# Composita 2.0: A Modern Reinterpretation of the Language

Hansruedi Patzen

Master's Thesis
December 24, 2020

University of Applied Sciences Rapperswil
8640 Rapperswil (SG), Switzerland

## Abstract

*Composita is a highly concurrent system built around the concept of self-contained components interacting with each other via channels through interfaces. This thesis ports the original Composita language implementation to a serverless web-application by recreating the compiler and runtime system in TypeScript and making it publicly available on the internet. This allows people interested in the language to play with, and quickly get a feeling for the language using any device with a modern browser installed.*

*The implemented compiler consists of a lexer, parser, checker, and code generator. The resulting intermediate language representation is then consumed by the runtime that handles code interpretation, component communication, and the system's concurrency. Communication handling and concurrency posed the biggest architectural challenges, due to the single threaded JavaScript environment.*

*Language modernization has been explored but is yet to be implemented. Modernization demands more than just a reimplementation hosted in a modern language, but also changes to the language itself. This thesis proposes several ideas to improve the syntax and semantics of the language, to make it more appealing to an audience already familiar with more modern languages like TypeScript, Kotlin or Swift.*

## 1. Introduction

Composita [1] as a system is built around the idea of having components communicate with each other via channels established through the defined interfaces. The majority of the original work was done during Dr. Luc Bläser's doctorate. In the scope of his thesis not only the language had been designed and implemented but also a runtime and a kernel had been written to support it. This implementation was done using the Oberon [2] programming language and was made available to the public by the means of a virtual machine and a bootable optical disc image. These results lay the foundation for this paper and explain the various concepts of Composita in depth.

The goal of this thesis is to take the original Composita programming language and make it available to a broader audience. This is achieved by two primary means. Firstly, the current dependence on a virtual machine or booting the image on a x86 system [3] is to be replaced with a new compiler and a runtime, that can be deployed as a serverless web-application [4], meaning nothing more than a modern web browser is required to write code in Composita. Secondly, changes to the Composita language are proposed, taking inspirations from currently popular programming languages like TypeScript [5], Kotlin [6] or Swift [7].

## 2. Revisiting Composita

Composita is a system mainly used as a research subject, seeing little development in the recent years. The core idea behind the system and the corresponding language with the same name was to get rid of pointers as a language feature [8, 9], most prominently seen in languages like C and its derivatives. This idea also applies to languages using references as a pointer replacement, for example Java [10].

The Composita language's main building blocks are interfaces and components. Components offer and require interfaces, the latter must be fulfilled before it becomes fully operational. Code listing 1 shows a com-

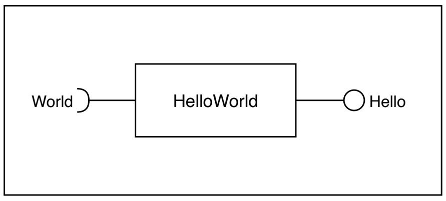


Figure 1: UML [11] notation of a component HelloWorld found in listing 1

ponent requiring the World interfaces to be connected and offers the Hello interfaces to other components.

```
COMPONENT HelloWorld OFFERS Hello
    REQUIRES World;
  IMPLEMENTATION Hello;
    (* ... *)
  END Hello;
END HelloWorld;
```

Listing 1: HelloWorld Component offers the Hello and requires the World interface

Figure 1 illustrates listing 1 using the UML [11] notation. Each offered interface must either be implemented directly using an IMPLEMENTATION block, or be redirected to an inner component. The offered and required interfaces further define the component's type, allowing for generic ANY type declarations. Listing 2 shows an example of a component containing a generic component member variable.

```
COMPONENT House;
  VARIABLE kitchen: ANY(Food |
      Electricity);
END House;
```

Listing 2: House component with a generic member variable

The kitchen variable can be instantiated by any component offering at least Food and requiring at most Electricity.

An INTERFACE uses EBNF notation [12] to define a sequence of messages exchanged between two components connected through the interface. The component offering the interface acts as the server and the requiring side as the client. Each message declares a direction in which it is going, IN messages go from the client to the server and OUT messages from the server to the client. The communication ends as soon as the interface specification has been fulfilled or a special finish message has been sent from either side. The Hello interface can be found in listing 3.

```
INTERFACE Hello;
  { IN GreetingRequest }
  OUT Greeting
END Hello;
```

Listing 3: Hello interface declaration

## 3. Implementation

As already mentioned in section 1, up until now the Composita system was available as a virtual machine or bootable optical disk image for x86 systems. Even with the convenience of a virtual machine image, the setup process is rather time consuming. By making the Composita compiler and runtime available as a serverless web-application, code can be compiled and executed on any device with a web browser and an active internet connection.

To fulfill this purpose, the existing compiler and runtime needed to be implemented in a language that can easily be integrated into a serverless website. This requirement only leaves a few feasible options: JavaScript [13] itself, any other language that can be transpiled into JavaScript, or WebAssembly (WASM) [14]. Due to these restrictions, its widespread use and support, and the static type safety guarantee, TypeScript has been chosen for the implementation.

To ease code integration, the compiler, runtime and Intermediate Language (IL) were split into node packages that are distributed through the Node Package Manager (npm) registry [15]. Each package contains the TypeScript code, type annotations, and two different transpilations. The first transpilation targets ECMAScript 5 [16] to support older browsers and Node.js [17] versions. The second one targets ESNext [18] supporting modern browsers and the newest releases of Node.js. While the type annotations allow for an easy integration into other TypeScript projects.

The Node.js packages @composita/compiler, @composita/runtime, @composita/il, and @composita/ts-utility-types have been published to the npm registry and can be installed using any node package manager, for example yarn [19] or the npm [20] tool.

### 3.1. Compiler

The compiler takes code in the form of a string and outputs its IL code representation, following the classic approach found in N. Wirth's Compiler Construction book [21].

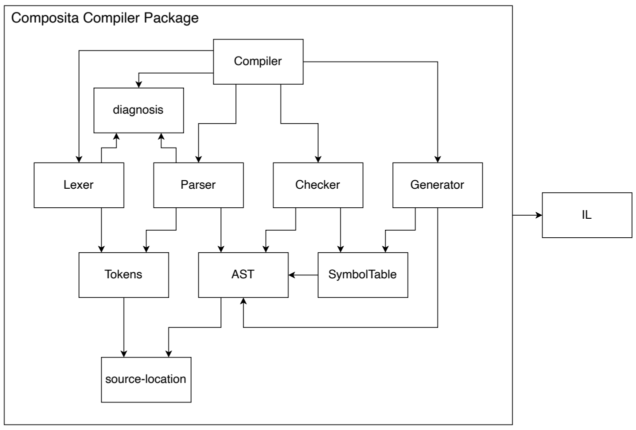


Figure 2: Compiler package overview, showing the main building blocks and its dependencies

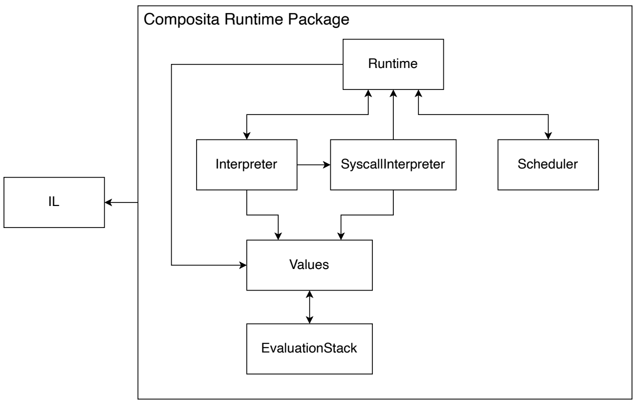


Figure 3: Runtime package overview, showing the main building blocks and its dependencies

The different compilation stages are split into lexer, parser, checker, and generator. Figure 2 provides an overview of the compiler package implementation, by showcasing the dependencies and outputs of each compile stage. The lexer consumes the input string and outputs an array of tokens. These are subsequently used as the input for the parser, which in turn outputs an Abstract Syntax Tree (AST), in the form of a program root node. Next is the checker, taking the AST and transforming it into a symbol table. During the last step the symbol table is passed to the code generator producing the IL representation.

## 3.2. Intermediate Language

The IL contains a list of all component and interface descriptors found during compilation. It further specifies a list of component descriptors considered to be valid entry points, as described in section 3.3.4. The different descriptors contain all the information required for them to be consumed by the runtime.

## 3.3. Runtime

Figure 3 presents an overview of the runtime, made out of a scheduler for the different active components and services, an instruction interpreter and an interpreter for system operation codes. The runtime provides a means for the program to communicate with the outside world. The communication is currently limited to taking the IL as its only input source and outputting text by calling a provided output function, taking a string as its argument. Furthermore, the runtime handles the creation and queuing of active components and services, as well as maintaining a mapping for each pointer to its value.

### 3.3.1. Scheduler

The implementation of the scheduler is based on the round-robin scheduling algorithm [22], whereby it will enqueue the next ready component or service after an instruction has been executed and, should it not already be completed, move the currently active one to the back of the queue.

### 3.3.2. Interpreter

There are two types of interpreters implemented. One for instructions, and one for system operation codes. The latter handles system provided proper and function procedures, like `WRITE(x)` and `SQRT(x)`, as well as type conversions, like `REAL(x)` or `TEXT(x)`. The instruction interpreter takes a pointer to a value as its input, where it will then try to fetch and interpret the next instruction from.

### 3.3.3. Values

The descriptors provided by the IL are used to create the different values managed by the runtime during execution. There are two main value types: active and built-in values.

An active value represents either a `COMPONENT`, an `IMPLEMENTATION` or a `PROCEDURE`. It contains a reference to its descriptor, where all the instructions and further declaration descriptions are stored. Access to the value is possible through its respective pointer value, which is stored in the runtime. An

IMPLEMENTATION is modeled as a service, that further holds a queue used for sending and receiving messages.

The Composita system provides the built-in INTEGER, REAL, CHARACTER, TEXT and BOOLEAN types. Their implementation contains a single TypeScript basic type field. Due to TypeScript's inability to differentiate between integer and floating point, as well as between character and string values, these basic types could not be used to represent the original Composita built-ins directly.

### 3.3.4. Program Entry Points

This implementation extends the original language definition in order to allow for an additional attribute. That attribute is used to mark a component as a possible program entry point, showcased by listing 4. With this attribute, a component is considered an entry point if it is marked as such and does not require any interface to be connected.

```
COMPONENT { ENTRYPOINT }
    HelloWorld;
  BEGIN
    WRITE("Hello World");
    WRITELINE
END HelloWorld;
```

Listing 4: Composita language extension allowing for components to marked as potential program entry points

It could be argued that this extension is not necessary as it certainly breaks backwards compatibility, and there are other approaches that would work just as well. For example, an initial implementation just assumed any component on a program level, that does not require any interfaces to be connected, to be an entry point.

One approach to solve this issue, could be to always take the first component complying to the required interface restriction. As an alternative, the list of potential entry points could also be provided to the user interface (UI), where the users then selects one or multiple entries. Each approach has its own set of advantages and disadvantages, that have to be considered. For example, with the approach of just taking all valid entry points, the user needs to keep track of what components are potential entry points themselves, which can easily lead to undesired behavior in the program.

Many languages define a single point of entry by denoting a special function usually called main as their program starting point. The C and C++ languages can be mentioned as a prominent examples for that. C++ defines the program start in its ISO standard under [basic.start.main] [23].

### 3.3.5. Creating New Values

Components are created by the means of the NEW function, in contrast to built-in types, that get default initialized during instantiation of their parent container. Services representing an implementation of a specific interface, do not have a dedicated NEW instruction. Instead they are created once the service implementation gets connected to a component.

### 3.3.6. Blocking Instructions

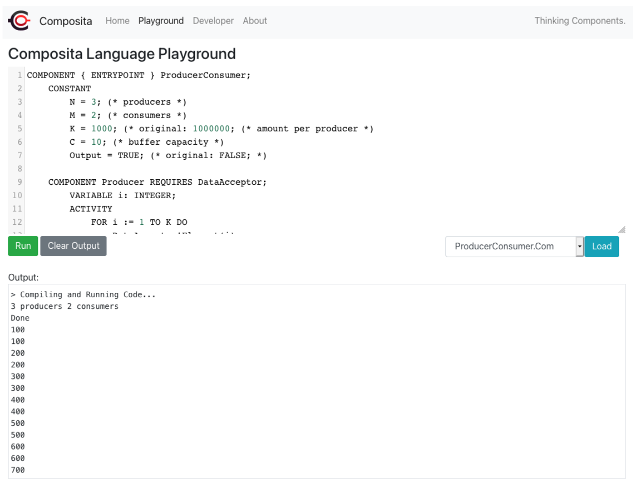
Encountering any blocking instruction like *receive*, *receive check* or an *exclusive monitor lock acquisition*, will cause the component or service to stop from advancing its instruction pointer, until the instruction has been successfully executed. If either the *receive* or *receive check* instruction are executed, the contents of the oldest queued message is checked against the expected message descriptor. The exclusive monitor lock requester has to wait until the lock is released, before being able to acquire it.

### 3.3.7. Concurrency

The underlying JavaScript architecture is single-threaded with an event queue. Into this queue synchronous and asynchronous events can be posted, however, each posted event has to be run to completion before the next one can be started. There is a way that allows to implement concurrency by using a WebWorker [24]. Such a worker keeps the Composita website reactive even if a playground program is blocking the runtime. The runtime uses only a single thread and simulates concurrency by keeping a queue of active values, and switching through them. Nevertheless, the implementation supports the AWAIT statement, as well as the EXCLUSIVE attribute. The await statement checks a boolean expression and releases a held exclusive monitor lock if the expression evaluates to false until it runs the check again. The SHARED monitor lock from the original language has yet to be implemented.

## 3.4. Website

The Composita language implementation resulting of this thesis is made available to the public at https://www.composita.dev/. The website is built using the React [25] web framework and depends on CodeMirror [26] for its editor capabilities. It is deployed using Github pages [27]. Figure 4 shows a screen capture of the Composita website playground.



```
COMPONENT { ENTRYPOINT } ProducerConsumer;
    CONSTANT
        N = 3; (* producers *)
        M = 2; (* consumers *)
        K = 1000; (* original: 1000000; (* amount per producer *)
        C = 10; (* buffer capacity *)
        Output = TRUE; (* original: FALSE; *)

    COMPONENT Producer REQUIRES DataAcceptor;
        VARIABLE i: INTEGER;
        ACTIVITY
            FOR i := 1 TO K DO
```

```
> Compiling and Running Code...
3 producers 2 consumers
Done
100
100
200
200
300
300
400
400
500
500
600
600
700
```

Figure 4: Screen capture of the Composita playground

### 3.5. Limitations

As mentioned earlier, there are some features missing or only partially available in this implementation's current form. The following list gives an overview of the currently known limitations:

- Interface protocol checks during program execution are not implemented
- Interface cardinality is ignored
- System provided interfaces are not implemented
- `DELETE` of components is not handled
- `CONNECT` interface redirection is not supported
- `DISCONNECT` is not handled
- `OFFERS` and `REQUIRES` expressions are ignored
- `MOVE` operation is not implemented
- `SHARED` monitor lock is not implemented
- `EXCLUSIVE` monitor lock implementation needs to be improved to support more complex use cases
- `PASSIVATE` is ignored
- Virtual time concept is not implemented

Other known issues not related to the language port:

- Error handling needs to be improved and compiler warning and errors should be enhanced
- Cancellation of a running program is not possible, due to the code running and not accepting external input
- Scheduler algorithm not always optimal and could be made selectable by the user

Both of these lists are not final and most probably will need to be extended once more users are actively using the website and its playground to learn about Composita.

## 4. Language Evolution

Alain Kay, one of the pioneers in object oriented programming, once said that “everything is an object” [28]. What if this concept could be applied to Composita by saying that “everything is a component”? Components and service implementations are the primary Composita constructs, yet they are not the only constructs. There are still procedures and function calls, used for either calling a procedure, or the system, like `WRITE` or `WRITELINE`.

The following paragraphs present some ideas that have come up during the implementation of the Composita language in its original design. Due to the limited time and resources the presented ideas have not made it past the idea stage and, therefore, no implementation showcasing the limitations and implementation difficulties exists.

### 4.1. Braces Syntax

The most visible language changes are updates to the syntax. Composita had been heavily inspired by the Oberon language. These changes would mostly effect the lexing and parsing stages of the compilation, but one could also continue with changes to the semantics of the Composita language to achieve further improvements.

Take for example the classic hello world program written in Composita, as shown in listing 5.

```
COMPONENT { ENTRYPOINT }
    HelloWorld;
  BEGIN
    WRITE("Hello World");
    WRITELINE
END HelloWorld;
```

Listing 5: The classic hello world program in Composita

By replacing the uppercase keywords with lower case ones, and using braces rather than an explicit END keyword for creating code blocks, the result will look something like in listing 6.

```
component HelloWorld {
  begin {
    write("Hello World");
    writeline
  }
}
```

Listing 6: The hello world program after transforming it to a syntax using braces

## 4.2. Components Only

At its core, Composita is built around the idea of having components, but it still supports other constructs like procedures, built-in operators, and record types defined in the interfaces. Unifying those has the potential of simplifying the language significantly. This would of course not come without downsides and many unanswered questions. One such question would be to define what happens whenever a component gets packed into a message and passed on to a service. Will it still continue to execute its instructions? Will a new component be instantiated? What happens to the existing connections? How are active service values depending on the component handled? Will the current instance be moved - making it inaccessible from the outside - or should a copy of the current state be created? Here, one could potentially look at other languages like C++, where copying and moving class instances are core concepts [29]. Inspired by that, the syntax could look something along the lines of what is shown in listing 7.

```
component Comp provides Copy {
  variable v: integer;
  implementation Copy {
    variable copy: Comp;
    begin {
      new (copy, Comp);
      !copy
    }
  }
}
```

Listing 7: One potential way a component providing a copy mechanism could look like

Most of these implementation blocks could potentially be default generated by the compiler in order to keep boilerplate code to a minimum.

