Connections between Language Semantics and the Query-based Compiler Architecture

Peter Lenkefi[a] and Gergely Mezei[b]
Department of Automation and Applied Informatics, Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, Műegyetem rkp. 3., H-1111 Budapest, Hungary

Keywords: Compilers, Query-based Compilers, Language Engineering, Memoization, Optimization.

Abstract: Modern software development has drastically changed the role of compilers with the introduction of responsive development tools. To accommodate this change, compilers have to go through an architectural transformation, diverging from the classic pipeline. A relatively new idea is called query-based compiler design, which took inspiration from build systems. It splits up the pipeline into smaller, individual operations, which - given some constraints - allows for some interesting optimizations. We argue that some programming language semantics introduce cyclic dependencies between certain compiler passes, which can naturally lead to rediscovering query-based compilers. In this paper, we present a framework that can be used to create compilers with a query-based architecture. Based on this framework, we introduce the Yoakke programming language, which we also use to explore our hypothesis regarding cyclic dependencies and rediscovering query-based compilers.

1 INTRODUCTION

Historically compilers have been designed as a simple pipeline that started with the source code and ended with the executable, treating the in-between steps as a strict sequence of operations (Aho et al., 2006), as illustrated in Figure 1. This was acceptable when the sole purpose of a compiler was to be ran in the command-line as part of a batch process, producing the output or a list of error messages. Since modern software development is a very different process, the requirements towards compilers have also changed.

Normally the developer works in an Integrated Development Environment (IDE), which aids coding in various ways by providing different services, for example:

- syntax (or semantic) highlighting
- error reporting
- auto completion
- refactoring operations (like symbolic renaming)

These services are usually implemented by a language analysis tool that runs in the background of the IDE, updating their output on each keystroke. Sometimes these services are implemented as a completely independent set of tools, which introduces a potential source of bugs. If one of these tools interprets the language specification incorrectly, they will have different interpretation of the code from the compiler, leading to inconsistent behavior. The severity of these inconsistencies can vary: a simple lexical misinterpretation can lead to the incorrect highlighting of tokens. A more severe case could be that the language tool reports an error but the code compiles correctly, or vice versa.

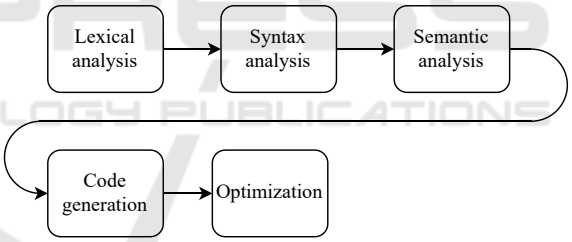

Figure 1: The classic compiler pipeline.

Compilers already implement and work with most operations and data structures that these language analysis tools require, such as the Abstract Syntax Tree - or AST for short - (Aho et al., 2006), the symbol table or type information. Ideally the compiler would expose its knowledge about the codebase, acting as a back-end service for the tooling, reducing

[a] https://orcid.org/0000-0002-9421-4151
[b] https://orcid.org/0000-0001-9464-7128

Lenkefi, P. and Mezei, G.
Connections between Language Semantics and the Query-based Compiler Architecture.
DOI: 10.5220/0011260400003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 167-174
ISBN: 978-989-758-588-3; ISSN: 2184-2833
Copyright © 2022 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

the potential source of errors. This is the idea the Roslyn [1] compiler went with, and it seems to work well in practice: most tooling in the .NET environment can be written as so-called analyzer packages, that use the Roslyn APIs. This approach however requires a different architecture to stay reasonably efficient and achieve the response times that are expected from these tools. A possible solution for this is the *query-based compiler architecture*, which splits up the pipeline into smaller, independent operations called *queries*. It is not too big of a divergence from the pipeline architecture, but will allow for optimizations that greatly helps responsiveness.

In Section 2 we discuss the idea behind query-based compilers and the optimizations they can offer. We also talk about languages with semantics that naturally require a different architecture, hinting towards the query-based approach. In Section 3 we describe the framework we have developed along with its main goals. While it is not the focus of our contribution, we present an experimental programming language used to showcase the capabilities of the framework. We believe that the main value in the programming language is the connection between its semantics and its architecture and how that connects to query-based compilers, so - due to it also being out-of-scope for this paper - we will not give a formal description of the language itself. In order to help better understand the framework, Section 4 contains an illustrative example.

