BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

AUTOMATED TRANSPILER GENERATION

Supervisor

Lect. Dr. Petrașcu Vladiela

Duma Horațiu

**2022**

# Abstract

The primary field of study addressed and discussed within this study is the field of Metaprogramming. Though it is one of the more well known specific areas of study under Computer Science, it is touched upon by most in a very shallow manner. Due to the fact that the field of Metaprogramming’s most significant realizations are tools used by other computer specialist, it is regarded as common knowledge to be aware and be capable of using these tools. Concretely, we are referring to tools such as compilers, interpreters and code generators. Understanding a programming language is considered to be synonymous to understanding the compiler or interpreter which conceives it. However, as modern tooling has advanced, abstractions are more and more commonplace, and the inner workings of these tools, already known by few, are obfuscated further.

As such, part of the purpose of this paper is to iterate upon the concept of metaprogramming, demystify its principles and expose them in such a manner that can be easily understood by a significant part of computer engineers. Furthermore, we elect a specific principle called General Purpose Metaprogramming and demonstrate the train of though implied by applying the principle.

As a final step in showcasing this process, we will depict and describe a pragmatic example, a concrete application designed through the application of that principle, as within the field of computer science practicality of solutions is one of the most important factors. The application showcased is named LangBuilder, coming from (Programming) Language Builder, named accordingly with its purpose and solution domain.

Table of Contents

[Abstract 1](#_Toc105682158)

[Chapter 1: Introduction 4](#_Toc105682159)

[1.1: Motivation and Purpose 4](#_Toc105682160)

[1.2: Structure 5](#_Toc105682161)

[1.3 Personal Contributions 6](#_Toc105682162)

[Chapter 2: Metaprogramming 7](#_Toc105682163)

[2.1: Metaprogramming in the large 7](#_Toc105682164)

[2.2: Code Generation 8](#_Toc105682165)

[2.3: Metacompilation 9](#_Toc105682166)

[Chapter 3: LangBuilder Application 11](#_Toc105682167)

[3.1: Requirements Elicitation 11](#_Toc105682168)

[3.1.1: Introduction 12](#_Toc105682169)

[3.1.2: Overall Description 13](#_Toc105682170)

[3.1.3: Functional Requirements 14](#_Toc105682171)

[3.1.4: Interface Requirements 15](#_Toc105682172)

[3.1.5: Nonfunctional Requirements 15](#_Toc105682173)

[3.2: Analysis and Design 16](#_Toc105682174)

[3.2.1 Analysis 16](#_Toc105682175)

[3.2.2 Design 17](#_Toc105682176)

[3.3: Implementation Details 21](#_Toc105682177)

[3.3.1: Rule List Processing 21](#_Toc105682178)

[3.3.2: Grammar File Generation 24](#_Toc105682179)

[3.3.3: Transpiler Executable Generation 25](#_Toc105682180)

[3.4: Testing 26](#_Toc105682181)

[3.5: User Guide 29](#_Toc105682182)

[3.5.1: User Interface 29](#_Toc105682183)

[3.5.2: API Interface 36](#_Toc105682184)

[Chapter 4: Conclusions, Further Work 37](#_Toc105682185)

[4.1: Conclusions 37](#_Toc105682186)

[4.2: Further Work 38](#_Toc105682187)

[References 39](#_Toc105682188)

# Chapter 1: Introduction

## 1.1: Motivation and Purpose

The field of Metaprogramming within the broader subject of Software Development is, at the same time, one of the most esoteric and one of the most integral areas of knowledge to Computer Science. Every single modern software developer, and all developers to come, make use of products of that field. Tools which have become synonymous with the act of coding, such as compilers, interpreters, transpilers [1] and even Integrated Development Environments (IDEs), all make use of the concepts of Metaprogramming. These advancements are the chisel and hammer with which the modern software programming world has been built, and continues being built on top of with.

However, despite the nigh-omnipresence of such tools, the average software developer is oblivious to the inner workings of such tools. Their use is an inherent requirement, their subject is the source code itself and their product is the software. Their capabilities are well documented, yet the means through which they realize them are disregarded. To most developers, the compiler (or interpreter) is a black box. It is fed the source code and it produces a runnable application, nothing more.

Among a select few, are these concepts such as compiler and interpreter, dissected to reveal their inner workings. Within the software development world there are groups working on these specific technical challenges, which power millions of other developers. They are most regularly known as ‘Maintainers’, a common term used to describe a developer working on an ‘already finished’ product. In this particular case, the product is a software technology or a programming language in of itself. Some of them independent, some of them part of bigger entities which rely on the technology they support. These maintainers take up the task of improving and repairing these technologies, with the help of the communities which use and rely on them.

However, despite the omnipotent reliance on what were once breakthroughs, the compiler and the interpreter, the field of metaprogramming has seen very few advancements, and nothing of note compared to the invention of the compiler which has revolutionized programming at its time. Instead, what metaprogramming has been affixed on is improving and adding to the existing concepts, alongside some minor newly discovered concepts, such as code analysis tools and source generators [2].

The motivation behind this paper is to propose a completely novel concept, with the potential to change mainstream programming practices in a similar way that the compiler did at it’s time, by applying 3rd degree metaprogramming, or metacompilation.

## 1.2: Structure

This paper is structured into 4 chapters. The former two chapters offer a broader view of the field which this paper pertains to, in order to shape the context of the paper’s proposition. The latter two chapters serve to approach and demystify the practical implementation of the proposition, as well as describe the technical difficulties, and as a consequence, the limitations of the implementation. The contents of each chapter summarily described:

Chapter 1: Introduction, this chapter serves to describe the motivation behind the paper, its structure and most notable personal contributions.

Chapter 2: Metaprogramming, takes up the effort of summarily describing the field of Metaprogramming for readers who find it foreign as well as offer relevant examples of metaprogramming.

Chapter 3: LangBuilder Application represents the bulk of the paper, it portrays explicitly the centermost idea contained within the paper, continues with displaying the pragmatic analysis and design of the proposal, formulated with the foresight of a concrete production in mind, whilst respecting the software engineering principles, and finalizes with the presentation of the specific construction accompanying this paper .

Chapter 4: The final chapter outlines the limitations of the current system and the extension points for further possible work to the proposal and the sample.

## 1.3 Personal Contributions

The current paper contains many personal contributions, as the metaprogramming field is seldom tackled by the mainstream software development community, even more scarcely by the computer science academic circles.

By far the greatest personal contribution is the main proposal of the paper - a completely innovative look at metaprogramming – metaprogramming, about metaprogramming, referred to within the paper as 3rd degree metaprogramming. It is a bewildering subject to touch upon in of itself. This concept represents the intellectual contribution of the paper.

Furthermore, accompanying the proposal, the paper brings a tangible implementation of the proposal – a software application which puts into practice the principles of 3rd degree metaprogramming, customarily named the ‘LangBuilder’, meaning Language Builder, interpreted as Programming Language Builder, an application that constructs new applications, which act upon other applications. This instance represents the engineering contribution of the paper.

The previously mentioned contributions and the effort devoted to discover, develop, shape, conceive, and finally present them, are fully owned by the author of this paper, with the corresponding guidance and groundwork conferred by other papers in the field, touching on related subjects, and their authors, as well as community-driven projects, technologies and tools utilized by the implementation, all having been cited as references for this work.

# Chapter 2: Metaprogramming

## 2.1: Metaprogramming in the large

Metaprogramming is the act of writing computer software whose data it manipulates, is the computer software. Metaprogramming came about as an effect of the desire to ‘generify’ already written code. Through the process of extracting reusability, there was reached a point where the code writing process itself became the target domain of a computer program.

The term ‘Metaprogramming’ is directly interpreted as ‘programming about programming’ and is used to refer to the paradigm of programs that write programs. However, the term has been popularized by the practice of ‘Template Metaprogramming’ native to the C++ programming language. Template Metaprogramming (TMP) is a compiling technique through which temporary source code is generated, based on ‘templated’ source code. The templates are evaluated at compile them and replaced with the values, resulting in new source code, which is then used instead of the templated source code. This is an example of General Purpose Program Transformation metaprogramming.

The metaprogramming paradigm is usually applied in three distinct ways:

* Exposing information about the run-time to the source code to be executed. As such, the program being executed, can know about the way it is being executed. A common use case is providing the code information about the operating system environment it is being executed by.
* Dynamically executing code, most commonly expressions or commands. As such, the program being executed can create and/or execute itself, or rather, a smaller subset of itself. A pertinent use case for this is dynamic query execution, allowing to parametrize otherwise rigid SQL syntax.
* General Purpose Program Transformation. As such, the program being executed analyzes and/or creates a new program. An example of this is represented by the compiler or interpreter of any general purpose programming language. The significance of this use case is detailed by the simply the widespread usage of compilers and interpreters.