## 4.3. Generics

Composita currently allows generic programming for components by using the ANY( (* OFFERED *) | (* REQUIRED *) ) syntax. Here one specifies the offered and required interfaces a component must satisfy in order to qualify as a valid substitution, whereby the pipe symbol | is used to separate the offered and required interfaces. Taking inspiration from C++ and its template language, one could extend the capabilities of ANY constructs, to increase the type safety of the language. The syntax in listing 8 could be an example for that.

```
component [T] TemplateComponent {
  variable temp: T;
}
component Main {
  variable templateInstance: [
      integer] TemplateComponent;
}
```

Listing 8: Extension of the current generic variable approach

One could go even further and allow disjunction types as generic constraints. The different types could be separated using a double pipe ||, similar to languages like TypeScript. This would look something like listing 9.

```
component [T: House || any(Food |
    Water)] TemplateComponent {
  variable foodStation: T;
}
component House {}
component Kitchen offers Food
    requires Water {}
component Main {
  variable templateInstanceA: [
      House] TemplateComponent;
  variable templateInstanceB: [
      Kitchen] TemplateComponent;
}
```

Listing 9: Component featuring a disjunction generic type constraints

Where the template parameter T must be either a House or any component that requires Water and provides Food.

The implementation is probably the most difficult part to this extension as changes to all compile stages are required. There are open questions as well, like how to handle generic variance, or what syntax to use.

### 4.4. Evolution Process

Before any changes can be implemented, a language evolution process should be defined. This process describes how changes to the language should be proposed and implemented if accepted. Every language has to choose one process or another if it is willing to evolve. Languages like C++ are governed by a committee [30], with the community designing the extensions to the language. Other approaches are, for example, Rust's RFC [31] or Swift's [32] language evolution process. Both allow for fast changes, making it easier for a language to keep up with with the surrounding technical evolution.

## 5. Related Work

Many modern languages feature an online playground, where they allow users to experiment and, as the word indicates, play with the respective languages. Notable members are TypeScript [33] and Kotlin [34]. There are also projects allowing multiple languages to be compiled and run online. An important example would be the Compiler Explorer [35], being most famous for its, but not limited to, C++ compiler support. Due to its seamless Python and math tooling integration, Jupyter [36] is heavily used in academia. Another category are the different programming-learning websites like CodeWars [37], where they use various forms of gamification to get users to program in a diverse set of programming languages.

## 6. Conclusion

The Composita language has for the most part been successfully ported and been made available to the public. The TypeScript implementation features strong type checking and a runtime to interpret the generated code. However, it is still incomplete and does not fully support all language features, as listed in section 3.5.

Performance was not a primary goal of this thesis, hence no further effort has been taken to improve on that topic. But one can assume that by the simple nature of being interpreted, the achieved performance can not be compared to what the original Oberon implementation can offer. If this should be taken into account, then one could think about porting parts to WASM as it is supported by most of the current browsers [38], and greatly outperforms JavaScript on computationally demanding tasks.

Having the language available as a website provides the means for everyone with access to the internet and a browser to play around and learn more about the language and its concepts. The proposed changes to the language syntax and semantics in section 4 can be used as an inspiration for future experimentation with the language and its design.

## References

[1] Luc Bläser. "A component language for pointer-free concurrent programming and its application to simulation". Doctoral Thesis. ETH Zurich, Nov. 2007. DOI: 10.3929/ethz-a-005539347. URL: https://www.research-collection.ethz.ch/handle/20.500.11850/72904 (visited on 03/18/2020).

[2] *Project Oberon.* URL: http://www.projectoberon.com/ (visited on 12/23/2020).

[3] "Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4". In: (2018), p. 4844.

[4] Faizan Bashir on. *What is Serverless Architecture?* URL: https://hackernoon.com/what-is-serverless-architecture-what-are-its-pros-and-cons-cc4b804022e9 (visited on 12/22/2020).