2 BACKGROUND

Language analysis tools that work in the background of the IDE need to have reasonably fast response times. When the developer edits the code, the tool needs to immediately show the errors or suggest the best possible auto completion options for example. If we simply opened up the classic, pipeline-based architecture, exposing all functionality as-is, the resulting process would be highly inefficient. The reason behind this is that the entire process of code analysis would have to run for the entire codebase each time the user alters the code, which is potentially very expensive. This could cause delays that are usually unacceptable for such a tool. A new architecture is needed that is not only incremental in nature, but would also only do the minimal required computation for the given analysis, hinting towards a demand-driven system. This is where query-based compilers can aid us.

[1] https://github.com/dotnet/roslyn

2.1 Query-based Compilers

The idea of query-based compilers has gained some popularity recently as major compilers - most notably the rustc compiler (Klabnik and Nichols, 2019) - have started to turn towards this architecture. General incremental computation frameworks like Rock (Fredriksson, 2020) and Salsa (Matsakis, 2019) extracted the principle into libraries, helping the understanding and development of such systems. The basic architectural transformation is quite simple: instead of the pipeline elements that transform the input in large batches, we define smaller declarative operations that work on individual entities in the compiler. These smaller operations are called *queries* (Fredriksson, 2020) (Matsakis, 2019).

For example, symbol resolution might be implemented as a single pass on the AST in a classic compiler, but in a query-based compiler we would have various queries defined such as:

- attaching a declared symbol to an AST node
- retrieving the declared symbol of an AST node
- asking for all available symbols in a given context

While a compiler working with the classic pipeline architecture also has to implement these operations in some form, the key is that these queries should be implemented in such a way that they can be invoked without assuming that any previous passes have been executed. A query will always assume that no work has been done before and starts with invoking all other queries that are required to do its own work. Figure 2 shows a possible tree of queries invoked when type checking a simple C statement. The query starts from the operation we want to perform, and invokes all computation to reproduce results, up until the source code is requested, which is considered as a given input for the system. While this might seem inefficient at first, it is a very important step to make the compiler into a demand-driven system. In the next section we will explain how redundant computations can be eliminated from the system, solving this inefficiency concern.

Solving the inefficiency concerns, this could mean that the compiler and the IDE tools would be able to share the exact same code, reducing the complexity, codebase sizes and the possibilities for errors. Despite this promising design, there are still very few systems that have been developed to support this architecture and idea (Fredriksson, 2020) (Matsakis, 2019).

2.2 Memoization

Memoization is an optimization technique that caches the results of expensive computations and looks up

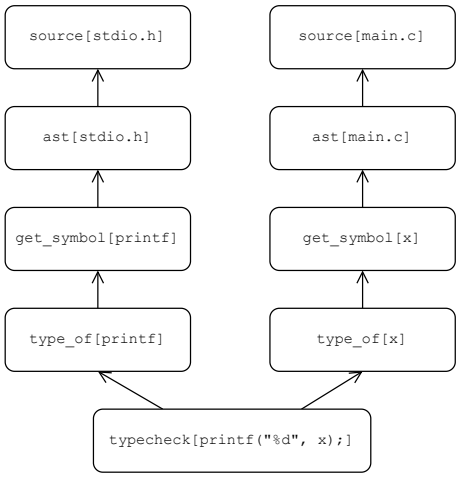

Figure 2: Portion of a possible computation-tree of queries.

these results the next time the computation is invoked (Michie, 1968) (Norvig, 1991). In the simplest form it is nothing more than a dictionary from the tuple of input arguments to the computed result. This can only be done for operations that are side-effect free, otherwise the results might be incorrect (Hughes, 1990). A small Python sample is provided below, demonstrating how one might memoize values by hand for computing Fibonacci-numbers.

```
cache = {}
def fib(n):
    if n <= 2:
        return 2
    if n not in cache:
        cache[n] = fib(n - 1) + fib(n - 2)
    return cache[n]
```

As long as the queries we define are side-effect free - which is usually desirable when developing a compiler -, we can safely memoize their results. This means that despite potentially calling certain queries multiple times that would do redundant computation, in reality we will not perform more work than a pass-based compiler. This is because once everything has been memoized once, the recall of that result can be considered insignificant compared to the computation.

2.3 Dependency Tracking

If query *A* calls query *B*, we can assume that the result of *A* depends on the result of *B* - given that the queries are side-effect free -, otherwise the call would be unnecessary. This allows for further optimizations.