Thus, metaprogramming is a complex, standalone field, unable to be wholly described in a single paper. The subdivision of metaprogramming which is tackled by this paper is part of General Purpose Program Transformation, it is the technique of code generation

## 2.2: Code Generation

Within this paper, when we refer to Code Generation, we refer specifically to ‘Source-code Generation’, which is the process through which a computer program is used to generate source code. A peer term which also details this concept is ‘Automatic programming’.

Code Generation, within the specific context of compilers is the process of converting a programming language’s intermediate representation into a readily executable format (e.g. machine code). Within this paper we will not detail this specific definition, as it already a well documented practice [3].

Source Code Generation is most commonly used through the medium of specific tooling developed with the purpose of fastening the development process. This is frequently done by automating common simple operations. The features of these tools span from generating simple methods, to generating entire multi-layered systems, an example for the former is automatic generation of getters and setters, a feature found in all IDEs developed by JetBrains [4], an example for the latter is Ruby on Rails’ Scaffolding system integrated in the CLI.

By far the most popular example of Code Generation is C++’s Template Metaprogramming, whose templates represent a compile-time Turing-Complete system, allowing complex calculations, such as defining within the source code the values of the first one thousand numbers in the Fibonacci sequence.

A lesser-known example is Microsoft’s Text Template Transformation Toolkit (T4) [5], which allows both design time and runtime arbitrary text generation. It is the spiritual predecessor of Razor Templates and Razor Pages which are most commonly used in ASP.NET Web Development [6].

The most complex use case of Code Generation however is represented by the concept of creating Domain-Specific Languages (DSLs). DSLs are computer languages designed to cater to a specific domain. DSLs have a wide range of usages and examples number HTML, Matlab and SQL. DSL Code Generation is a process synonymous to creating a General Purpose Programming Language (GPPL), the main point of discrepancy between the two however, is their use case. DSLs have a much wider range of use cases, and as such are designed for extremely specific or niche use cases. It is also common that DSLs are intended to be human-readable, whilst at the same time being machine-interpretable. Whereas GPPLs serve to abstract programming-specific concepts such as memory allocation or operating system interaction, DSLs are used to abstract computer interpretation of specific concepts. As such, they are the most complex example of code generation, the process of making a program understand that which it cannot understand, by translating it into something that it can. An example of this in practice is JetBrains’ MetaProgramming System (MPS) [4]. It is essentially a tool which customizes the process of building a DSL.

## 2.3: Metacompilation

One step above the process of Code Generation, is the process of generating the code that generates the code, through the usage of a Metacompiler. A Metacompiler, or compiler-compiler [7] is a tool used to generate the tools used by a programming language, such as a code generator, parser or even compiler. The purpose of this paper specifically is to explain and demystify the practice of Compiler Generation, as well as present an original contribution within this domain – a transpiler generator.

In their most simple form, metacompiler receive as input a program developed in a specializing programming metalanguage, created with the intent of constructing programming language tooling.

The earliest metacompilers, META I and META II developed by D. Val Schorre are functional programming languages used to describe rudimentary input parsing expressions. At the time of writing, the most widespread metacompiler is the ANTLR parser generator, originally developed by Terence Parr [8]. It is a tool integrated with most other widespread GPPLs and makes use of Grammar files to generate a parser which will output an interpretable syntax tree [9]. Other parser generator examples include Yacc and GNU Bison.

An important aspect of note, is that the tools noted above, are intended for design-time tool generation. They were developed and intended to be used within the development process itself, for specific use cases. Most specifically they are used for static generation use cases.

At the time of writing of this paper, there is no widespread, documented example of a metacompiler metacompiler, a program which compiles metacompilers. As such, the subject of this paper is an example of such a program, intended to, at run-time generate a programming language compiler.

# Chapter 3: LangBuilder Application

## 3.1: Requirements Elicitation

The centermost proposal of this paper is to apply the principles of 3rd degree metaprogramming, the concept represents the idea to apply metaprogramming principles, to a metaprogramming concept. However, in order for this proposal to provide value from an engineering standpoint, it must also lead to the production of an application with an appreciable practical appliance [10].

For the specific implementation of the theoretical notion of the paper, the metaprogramming concept reflected upon, is the process of compilation of source code, with its brother concept of interpretation. The metaprogramming tool in question is the compiler (or interpreter) and the information it acts upon is the source code. To this process, we apply the 3rd metaprogramming concept – General Purpose Transformation, more specifically, we will apply general purpose transformation to the source code itself. However, source code generation and transformation in of itself is not a novel concept. Furthermore, we once again apply General Purpose Transformation, to the mechanism of source code generation.

The resulting concept, is to programmatically generate an application, that programmatically manipulates and/or generates source code. This is the bare concept of the application [11].

The specific requirements and details of the application are outlined as follows, in a Software Requirements Specification (SRS) format:

### 3.1.1: Introduction

* 1. Purpose

The purpose of the application is to provide a concrete implementation for the bare concept, in a reasonably detailed and pragmatic manner and containing enough complexity to prove practical applicability of the concept.

* 1. Document Conventions

This document uses the following conventions – see Table 1:

|  |  |
| --- | --- |
| LangBuilder | The name of the application |
| OS | Operating System |
| FP | Functional Programming |
| UI | User Interface |
| HTTP | Hyper-Text Transfer Protocol |
| ANTLR | Another Tool for Language Recognition, version 4 - https://www.antlr.org/ |
| .NET | Microsoft’s .NET Framework - https://dotnet.microsoft.com/en-us/ |
| ASP.NET | .NET Subset oriented for web development |
| React | JavaScript library - https://reactjs.org/ |
| Docker | Containerization tool - https://www.docker.com/ |

Table 1 - Conventions

* 1. Project Scope

This application project represents a Proof of Concept (POC) type of application, and as such, it’s feature scope is restrained to only implementing the bare minimum of the concept it serves to prove. It is both an implementation as well as a showcasing application for the concept, and is designed with these goals in mind.

### 3.1.2: Overall Description

* 1. Intended Use

The application is intended as a programmatically consumable API, invoked for the generation of the byproduct applications, as well as subsequently re-invoked for the purpose of refinement and fine-tuning of the applications.

* 1. Intended Audience

The Intended Audience for the application is Software Developers, Engineers, Architects and any technical person interacting, working on and modifying source code, written in any programming language.

* 1. Definitions

The LangBuilder application makes use of the following concepts

* **Transpiler**

An executable application programmatically generated by the application. It represents a console application, used to parse a source file, and producing an output file.

* **Transpilation Rule**

Is an arbitrarily formulated rule, containing an input format, and output format, which may be based on the input. They are the fundamental component of the Transpiler. Transpilation Rules are of several types.

* **Transpiler RuleSet**

Is the collection of rules used by a particular transpiler

### 3.1.3: Functional Requirements

* 1. Product Features

The main feature of the application is the mechanism of receiving a Transpiler RuleSet and producing a Transpiler that uses it.

The main feature of the Transpiler applications is to receive a source code file as input and transform it, producing a source code output file.

* 1. User Class and Characteristics

Users of the application are able to create their own Transpilation Rules, and then submit them as a Transpiler RuleSet in order to generate a Transpiler.

The Transpilation Rules must be personalizable and configurable.

The Transpiler must be executable in any environment and must produce the expected output according to the rules input.

* 1. Use Cases

The application’s use cases are to alleviate any potential pain-points in currently existing compilation/interpretation processes through augmentation of source code as an additional step before this. Some particular examples are listed below:

* + - Redefining language-imposed syntax to a more intuitive representation – e.g. obligatory Task return type of asynchronous methods in C#
    - Simulating inexistant language concepts – e.g Operator Overloading in Java
    - Shortening boilerplate syntax – e.g main function conventional signature in Java or C#

### 3.1.4: Interface Requirements

* 1. The LangBuilder application exposes a HTTP API with a single endpoint. The application receives its input in the body of the HTTP request sent to the endpoint, in JSON format, and delivers the output as a downloadable zip file containing the required files for running the Transpiler application.