[5] *TypeScript Programming Language.* URL: https://www.typescriptlang.org/ (visited on 12/22/2020).

[6] *Kotlin Programming Language.* Kotlin. URL: https://kotlinlang.org/ (visited on 12/22/2020).

[7] *Swift Programming Language.* URL: https://developer.apple.com/swift/ (visited on 12/22/2020).

[8] Luc Bläser. "How can we liberate ourselves from pointers?" In: *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07.* Companion to the 22nd ACM SIGPLAN conference. Montreal, Quebec, Canada: ACM Press, Oct. 2007, p. 813. ISBN: 978-1-59593-865-7. DOI: 10.1145/1297846.1297901. URL: http://portal.acm.org/citation.cfm?doid=1297846.1297901 (visited on 03/17/2020).

[9] Luc Bläser. "A Component-Oriented Language for Pointer-Free Parallel Programming". In: (Oct. 2007), p. 6.

[10] *Java Programming Language.* URL: https://go.java/ (visited on 12/22/2020).

[11] *UML*. URL: https://www.uml.org/ (visited on 12/22/2020).

[12] International Organization for Standardization, ed. *ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF.* 1996.

[13] *JavaScript Programming Language*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript (visited on 12/22/2020).

[14] *WebAssembly*. URL: https://webassembly.org/ (visited on 12/24/2020).

[15] *npm — build amazing things*. Library Catalog: www.npmjs.com. URL: https://www.npmjs.com/ (visited on 07/11/2020).

[16] *ECMAScript Language Specification - ECMA-262 Edition 5.1*. URL: https://www.ecma-international.org/ecma-262/5.1/ (visited on 07/12/2020).

[17] Node.js. *Node.js*. Node.js. URL: https://nodejs.org/en/ (visited on 12/23/2020).

[18] *ECMAScript 2021 Language Specification - ECMA-262 Draft*. URL: https://tc39.es/ecma262/ (visited on 07/12/2020).

[19] *yarn*. URL: https://yarnpkg.com/ (visited on 12/23/2020).

[20] *npm*. URL: https://www.npmjs.com/ (visited on 12/23/2020).

[21] Niklaus Wirth. *Compiler construction*. International computer science series. Harlow, England ; Reading, Mass: Addison-Wesley Pub. Co, 1996. 176 pp. ISBN: 978-0-201-40353-4.

[22] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.00. Arpaci-Dusseau Books, Aug. 2018.

[23] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[24] *Using Web Workers*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (visited on 12/16/2020).

[25] *React – A JavaScript library for building user interfaces*. URL: https://reactjs.org/ (visited on 07/01/2020).

[26] *CodeMirror*. URL: https://codemirror.net/ (visited on 07/01/2020).

[27] *GitHub Pages*. GitHub Pages. Library Catalog: pages.github.com. URL: https://pages.github.com/ (visited on 07/11/2020).

[28] Alan C. Kay. "The Early History of Smalltalk". In: *The Second ACM SIGPLAN Conference on History of Programming Languages*. HOPL-II. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1993, pp. 69–95. ISBN: 0897915704. DOI: 10.1145/154766.155364. URL: https://doi.org/10.1145/154766.155364.

[29] JTC1/SC22/WG21. *ISO/IEC 14882 - Information technology — Programming languages — C++*. ISO, 2017, pp. 1–1605. ISBN: 0738140074. DOI: 10.1109/IEEESTD.2010.5733835.

[30] *Standardization : Standard C++*. URL: https://isocpp.org/std (visited on 07/24/2020).

[31] *rust-lang/rfcs*. original-date: 2014-03-07T21:29:00Z. July 24, 2020. URL: https://github.com/rust-lang/rfcs (visited on 07/24/2020).

[32] *apple/swift-evolution*. GitHub. URL: https://github.com/apple/swift-evolution (visited on 07/10/2020).

[33] *TypeScript Playground*. URL: https://www.typescriptlang.org/play/ (visited on 07/08/2020).

[34] *Kotlin Playground*. URL: https://play.kotlinlang.org/ (visited on 07/02/2020).

[35] Matt Godbolt. *Compiler Explorer*. URL: https://godbolt.org/ (visited on 07/02/2020).

[36] *Project Jupyter*. URL: https://www.jupyter.org (visited on 07/02/2020).

[37] *Codewars: Achieve mastery through challenge*. Codewars. URL: https://www.codewars.com (visited on 07/08/2020).

[38] *Can I use... WASM*. URL: https://caniuse.com/#feat=wasm (visited on 07/10/2020).