If we can build a dependency graph for the queries, we can track what queries were affected by certain changes of the source code and only invalidate the memoized results where necessary. This can cut the required work even shorter for many queries, but only if the entire codebase has been processed previously. This is ideal for a tool working behind an IDE, where the user mostly makes localized changes, leaving most of the codebase largely unaffected. There will be an example showing off this kind of optimization in Section 4.

Interestingly, the problem of dependency tracking, namely, how it can be solved and optimized - and the query-based architecture as a whole - is very similar to problems build systems face: caching, versioning, invalidating results and tracking dependencies are usually associated with build systems (Mokhov et al., 2018).

2.4 Languages with Cyclic Compilation

While developing the framework, we have noticed that certain language semantics naturally lead to a query-based compiler architecture. Since we wanted to develop a compiler to study the architecture in more depth, we have decided to investigate languages with these semantic properties. The language that has mainly inspired us while creating our own was the JAI programming language [2], which syntactically unifies function calls and generic parameterization. A small sample of a type definition and instantiation in JAI can be found below.

```
List :: struct(T: type) {
    // ...
}

list := new List(i32);
```

From this small syntactic change, the Zig programming language (Zig Software Foundation, 2016) and our own programming language, Yoakke have both derived a very similar semantics, bringing types and values onto the same level. This will be described in more detail in Section 3.2.

3 CONTRIBUTION

Our contribution consists of two main parts: architectural research and language research. First, we wanted to see how we could extract the essential logic for query-based compilers into a reusable component. Our results are shown in Section 3.1, where we present the framework we have developed as a general tool for query-based compiler development. We

[2] https://github.com/BSVino/JaiPrimer/

also argue that certain programming language semantics naturally give rise to the query-based compiler architecture. In Section 3.2 we will discuss the details of such a language we have designed and how it naturally leads to the rediscovery to the query-based compiler architecture.

3.1 Query Framework

Developing a query-based compiler manually is a repetitive task and prone to errors. The logic of memoization and dependency tracking would have to be written manually, all intermixed with the actual compilation logic. To solve this concern, we have developed a framework [3] with the following goals:

1. The memoization logic is provided by the system, completely separated from the compilation logic.
2. The dependency tracking between queries is implicit and automatic, meaning that the framework should discover dependencies between queries without the user having to do that manually.
3. The query operations are cut short and terminated as soon as possible, when the framework can detect that the previous results are reusable. There will be a detailed example in Section 4.
4. Queries can be interrupted with a cancellation token, facilitating asynchronous use.
5. Old and irrelevant results can be periodically collected by a garbage collector.
6. The queries can be exposed as a public API in a way that is easy for language tools to consume.

The basic API describing how the framework can be used is very similar to the one Salsa presents but is written for C# instead. Queries are defined in groups, which carries no significant meaning other than that queries are related in functionality. There are two kinds of query groups:

- Input query groups: queries that provide the input for the other computations. These invoke no other queries. They are declared with the *InputQueryGroup* attribute.
- Computed query groups: queries that depend on input queries and other computed query results. They are declared with the *ComputedQueryGroup* attribute.

The declaration of query groups is done through a C# interface annotated with the proper attribute. The input queries are special in the sense that each query will have an appropriate setter generated in the interface definition. Since these interfaces and their implementations are very similar in their look and use to the usual service-pattern in the .NET world, we will also refer to them as *services*. A simple input- and computed query group declaration can be found in the snippet below.

```
[InputQueryGroup]
public partial interface ICompilerInputs
{
    public Manifest ProjectManifest();
    public string SourceText(string filename);
}

[ComputedQueryGroup]
public partial interface ISyntaxService
{
    public Sequence<Token> Lex(string filename);
    public SyntaxTree Parse(string filename);
}
```

Since *ICompilerInputs* is an input query group, the setters *SetProjectManifest* and *SetSourceText* are automatically generated in the interface definition.

Separating the interface and the implementation is very important from a usability standpoint. The framework internally generates a proxy serivce implementation for these interfaces that wrap the user-implemented service logic. This proxy is responsible for carrying out the memoization logic and dependency tracking, ensuring that the actual compilation logic and memoization logic are not mixed. The service interface and the implementation can then be registered into a dependency injection framework. When a service is requested through the interface, the user implementation of the service is wrapped up in the appropriate generated proxy, which is returned instead. This makes sure that only proxy implementations are handed out, which means that every operation will be tracked and memoized correctly.