### 3.1.5: Nonfunctional Requirements

* 1. Assumptions and Dependencies

The LangBuilder application specifications are listed below

* Functional System
* Persistence Agnostic
* OS Agnostic
* Application Platform: ASP.NET, .NET 6, C#
* Other Requirements: ANTLR with C# target, Java for running ANTLR

Additionally, the LangBuilder application contains an additional clientside application with the following specifications:

* Persistence Achieved using Browser Local Storage and/or FileSystem
* Browser Environment: Google Chrome
* Application Platform: React, Typescript
* Other Requirements: HTTP access to main LangBuilder application
  1. Design and Implementation Constraints

The currently encountered constraints are the following:

* + - The produced Transpiler application represents a .NET Console Application, and as such any system it runs on must have installed a .NET 5+ SDK corresponding to the application’s version, and must be executed using the .NET CLI.
    - The produced Transpiler application requires a JSON config file to be run.
    - The produced Transpiler may requires DLL files containing dependent assemblies, at the location it is execute from.
  1. Performance Considerations

In order for the application to maintain practical applicability, the execution time of a Transpiler application must be of length equatable, or lower than the compilation time. An execution time an order of magnitude higher will severely hinder practicability of the system.

## 3.2: Analysis and Design

### 3.2.1 Analysis

To concisely restate and detail the problem domain, the application’s purpose is to showcase 3rd degree metaprogramming, by implementing a system which facilitates and/or enables configuration and customization of Source Code Generation. Additionally, the mechanism through which this is done must be practical enough to cover most common use cases of current source code generation mechanisms, and fulfil additional use cases, unable to be covered by existing implementations.

The domain of the problem is quite abstract, but the general scope of it is that the result must be a system, capable of, in a configurable way, generating source code. The input of the system is a set of arbitrarily chosen parameters, specific to the implementation, and the output is represented by text representing source code.

The end user of the system is a technically capable person, and as such we assume he is capable of understanding common software concepts and formats, e.g Regex, JSON, Yaml. The main user need we are trying to solve is to be able to programmatically resolve code duplication, or code syntax restrictions imposed by current mainstream programming languages.

The constraints of the system are, due to the nature of the design being innovation itself, time and resource constraints. As such, the current project’s scope is restricted to being merely a POC application, resolving a very small number of features, yet demonstrative enough to show potential of extension.

### 3.2.2 Design

In this section we will specify elements of the design around the application, as well as a proposed architectural design for the implementation of the application. This section is intended to only to aid in understanding the problem using diagrams and compositions serving to simplify the details of the problem. Further details about the specific design of the fully implemented application will be provided in a subsequent subchapter.

The most important aspect of the application is the data itself, data is a collection of names, attributes or elements representing information. The most primordial step in shaping the design is to model the data flow.

For the specific instance of the arrangement of this paper, we have greatly reduced the input and the output prospects for simplicity’s sake, due to the fact that the underlying context of the issue allows it. A visualization of the data flow is provided below – see Figure 1.

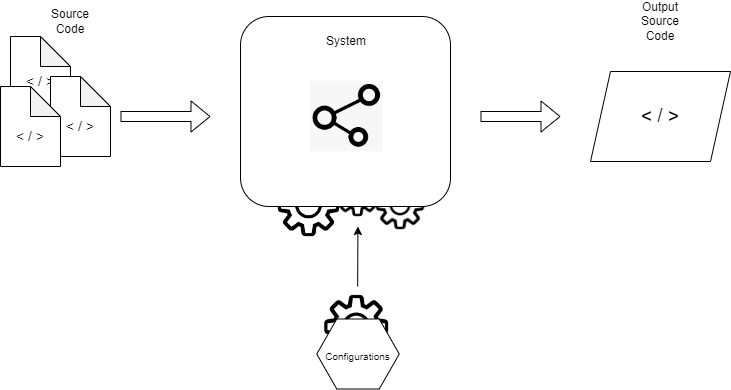


Figure 1 - Data Flow Diagram

To the diagram above we provide the following clarifications:

* The Input is made out of the ‘Source Code’ element as well as the ‘Configurations’ element
* The Output is solely represented by the ‘Output Source Code’

As well as mentioning the following assumptions, made as well for simplicity

* We will concede that the ‘Source Code’ be represented by the simplest format possible – simple string. Additionally, we will ignore all whitespace and any other insignificant characters, as well as ignore formatting rules as these aspects are non-essential to transferring of information in most mainstream computer programming languages.
* The ‘Output Source Code’ will respect the same constraints as the `Source Code`
* The inner design of the system is purposefully abstracted and will be detailed subsequently
* The `Configurations` element is considered to be simple data, input to the system, in a distinct way to the `Source Code` and will directly dictate how the Output is generated. It is also abstracted as it is tightly linked with the implementation, different from the Source Code.

The proposed design of the inner system has been carefully constructed taking into account the time limitations imposed on the project, it is by no means a definitive way of implementing such a system. It does however, represent a very pragmatic and relatively simplist way of realizing the requirements.

A diagram showcasing the proposal can be seen below, followed by further clarifications below it – see Figure 2.

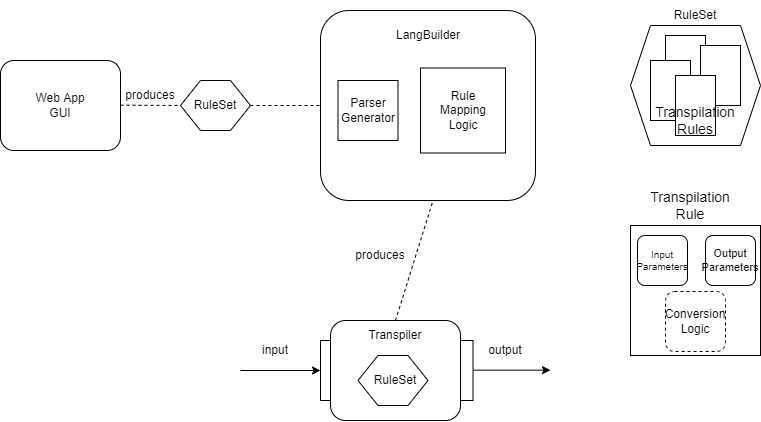


Figure 2 - LangBuilder System Design

To the diagram above we provide the following clarifications:

* The system is composed of 3 main parts – “Web App GUI”, “LangBuilder” and “Transpiler”.
* The system is composed of the following subcomponents – “RuleSet” and “Transpilation Rule”.
* The Web App GUI produces a RuleSet and sends it to the LangBuilder
* The LangBuilder prouces a Transpiler
* The Transpiler handles input and creates output
* The LangBuilder makes use of a Parser Generator as well as Custom Logic for Rule Mapping
* The RuleSet contains multiple Transpilation Rules
* Transpilation Rules contain as data fields Input Parameters and Output Parameters, and implicitly, Conversion Logic which specifies through a convention how the output is generated based on the input

The last aspect of the design on which we will touch upon is the application lifecycle. The application lifecycle is composed of a set of states of being, which are defined and delimited using set criteria/parameters of an application, as well as a set of transitions or methods through which the application changes from one perceived state to another.

The life cycle of the components of the system:

* The LangBuilder is stateless, it is a functional system and thus has no lifecycle
* The Transpiler is stateless, it is as well a function system and thus lacks a lifecycle
* The Web App GUI is partly used to create rulesets used by the Generator, and partly to use those rulesets to consume or ‘trigger’ the Generator. The diagram below depicts the flow of the frontend application. It is stateful, storing sequentially added rules as a ‘ruleset’ used afterwards. See Figure 3.

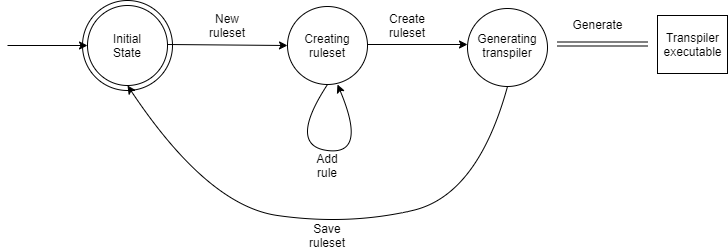


Figure 3 - Generator Lifecycle

The lifecycle element of the design is very succinct for the proposed design, as two of the three components are considered stateless. This is due to outside constraints forcing to simplify the implementation, and as such the Functional design was adopted.

## 3.3: Implementation Details

Within this section we will detail interesting points part of the implementation, as presenting every aspect of it would be unreasonable and impractical. For additional details, this paper will have an accompanying repository showcasing the code.

### 3.3.1: Rule List Processing

The Input Parameters of the application are represented by the Rule List domain model, and the customizability of the application is directly determined by the way the rules are defined. The initial implementation contains 4 rule types – Direct Translation, Expression, Rule Option Sequence and Rule Sequence. The former two are ‘simple’ rules which translate input tokens to output tokens directly. The latter two are ‘complex’ rules due to ‘depending’ on other rules out of the Rule List.

 For simplicity, the rule types are deserialized used dynamic deserialization and using a discriminator property ‘RuleType’. This is specifically done using the JsonSubTypes NuGet Package library [12] – see Figure 4.

