
```

```
                              : this.rest().map(f).add(f(this.first()));
  filter = (p: (e: X) => B): FilterableCollection<X> =>
    this.empty()              ? this.constructor()
    : p(this.next())          ? this.rest().filter(p).add(this.first())
                              : this.rest().filter(p);
}
```

Abstract class with the signatures and folding-based implementations of programming theorems

4. Educational considerations

In classic programming education we usually follow an algorithm-first, object-oriented-second approach. With this method the focus in early programming is on the *low-level* concepts, i.e. *how things work* and how to implement basic algorithms. With a functional-first approach the same can be said if we start with the *low-level* concepts first, like pattern-matching or folding. This approach results in a *bottom-up* learning process.

However, there is the possibility to start programming education with a *higher-order-first* approach. This would mean that programming is introduced with *high-level*, functional-style programming: using collections and the `reduce`, `map` and `filter` functions as shown in Section 2.2. This would allow students to easily solve most data-processing tasks easily using high-level tools. Based on our experiences in various introductory and web programming courses and our experience with teaching pupils in summer camps [15] this *top-down* learning approach can emphasise *problem-solving*, giving student early satisfaction and quick success to keep their motivation high. Principles of object-oriented programming could also be introduced early by data-encapsulation with classes and objects and data-processing methods. It would also be possible to use this approach combined with web technologies, web programming in the browser, and the principles of the constructionist learning theory to create a motivating and efficient framework for learning programming [15].

In later stages of a *higher-order-first* educational approach, it could be possible for students to learn about the internal operation of the functional theorems by creating custom enumerable data structures. The implementation of the theorems on custom enumerators could be either in a functional or imperative style thus focusing on *how things work*. This approach could help students to learn about various programming paradigms and how to combine them.

As such *higher-order-first* approach would initially only use the Reduction, Mapping and Filtering theorems as shown in Section 2.2, some other theorems will be less efficient than some other implementations. One example would be the Partitioning theorem that is solved by applying the Filtering theorem twice in succession (thus iterating through the input two times), however on a lower level it can be solved by a single iteration over the input collection. In a top-down learning approach this lack of efficiency is not necessarily an issue, as the focus is on solving problems. Most of the times we do not require highly efficient programs and working with the occasional sub-optimal theorem implementation is perfectly fine. Also, some programming languages provide many built-in higher-order functions that solve more theorems efficiently similarly to how the `reduce`, `map` and `filter` functions solve the Reduction, Mapping and Filtering theorems. In later stages of the learning process the efficiency aspect can be covered in more detail as well, and more effective solutions can be implemented with imperative or low-level functional programming.

5. Conclusions

Programming theorems and enumerators form a solid foundation for classical *algorithm-first*, *object-oriented-second* programming curriculums that are very popular all over the globe. This approach emphasises understanding the low-level concepts of programming and how to use those concepts to build more and more complex algorithms and data structures to solve problems. It is possible to use functional programming to implement the same programming theorems and enumerators. This means that it is possible to create a *functional-first* programming curriculum that is analogous to this classic method as it uses the same programming theorems and enumerators as its foundation.

With functional-style programming and higher-order functions three programming theorems are enough to provide concise albeit form an efficiency perspective sometimes sub-optimal solutions to the rest of the theorems. Practically every programming language that supports functional programming have these three programming theorems available as higher-order functions out of the box. Based on these functions it could be possible to create a *higher-order-first* curriculum for teaching programming. This approach would facilitate a problem-solving centred learning process and would focus less on the low-level inner workings of programming theorems in the early stages of learning programming. In later stages it could also be possible to work our ways towards implementing new enumerators and other data structures that give a deeper understanding of the underlying algorithms or functional constructs. The browser and web programming combined with a constructionist learning methodology could provide a suitable environment for learning activities using this *higher-order-first* approach, thus they could hold great potential as a platform for learning programming using this methodology.

Acknowledgement

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

Bibliography

1. Vujošević-Janičić, M., Tošić, D., *The Role of Programming Paradigms in the First Programming courses*, *Teaching of Mathematics*, vol. 11, no. 2, pp. 63–83, (2008).