3.1.1 Constraints within the Framework

A few constraints have to be satisfied while using the framework. These constraints are either harder to automatically validate or might introduce inefficiencies in the system:

- All queries must be side-effect free. Otherwise, memoizing the result leads to incorrect behavior.
- Services should not call other queries directly through the *this* instance. The proxy service should be injected and called instead, otherwise the call will not participate in memoization, since it is not made through the proxy.
- Queries must not be called conditionally. The dependencies of each query are discovered on the

[3] https://github.com/LanguageDev/Fresh/

first invocation only, to be more efficient. This means that if the condition changes, a query that is only invoked later might not be tracked properly as a dependency.

- Query parameters and return type must have value-based equality. The memoization logic assumes that these types can be stored in a standard C# *Dictionary*.
- The query return type should either be cloneable or immutable. Otherwise, the stored return value could be accidentally mutated from the outside, making it invalid for reuse.

Fortunately these rules do not introduce a lot of unreasonable constraints. We believe some of these are even desirable for compilers - like the side-effect free nature and immutability.

3.2 The Yoakke Programming Language

For experimentation purposes we have developed the Yoakke programming language [4]. The original goal of Yoakke was to reduce the number of required concepts a developer needs to know about, while still providing static type safety:

- Almost every language element can be used in compile-time computations, no need for separate mechanism and constraints.
- Types are values at compile-time, allowing generic structures to be simple compile-time executed functions.
- Every top-level construct is either a constant or a variable binding.

This leads to a simplification of language features in general. For example, there is no explicit need for supporting generics as they can be modeled simply by a function constructing a structure, parameterized by the generic types. This is showcased by the code snippet below, which defines a generic two-dimensional vector as a function, that constructs the vector structure from the given type parameter.

```
const Vec2 = function(T: type) -> type {
    return struct {
        x: T;
        y: T;
    };
};
```

The language allows arbitrary expressions where the syntax would expect a type, and those will be evaluated at compile-time to compute the specified type.

[4]https://github.com/LPeter1997/YoakkeLang

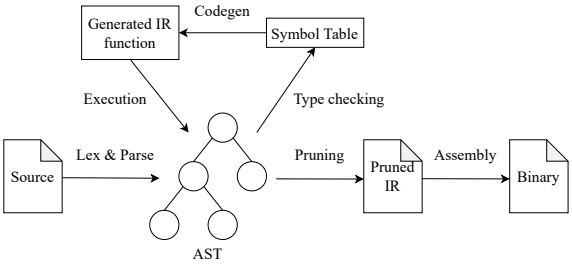

Figure 3: High-level architecture of the prototype compiler.

This means that the above function can be called in a types place, with a type supplied as a parameter, to create a two-dimensional vector type of any coordinate type.

3.2.1 Cycles in the Compilation

It is easy to see that a language with such unrestricted features cannot be checked and compiled with classic, pipeline-like compiler passes. Namely, we can introduce dependencies inside the code that require code-generation and execution of a function to be properly type-checked. To not give up type-safety, that executed code also has to be type-checked beforehand, which in turn can cause more code-generation and execution. This is demonstrated in the code snippet below, where the type-checking of the variable x in the *main* function requires evaluating the expression $get\_type(true)$, which requires type-checking the *get_type* function.

```
const get_type = function(value: bool) -> type {
    if (value) {
        return int;
    } else {
        return double;
    }
};

const main = function() {
    var x: get_type(true) = 0;
};
```

This cyclic nature of the compilation has caused us to separate the semantic checks and compilation into smaller, individual steps and group them into services that can depend on each other. This was essentially the rediscovery of the query-based compiler architecture. While the architecture can be used for a wide variety of language semantics, it is interesting to see that certain language semantics naturally lead to this architecture. Unsurprisingly, these semantics are very closely related to the problems that need to be solved by build systems. The high-level architecture of our compiler can be seen in Figure 3.