Figure 4 - Transpiler Rule ViewModel

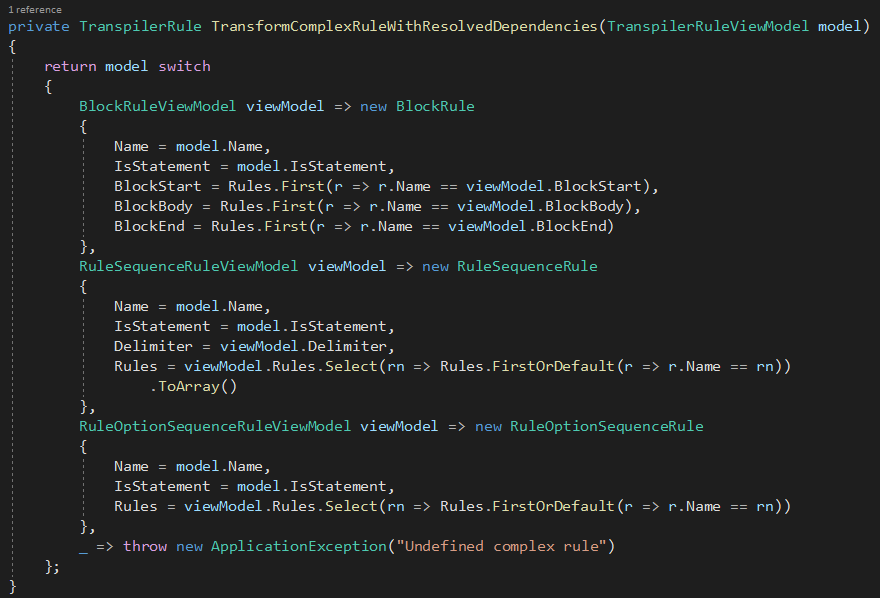
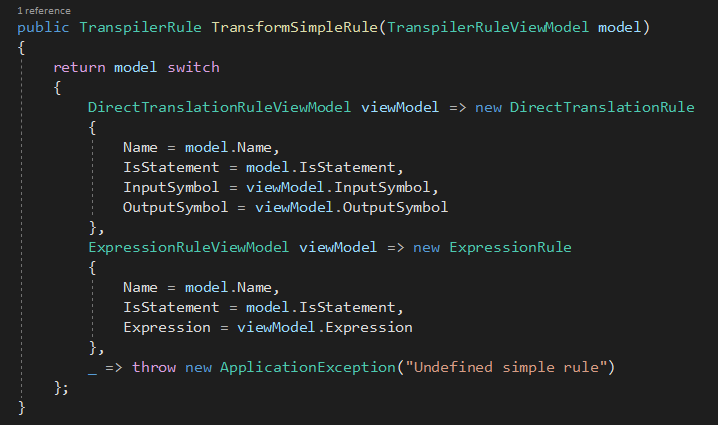
 Furthermore, the Rule List itself is processed and each rule is transformed from a ViewModel into the Rule Model itself – see Figure 5 and Figure 6.

Figure 5 - Transform Simple Rule

Figure 6 - Transform Complex Rule

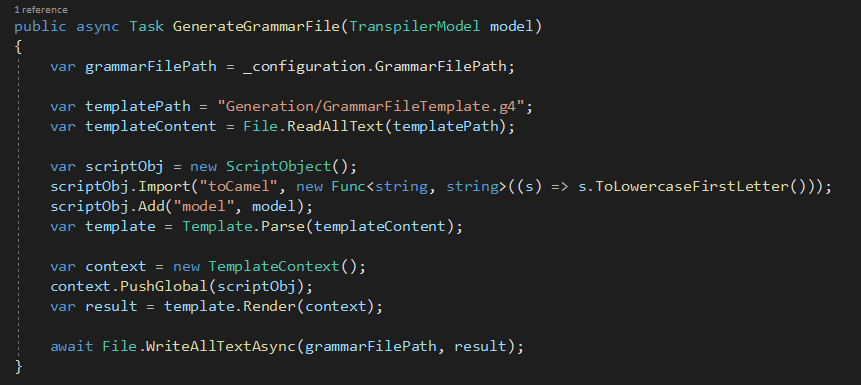
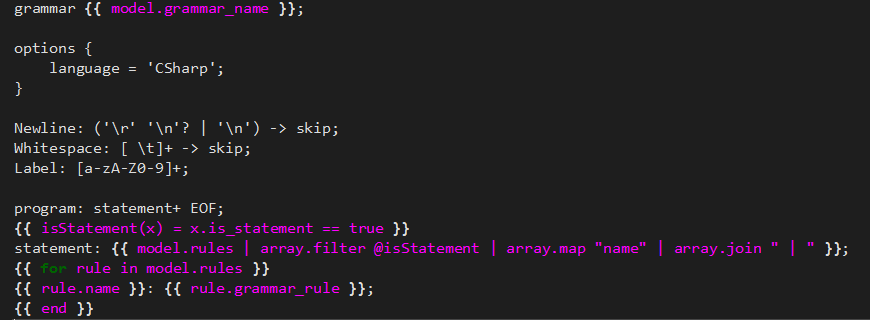
The Complex Rules are transformed after all of their dependencies are resolved. This is done using a class called ‘ComplexRuleTransformer’ which executes an Inverse DFS algorithm on the Rule List’s dependency graph.

### 3.3.2: Grammar File Generation

The g4 Grammar File used by the ANTLR CLI is generated directly by the web application using the RuleSet. This is done using a templating language called ‘Scriban’, using the Scriban NuGet Package [13]. See Figure 7 and Figure 8.

Figure 7 - Grammar File Template

Figure 8 - Create Grammar File



### 3.3.3: Transpiler Executable Generation

The final step of the pipeline is generating the executable itself. The Transpiler executable is a .NET Core Console Application. The executable is generated using C#’s Roslyn Compiler [14].

The ANTLR CLI is invoked in order to automatically generated the C# code doing the parsing of the executable [15] [16]. This will generate within the C# project a Lexer, a Parser, and a Visitor. The Visitor archetype is used to traverse the Abstract Syntax Tree (AST) resulted from the parsing process. The Visitor’s implementation is generated using the Roslyn Syntax Tree API, using the Transpiler Rules. Each Rule has a corresponding Visitor method. See Figure 9.

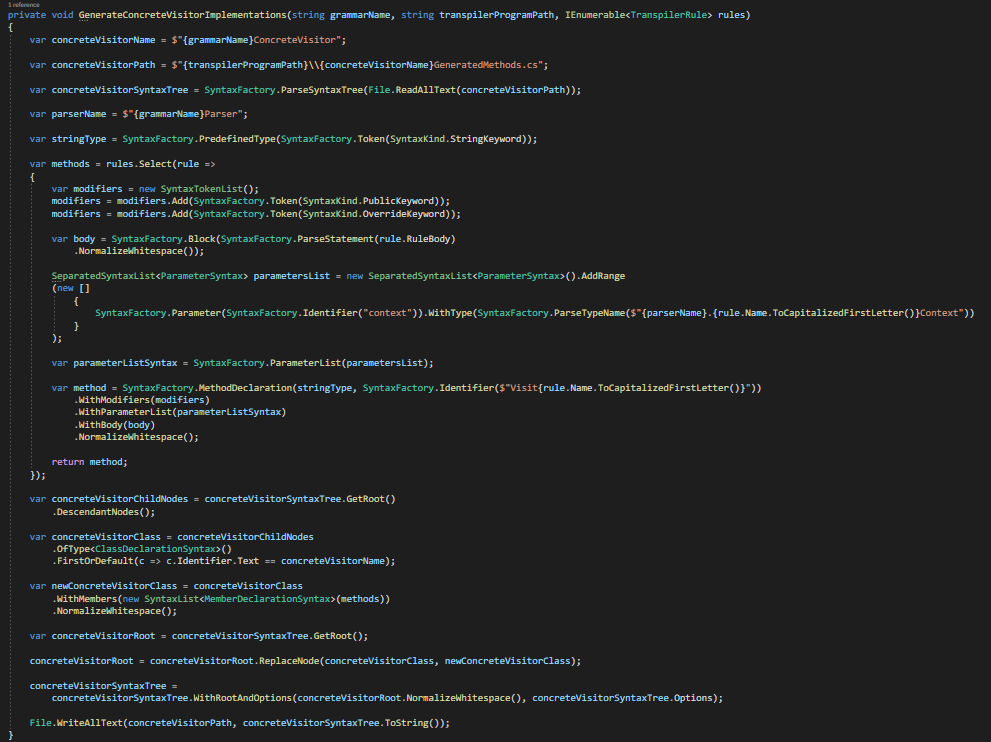


Figure 9 - Generate Visitor Implementation

Finally, the Rolsyn Compiler is used to compile the Transpiler project using all dependent libraries and bundled into an executable.

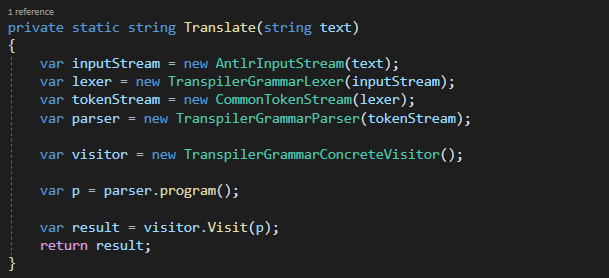
The Transpiler Executable is a simple console application, with the added option of reading text from a file, which will instantiate the Parser and Visitor, and use it to process the input text. See Figure 10.

Figure 10 - Translate

## 3.4: Testing

The solution domain of the application, being plain text representing code, is complex and purposefully left ambiguous. The purpose of this application is merely to showcase the possibility of implementing the idea. The specific use cases (and thus, test cases) of the application remain to be determined in a real-world scenario.

Consequently, for demonstration purposes we have chosen a small number of test cases showcasing the pragmatism of the idea, as well as the large potential for customizability and extensibility. The use cases chosen for this iteration of the implementation are depicted below:

1. Simple Direct Transpilation

Story

* A given token ‘symbol1’ will be transpiled to ‘symbol2’
* The tokens ‘symbol1’ and ‘symbol2’ are parameterized

Constraints

* The input is comprised of a single token

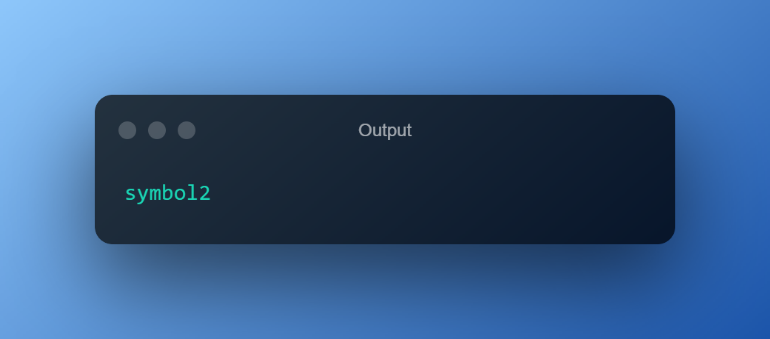
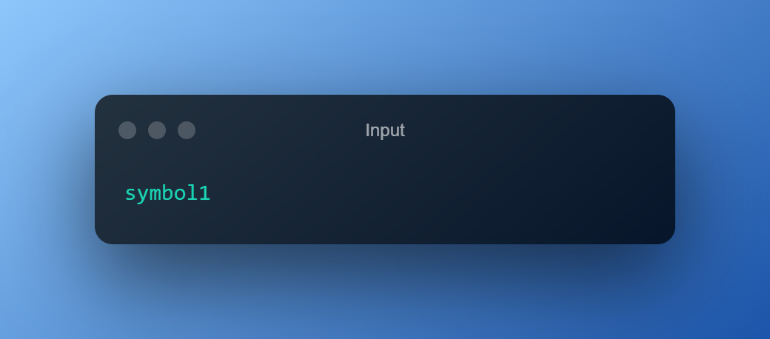


Figure 11 - Simple Direct Transpilation - Input

Figure 12 - Simple Direct Transpilation - Output

1. Direct Transpilation

Story

* A given token ‘PI’ will be transpiled to ‘3.14’
* The tokens ‘PI’ and ‘3.14’ are parameterized
* Unrecognized tokens will be unaffected

Constraints

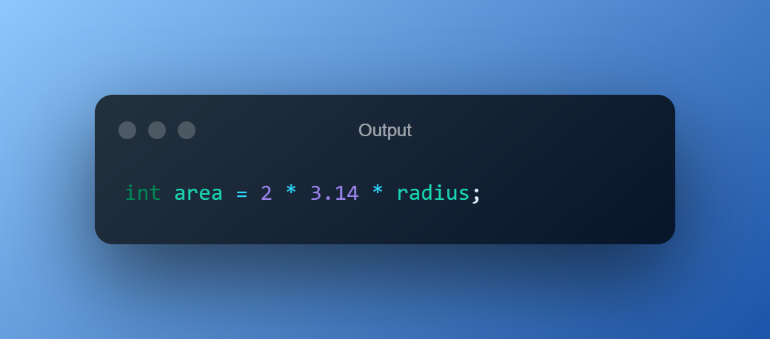
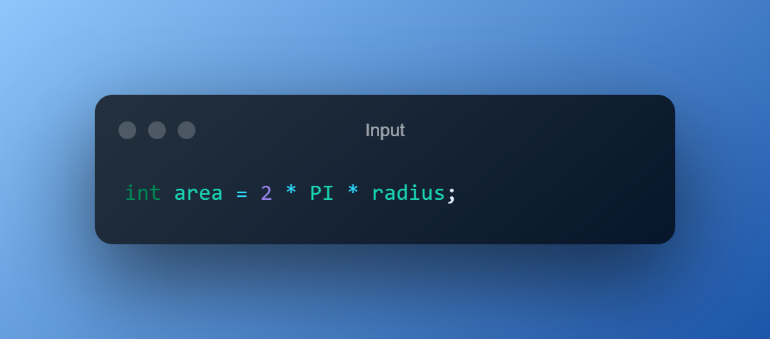
* The input contains different tokens representing code
* The input will contain the given token ‘PI’

Figure 13 - Direct Transpilation - Output

Figure 14 - Direct Transpilation - Input

1. Composite Transpilation

Story

* A given token `expose` will be transpiled to a composite token `public static`
* The token `expose` and the expression `public static` will be parameterized
* Unrecognized tokens will be unaffected

Constraints

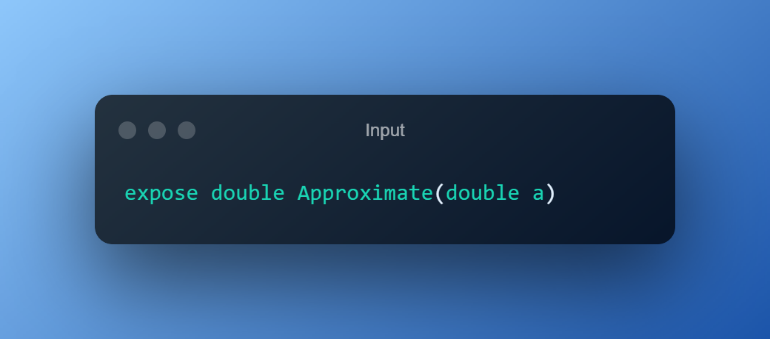
* The input contains different tokens representing code
* The input will contain the given token ‘expose’

Figure 15 - Composite Transpilation - Input

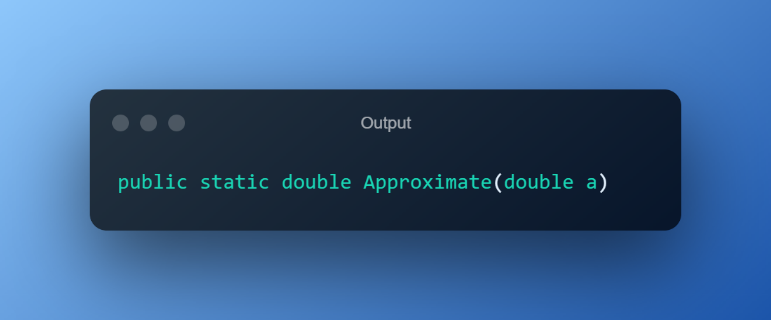


Figure 16 - Composite Transpilation - Output

1. Practical Example – User-defined programming language transpiled to Java

Story

* A user-defined programming language will be transpiled to Java code

Constraints

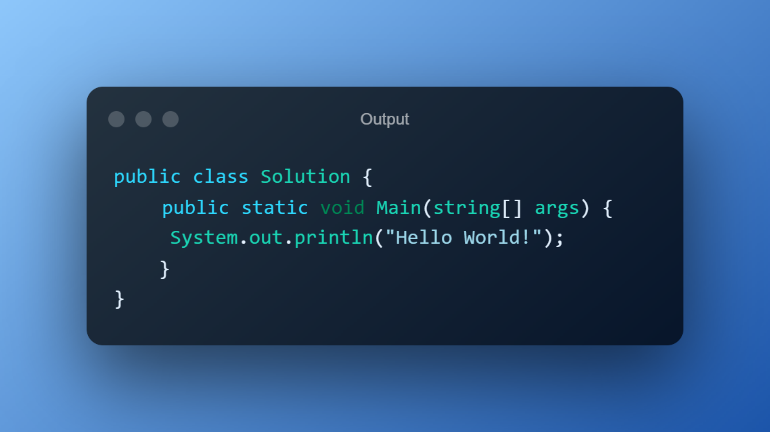
* The input will contain valid syntax for the user-defined programming language

Figure 17 - Practical Example - Input

Figure 18 - Practical Example - Output

## 

## 3.5: User Guide

### 3.5.1: User Interface

The implementation accompanying this paper is intended for demonstration purposes. As such, for facility purposes it also contains a rudimentary Graphical User Interface (GUI) in the form of a web application showcasing the features of the implementation.

An example application usage flow with additional explanations provided below:

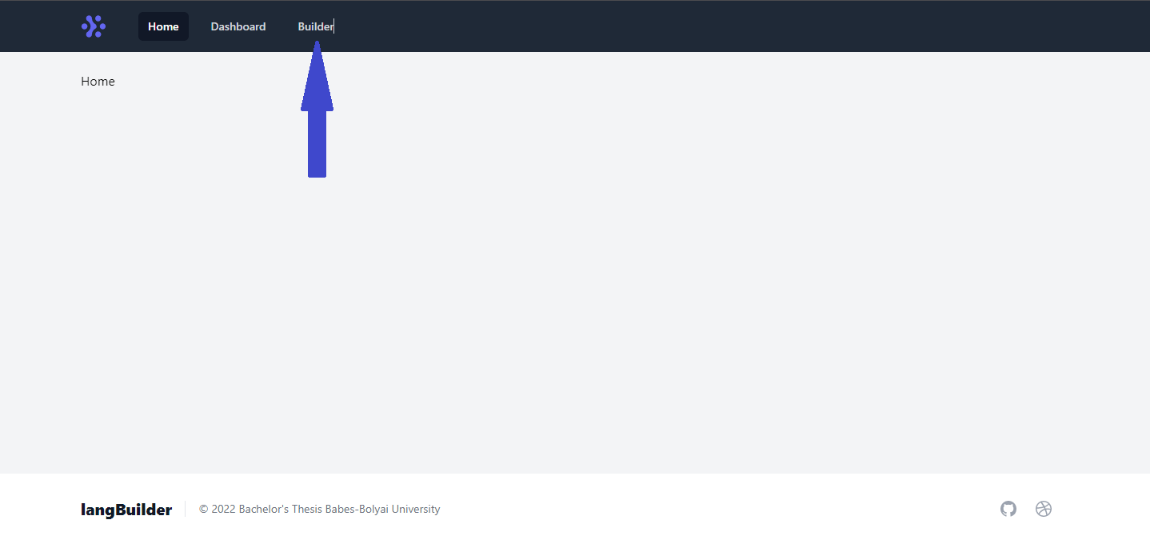
1. Initial Screen (Home Page)
   * Proceed to Builder page by clicking ‘Builder’

Figure 19 - Home Page

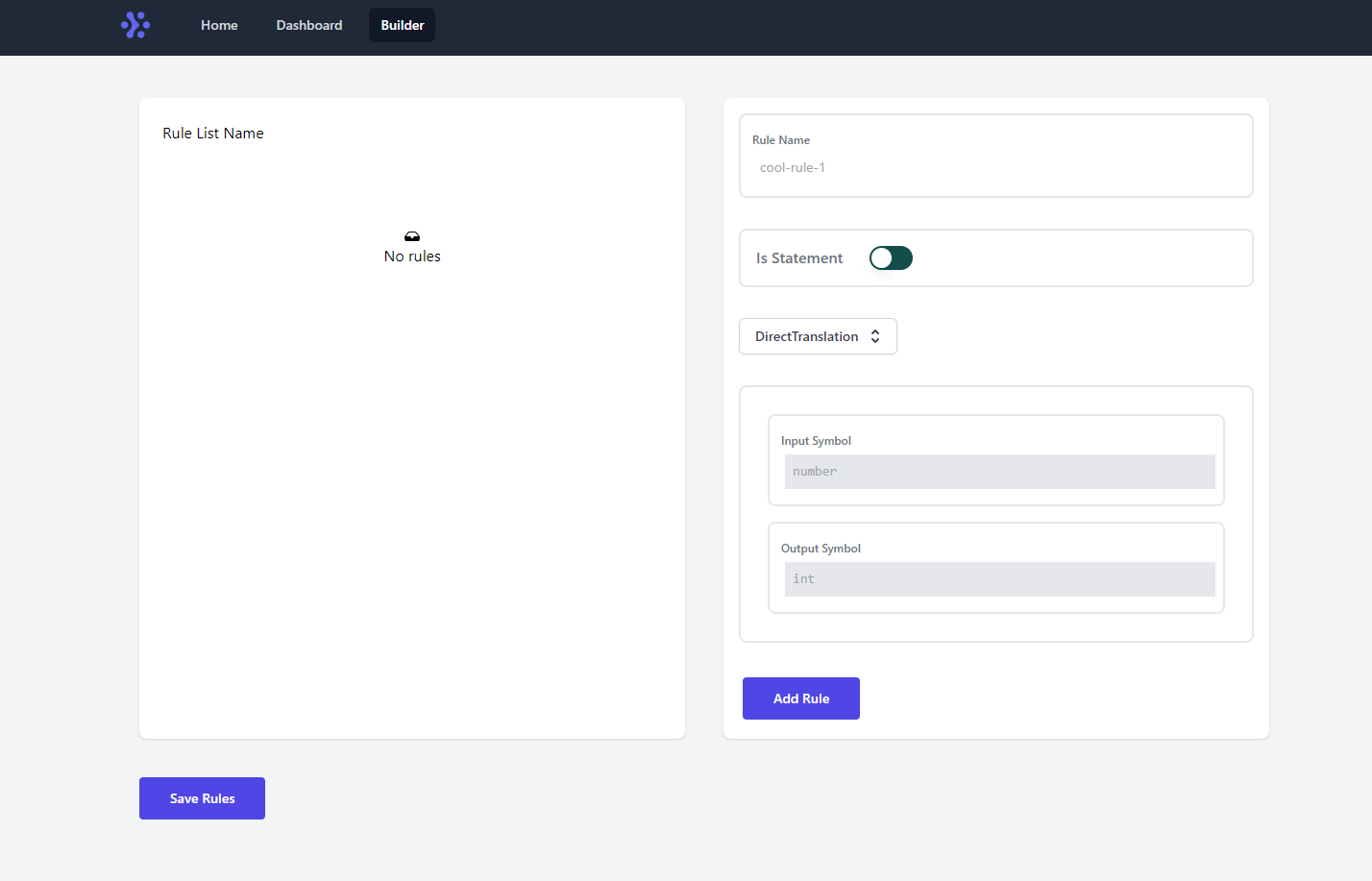
1. Builder Page – Rule List Section
   * The first section is the Rule List Building Section. Rule Lists are used to generate a transpiler executable.

Figure 20 - Builder Page

* + Rule Lists must have a unique name. The name can be edited by clicking on it

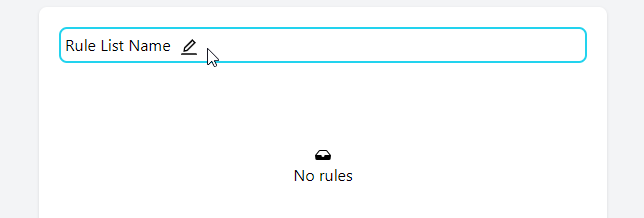


Figure 21 - Edit Rule List Name

* + Rule Lists are made up of Rules. Rules have a name that must be Unique, a switch dictating if they are a statement or not and a type.

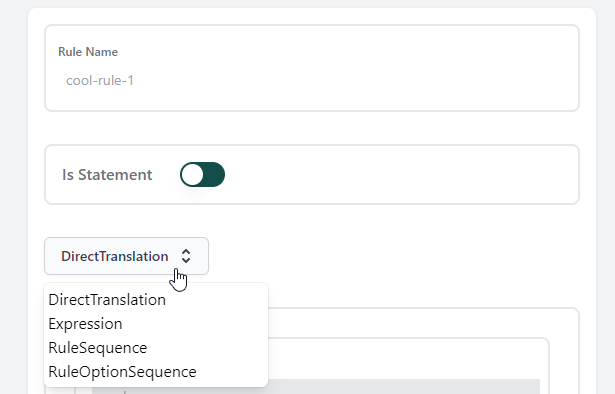


Figure 22 - New Rule Form

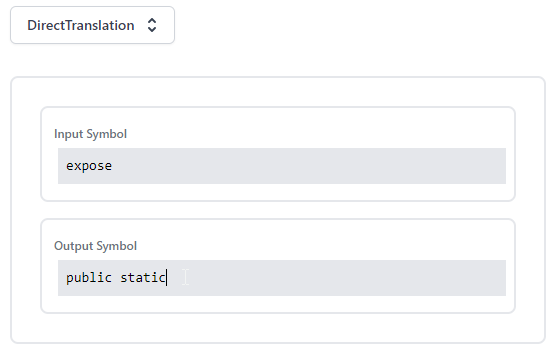
* + There are 4 Rule Types. Rule Types have additional, specific fields which are used to create the transpilation algorithm

Figure 23 - Rule Types Form

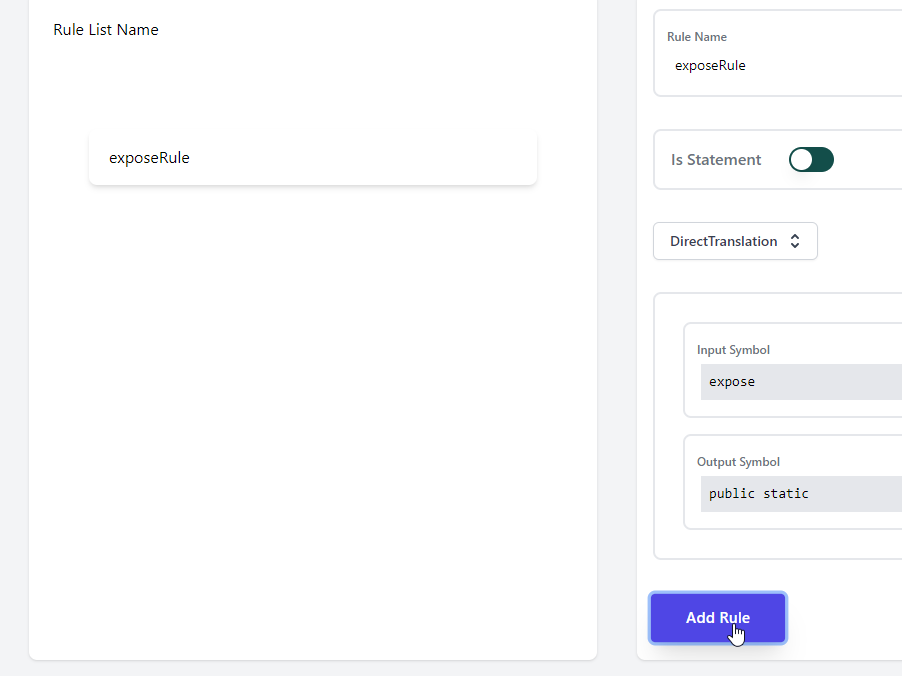
* + Rules are then added to the Rule List using the ‘Add Rule’ button

Figure 24 - Add Rule Button

1. Builder Page – Save Rule List
   * Once Rule List creation is done, press the Save Rule List button, which will save the Rule List in JSON format for storage

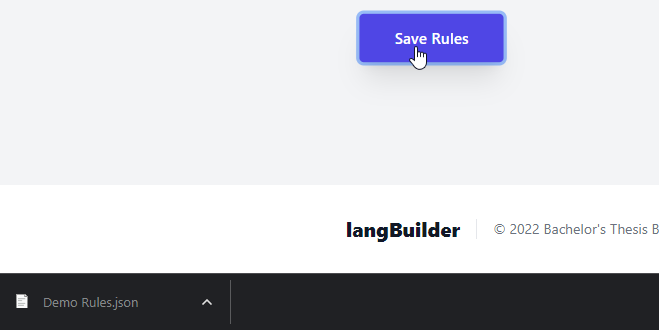


Figure 25 - Save Rule List

* + The Rule List JSON format contains the values of the fields of the Rules

Figure 26 - Rule List JSON File

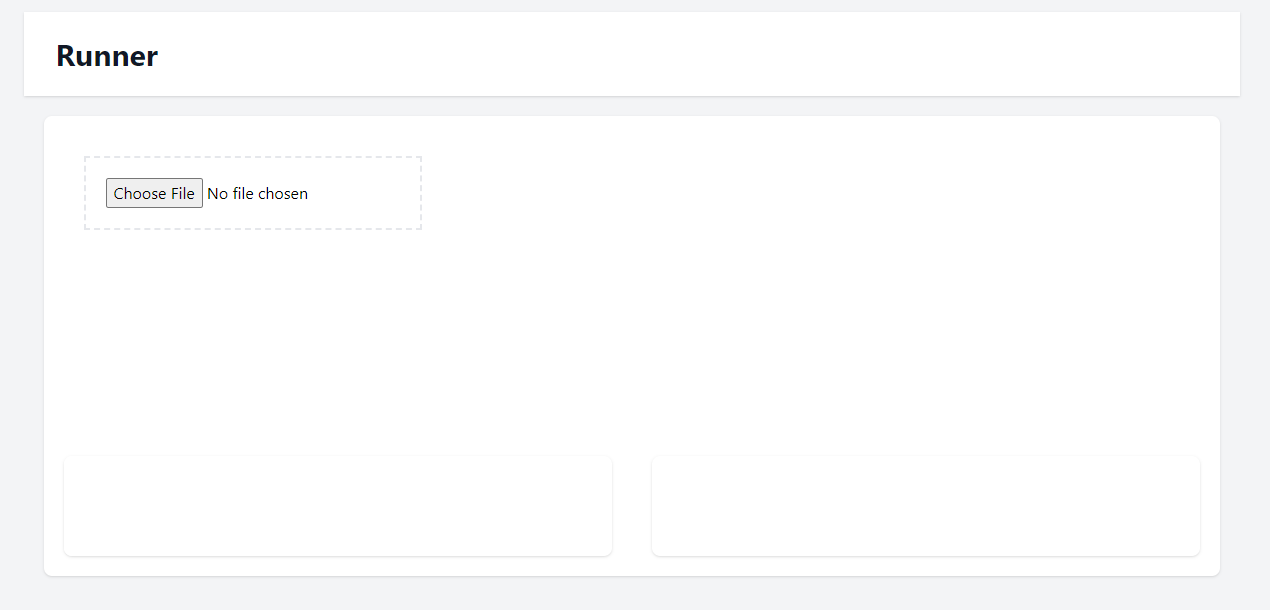
1. Builder Page – Runner Section
   * Scrolling down will reveal the ‘Runner’ section. This section uses the Rule List to showcase the Transpilation algorithm

Figure 27 - Transpiler Runner

* + Click the ‘Choose File’ button and select the JSON Rule List file

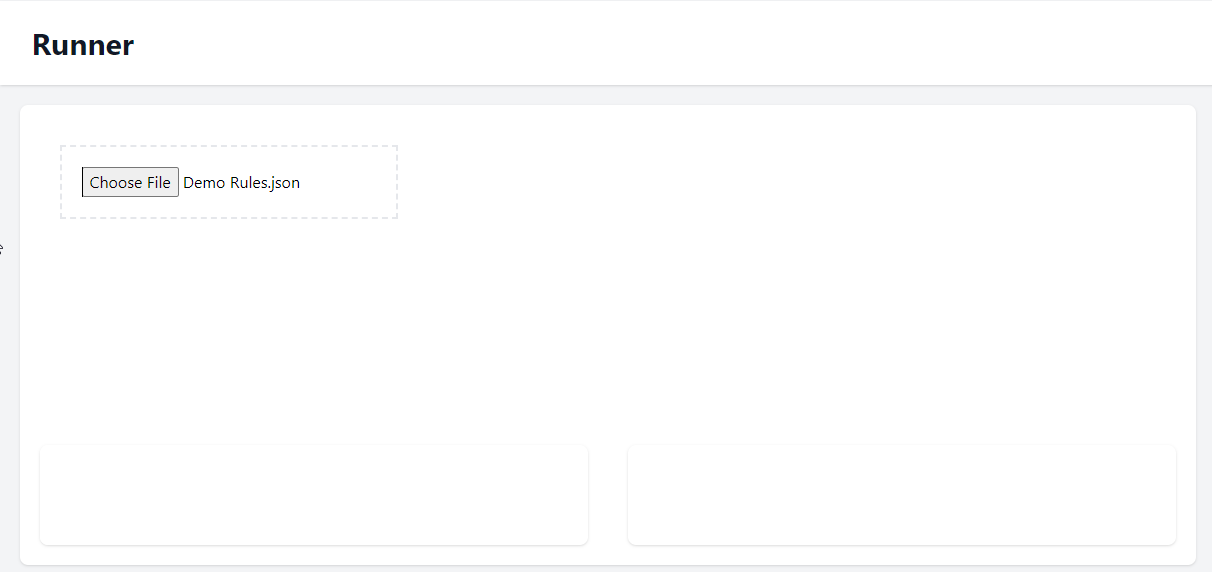


Figure 28 - Runner File Upload

* + Once selected, write the input in the text box and it will be automatically transpiled

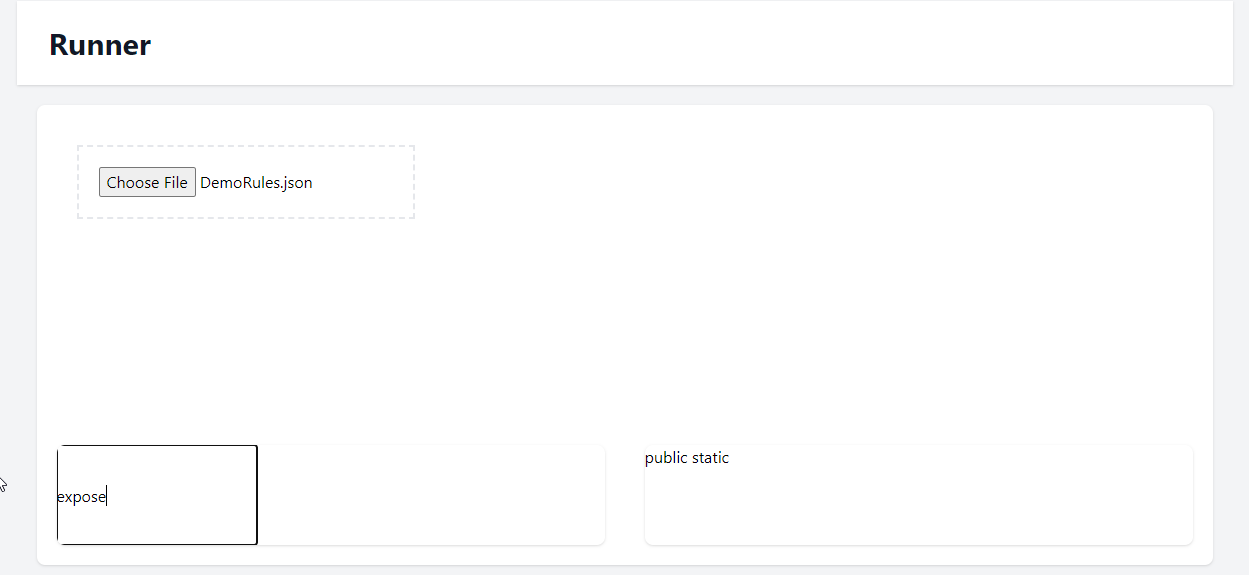


Figure 29 - Runner Transpilation

### 3.5.2: API Interface

The main feature of the LangBuilder application – Transpiler Generation is exposed as a REST Endpoint intended to be consumed. The schema of the endpoint’s parameters is exposed below (Typescript Syntax) – see Figure 30.

Additional constraints:

Figure 30 - Rule List Schema

* The Direct Translation Rule’s Input and Output symbols must not contain special tokens
* The Expression Rule’s Expression field represents a Regex expression which should disregard white space
* The Rule Option Sequence and Rule Sequence Rules’ RuleNames fields must contain names of other Transpilation Rules
* The Rule Option Sequence and Rule Sequence Rules’ RuleNames fields must not create cyclical dependencies within the RuleList

# Chapter 4: Conclusions, Further Work

## 4.1: Conclusions

As the mental panorama of the information within this paper draws to a close, we would like to step back and present the synopsis of it in which we will highlight the most important aspects presented by the paper.

In this document we have displayed and explained the notion of metaprogramming, as well as its significance to the software creation world in order to initialize the context in which this document was conceived.

Furthermore, we describe a number of different empirical methods of manifestation of metaprogramming, and we exemplify how they were used in the modern programming scene to create the very tools on which it is built upon.

Finally, we propose an innovative concept through applying a metaprogramming principle to two virtually omnipresent already existing brother concepts – compilation and interpretation. We continue by shaping the concept, argument its potential and applicability in actual modern scenarios and/or pain points, before culminating with a corporal manifestation of the concept serving to demonstrate the feasibility of the idea.

To add specificity to the statements above, the presented application – the LangBuilder system is an exemplification of general purpose metaprogramming, applied with the purpose of customizing the compilation/interpretation process. This is peculiarly done by generating another application, denoted as transpiler, whose purpose is to transform non-compileable/interpretable source text, into compileable/interpretable source text. In essence, it is the idea of customizing an element of the development context itself – the programming language. As such, the application is appropriately named LangBuilder – (Programming) Language Builder.

## 4.2: Further Work

The current paper is predominantly focused on initially proposing a novel concept, detailing it and enumerating a small-scale set of pragmatic use cases of the concept. Additionally, the application accompanying this paper represents a mere POC, with a very narrow feature set, mostly due to outside factors exacted on the process of composing this document.

As such, there are multiple extension points for the current intellectual patent as well as the current form of the application.

From the abstract, conceptual standpoint, a first method of extending the work is by applying general purpose transformation metaprogramming in a similar fashion, to a different concept related to programming, such as application feature test manufacturing. A different method of approach is attempting to customize, in a similar fashion, the programming context environment – the development machine. This, however, is actually not a novel or foreign concept. It is an idea related to the concept of containerization used by technologies like Docker. The link between the approach, and the result, which is a concept that actually exists can prove meticulous to explore in a new document.

From the concrete, implementation standpoint, potential for further work lays firstly in the possibility of providing an entirely new design for the system, with a different approach to the implementation, yielding contrasting yet peculiar results as well as newfangled use cases. Additionally, the current implementation as it stands can be greatly improved upon, through additional features, user experience improvements, and one area of which proves to have a lot of extensibility potential is the set of Transpilation Rules, which are determined by convention, yet can be extended with new forms in order to fill the provenly most common use cases.

# References

|  |  |
| --- | --- |
| [1] | A. P. David, "Source-to-Source Translation and Software Engineering," *Journal of Software Engineering and Applications,* vol. 6, no. 4A, 2013. |
| [2] | X. Yan, "4 ways to generate code in C# - Including Source Generators," 19 January 2021. [Online]. Available: https://levelup.gitconnected.com/four-ways-to-generate-code-in-c-including-source-generators-in-net-5-9e6817db425. |
| [3] | A. Kumar, "Source to Source Compiler," [Online]. Available: https://www.geeksforgeeks.org/source-to-source-compiler/. [Accessed 6 June 2022]. |
| [4] | JetBrains s.r.o, "JetBrains MPS," [Online]. Available: https://github.com/JetBrains/MPS. |
| [5] | E. Potter, "T4: C#’s Little Known Code Generator," [Online]. Available: https://humbletoolsmith.com/2018/04/05/t4-csharp\_s-little-known-code-generator/. |
| [6] | Microsoft, "ASP.NET Web Apis," [Online]. Available: https://dotnet.microsoft.com/apps/aspnet/apis. [Accessed May 2022]. |
| [7] | Wikipedia, "Compiler-Compiler," [Online]. Available: https://en.wikipedia.org/wiki/Compiler-compiler. |
| [8] | T. Parr, "ANother Tool for Language Recognition (ANTLR)," [Online]. Available: https://www.antlr.org. |
| [9] | N. Tahir, "An introduction to language lexing and parsing with ANTLR," 24 October 2017. [Online]. Available: https://www.willowtreeapps.com/craft/an-introduction-to-language-lexing-and-parsing-with-antlr. |
| [10] | K. Justin, L. Haakon and A. Jeffrey, "SML2Java: A Source to Source Translator," *Cornell University PLI,* August 2003. |
| [11] | Jarble, "Universal-transpiler," [Online]. Available: https://github.com/jarble/transpiler. [Accessed May 2022]. |
| [12] | E. Counasse, "JsonSubTypes," [Online]. Available: https://github.com/manuc66/JsonSubTypes. |
| [13] | A. Mutel. [Online]. Available: https://github.com/scriban/scriban. |
| [14] | K. Hazzard and B. Jason, Metaprogramming in .NET, ISBN 9781671290268, 2012. |
| [15] | Full Boar LLC, "ANTLR4, .NET Core 2.1, and C#: Getting Started," 28 August 2018. [Online]. Available: https://fullboarllc.com/using-antlr-4-with-net-core-2-1-and-c-getting-started/. |
| [16] | R. Reynold-Haertle, "Create a Text Parser in C# with ANTLR," 12 November 2022. [Online]. Available: https://docs.microsoft.com/en-us/events/dotnetconf-2020/create-a-text-parser-in-c-with-antlr. |