2. Szlávi, P., Zsakó, L., *Methods of teaching programming*, *Teaching Mathematics and Computer Science*, vol. 1, no. 2, pp. 247–257, (2003), DOI: 10.5485/tmcs.2003.0023.
3. Szlávi, P., Zsakó, L., *Módszeres programozás: Programozási bevezető* (in Hungarian), in Mikrológia vol. 18. ELTE TTK Általános Számítástudományi Tanszék: Budapest, (1994).
4. Joosten, S., van den Berg, K., van der Hoeven, G., *Teaching functional programming to first-year students*, *Journal of Functional Programming*, vol. 3, no. 1, pp. 49–65, (1993), DOI: 10.1017/S0956796800000599.
5. Chakravarty, M. M. T., Keller, G., *The Risks and Benefits of Teaching Purely Functional Programming in First Year*, *Joural of Functional Programming*, (2004), DOI: 10.1017/S0956796803004805.
6. Szlávi, P., Zsakó, L., *Módszeres programozás* (in Hungarian). Műszaki Könyvkiadó: Budapest, (1986).
7. Szlávi, P., Zsakó, L., *Módszeres programozás: Programozási tételek* (in Hungarian). ELTE Informatikai Kar, (2008).
8. Szlávi, P., Temesvári, T., Zsakó, L., *Módszeres programozás: A programkészítés technológiája* (in Hungarian), in Mikrológia vol. 21. ELTE TTK Általános Számítástudományi Tanszék: Budapest, (1994).
9. Gregorics, T., *Programozás 1. kötet Tervezés* (in Hungarian). ELTE Eötvös Kiadó: Budapest, (2013).
10. Gregorics, T., Kovácsné Pusztai, K., Fekete, I., Veszprémi, A., *Programming Theorems and Their Applications*, *Teaching Mathematics and Computer Science*, pp. 213–241, (2019), DOI: 10.5485/TMCS.2019.0466.
11. Szlávi, P., Zsakó, L., Törley, G., *Programming Theorems Have the Same Origin*, *Central-European Journal of New Technologies in Research, Education and Practice*, vol. 1, no. 1, pp. 1–12, (2019), DOI: 10.36427/cejntrep.1.1.380.
12. Kruglyk, V., Lvov, M., *Choosing the First Educational Programming Language*, in *CEUR Workshop Proceedings*, (2012), vol. 848, pp. 188–198.
13. Van Roy, P., Haridi, S., *Teaching Programming Broadly and Deeply: The Kernel Language Approach*, in *IFIP Advances in Information and Communication Technology*, (2003), vol. 117, pp. 53–62, DOI: 10.1007/978-0-387-35619-8_6.

14. Horváth, G., Menyhárt, L., *Teaching introductory programming with JavaScript in higher education*, in *Proceedings of the 9th International Conference on Applied Informatics*, (2015), pp. 339–350, DOI: 10.14794/icai.9.2014.1.339.
15. Visnovitz, M., Horváth, G., *A Constructionist Approach to Learn Coding with Programming Canvases in the Web Browser*, *CONSTRUCTIONISM 2020*, pp. 1–8, (2020).
16. Gregorics, T., *Force of Summation*, *Teaching Mathematics and Computer Science*, pp. 185–199, (2014), DOI: 10.5485/TMCS.2014.0365.
17. Gregorics, T., *Programming Theorems on Enumerator*, *Teaching Mathematics and Computer Science*, pp. 89–108, (2010).
18. Gregorics, T., *Programozás 2. kötet Megvalósítás* (in Hungarian). ELTE Eötvös Kiadó: Budapest, (2013).

Authors

VISNOVITZ Márton

Eötvös Loránd University, Budapest, Hungary
3in Research Group, Martonvásár, Hungary
e-mail: visnovitz.marton@inf.elte.hu

About this document

Published in:

CENTRAL-EUROPEAN JOURNAL OF NEW TECHNOLOGIES IN RESEARCH, EDUCATION AND PRACTICE

Volume 2, Number 2. 2020

ISSN: 2676-9425 (online)

DOI:

10.36427/CEJNTREP.2.2.965

License