3.3 Tooling Integration

One of the most important aspects of this architecture is the ability to make tooling integration easier. We have implemented a Language Server [5] for Visual Studio Code. Since the extension protocol is request-response based, it has fit in well with the query-based compiler API. For example, on a text change notification, the input handling service is notified of the change, then the compiler is queried for all diagnostics - such as errors or warnings -, which are then presented to the user. A simplified version of the handling of the text change notification can be seen in the code below. Asynchronous elements are removed for the sake of compactness.

```
void TextChange(ChangeParams req)
{
    sourceRepo.Apply(req.Uri, req.Changes);
    var newText = sourceRepo.GetText(req.Uri);
    inputService.SetText(req.Uri, newText);
    var diags = diagService.AllDiagnostics();
    return PublishDiagnostics(diags);
}
```

The declarative nature of query-based APIs make it short and simple to write Language Servers and other tooling. Basic editor support - like error reporting - usually becomes a simple wrapper around the protocol the editor uses to call into one of the compiler queries.

4 ILLUSTRATIVE EXAMPLE

To present the framework in a more concise manner, we have prepared a small example to demonstrate its ease of use. Please note, that even in the relatively small language we have developed, even the smallest computations would span multiple pages, so we needed to find a smaller sample to demonstrate the usage of the framework. *MathService* calculates the nth Fibonacci number using the recursive method. The recursive method is considered highly ineffective because of the multiply-redundant computation it does, so it is an ideal candidate for memoization (Dijkstra, 1978). To make the computation represent a scenario closer to compilers, the system can accept the parameter as an input string, from which an integer is parsed. This attempts to bring the example a bit closer to the targeted use case, which is usually working with a textual input. The example uses two services, one for handling the input and one for the actual computation, the source of both can be found below. Note, that the reason for injecting the math service to its implementation is because this is the only way memoization can take place. This was further explained in Section 3.1.1.

```
[InputQueryGroup]
public partial interface IInputService
{
    public string Var(string name);
}

[ComputedQueryGroup]
public partial interface IMathService
{
    public int Fib(int n);
    public int ParseVar(string name);
    public int FibFromVar(string name);
}

public sealed class MathService : IMathService
{
    public readonly IInputService input;
    private readonly IMathService math;
    // Constructor omitted for brevity

    public int Fib(int n)
    {
        if (n < 2) return 1;
        return math.Fib(n - 1)
             + math.Fib(n - 2);
    };

    public int ParseVar(string name) =>
        int.Parse(input.Var(name));

    public int FibFromVar(string name)
    {
        var n = math.ParseVar(name);
        return math.Fib(n);
    }
}
```

There are two interesting optimizations the framework can do. To showcase the simpler one, first we will compute the 5th Fibonacci-number - using the input $n =$ "5", which can be seen on Figure 4. The redundant computations are picked up by the framework, making it essentially a linear operation. After changing the input to $n =$ "8" and invoking the computation again, the computation is even shorter, only computing 3 new numbers, recalling the rest from the cache, as can be seen on Figure 5.

A more interesting case of optimization can be observed when the dependencies of a query change, but the result stays unaffected. This is also picked up by the system, short-cutting the computation. This can be seen on Figure 6, after changing $n =$ "8␣", intentionally adding a space after the input. After the variable parsing query notices that its output is unchanged - despite the input change -, it immediately terminates the computation, as it is considered up to date. This is done by the framework, because all dependencies are

[5] https://microsoft.github.io/language-server-protocol/

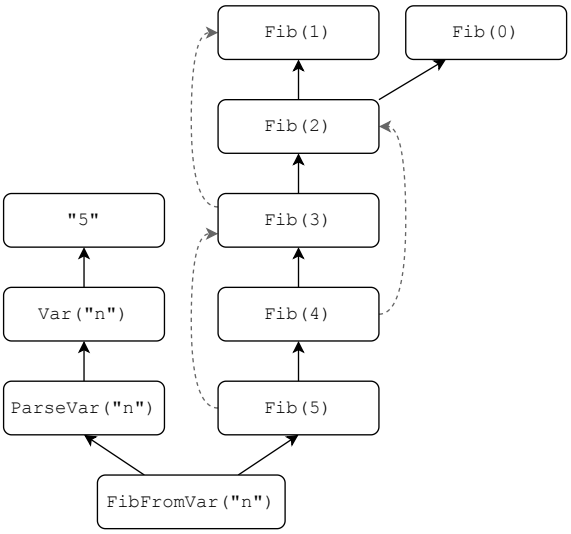

Figure 4: Calculating the 5th Fibonacci-number, assuming an empty system.

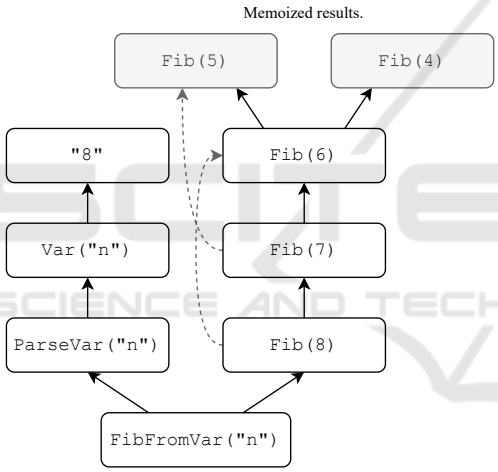

Figure 5: Calculating the 8th Fibonacci-number after the 5th one.

automatically discovered. Since the only dependency - the variable n - has been recomputed, but its value remained unchanged, the dependent is considered up to date.

5 CONCLUSION

Compiler development has had a long period where - besides special cases - no architectural improvements were done, and with a good reason. Compiler development is considered a difficult task, having a proven architecture gave a fulcrum during development. In this paper, we aimed to show that diverging from this architecture and utilizing queries can be relatively simple and does not require a completely different approach. Besides the simple change, it has the additional benefit that opening up the compiler for tooling integration becomes a lot easier. This is important in the modern era of software development where the tooling plays a lot bigger role than when the first compilers were written. We have also shown that this architectural shift is mandatory for languages with certain semantics, where compilation is not necessarily a strictly sequential operation anymore.

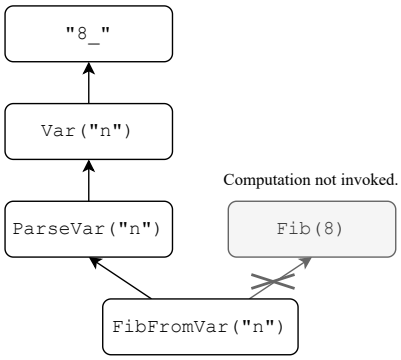

Figure 6: Calculating the 8th Fibonacci-number after adding a white space at the end of the variable value.

We have presented the framework we have developed to research and test this architecture, and talked about the constraints and advantages it introduced. We have also developed a language that by nature requires an architecture resembling query-based compilers, thus reinforcing the idea that this architecture is something that will naturally emerge for certain language semantics. The results of the research have shown us that this architecture is a viable and efficient way to open up compilers to tooling and IDEs in general. While the illustrative example in section 4 does not show usage from the compiler itself - as that would span many pages, even for trivial cases -, we hope that it motivates the idea behind the efficiency concerns mentioned in section 2.

Regarding future work, there are two paths we would like to explore. First, we plan to see if the query-based architecture could be extended to help decoupling non-primary information from the primary query results. Currently, all operations that might produce diagnostics return the list of produced diagnostics alongside the actual result of the queries, making the API a bit more noisy. We have been experimenting with information channels to publish this non-primary information, that would also be recorded and played back on requesting the results of a query. Second, we would like to further explore the discussed language semantics. Currently, the compiler can get stuck in an infinite loop from a compile-time

computation. While this cannot be resolved in an arbitrary case - as it would be the equivalent of solving the Halting-problem (Hopcroft et al., 2006) -, we aim to explore if there are any sensible boundaries we could introduce.

ACKNOWLEDGEMENTS

The work presented in this paper has been carried out in the frame of project no. 2019-1.1.1-PIACI-KFI-2019-00263, which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2019-1.1. funding scheme.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

Dijkstra, E. W. (1978). In honour of fibonacci. In *Program Construction, International Summer School*, page 49–50, Berlin, Heidelberg. Springer-Verlag.

Fredriksson, O. (2020). Query-based compiler architectures (https://ollef.github.io/blog/posts/query-based-compilers.html).

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

Hughes, J. (1990). *Why Functional Programming Matters*, page 17–42. Addison-Wesley Longman Publishing Co., Inc., USA.

Klabnik, S. and Nichols, C. (2019). *The Rust Programming Language (Covers Rust 2018)*. No Starch Press.

Matsakis, N. (2019). Responsive compilers.

Michie, D. (1968). "memo" functions and machine learning. *Nature*, 218:19–22.

Mokhov, A., Mitchell, N., and Peyton Jones, S. (2018). Build systems à la carte. *Proceedings of the ACM on Programming Languages*, 2(ICFP).

Norvig, P. (1991). Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98.

Zig Software Foundation (2016). Official website of the zig programming language (https://ziglang.org/).

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS