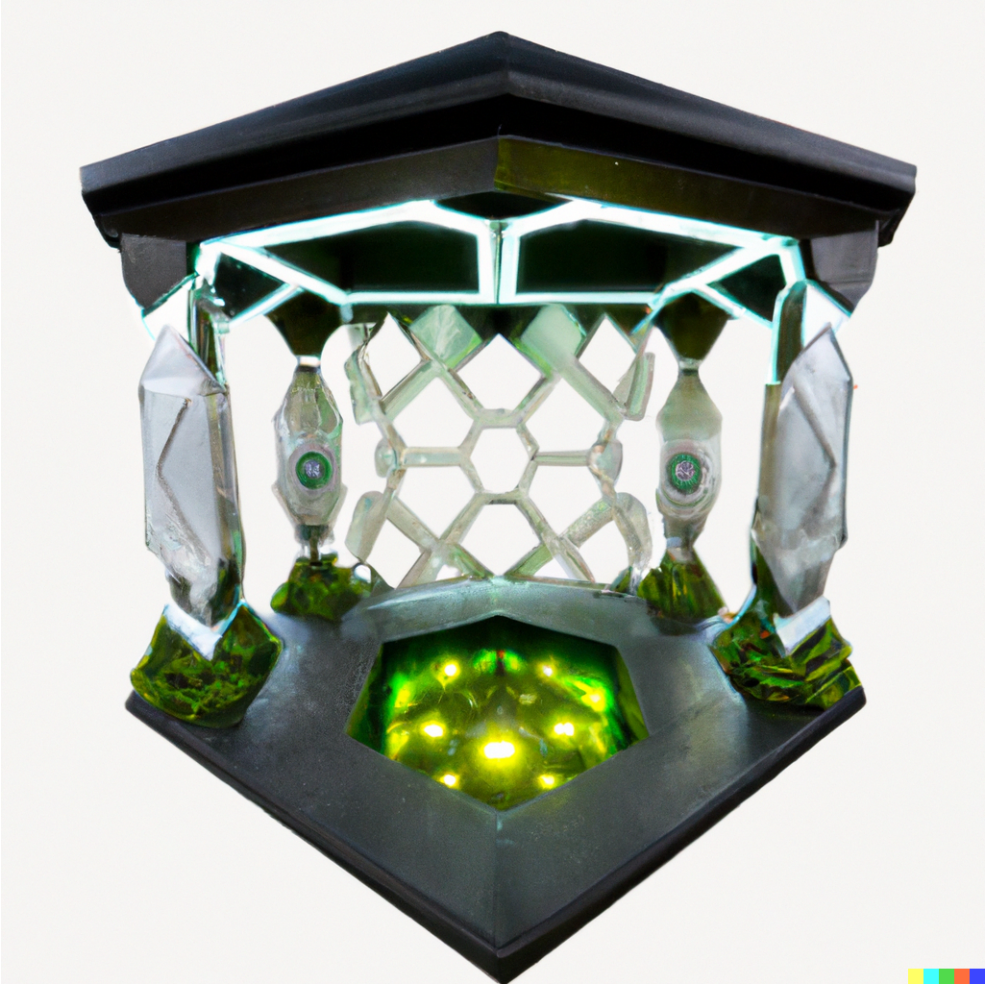
A Domain-Specific Language for Minecraft

Honours Programme Bachelor Report



J.H. (Jochem) Broekhoff

A Domain-Specific Language for Minecraft

REPORT

submitted in partial fulfillment of the
requirements for the

HONOURS PROGRAMME BACHELOR

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.tudelft.nl/eemcs

Cover picture: DALL·E art: `https://labs.openai.com/s/b14qOH9XBYCRtyhcCGDb6doF`

A Domain-Specific Language for Minecraft

Author: J.H. (Jochem) Broekhoff
Email: J.H.Broekhoff@student.tudelft.nl

Abstract

The popular video game Minecraft offers a feature-rich environment for building extensions. Unfortunately, the learning curve is steep for new users as it requires a non-conventional way of thinking about data organization and program flow in a highly verbose and error-prone syntax. We propose Gazebo, a new domain-specific language, inspired by existing community efforts and our earlier work, that aims to provide a more natural and expressive way of programming for Minecraft. The language is fully implemented using the Spoofax Language Workbench, resulting in a product with editor support and a stand-alone command line interface. We report on the design of the language, its compilation pipeline and our experiences with using Spoofax to achieve this.

Supervisors:

Dr. J.G.H. (Jesper) Cockx, Faculty EEMCS, TU Delft
A.S. (Aron) Zwaan, Faculty EEMCS, TU Delft
Prof. dr. E. (Eelco) Visser, Faculty EEMCS, TU Delft

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The video game Minecraft is the best-selling game of all time, released in 2011. Especially among the younger generation it is, or has been, highly popular and has played a significant role in their lives. Starting in the early days, the game has offered a chat-line based interface for sending commands to the game engine. With recent new releases of the game, these mechanisms have become more and more powerful, efficient and expressive. Notably, with the release of version 1.12 in 2017, it is possible to cluster these commands in batch, referred to as a 'function'. Even though the command system offers a vast feature set, its learning curve and general accessibility leave a lot to be desired.

To aid programmers, community projects targeting early versions of the game, such as *Command Block Parser* (ZipKrowd Team 2016) and *Redstone Programming Language* (Gromov 2014) implemented preprocessor engines. This trend has continued with *Mecha* (Berlier 2021) and *McScript* (Stevertus 2020). Other recent developments, such as *Trident* (Energyxxer 2019) and *Command Block Assembly* (Simon816 2017), have put more focus on introducing higher-level features and creating a conventional language experience. In this line, *Scarpet*, as part of the *Fabric Carpet Mod* (Gnembon 2019), has been developed to provide a native scripting language for Minecraft, however it requires the installation on a mod, which requires more effort from end-users, but delivers the most powerful features.

Their feature sets differ greatly, and only have limited overlap, as can be seen summarized in table 1.1. *Command Block Assembly* attempts to provide a C++-like experience, while *Trident* on is closer to our developments, as it puts the most focus on the DSL aspect, while still strongly trying to adhere to and integrate with the common Minecraft command conventions. The preprocessor-based approaches are not true programming languages with nontrivial compilation processes, but rather tools to aid the writing of repetitive Minecraft commands.

Another approach found in the community is that of a holistic data pack development

Project	Target	Build logic embedded	Non-trivial compilation	Type-safe
Command Block Assembly	MCFunction	●	●	○
Command Block Parser	command block	●	○	
Redstone Programming Language	command block		○	
McScript	MCFunction		○	
Mecha	MCFunction		●	○
Trident	MCFunction	●	●	○
Ours (*Gazebo*)	MCFunction	●	●	●

Table 1.1: Comparison of existing community efforts related to our new developments.

framework. Two prominent examples are *Beet* (Berlier 2020) and more recently *ObjD* (Stevertus 2022). These projects do not necessarily focus on function-level programming, but rather on the whole process of creating a data pack, of which writing functions is just a part.

Despite these efforts, none focus on developing a language that is both truly natural to the developer audience and that offers a feature set comparable to conventional languages and is fully type-safe. This technical report describes the process of implementing a new programming language, named Gazebo, that attempts to address some open issues with regards to type safety and ease-of-use.

Most notably, existing solutions do not offer full type safety, if any at all, as conventionally, types are only checked at runtime. The game itself only verifies the validity based on syntax rules, which by far do not account for all possible typing errors. Thus, tools that *do* perform some deeper level of analysis, be it linting or full type checking, need more context than the game itself has. That is the essential reason why a DSL is worth exploring, as the required context can inherently be gathered.

Spoofax Language Workbench

In particular, this report describes the process of implementation using the *Spoofax Language Workbench* (Spoofax Team 2021; Kats and Visser 2010).

The project first and foremost serves as an evaluation of the current state of Spoofax 2. We do this by means of an in-depth review of the experiences, which can be found in section 5. Since the inception of Stratego/XT, followed by MetaBorg and finally Spoofax, many aspects have changed. More will change in the future, especially with Spoofax 3 in development. In our review, we try to identify possible points of improvement that may be incorporated in these future releases. Our work covers the entire breadth of Spoofax, from high-level usage to the inner workings of the Java application programming interface (API).

In literature there have been numerous publications of languages or projects being implemented using Spoofax, such as the first version of *WebDSL* (Groenewegen et al. 2008) and *Apply* (Hamey and Goldrei 2008) in Stratego/XT, several languages (Bravenboer, Groot, and Visser 2006) in MetaBorg, and *Grace* (Haisma 2017) in Spoofax. This report attempts to put more focus on the usability aspect of the workbench, while also serving as a walk-through for new users.

Spoofax offers most of its features in the form of a (meta) DSL. This means that the languages in which we define particular aspects of our language are themselves defined using Spoofax, making it a self-hosting environment. Of these, we use the following:

- Syntax Definition Formalism 3 (SDF3): syntax specification.
- Statix: static semantics specification: type checking and name binding.
- Stratego: transformations between intermediate stages; final assembly.
- Editor Services (ESV): Eclipse editor configuration: menu action items and syntax highlighting.

Report Outline

The first two chapters are dedicated to the design and implementation of the language itself. We first summarize the entire processing pipeline in chapter 2. Then, in chapter 3 we elaborate on some key aspects of the implementation.

We move on to project-related content starting with chapter 4, where we evaluate the state of the project with respect to the Honours Programme. In chapter 5, we cover our experiences

with Spoofax and give some points of criticism. Finally, with chapter 6, we conclude the report and give some pointers for future work.

For completeness and more background information, we have included several appendices. Appendix A goes into more detail on the feature set of the language. Appendix B explains how a stand-alone command-line interface (CLI) has been constructed using the Spoofax Core API, combining all aspects of our language. Finally, appendix C covers the contents and automated construction of the standard libraries.

The full source code of the project implementation can be checked out from the following Git repository: `https://github.com/MetaBorgCube/gazebo`.

Chapter 2

Pipeline Summarized

This chapter covers the entire Gazebo compiler pipeline from start to end. As we almost exclusively rely on Spoofax and its meta-DSLs, a brief introduction will be given for starters. We only cover the parts of Spoofax that are relevant for Gazebo, so this is by no means a definitive guide. Additionally, for those new to the field of compiler construction, we explain the relevant areas on a high level.

Note that this chapter covers the Gazebo 'core pipeline', which is only readily usable from within the Eclipse integrated development environment (IDE). In appendix B, we cover the final big picture, of which the design described in this chapter is a part.

2.1 A Brief Intro to Spoofax and Compiler Construction

The *Spoofax Language Workbench*, commonly referred to as *Spoofax*, is "a platform for developing textual (domain-specific) programming languages" (Spoofax Team 2021). In other words, Spoofax is a complete toolchain for developing programming languages. Its reference manual can be found at `https://www.spoofax.dev/`. Visser (2021) published an excellent writeup on the history of Spoofax, which we recommend for more background information.

Projects using Spoofax usually model the conventional compiler pipeline, which comes down to the following: parsing, static analysis and transformations. It has to be noted that practical compilers do not necessarily follow these stages all that strictly. For many reasons, not limited to bad historic ones, they may be constructed differently.

A slightly different common perspective on the compilation stages is a more strictly pipelined variant. In this model, a 'front-end' handles tasks related to the surface language, and converts it into some intermediate representation (IR). This, in turn, passes through one or more 'middle-ends', which optimize the IR. Finally, a 'back-end' is selected, depending on the configuration, which emits the final output, a process called 'code generation'.

Parsing First, the source files are read and parsed. Parsing is the art of converting the textual form of a program, which is the source file, into a data structure that the compiler itself can easily work with.

The kind of information the parser outputs, depends on the particular implementation. In the case of Spoofax Core, we get an abstract syntax tree (AST). Other styles of parsers may emit a richer data structure, which contains lay-out information, such as a concrete syntax tree. The output from the parser forms the basis for the rest of the compiler pipeline.

Static Analysis The next typical major stage is 'static analysis', a very broad term for different analyses run on an AST without executing the program. Compilers perform static analysis to reject programs that cannot be executed.

Typical tasks range from simple type checking, to more complex type inference, import analysis, symbol origin location resolution (name binding), or termination checking. For the purposes of Gazebo, we perform type checking and name binding.

Spoofax offers Statix as the go-to solution for implementing this kind of static analysis. In fact, we do not even have to implement the analysis manually. Statix is a DSL in which we merely have to formalize the specification of our type system. This gives us a type checker almost for free.

Transformation In the last stage, transformations are applied. Similar to static analysis, transformation is an umbrella term for many different actions. Prominent transformation applications are desugaring, optimization and code generation. In practice, compilers often have many more (different) transformation steps, that are not as clearly defined as these three. In Spoofax, transformations are conventionally written in Stratego, a meta-DSL for program transformations.

Desugaring is the transformation of an initial program's AST (derived from what the programmer writes) into a more simplified syntax. The reason for this is that, usually, a language contains some 'syntactic sugar', which is where the name of this transformation comes from. Syntactic sugar is syntax that is useful for a programmer, but does not correspond to a concrete language feature. The desugaring transformation therefore normally transforms the input AST to a simpler subset of the grammar of the surface language, which is easier to work with.

Similarly, in optimization the transformation restructures the AST into a way that is likely more efficient. Generally, it cannot be predicted exactly how to execute any program in the most efficient way. Optimization transformations are therefore usually said to be a heuristic.

Code generation does, in contrast, transform to a different syntax. Depending on the use-case, this can be directly to a string, written to a file, or another tree-based representation of the target language. Usually, code generation is the last step in a compiler: the generated code is the end-goal of a compiler.

The beautiful thing of Spoofax is that these steps we just described are not fixed. Instead, Spoofax has been designed with modularity in mind: each stage is independently replaceable with a complete different implementation. Similarly, introducing new stages should be similarly straightforward.

2.2 Schematic Overview

Gazebo's compilation pipeline is composed of several stages. The components that make up these stages are listed in table 2.1, where the meta-DSLs that each relies on are indicated. Note that three different kinds of components can be clustered. First, the 'pure' language projects (`lang.*`), which are primarily used to define syntax and static semantics. Secondly, the extension projects (`ext.*`), forming the connection between two syntaxes, by means of Stratego strategies. Finally, `str-common`, which is a Stratego-only project, providing reusable strategies on top of the regular Stratego standard library, used by all extension projects. We collected the strategies in this package over the course of the project, mainly to avoid code duplication.

All these components and their interdependencies are visualized in figure 2.2. This is still a simplification compared to reality, but it gives a good overview, without worrying about the details. Note that it captures more details than described above, which we will now cover step-by-step.

We start at the `lang.gazebo` component, which is the main component of Gazebo. Here, the input files are parsed into an AST by JSGLR, the Java implementation of the scannerless

Component	SDF3	Statix	Stratego	ESV
`lang.gazebo`	•	•	•	•
`lang.gazebo-core`	•			•
`lang.llmc`	•			•
`ext.gzb2gzbc`		API	•	•
`ext.gzbc2llmc`			•	•
`ext.llmc2mcje`			•	•
`str-common`			•	

Table 2.1: Explicit usage of meta-DSLs by the different components of Gazebo

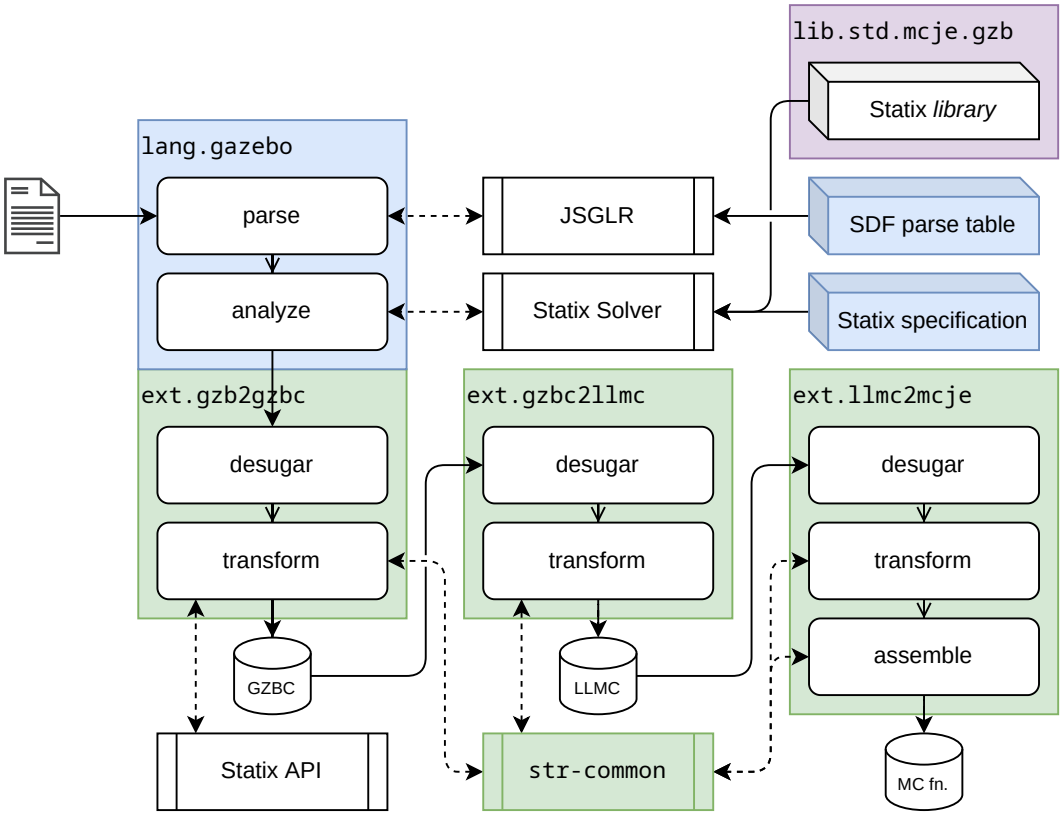


Figure 2.2: High-level schematic overview of the Gazebo compilation pipeline in Spoofax

generalized LR (SGLR) algorithm (Visser 1997). JSGLR relies on parse tables, which are generated by SDF3. This way, the parser is fully independent of both the language that it is parsing and the framework used to formalize the syntax.

Next, by leveraging the Statix solver, we perform static analysis on the AST, annotating it with relevant extracted information along the way. Similar to JSGLR, Statix relies on a pre-packaged specification of the type system in the form of constraints. In addition, we provide the solver with a pre-analyzed library package (the 'Statix *library*') of our standard library to reduce analysis time (see appendix C). The static analysis mainly consists of type checking, such as checking that all variables are declared when used, and that all types are well-formed. Additionally, we annotate the AST with some meta-data, such as the derived type information or fully expanded references. Indirectly, this information is used by ESV to support control-click navigation of the source code in the editor.

The analyzed and annotated AST can now be processed by the extension projects, to transform it into the target language. All three transformation projects are implemented in Strat-

ego and depend on ESV to add menu items to the editor[1], used to run the transformations manually. Only `ext.gzb2gzbc`, which transforms GZB to GZBC, has an extra dependency, the Statix API, to extract information from the annotated AST.

The basis for each transformation project is a two-step process, namely desugaring followed by the 'real' transformation. The desugaring step takes the input AST and simplifies is, whereas the latter takes that simplified AST and transforms it into the target language. From project to project, these transformations differ greatly, which is why we cannot possibly cover everything in this report. Instead, section 3.3 covers the basics.

The target AST of each transformation is written to disk, and can be used as input for the next transformation. Normally, the output would be pretty-printed and consequently parsed, but we skip over this step for performance reasons, and instead only serialize, and in turn deserialize, the AST (see section B.5).

The last transformation stage, `ext.llmc2mcje`, contains an extra step, which is to assemble the internally used assembly format into the final target format: an MCFunction file. This is the text format that is finally usable by the game. We only have to perform some small organizational tasks, such as packaging up all these files into a ZIP archive with a structure that Minecraft accepts.

[1]Technically, ESV only serves as a configuration source for the action facet, which reads directives from the ESV source to determine which menu items to provide to the editor and which strategy to execute when the associated action is to be performed.

Chapter 3

Implementation Key Aspects

In order to get an idea of how Gazebo works and is implemented in practice, we will discuss some key aspects related to that in this chapter. We will take a look from four different aspects.

To start, we cover the high-level design of the surface language, mostly from a syntax-perspective, while also discussing the relevancy of DSL-typical features. Next, we outline the ideas behind the type system used. Then, in order to get a better understanding of how the internals of the language work together as a whole transformation pipeline, we discuss that particular architecture. Finally, we summarize how the stand-alone compiler was constructed from all separate parts.

3.1 High-Level Language Design and Features

Even though Gazebo composes of several languages internally, there is only one syntax that a programmer needs to know. We refer to this as the *surface language*. Where relevant in section 3.3, we will discuss the internal languages in more detail.

For brevity sake, we will not discuss all features, as there are too many. Instead, we will focus on the features and ideas which are most interesting from a DSL-design perspective. For the full overview of surface language features, see appendix A.

3.1.1 Modularity with Functions

One of the most important features that Minecraft misses, is proper functions. Although a community standard for interoperability has been proposed (Arcensoth 2020), there is no native way. Gazebo does provide functions similar to other high-level languages, including full support for argument passing, return values, and recursion. We achieve this by implementing a true call stack, which is cumbersome to use manually as it requires a lot of bookkeeping because it is not natively available.

These functions can call each other, not only from the same source file, but across multiple files. We achieve this by providing a module system, which allows (selectively) importing functions from other modules.

As it is a good practice to keep functions small and focused, we allow for very short function notations. The shortest functions can be defined in a single line, without any braces. For convenience, these short functions can be written in statement-form or expression-form, depending on the operator used. The former is used for short functions that do not return anything, but instead perform some side-effect, while the latter is used for short functions that return a value.

We give some examples in listing 3.1, showing functions and globals being used across multiple files which correspond to modules.

```
/// file: main.gzb
from my_mod use step,
                counter as cnt           // local import renaming is supported too
func increment ⇒ cnt += step(cnt)        // short function in statement-form
/// file: my_mod.gzb
from text use say        // resolves to the standard library
counter := 0
func step → Int = 3      // short function in expression-form
#tick func my_loop       // registering in tag #minecraft:tick causes the function
{                        // ... to be called every game tick (20 times per second)
  main~increment()       // use-statement not necessary if a relative path is used
  say_if_even(counter, "Counter is even")
}
func say_if_even(v Int, msg String)
{
  if v % 2 == 0
  {
    say(msg)
  }
}
```

Listing 3.1: Functions and globals with modularity

3.1.2 Global Variables

Similarly to how functions can be declared in a module and referenced from others, we also allow this for variables. Those which are put at the top-level of a module thus become global variables, which can be imported from elsewhere.

3.1.3 Selectors and Aliases with First-Class Data Access

Selectors are a very powerful core feature of Minecraft, being the mechanism to query entities in the world. Their usage, however, is relatively cumbersome, especially when they are used in non-trivial ways with filter constraints.

Although the game offers a handful of built-in selectors, it is often necessary to specify queries further with these constraints. In situations where such a query, or slight variations thereof, are done often and in multiple places, it is not desireable to have to write the same query over and over again. Not only is this error-prone, but is also degrades readability.

This is why we offer selector aliases, a way of defining a reusable selector that already applies some filter constraints. These aliases may then be used similarly to how one of the built-in selectors would be used.

Entities queried with selectors are always used to perform some action on. Some of these actions are related to data access, such as retrieving or storing some entity data. In plain Minecraft, there are dedicated commands to achieve this. In Gazebo however, we make it possible to use selectors as proper L-values, allowing relatively concise data access expressions. An example of this can be found in listing 3.2, including a demonstration of the selector alias feature.

3.1.4 Easier Control Flow

Strictly speaking, Minecraft offers no control flow. Instead, it offers one command which can conditionally execute another command. Usually, the chained command would invoke an-

```
alias @MyCreeper = @e[type == $creeper, is "super_secret_tag"]
func configure_my_creepers
{
  as @MyCreeper
  {
    @s.ExplosionRadius = 100  // fully type-safe entity data operation
    // ^^^^^^^^^^^^^^^^^—— refers to a key in the entity definition
    //                     which is inferred via the contextual selector @s
    //                     which in turn is known to point to entities
    //                     of instance $<minecraft:entity>minecraft:creeper
    @s.NonExisting = "test"  // error: key "NonExisting" does not exist
  }
  // double the explosion radius for two random entities
  as @MyCreeper[sort = "random", limit = 2] ⇒
    @s.ExplosionRadius *= 2  // @s refers to the entity of the current context,
}                            // as the body block is executed for each match
```

Listing 3.2: Selectors and aliases with first-class type-safe data access

other MCFunction, but the body of that inherently must reside in a separate file. To improve on this situation, we offer proper control flow statements, such as if-else branching, loops and case-match statements. All of these are demonstrated in listing 3.3.

3.2 Type System

Generally speaking, Minecraft does not perform any type-checking. It does perform some basic sanity checks while parsing, but that is everything. If there are any errors, they will only be detected at runtime, if at all. Although there is no undefined behavior or a way to crash the game due to these mistakes, it could still result in unexpected behavior.

In Gazebo, all programs are required to adhere to much stricter typing rules. All data is strongly typed, thus guaranteeing most allowed operations theoretically safe. The type system follows a structural approach with recursion support, on which we elaborate in the next section.

The type system defines a set of rules that constrain which operations are allowed on which data, but also defines how data is related to other data. We define all rules with Statix, applying the scopes-as-type paradigm (Antwerpen, Bach Poulsen, et al. 2018). This approach suits our needs well, as we use a structural type system. The reason for this is that a it is the closest safe way of modelling the actual runtime behavior of Minecraft. The only constraint that is imposed on us, is that there is no any-type, requiring each possible location to be typed and initialized with a default value. For the representing syntax, we took some inspiration from Go.

Although most typing rules are intuitive, there are some features that are worth discussing in more detail. Before we can do that, we need to give a brief overview of types that are available.

3.2.1 Type Classes

Table 3.1 lists all available type classes and their corresponding syntax. Additionally, for the position primitive, the symbol between ` and T has a significant meaning. Depending on what it is, it means the following:

```
func example
{
  a := 4

  if a > 3 ⇒ say("a is greater than 3")
  else
  {
    // ... something else
  }

  for a > 0 ⇒         // conditional loop ("while")
    a -= 1

  for i ← 0..10 ⇒     // range-based loop ("for-each", "iterator")
    a += i

  for ⇒               // infinite loop
    say("Infinite Power!")

  match a
  {
    0 ⇒ say("zero")
    1 ⇒ say("one")
    _ ⇒ say("something else")
  }
}
```

Listing 3.3: Conventional control flow in Gazebo

- *none*: absolute position;
- ~: may contain parts relevant to cartesian coordinates;
- ^: may contain parts relative to facing-direction;
- *: do not care. All of the above are subtypes of this.

An instance of a type of a particular class is either derived ad-hoc from an expression (and is in fact done so for each expression), or explicitly given as part of a function signature or type alias in the aforementioned syntax. It is thus not possible to instantiate an empty variable by only providing its type; every value's type must be derived from its concrete value. Table 3.2 lists some examples per type class and their exact derivation. Still, if it is desired to self-enforce a particular type of a variable, it is possible to do so with the :: symbol, which plays an important role in the language.

3.2.2 Type-Ascribed Construction (::)

As we mentioned earlier and have shown in table 3.2, it is not always possible to get a value of a particular type without contextualizing the expression. This missing context can be provided by the typed-ascribed construction symbol (::), which can be used create instances of three different type classes in two different ways.

The :: symbol, signifying a type ascription to an expression, is placed in between a type and and a construction expression. The type can be any valid type, but the syntax of the

Class	Representation	Remark
Alphanumeric primitive	`String`, `Bool`, `Long`, `Int`, `Short`, `Byte`, `Double`, `Float`	
Position primitive	`` `T ``, `` `~T ``, `` `^T ``, `` `*T ``	`T` is normally `Int` or `Float`.
Resource registry	`$<namespace:path~to~registry>`	
Function tag	`#`	
Unconstrained selector	`@`	
Array and list	`[T;]`, `[T]`	All operations are equally valid on both arrays and lists, but in general they are not interchangeable.
Enum	`enum {A, B}`, `enum T {A= ... , B= ... }`	Without `T` provided, `Int` is assumed and values are auto-numbered.
Interface / compound	`interface {key1 T1 ... }`, `interface : A, B, ... { ... }`	Overlapping composed members' types are merged by intersection (L.U.B.).

Table 3.1: Available type classes and their syntactical representation

Class	Expression	Derived Type
Alphanumeric primitive	`.5f`, `"Hello"`	`Float`, `String`
Position primitive	`` `(1 0 ~1) ``, `` `(^ ^-.3 1) ``	`` `~Int ``, `` `^Double ``
Resource registry	`$stone`, `$<item>stone`, `$dolphin`	*error—ambiguous*, `$<minecraft:item>`, `$<minecraft:entity>`
Function tag	`#minecraft:tick`, `#:my_tag`	`#`
Unconstrained selector	`@e[level > 5, is "tagged"]`, `@s`	`@`
Array and list	`[; "Hi"]`, `[1, 2.3]`	`[String;]`, `[Double]`
Enum	*cannot be instantiated ad-hoc without a type ascription* (`::`)	
Interface / compound	`{a: [1], "b c": 2}`	`interface {"a" [Int] "b c" Int}`

Table 3.2: Example expressions for each type class and their derived types

```
type MyEnum enum { A, B }
func my_enum_equal(p, q MyEnum) → Bool = p == q
func caller
{
  b := MyEnum::B
  my_enum_equal(::A, b)
  //            ^^^——— inferred to be MyEnum::A,
  //                   because my_enum_equal expects a MyEnum
}
func local_enum
{
  p := enum{X,Y}::X
  p = ::Y //--^ bound to locally defined enum key, with click-through support
  //  ^^^——— inferred to be <anonymous_enum>::Y,
  //           because `p' is of type <anonymous_enum>
  match p
  {
    ::X ⇒ say("X")
    ::Y ⇒ say("Y")
    ::Z ⇒ say("Z")  // error: key "Z" does not exist on enum
  }
}
```

Listing 3.4: Constructing and using enums

right-hand side expression is constrained. If no left-hand side type is provided, the ascription works as if the type that is expected at the current position is used. The benefit of this will become clear in the examples.

The first type classes that can be constructed, are arrays, lists and interfaces. Support for these classes is necessary in order to propagate the type context to subexpressions that may be used in list or array members, or in values of interface members. For interfaces specifically, this is also a way of moving errors closer to the offending location, as the compiler can now check whether some keys are missing or superfluous *within* the interface, instead of only after a full type has been derived.

The other other possible type class to construct is the enum class. To construct an enum instance, the right hand side of the ascription is just the name of the enum member. With this, it is also possible to use a nameless scoped enum. Both use cases are demonstrated in listing 3.4.

The third and final use-case for the type ascription is to construct arrays, lists or interfaces with default (empty) values. This is possible because each type has a determined default value. Using the ascription like this, is also the only way to pre-allocate slots in an array or list. The operation is demonstrated in listing 3.5. Note that for arrays and lists, the number of slots has to be provided, whereas for interfaces, nothing can be provided.

3.3 Transformation Architecture

As we have seen in chapter 2, the compiler internally deals with a number of different intermediate languages to get the job done. In this section, we explain some of the reasoning behind the design of these languages and their purpose. Where relevant, we mention more details about the transformations that are responsible for transforming the input to the output form.

```
type MyStruct interface { a Double }
func default_array_and_list
{
  default_array := [String;]::(4)  // value: [String; "", "", "", ""]
  default_array = ::(2)            // value: [String; "", ""]
  default_list := [MyStruct]::(2)  // value: [{"a":0.0}, {"a":0.0}]
}
func default_interfaces
{
  dynamic_instance := { a: 5 }     // inferred to have type interface { a Int }
  dynamic_instance = ::()          // reset to default value: { "a": 0 }
  dynamic_instance = ::{}          // error: missing value for key "a"
}
```

Listing 3.5: Constructing default values for arrays, lists and interfaces

The implementations of all the transformations described in the following subsections can be found in the source code repository[1] in the following respective directories: `ext/ext.{gzb2gzbc,gzbc2llmc,llmc2mcje}/trans/`. Some common transformation strategies can be found under `str-common/trans/`.

3.3.1 Gazebo Main to Gazebo Core

The surface language, as discussed in the previous section, contains a relatively high amount of syntactic sugar. The very first step is to desugar this. Although desugaring can normally operate fully within one syntax, it is beneficial to have a dedicated core language to desugar to. This enforces a bit more simplicity and regularity in the format of the desugared code, which makes it easier to process further on, without the possibility of confusing non-desugared code with desugared code.

The step from GZB to GZBC is relatively small for most parts. Where the majority of transformations are spent on, is extracting type information that was inferred during the static analysis phase in Statix. We decided to make GZBC a language of which the files are fully self-contained. There are two reasons for this. First, it makes it theoretically trivial to process files in parallel. Second, it requires no static analysis to be performed, because all type and name binding information is either present in the file, or can be trivially inferred. Not wanting to perform static analysis is especially relevant with regards to the current performance characteristics of Statix.

3.3.2 Gazebo Core to Low-Level Minecraft Commands

The GZBC AST received as input in this transformation stage is still relatively high-level and feature-rich. LLMC on the other hand, is supposed to be a very low-level abstraction of the Minecraft's commands, as the name suggests.

The first step in this transformation is to once again apply desugar transformations. Although some of these could have been applied in the previous stage already, the desugar transformations in this stage are more lossy in terms of context information and already geared towards the LLMC format. There are three major parts which are desugared in this stage: for-in loops to conditional loops, match statements to if-else chains and selector mixin lifting and reduction. For the last part, we given an example in listing 3.6.

[1] `https://github.com/MetaBorgCube/gazebo`

```
// Gazebo Main
alias @First = @e[is "example"]
alias @Second = @First[level > 5+6]
const my_sel := @Second
// ⟶ Gazebo Core (aliases transformed to mixins)
mixin First [tag = "example"]
mixin Second [level > 5I + 6I]
const my_sel : @ = @(e, First, Second, [], __UNK)
// ⟶ Gazebo Core (mixins desugared, non-literal expressions lifted)
const _mixin$Second$level0 : __UNK = 5 + 6
typeof :(ns, name _mixin$Second$level0) = __UNK
const my_sel : @ = @(e, [tag = "example", level > _mixin$Second$level0], __UNK)
```

Listing 3.6: Desugaring of selectors and mixins in GZBC

The transformations leading up to the output in LLMC can now be started. At the top-level of an GZBC file, there are only three types of items remaining: function definitions, global variable definitions, and accompanying global variable type declarations. After passing over the type-of declarations and recording them, there are only two remaining.

All globals have to be initialized at some point, as there is no native concept of globals. Although global variables are properly scoped by the module name, they do not exist unless explicitly created. After initialization, it is sufficient to just know about the address of the variable. For this reason, global variable definitions can also be translated into a function definition, namely their initializer. This is a special function that cannot be called externally, but is automatically called during initialization, by registering it to the tag `#minecraft:init`.

Now we are left with just functions. These in turn exist of statements, containing expressions or other statement blocks, possibly executed conditionally. At this point, the design of LLMC dictates the remaining transformations: it only knows about functions, each of which lives in a separate file.

To allow for simplified processing in the next stage, we decided to explicitly include the concept of *basic blocks*, blocks of continuous statements, in LLMC. To allow control flow between basic blocks, we introduce a meta-block construct, called a *flow group* existing of one or more *flow blocks*. A flow block is a condition that determines whether an encapsulated block is executed. This body block may in turn be another flow group, or just a basic block.

Note that basic blocks cannot contain anything else than statements. As mentioned earlier, this enforces a very strict view of what is executed together and what is not. Also note that a flow block that executes its encapsulated block will always return and execute the next flow block. This is analogous to how Minecraft executes nested blocks, causing the logic for, for example, if-else statements to be implemented in the transformation from GZBC to LLMC.

We constrain ourselves even further by requiring all statements in a basic block to be in the form $d \leftarrow a$ or $d \leftarrow\!\shortmid\ o\bar{a}$, where d is a destination (or L-value), a is an argument (or R-value), and o is an opcode. The first case is a trivial reassignment, while the value stored in the destination for the second case depends on the combination of operator and arguments. There are only a few opcodes necessary to implement the entire feature set of Gazebo: arithmetic (`add`, `sub`, ...), element counting (`count`), data access (`find`), data deletion (`del`), empty data construction (`new`) and function invocation (`ivk`). If necessary, this format can trivially be converted into a static single assignment (SSA) form, which can be helpful for further optimizations, although it would require the addition of the Φ function as an opcode.

Arguments in LLMC are the result from transforming GZBC expressions. Most expressions correspond cleanly with one or more LLMC arguments, but some require a more com-

```
/// Gazebo Main
func say_list(items [String]) ⇒ for s ← items ⇒ say(s)
/// Gazebo Core
func say_list(val items [String]) → __VOID
{
  iterate items as s {
    discard :(minecraft, text say)(__VOID; text: String = s);
  };
}
/// Gazebo Core (desugared)
func say_list(val items [String]) → __VOID
{
  var flag : Byte = 1B;
  var subj : [String] = items;
  loop flag {
    if __count(subj) (Int)>(Int) 0I {
      val curr : String = subj.[0I];
      __del subj.[0I];
      val s : String = subj;
      discard :(minecraft, text say)(__VOID; text: String = s);
    } else {
      flag = 0B;
    };
  };
}
```

Listing 3.7: Transformation of a function definition from GZB to GZBC, continued in listing 3.8 to LLMC

plex treatment, such as lifting computations into several operational assignment statements. Note that logic operations are not implemented as opcodes; instead they are LLMC arguments, because they can also appear in the condition of a flow block.

As we cannot discuss all transformations, we show a reduced example of a function definition in GZBC and its resulting LLMC representation, in listings 3.7 and 3.8.

3.3.3 Low-Level Minecraft Commands to MCFunction Format

Although LLMC was designed to closely resemble the MCFunction format, this last transformation phase is not trivial. The majority of the transformations are concerned with figuring out how data is actually stored, moved around, manipulated and persisted.

For the emission of commands, we rely heavily on Stratego's *dynamic rules* extension (Bravenboer, van Dam, et al. 2006). In contrast to the previous phases, the actual order of execution is determined here. The way we use dynamic rules requires the use of regular collection points, where auxiliary commands are wrapped into a single atomic set of commands, called *units*. This leads to a lot of nesting, as each of these collection points conceptually corresponds to the smallest scope that exists, and every wrapping scope thereof.

As this behavior is undesirable, we abstract the emission of commands into yet another small assembly language. With this, the transformations from LLMC can be kept at a slightly higher level, as the assembly format only encapsulates the bare minimum of MCFunction structures.

The final step is to pass the assembly tree through a tiny optimization pass, which re-

```
/// Low-Level Minecraft Commands
function example:sandbox/say_list
// declarations with associated type, used to determine the stack frame layout
sign items l[s]  // signature declaration, the remaining are locals
local flag iB    local subj l[s]    local $if_not_taken0 iB    local $count0 iI
local curr s     local $find0 s     local s s
body
  { // unconditional flow block
    flag ← 1B
    subj ← items
  }
  loop flag = 1B {
    { $count0 ← count subj }
    if $count0 > 0I {
      $if_not_taken ← 0B
      $find0 ← find subj '[0]'
      curr ← $find0
      subj, '[0]' ←| del
      s ← subj
      _ ← ivk(?) minecraft:text/say (text s)@s
    }
    if $if_not_taken0 = 1B {
      flag ← 0B
    }
  }
```

Listing 3.8: Transforming a function definition from listing 3.7 in GZBC to LLMC, demonstrating how flow control in LLMC becomes tedious

moves most of the unnecessary unit nesting, before assembling every remaining unit into a MCFunction content string.

The outputted content string is prefixed with some metadata in a comment string, in the Interoperable Module Protocol (IMP) format. IMP is a collection of community-proposed standards for authoring data packs (Arcensoth 2020). Catering to the community, we try to stay compliant to this standard.

3.4 Stand-Alone Compiler

Knowing how a program written in GZB is to be transformed all the way down to several MCFunctions, we can start tying things together. An end-user, a Gazebo programmer, should be able to make use of our compiler just as easily as they would use any other. Practically all compilers are distributed to these users as either a stand-alone executable or a library that can be imported. Usually the former is a convenience CLI wrapper around the latter, as is be the case for us.

Although there are many interesting aspects of our stand-alone implementation, we cannot go into all detail here. For a deeper dive into the implementation, including some drawings and code examples, see appendix B.

3.4.1 Goal

Although Spoofax is extremely feature-rich, it does not provide all features out-of-the-box in a single package that one might need to implement a fully stand-alone CLI such as Gazebo stand-alone (GZBS). Sunshine[2] is one of the efforts of the Spoofax Team to provide CLI support. Unfortunately, it is not actively maintained nor well-documented, and only implements a basic feature set. On the other hand, the Spoofax Core API is very extensive. Sunshine can be regarded as more of a tool for very specific use-cases of the API instead. We will discuss this situation in more detail in section 5.2.2.

End-users of a programming language cannot be expected to program inside the *Spoofax Eclipse* environment, even though that may be the easiest way to get the full experience. This is a full-blown IDE which is not only resource-intensive, but also may be too complex and too steep of a learning curve for some users, as not everybody can be expected to be familiar with Eclipse. What *is* expected by end-users is that the language *at least* provides a CLI; IDE support is generally regarded as a quality of life feature not strictly necessary. In order to fulfill this expectation, GZBS was developed.

With the modularity spirit of Spoofax in mind, a design goal of GZBS was to achieve modularity, which is detailed in the next section. It has practically already proven useful. The heavy-lifting of the compiler is done in a reusable library component, which has been used in practice to construct a graphical user-interface (GUI) for demonstration purposes at the Honours Symposium, a poster-presentation session, held in March 2022. This particular GUI was created in just a few hours of quick work, only relying on JavaFX as other dependency.

The other strong reason for implementing this sub-project was to run tasks alongside Spoofax which are not part of a typical Spoofax pipeline. Packaging all outputs into a final ZIP archive, with appropriate meta-files, is the primary use-case for this. Another one is gathering *Statix* Libraries (StxLibs) for standard library packaging, as is described in appendix C.

3.4.2 Languages, Build System & Frameworks

All code of GZBS is written in the Kotlin[3] programming language. The Gradle build system is used to produce Java Archives (JARs) and corresponding executables. The choice for Kotlin was by no means necessary, but demonstrates that the Spoofax Core API is perfectly usable from other Java Virtual Machine (JVM) languages as well. Using Gradle, instead of Maven, is a logical consequence of choosing Kotlin, as it integrates most natively.

Other than Spoofax Core there is only one other direct and one indirect dependency that is used actively: Picocli and Guice. Picocli is a helper library for command-line applications. Guice is a dependency injection framework, already used by Spoofax Core, requiring GZBS to use it as well.

3.4.3 Wiring Things Up

Spoofax projects are packaged into `.spoofax-language` files, conventionally called language archives. Although the name suggests that such a package only serves for 'real' languages, it is perfectly possible to only use a language archive for general utilities, or in this case, for transformations between the different intermediate languages.

In chapter 2 the complete pipeline of the compiler was given. Each part of the compiler seen from that perspective corresponds to one language archive. When used from within the *Eclipse* environment, some actions such as static analysis are performed automatically, whereas others need to be invoked manually by the user via the menu.

[2] `https://github.com/metaborg/spoofax-sunshine`
[3] `https://kotlinlang.org/`

Wiring up these language archives and all corresponding actions between them is the task of GZBS, similarly to how Sunshine did it. Fortunately, the heavy-lifting is implemented by Spoofax Core's build system. However, it still needs to be informed and configured, for example about where these language archives reside, and how to load them.

Chapter 4

Project Evaluation

The Honours Programme for Bachelor students of the Computer Science and Engineering programme here at Delft University of Technology aims to provide students ways of broadening their knowledge and skills. Not only in the field of Computer Science, but also interdisciplinary, for which several courses were offered and taken.

This report only covers the faculty part of the programme, which is a scientific project in the broadest sense. Normally, we would have to write a scientific article of four to six pages, reporting on our findings of the work. However, specifically applying to our project, we were allowed to write a report instead, "detailing what [we] have done and learned".

Regarding to what has been done, that is the main body of the report and its appendices, but regarding to what we have learnt, we will now discuss that now in more detail. Following that, we will put the project as a whole into more context, discussing some miscellaneous important project-wide concerns.

A formal evaluation of the project and our final work has been done by our supervisors, but is not covered in this document.

4.1 Learning Points

The most obvious and important learning point is the fact that I have learned a lot about how compilers *actually* work. Although this experience occurred mostly within the bounds of Spoofax, I still think it has been a very valuable experience, broadening my knowledge significantly.

From working directly with the Spoofax Core API, over the course of the entire project, I have also picked up a lot about Spoofax's internal design. Even though I already have several years of professional software development experience, I had never worked with a project of this size and complexity before. Again, from an insight-broadening perspective, this has been very valuable.

During the project, but especially in the beginning and at some point towards the end, I participated in a significant fraction of Spoofax Progress meetings. Similar to the Slack channel, these meetings were a great way to get to know the Spoofax team. Although I have not worked with anybody together intensively or implemented new features in Spoofax itself, these meetings and the Slack channel both have been good resources to know who was working on what and what the (short-term) roadmap looked like.

Similarly, in the very beginning of the project, even before any serious implementation work had been done, I have read a lot of the Spoofax documentation and backing scientific publications. Simply diving right into the matter has been a good way for me to gain a better understanding of Spoofax, its history, its design and implementation specifics. Of course, it was not relevant for me to know every detail about the different meta-DSLs and their implementation, but it was very useful to know –in broad terms– what was possible

and what was not, what they were intended for and what they were not, giving me a solid basis to start working on the project.

4.2 State of Completion

Unfortunately we were not able to bring the project to a full state of completion. In this report, we have discussed many features that are implemented and working to some degree, but there were many more features that we initially planned on. However, in hindsight it was too ambitious from the start. With the unfortunate passing of Eelco, the project also came to a halt for a significant period of time, after which it was difficult to pick up the project again. Still, were this to not have happened, we would not have been able to implement all features anyway.

The current state of the project is that it is usable to some degree. From start to finish, the full compilation pipeline can be run, with some inputs producing actually usable outputs which can be loaded into and executed in the game. With the help of this document to convey some of the ideas, and the code itself to showcase an implementation with several examples to illustrate concepts even further, the project is in a state where it could be used as a basis for further development. Although the idea of compiling to a game engine may not be particularly useful in practice, the project as a whole serves as a possible real-world example of how to use Spoofax in a significantly more elaborate way than examples in the documentation show.

4.3 Minecraft Support Considerations

Minecraft can be found on may different platforms, but not all features are shared among them. The game was initially launched exclusively for the Java platform, but soon Mojang released *Minecraft: Pocket Edition* (MCPE). In 2017, a major rebranding took place, causing MCPE to now be known as *Minecraft: Bedrock Edition* (MCBE). The 'original' Java edition became known as *Minecraft: Java Edition* (MCJE), to which we have been referring to throughout this report.

These two editions of the game are developed by separate teams within Mojang. Although the feature disparity has been largely resolved over the years, implementation details still differ significantly. We do not expect this to change anytime soon, as there does not appear to be any reason for doing so.

Hence, we had to make the choice of which edition of the game to support. There was not much of a choice, as our personal experience was almost exclusively with *Java Edition*, and not so much in the *Bedrock Edition*. Another reason is simply that the technical community around MCJE is much more active and developed. For example, tooling, not necessarily related to programming languages that target the command system, is well-established: community projects such as *Data Pack Helper Plus* (SPGoding 2019), an extension for VS Code or *Beet* (Berlier 2020), a data pack assembler written in Python, are widely used by the 'classic' developers.

Chapter 5

A Review of Spoofax 2

Finally, in this last chapter, we will discuss our experience working with Spoofax 2. All suggestions or comments we make have been checked with the responsible authors before publication, whether or not they are valid and grounded. The order of sections, each dedicated to a different area, is relevant in the sense that we believe the first subsections are the most important. We only cover topics that were related to, or that were noticed during, the development of Gazebo. Moreover, some problems described here are in fact already solved in Spoofax 3.

5.1 Architectural Limitations

Spoofax 3 is on its way, and for good reason. The stability of Spoofax 2 has served us well in the past, and will probably continue to do so in the following years, especially in enterprise environments. However, there are some limitations that we currently either have to deal with, be it using a workaround or simply not using it at all. Additionally, both for performance reasons and pure architecture sake, we have some comments.

5.1.1 Multi-File Transformation

Stratego, the meta-DSL used to write AST-level transformation in, is designed to work with one input and one input only. On several occasions, we have noticed that sharing state between several files is not trivial this way. Statix, for example, *does* support multi-file analysis. However, it is not implemented in a reusable way, but built for its very specific use-case. It has already resulted in some confusing bugs[1].

Whole-program optimizations are thus not possible in normal Stratego. Even with manual workarounds, different from how Statix has actually solved it, are suboptimal. This is very unfortunate, as Hendrik van Antwerpen stated: "There are no true multi-file transformations. They would indeed be very useful, but I expect we have to wait for Spoofax 3 before we get that."[2]

5.1.2 The Central Place of ESV

In several areas, Spoofax Core is seemingly coupled to support editors in general: it consists of many facets and services, some of which are dedicated to editor support, even though they may never be used. A language archive's configuration is largely defined in terms of its ESV specification, which is intentional (Kats, Kalleberg, and Visser 2010). The primary

[1] See, for example, `https://github.com/metaborg/nabl/issues/94`

[2] `https://slde.slack.com/archives/C7254SF60/p1612776974137000?thread_ts=1612769297.136500&cid=C7254SF60`

configuration is a neutral YAML file, while the useful configuration has to be defined in the ESV file. This, in turn, is parsed in terms of different 'facets', which define some particular (class of) functionality of a language, for example, which action should be run whenever a file is saved.

Although ESV can be looked at as simply a misnomer for an evolved subsystem, that should instead be called something more appropriate, such as 'Spoofax Runtime Configuration Language', it still is a singular and very centralized part of Spoofax. Due to the static nature of facets, Spoofax Core contains many which are, arguably, not related to core functionality at all. Ideally, all non-essential facets should be part of separate modules. In turn, not everything should be defined in ESV format, requiring the language packaging process to be more flexible as well. Such a design would allow for a much more modular core design, which is in the spirit of Spoofax. In our case, we would simply not have to load all facets and related services for features that are not related to a stand-alone compiler, resulting in a smaller footprint overall.

The interdependence of some ESV features with non-editor related things manifests itself in some non-intuitive ways to configure a Spoofax language. The prime example is that all ESV files must be placed in the `editor` directory of a language project. This does allow the usage of separate files, to achieve some modularity. Still, options such as the start symbol of a language, the file extensions it supports, or configuration of the Stratego runtime do not make sense to be located here, as the connection with editor services is far-fetched.

5.1.3 Message Reporting

Most meta-DSLs emit some messages via the `IMessagePrinter` interface, often indirectly. Statix however in particular adds HTML content to the messages, which cannot be expected to be understood by all implementations of the interface. Within the Eclipse environment this is fine, as the GUI elements can be rendered with HTML. For our stand-alone compiler we needed to implement workarounds to make the messages properly renderable in a terminal. In general, it appears that the messaging system's architecture can use some improvement.

The usage of HTML is messages is not desired for the above reason, but especially because it is not a portable format. It may sometimes be desireable to add some markup or meta-data to messages, as this could enrich the user experience. Examples that come to mind are links to documentation, or the ability to link a piece of the message body to a specific source code location.

Another likely use case, at least within editor environments, is navigable (stack) traces, such as those produced by Statix. Currently, the trace is formatted as a fixed string, based on predetermined depth limits for the stack itself and the relevant context term. In an interactive environment, it would be much more useful to be able to explore this data interactively, for example by clicking on a term to expand it.

In any case, how feature-rich the messaging system may be, it must still be possible to format everything into a fixed string format. This is useful for command-line tools, but also for logging purposes.

5.1.4 Concurrent Task Execution

Concurrency, in general, is something Spoofax makes no use of. Only Statix's solver specifically implements a complete actor-based concurrency framework, which is used to parallelize the analysis of multiple files (Antwerpen and Visser 2021). Other parts of the Spoofax pipeline do not leverage any parallelism. It seems like an important aspect to focus on, as Spoofax is generally not known for its efficiency.

Even though Antwerpen and Visser (2021) have shown that modern compilers only parallelize a small fraction of their work, Spoofax delivers more than a compiler. It performs

housekeeping tasks, such as discovering input changes, that would conventionally be managed by a build system, while it also manages tasks that are very compiler-specific. Because Spoofax is partially a build framework, some tasks ought to be suited for concurrent execution, while the compiler-internal tasks may also be. Parsing seems to be the most obvious candidate: there cannot yet be any interdependencies between files. AST transformations in Stratego operate, as mentioned earlier, on a single AST. Assuming the strategies do not have any conflicting I/O side-effects, they should be fully parallelizable.

5.2 Using Spoofax in Stand-Alone Fashion

Most users of Spoofax will probably never use the API directly, however we did make extensive use of it in the implementation of GZBS (see appendix B). The comments in this section originate from our experience using and extending Spoofax Core.

5.2.1 Disk Access

Spoofax projects that use a multi-stage pipeline, write their intermediate results to disk. The build order is determined by Spoofax, such that the right files are written to disk before the next stage that depends on them is executed. From the perspective of the next stage, the outputs of the previous stage are indistinguishable from preexisting files.

In theory, this is great, as it allows for complete separation of concerns. However, the practical implementation leaves some things to be desired. We mention one optimization in section B.5, namely to skip intermediate parsing overhead. Another bottleneck may present itself in the fact that all these files are physically written to disk. For our intents and purposes, all intermediates could just as well be written to a temporary directory mounted in memory[3]. For some purposes, such as incremental compilation, it might be beneficial to permanently cache intermediate results.

We have tried to add native transparent support for in-memory files to Spoofax Core, but this proved to require changes in several areas. Ideally, the strategies that emit intermediate results are unaware of the fact that they are not writing their result to disk. With the current design, this is impossible, as the only way of indicating that a file should be written to a non-default location is by providing a strategy output location starting with `tmp:`. Security concerns aside, this requires changes from the strategy author, and it causes the transformation result to be placed at a location that is not managed by default, and thus to not scanned for source files for the next stage. Instead, we need to provide this location explicitly as a location to scan through the project configuration.

The convention is to put all temporary, generated or intermediate files in the `src-gen` directory, relative to the project root. A more transparent approach would map all files in the `src-gen` directory to a temporary location in memory, but this is a drastic change and not easily possible with Apache VFS. [4]

5.2.2 Spoofax Sunshine

For testing purposes, Sunshine is a neat tool. We have evaluated its usefulness early on in the development process, in order to determine whether or not it was necessary to write a new piece of software around the Spoofax API. Most features that GZBS, our implementation of a similar tool, supports, are also supported by Sunshine, because they are wrappers around Spoofax Core.

[3] Such as `/tmp` on Linux: `https://www.freedesktop.org/software/systemd/man/file-hierarchy.html`.

[4] An actual implementation would likely work with a custom filesystem implementation, which delegates some directories to an in-memory filesystem and the rest to the default filesystem. Another approach might use a mechanism similar to Linux's `overlayfs`.

Sunshine is generally a well-designed tool, and easy to use. Although it makes proper use of OOP principles, it does not allow for the functionality extension that we would have needed. This is due to the fact that command implementations are not designed to be extended, as they do not return any value or modify any class-local state. The reason for this seems to be coupling with derived local and remote classes for usage in the Nailgun framework[5]. We were especially interested in the instance of `ISpoofaxBuildOutput`, but this is kept as a local variable in the `BuildCommand` class, and never returned.

Furthermore, Sunshine is not open to extension of the build input[6], and thus we cannot modify any flags other than those exposed as command-line options. Part of the build input is logger configuration, which we also wanted to modify with respect to the defaults in Sunshine, but is also impossible.

At this point, we could either modify Sunshine and upstream the changes, or write a completely new implementation, without having to risk future impossibilities and need for upstream patches. As can be read in appendix B, we have actually chosen to implement it from scratch. The conclusion here is that it would likely be useful if Sunshine would be slightly more extensible.

5.2.3 Documentation on Internals

Implementing GZBS, based on Sunshine was a bit of a challenge. Although there was some additional documentation about the API[7], this was not complete. Especially when debugging, it was often required to delve into Spoofax's internals. Personally, that was not much of a problem, as we were curious to learn and able to understand it relatively easily. However, from a broader audience this cannot be expected.

Javadoc comments are placed in some places, but are often not really helpful or simply redundant. Although no comments are better than redundant comments, in many places where comments would be useful, there are none actually. Admittedly, this is a problem that a lot of software suffers from. It is difficult to say how much more documentation is required for the source code to become significantly easier to understand, but for Spoofax to gain more widespread adoption, some more focus on industry-grade (lean) software development practices might be beneficial.

[5] Used to run the application as a daemon to speed up repeated execution, because initialization overhead is only incurred once the server starts.

[6] The build input (builder) covers a large amount of configuration options, which are consequently passed as a big single object to the Spoofax builder.

[7] `https://www.metaborg.org/en/latest/source/core/manual/index.html`

Chapter 6

Conclusion & Future Work

We implemented a prototype of a new programming language targeting the Minecraft commands ecosystem, with the aim of exploring the possibilities of the game's platform and in order to evaluate Spoofax as a whole. We have extensively reported on both aspects in this document, including some more background information in the appendices.

Although this report does not describe the language in all detail, some parts of the language that are not mentioned are still missing. Initially the feature set seemed reasonable, but reality has proven more than once that the smallest features require the most work, as they interplay with the entirety of the system. Due to the fact that we had never worked with Spoofax before, and the fact that most people who do work with it have a gradual introduction, it was difficult to estimate the amount of work required for each feature in advance. Moreover, earlier this year, with Eelco Visser's passing, work on the project also temporarily halted. However, in the end, most features were implemented to the extent that we could report about them.

As we described in chapter 5, not everything was smooth sailing with Spoofax. Even though the community was very helpful, there were many things that we simply had to figure out for ourselves. As the project progressed, Spoofax as a whole became slower and slower, decreasing the work efficiency. We also spent a significant portion of our time on debugging Spoofax itself (and learning how to do so in the first place), searching for and reading through documentation, and understanding the workings of the Spoofax API.

We have designed our language as a three-stage pipeline, in which the surface language (GZB) is transformed and enriched into a core language (GZBC), which in turn is transformed into the LLMC format, which then is compiled into Minecraft commands. In the end, this turned out to be a reasonable approach, however it was probably not necessary to have the core language. The overhead of a fully separate language that basically shares all concepts with the surface language is not worth the benefit of separation of concerns: all enrichments and simplifications could just as well have been done within the surface language. The fact that the core language bears all type information in a fully explicit way is also wasteful in terms of information passed between stages, as it can add up quickly if more complex types are used often.

There are several other areas in which our language could be improved or extended. First, we currently do not enforce visibility and mutability modifiers, meaning that everything is public and mutable. Seconds, the LLMC can be extended with static analysis and optimization passes, of which inlining and partial evaluation are critical, as some features rely on them[1]. Third, as all features are currently implemented in a relatively straightforward way, it is likely that it is not efficient. The Minecraft platform is inherently a very inefficient target,

[1]For example, concatenating strings is not feasible to support at runtime. Similarly, functions that use raw statements, such as in the standard library, must be fully inlined as it is not possible to interpolate this at runtime.

so optimizations can be critical in determining whether using this language is feasible at all, hence we need to benchmark the performance. Fourth, a dependency or library management system would be useful to allow a more streamlined experience for users to reuse code.

The language we have implemented, is an imperative language. For most programmers and the target audience, this is the natural way of thinking about programs, however the functional paradigm is more interesting from a scientific perspective. All previous related work has been dedicated to imperative languages too. As such, there appear to never have been any attempts to implement a functional language targeting the Minecraft platform. The question remains whether this makes sense, and if it is possible to do so at all.

One of the goals of this project was to evaluate Spoofax and its ecosystem as a whole, from a programming language designer's perspective. Having bridged the initial learning curve, we can say that it is indeed a very powerful tool. Although we have not experienced the full breadth of all its meta-DSLs, everything that we did use was generally well designed and useful for its intended purpose.

Specifically Statix has proven itself to be extremely useful. Especially compared to its predecessor NaBL2, it is still relatively new in the ecosystem. The learning curve is mild, as the syntax is relatively simple, while there were initially not many examples available.

The other two meta-DSLs that we used, SDF3 and Stratego are also both very powerful. SDF3 (with JSGLR) was very intuitive to use with almost no learning curve, while Stratego was by far the most difficult to learn. Even with the help of extensive documentation, papers and examples, it was initially very difficult to understand the logical flow and interplay of strategies. However, once we understood all of this, it proved to be a useful tool for implementing our program transformations, especially with the Statix integration.

In the end, it is difficult to say whether Spoofax enabled us to implement our language faster than we would have been able to otherwise. Although the answer seems to be *yes*, it is difficult to quantify. As we had no proper prior experience in compiler construction, there is little experience to compare to. Having experienced and described that it can sometimes be a bit rough around the edges, it still feels like a proper tool for the job.

Bibliography

Antwerpen, Hendrik van, Casper Bach Poulsen, et al. (Oct. 2018). "Scopes as Types". In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276484. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3276484.

Antwerpen, Hendrik van and Eelco Visser (2021). "Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers". English. In: DOI: 10.4230/LIPIcs.ECOOP.2021.1. URL: http://resolver.tudelft.nl/uuid:ccd9fcfc-6cf5-4447-b8b4-b862ce1b4483.

Arcensoth (2018). *Version-controlled history of Minecraft's generated data*. URL: https://github.com/Arcensoth/mcdata (visited on 05/21/2022).

— (2020). *IMP: Interoperable Module Protocol design specification and recommendations for Minecraft datapacks*. URL: https://github.com/Arcensoth/imp-spec (visited on 05/14/2022).

Berlier, Valentin (2020). *Beet - The Minecraft pack development kit*. URL: https://mcbeet.dev (visited on 03/13/2022).

— (2021). *Mecha - a powerful Minecraft command library*. URL: https://github.com/mcbeet/mecha (visited on 03/13/2022).

Brand, M. G. J. van den et al. (2000). "Efficient annotated terms". In: *Software: Practice and Experience* 30.3, pp. 259–291. DOI: 10.1002/(SICI)1097-024X(200003)30:3<259::AID-SPE298>3.0.CO;2-Y. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-024X%28200003%2930%3A3%3C259%3A%3AAID-SPE298%3E3.0.CO%3B2-Y.

Bravenboer, Martin, René de Groot, and Eelco Visser (2006). "MetaBorg in Action: Examples of Domain-Specific Language Embedding and Assimilation Using Stratego/XT". In: Lecture Notes in Computer Science 1611-3349. Berlin, Heidelberg : Springer Berlin Heidelberg, pp. 297–311. DOI: 10.1007/11877028_10. URL: https://doi.org/10.1007/11877028_10.

Bravenboer, Martin, Arthur van Dam, et al. (2006). "Program Transformation with Scoped Dynamic Rewrite Rules". In: *Fundamenta Informaticæ* 69. 1-2, pp. 123–178.

Energyxxer (2019). *Trident - A programming language for Minecraft data packs*. URL: https://energyxxer.com/trident (visited on 03/19/2022).

Gnembon (2019). *Fabric Carpet - a mod for vanilla Minecraft that allows you to take full control of what matters from a technical perspective of the game*. URL: https://github.com/gnembon/fabric-carpet (visited on 11/26/2022).

Groenewegen, Danny M. et al. (2008). "WebDSL: A Domain-Specific Language for Dynamic Web Applications". In: *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*. OOPSLA Companion '08. Nashville, TN, USA: Association for Computing Machinery, pp. 779–780. ISBN: 9781605582207. DOI: 10.1145/1449814.1449858. URL: https://doi.org/10.1145/1449814.1449858.

Gromov, Anton (2014). *Redstone Programming Language*. URL: https://tossha.com/rpl/.

Haisma, M.A. (author) (2017). *Grace in Spoofax*. English. URL: http://resolver.tudelft.nl/uuid:76ab40b5-ccdb-4db1-95ef-db71c01e0c7f.

Hamey, Leonard G.C. and Shirley N. Goldrei (2008). "Implementing a Domain-Specific Language Using Stratego/XT: An Experience Paper". In: *Electronic Notes in Theoretical Computer Science* 203.2. Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007), pp. 37–51. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2008.03.043. URL: https://www.sciencedirect.com/science/article/pii/S1571066108001485.

Kats, Lennart C.L., Karl T. Kalleberg, and Eelco Visser (2010). "Domain-Specific Languages for Composable Editor Plugins". In: *Electronic Notes in Theoretical Computer Science* 253.7. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009), pp. 149–163. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2010.08.038. URL: https://www.sciencedirect.com/science/article/pii/S1571066110001179.

Kats, Lennart C.L. and Eelco Visser (2010). "The Spoofax language workbench". In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. ACM, pp. 237–238. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869592. URL: http://doi.acm.org/10.1145/1869542.1869592.

Simon816 (2017). *Command Block Assembly*. URL: https://github.com/simon816/Command-Block-Assembly (visited on 03/19/2022).

SPGoding (2019). *Data-pack Helper Plus*. URL: https://marketplace.visualstudio.com/items?itemName=SPGoding.datapack-language-server (visited on 03/19/2022).

Spoofax Team (2021). *Spoofax: The Language Designer's Workbench*. URL: https://spoofax.dev (visited on 03/19/2022).

Stevertus (2020). *McScript - a programming language for Minecraft*. URL: https://mcscript.stevertus.com/ (visited on 11/26/2022).

— (2022). *ObjD - a framework for developing Datapacks for Minecraft*. URL: https://objd.stevertus.com/ (visited on 11/26/2022).

Visser, Eelco (1997). *Scannerless Generalized-LR Parsing*. Tech. rep. P9707. Programming Research Group, University of Amsterdam.

— (2021). *A Brief History of the Spoofax Language Workbench*. URL: https://eelcovisser.org/blog/2021/02/08/spoofax-mip/ (visited on 05/21/2022).

Yurihaia (2019). *Machine and human readable documentation for all of Minecraft's in-game NBT data*. URL: https://github.com/Yurihaia/mc-nbtdoc (visited on 05/21/2022).

ZipKrowd Team (2016). *Commandblock Parser*. URL: https://web.archive.org/web/20190626002750/http://zipkrowd.com/tools.htm#cbp (visited on 05/07/2022).

Glossary

Apache VFS Virtual filesystem library for Java, supporting many different filesystems via a common API. 25

ATerm Data exchange format used in and around Spoofax. Primarily used in the Stratego meta-language to represent abstract syntax trees, optionally accompanied by annotations. 46

class path Locations where the Java runtime searches for files, such as classes, at runtime. Often used to load packaged resources. 46

command Line of text containing a single instruction for Minecraft to parse and execute. 1, 10, 15, 17, 27, 37, 39, 50

command block Building block in the game Minecraft, which can be used to execute a command. See `https://minecraft.fandom.com/wiki/Command_Block`. 1

Eclipse Platform for IDE development. See `https://wiki.eclipse.org/Platform`. May also refer to Eclipse IDE. See `https://eclipse.org/ide`. 2, 5, 19, 24

exit code Numeric value returned by a system process upon exit. Usually, a negative value indicates termination by a signal, zero for success, and postive indicating an error. 44

Gazebo Name of the project described in this report. v, vii, 2, 5–7, 9–12, 16, 18, 19, 23, 33, 35, 37, 39, 40, 43–46, 49

Gazebo Runner Part of Gazebo Standalone, wrapping most low-level interactions with Spoofax Core. 43, 45

Go The *Go* programming language. 11

Gradle Generic build automation system. See `https://gradle.org/`. 19, 46

Guice Dependency injection framework for Java. See: `https://github.com/google/guice`. 19, 43, 46

Java The *Java* programming language. 2, 6, 19, 33, 36, 43, 46

Javadoc Standardized format for documentation comments in Java code. 26

JavaFX Graphics framework for Java. See `https://openjfx.io/`. 19

JSGLR Java implementation of the SGLR algorithm (Visser 1997). The default parser used in Spoofax, relying on parse tables generated by SDF3. 6, 7, 28

Kotlin The *Kotlin* programming language. 19

language archive ZIP-compressed package containing a compiled Spoofax project. 19, 20, 23, 43, 45, 46, 50

Maven Generic build automation system. See `https://maven.apache.org/`. 19

MCFunction Text format for batch execution of commands. 1, 8, 11, 17, 18, 40

Minecraft Popular video game. See `https://minecraft.net/`. i, 1, 8–11, 15–17, 22, 27, 28, 35

NaBL2 Spoofax's Name Binding Language ('enable'), version 2. 28

namespaced ID style of idendifiers used in Minecraft to identify arbitrary objects, always within some given context (such as functions, tags, blocks). 35

Picocli Command-line arguments parser and dispatcher. See: `https://picocli.info/`. 19, 43

protocol ID Numeric identifier used by Minecraft to uniquely identify a thing within some context. Usually only used on the wire. 49

Python The *Python* programming language. 22, 35, 37, 50

selector Query mechanism to select a subset of entities in a Minecraft world. vii, 10, 16, 40, 41

Spoofax The Spoofax Language Workbench. v, 2, 3, 5–7, 19, 21–28, 43–46, 50

Spoofax Core Collection of Java libraries which contain the parts of the Spoofax Language Workbench which are not directly related to (implementation details of) a frontend, such as an IDE. 3, 5, 19–21, 23–25, 43–46

Statix Meta-DSL for static semantics specification. See `https://www.spoofax.dev/references/statix/`. 2, 6–8, 11, 15, 23, 24, 28, 50

Stratego Meta-DSL for program transformations. See `https://www.spoofax.dev/references/stratego/`. 2, 6, 7, 17, 23–25, 28

strategy Unit of transformation logic in Stratego. 6, 8, 25, 28

Sunshine Project adding command-line support to Spoofax. See `https://github.com/metaborg/spoofax-sunshine`. 19, 20, 25, 26, 46

VS Code Visual Studio Code. Popular open-source IDE by Microsoft. 22

ZIP Archive compression file format. 8, 45, 47

Acronyms

API application programming interface

AST abstract syntax tree

CLI command-line interface

DSL domain-specific language

ESV Editor Services

GUI graphical user-interface

GZB Gazebo surface/main syntax

GZBC Gazebo core syntax

GZBS Gazebo stand-alone

IDE integrated development environment

IR intermediate representation

IMP Interoperable Module Protocol

JAR Java Archive

JSON *JavaScript* Object Notation

JVM Java Virtual Machine

LLMC Low-Level Minecraft Commands

NBT Nabed Binary Tag

SDF3 Syntax Definition Formalism 3

SGLR scannerless generalized LR

SPI service provider interface

SSA static single assignment

StxLib *Statix* Library

Appendix A

Language Design & Features

In this appendix, we cover the feature set of the surface language, GZB, in more detail compared to chapter 3. We divide the feature set into four major different categories: project/-module structure, expressions, statements and the file structure. Additionally, some features related to internals are discussed, such as registries and global aliases. Each feature, or several related features, are illustrated with a small example and a rationale.

A.1 Modules: Project Directory and File Structure

In order for a Gazebo project to be compiled properly, it must follow a certain file structure. Our intention is to allow cooperation with existing Minecraft project structures, which is why we stick to the convention of thinking in namespaced ID-oriented terms. A typical project structure is shown in listing A.1. Here, the files `a.gzb` and `b.gzb` have the full module names `my_namespace:a` and `my_namespace:my_nested_module~b`, respectively. If for some reason a different module name is desired, it can be changed by explicitly specifying it in the GZB file on the first line, for example, `module my:override~module`.

The differences between namespaces, names and modules are subtle, but important. A module is a Gazebo concept, while namespaces and names are Minecraft concepts. A namespace with a name combined is called a namespaced ID, which, when used in the context of Gazebo files, refer to modules. The namespace part is separated from the name part by a colon (`:`). So, modules are identified by a namespaced ID, but an arbitrary namespaced ID does not necessarily refer to a module.

Modules are useful for grouping re-usable code, as well as for proper organization. Therefore, modules can import members from other modules. There are two ways of accessing members of other modules: explicitly importing them, or referring to them by their module-qualified name.

Explicitly importing module members can be done in several different ways, comparable to the import syntax of Python. Examples of the import syntax are shown in listing A.2.

```
project_root/
  data/
    my_namespace/
      gazebo/
        my_nested_module/
          b.gzb
        a.gzb
```

Listing A.1: Typical project structure of a Gazebo project

```
use other_ns:path~to~member       // becomes available as 'member'
use some_ns:a~b as custom_name    // becomes available as 'custom_name'
from relative_path use a, b as c  // become available as 'a' and 'c', respectively
from :rel~to~root use *           // all members from :rel~to~root become available
```

Listing A.2: Explicitly importing members from other modules

From these examples, it can be seen that there are not only different ways of expressing *how* to import members, but also different ways of expressing *where* to import from. Fully qualified identifiers are written with a namespace and name part, separated by a colon. However, both the namespace part and the namespace part in conjunction with the colon are optional. Their meaning is context-sensitive, as the following rules apply, independent of the import syntax:

- If the namespace part is omitted, the current namespace is used. For example, `:a~b` resolves to `current_namespace:a~b`.
- If the namespace part including the colon is omitted, the member is resolved relative to the current module. However, if that appears to be invalid, it is resolved relative to the `gzb:` or `minecraft:` namespace instead (in that order). For example, `a~b` resolves to `current_namespace:current~module~a~b`, while `text~say` resolves to `minecraft:text~say`, assuming there is no relative module named `text`.

Only the first rule applies to the ad-hoc module member reference syntax. This less recommended syntax can be used to refer to members of other modules without having to first import them. Use-cases of this syntax will become clear in the following sections, as we show more examples.

A.2 Body: Expressions

Starting from the expressions we gradually build up to the file structure. In order to be able to actually write any statements, we need to know which expressions are available and how they are supposed to be used.

A.2.1 Literals and Named Binary Tag

All literals in GZB follow the same syntax as that from the game, which effectively is Java's syntax for string and numeric literals. One exception is the syntax for booleans: the game does not have a concept of boolean values, but represents them as byte values of 0 and 1. Therefore, the boolean type and syntax in GZB is purely syntactic sugar, as they are translated to byte literals in the end. They do still serve a purpose, as integral values are not valid to be used interchangeably with boolean values.

The data structures lists, arrays and compounds are also fully inspired by the game's syntax. The notations for lists is intuitive, but the notation for arrays is a bit more exotic: after the opening bracket a semicolon indicates the fact that this is an array. This notation corresponds to the type syntax, but without the type explicitly specified, as it is inferred from the array's elements. Compounds are again straightforwardly modelled after *JavaScript* Object Notation (JSON).

These two clusters of literals and data structures are commonly referred to as a textual representation of the Nabed Binary Tag (NBT) format, a JSON-like binary data format, used by the game to serialize the world state for savegames. Listing A.3 demonstrates all literals, in conjunction with data structures.

```
string := "Hello"
bool_true := true ; bool_false := false
long := 2L ; int := 1 ; short := 3s ; byte := 4b
array := [; int, long, short]
list := [1, 2]
compound := { key1: string, "key 2": 123 }     // string keys are fine too
```

Listing A.3: Literals and data structures in GZB

```
@e[type=$chicken, is "My_Tag"]    // eq. @e[type=$chicken, tag="My_Tag"]
@r[dx=10, dy=10, dz=10]           // select a random player in a 10x10x10 radius
```

Listing A.4: Selectors in GZB

A.2.2 Selectors

Selectors are used to query entities in game. It always exists of one or two parts: an at-symbol (`@`) followed by the selector base, optionally followed by a list of filter conditions, called props. Given props, the query result of the base selector may be narrowed down more.

In the game's command syntax, it is only possible to use one of the primitive selector bases, as listed below. In Gazebo however, any other valid identifier may be used, which we then refer to as an alias. Details about these aliases will be further expanded on in section A.4.2.

The base selectors and their behavior explained:

- `@e`: select all entities.
- `@a`: select all players, i.e. entities of the type player (`[@e[type=$player]]`).
- `@p`: select the nearest player. This is the same as `@a` but sorted by nearest distance and limited to 1 result (`@a[sort="nearest", limit=1]`).
- `@r`: select an arbitrary player. Similar to `@p` but with random sorting (`@a[sort="random", limit=1]`).
- `@s`: select the sender, which is always derived from some entity, but may have some internal state differ from the original entity. Who is the sender is highly context-dependent, as it refers to the entity that issued or was issued to execute the command. The closest syntactical representation of the sender is `@e[type=magically-inferred, limit=1]`.

A plethora of different filter props are available. Describing each of them in detail is beyond the scope of this report. In general, without further explanation in this section, there are two kinds of filter props: key-value and comparison. This leads to two different syntaxes, primarily geared towards readability of the GZB code, as the key-value-only syntax that the game uses conveys little semantic information.

In listing A.4, several examples of selectors are given, each with different filter props. Note the usage of the `is` keyword, which is syntactic sugar for the `tag` filter prop.

A.2.3 Arithmetic and Boolean Logic

All conventional arithmetic and boolean logic operators are supported, with the addition of a few domain-specific operators.

Boolean conjunctions, disjunctions and negations use the Python-style syntax, i.e. using the `and`, `or` and `not` keywords, respectively. Comparison operators resulting in a boolean

```
a := 1
func example(c Int) {
  complex := { a: [1, 2] }
  complex.a[0] *= 3
  b := complex."a"[1]    // ."a" is equivalent to .a
  b += a
  b -= c
  if b > 0 { d := 123 }
  b = d                  // error: d not defined (out of scope)
}
```

Listing A.5: Variable declaration, referencing and assignment in GZB

value are supported depending on the type of the values that are compared. Specifically to check whether a numeric value falls within a particular range, the `matches` operator is used.

A.3 Body: Statements and Control Flow

In this section, we cover some statements and control flow mechanisms that were not discussed in the main text.

A.3.1 Variable Declaration, Referencing and Assignment

Variables are declared by a name and a value, separated by the walrus operator (`:=`). Types are intentionally not possible to specify, as this is guaranteed to never be necessary. A variable can be reassigned by using the single-equals operator (`=`), including shorthand operators such as `+=` and `-=`.

Variables are referenced by the exact same name as they are declared. However, it is possible to narrow down on a variable before assigning to or referencing it. This can either be done using the bracket index notation for lists and arrays (`[ ... ]`), or by using the dot access notation (`.`) for compound types. The access notation is normally used with regular identifiers, but any string key can be used as well.

The reason to have two separate operators for declaration and reassignment is to avoid confusion about where the initial value is assigned. It also serves as an explicit reference point.

Listing A.5 shows a few examples.

A.3.2 Execute

The execute statement is a control flow statement that can execute its body block in a wide variety of ways. Contrary to all other control flow statements, the execute statement is very feature-rich and forms the core of the domain-specificity of the language.

The keyword for this statement is actually not `execute`, but differs based on the variant used. There are about a dozen different variants, each with a different purpose. As often several variants are desired to be combined, we allow syntax-free chaining of the variants, called fragments: all other statements require the body block to be either put in curly braces, or be a single statement with a fat arrow.

As with all other features, this statement corresponds to some in-game mechanic. In particular, this corresponds to the `execute` command, which happens to be more expressive than this statement. The reason is that the `if` and `unless` sub-commands are actually imple-

```
as @a ⇒ say("Hello!")   // prints in chat "[player name] Hello!", for each player
// the following two examples assume the sender is a player in 'The Overworld',
// positioned at (80, 32, 80)
in $the_nether ⇒ teleport(`(~ ~ ~))    // teleports to (10, 32, 10) in 'The Nether'
in $the_nether positioned as @s ⇒ teleport(`(~ ~ ~))   // idem, but to (80, 32, 10)
// the trick here is that ——^ refers to the sender, which has the 'old' properties,
//  and thus holds the untranslated coordinates ('The Nether' has a 1:8 scale)
```

Listing A.6: Execute statement in GZB

mented via a higher-level statement, namely the if-statement, which additionally offers an else-branch.

We remain with eleven different fragments to be used in the execute statement:

1. **`align`**: align the sender's position along the x-, y-, z-axis, or any combination of these, by flooring the values of the given axes.
2. **`anchored`**: update the execution anchor. Either eyes or feet.
3. **`as`**: execute subsequently as *all* matching entities, but do not modify the sender position.
4. **`at`**: execute subsequently at *all* matching entities' positions, rotation and dimension, but do not modify the sender's identity.
5. **`facing`**: execute subsequently with rotation (yaw and pitch) set such that the sender is facing the given coordinates.
6. **`facing entity`**: execute subsequently with pitch and yaw set such that the sender is facing the target, for all matching target entities.
7. **`in`**: switch to the given dimension. Performs coordinate translation if applicable.
8. **`positioned`**: set the execution position to the given coordinates.
9. **`positioned as`**: execute subsequently at *all* matching entities' positions only, without affecting any other properties.
10. **`rotated`**: set the execution rotation to the given yaw and pitch angles in degrees.
11. **`rotated as`**: execute subsequently with rotations of *all* matching entities, but do not modify any other properties.

To demonstrate the usage of this statement, including the 'chaining' of different fragments, we give an example in listing A.6.

A.3.3 Raw

Some features of the game are not available through the surface language, but may still be useful to use. For this purpose, the raw statement is provided. This allows arbitrary commands to be emitted, comparable to inline assembly in low-level programming languages. Our current implementation does not check whether the raw command is actually valid.

This feature is mostly useful for the standard library, as it has to provide a Gazebo-compatible interface to the game's commands. A piece of the standard library, the `say` function, is listed in listing A.7.

As can be seen there, everything between the pipe symbols (`|`) is the raw command. Variables in scope can be formatted in-place by the percentage sign (`%`).

```
func say(text String) {
  raw |say %text|
}
// calling say("Hello") will emit exactly the following command:
//   say Hello
```

Listing A.7: Raw statement in GZB: the `say` function

```
func sum(a, b Int) → Int = a + b
func mul(a, b Int) → Int { return a * b }
func say_hello { say("Hello") }
```

Listing A.8: Functions in GZB

A.4 Files: Top-Level Entries

Each file being a module is further divided into members of that module. From the perspective of a file, these are its top-level entries. There are four kinds of top-level entries, each of which we cover in a subsection: functions, selector aliases, variable declarations, and type declarations.

A.4.1 Functions

Functions form the core of Gazebo are the most important feature for programmers. In all ways, these are similar to the general idea of what a function is, contrary to a MCFunction. Gazebo functions may take arguments and may return a value, which is demonstrated in listing A.8, while MCFunctions cannot take arguments; they can only be executed in full and will always continue at the caller, without the possibility of explicitly receiving or returning values.

Note that there are different ways of writing a function: most elements of the signature are optional. A minimal function definition is a name accompanied by a body. The body can either be an expression (`= …`), a single statement (`⇒ …`), or a regular block (`{ … }`).

Should a function take parameters, it is possible to specify them in parentheses after the name, otherwise parentheses can be omitted. Each parameters is denoted by its name followed by its given type. Arguments of the same type may be grouped, by providing a list of names followed by a type that applies to all in the preceding list. For example, `(a, b Int, c, d String)` means that `a` and `b` are of type `Int`, whereas `c` and `d` are of type `String`.

If the function should return a value, it can be specified after the name and parameters by → followed by the expected type. If no return type is specified, the function cannot return a value, and is typed as void.

A.4.2 Selector Aliases

As a convenience, Gazebo provides a way to refer to commonly used selectors by an alias. These aliases can be used in the same way as any other built-in selector. The syntax of aliases is very straightforward and demonstrated in listing A.9.

A.4.3 Variable Declarations

Any variable can be declared as a module top-level entry. This means that they become available for usage in other modules. In all other regards, global variables behave identically

```
alias @MySelector    = @e[sort="nearest", limit=20]
alias @OtherSelector = @MySelector[is "my_tag"]
```

Listing A.9: Selector aliases in GZB

to local variables (see section A.3.1).

A.4.4 Type Declarations

Similar to how selectors can be aliased, types can too. As was explained in section 3.2, the type system is purely structural, therefore type declarations are only a convenience feature.

A type is declared by a name followed by the expansion of the type. There, any type can be used. Self-references are allowed, but may lead to scenarios where it is not possible to construct a value for the type.

Appendix B

Architecture of the Stand-Alone Compiler

An integral part of the user experience with Gazebo is its stand-alone compiler. Conventionally, Gazebo offers this as a CLI, referred to as GZBS. This chapter covers most high-level implementation considerations, lists which technologies were used, and discusses some Spoofax-specific details.

As this appendix is intended to be a continuation of section 3.4, we assume the reader to be familiar with the ideas discussed there. The first section gives a schematic summary of the entire architecture. Remaining sections are dedicated to specific important implementation details.

B.1 Visual Summary

Figure B.1 shows a visual representation of the global summary that now follows.

The JVM starts execution at the main class of the CLI distribution, which immediately transfers control to Picocli to parse the command-line options. By means of the Java service provider interface (SPI), as described in section B.4, the language archives are gathered. Together with some configuration options for the standard libraries, the Gazebo configuration is created.

Transferring control to the main logic, the Gazebo module is initialized by the `GazeboSpoofaxFactory`. According to the procedure described in section B.5, the Guice modules are configured and language archives are registered. As the `GazeboModule` extends the `SpoofaxModule` (which in turn extends the `MetaborgModule`), all Spoofax Core services are bound and initialized.

Now that Spoofax is properly initialized, control is handed back to `:cli` where relevant overlay tasks are picked from what `:lib` offers. Together with the path to the root of the project and message printer configuration, a configuration for the Gazebo Runner is created. These tasks are described in more detail in section B.3.

Once again, control transfers to the main logic, where the Gazebo Runner communicates with Spoofax to initialize the project instance, configuring meta-languages (for example, enabling *Statix* message formatting), source paths within the project, and internal dependencies on the Gazebo language projects.

Now the 'build input' is configured, describing how Spoofax should build the project, which includes message-printing configuration. Full control is then handed to the Spoofax Core build system. GZBS now synchronously waits until the build result becomes available.

If the Spoofax part of the build completed successfully, all overlay tasks will be executed in sequential order. The project and output information is passed to the overlay tasks, as they may use this.

Finally, the final result status is printed and GZBS terminates with the proper exit code.

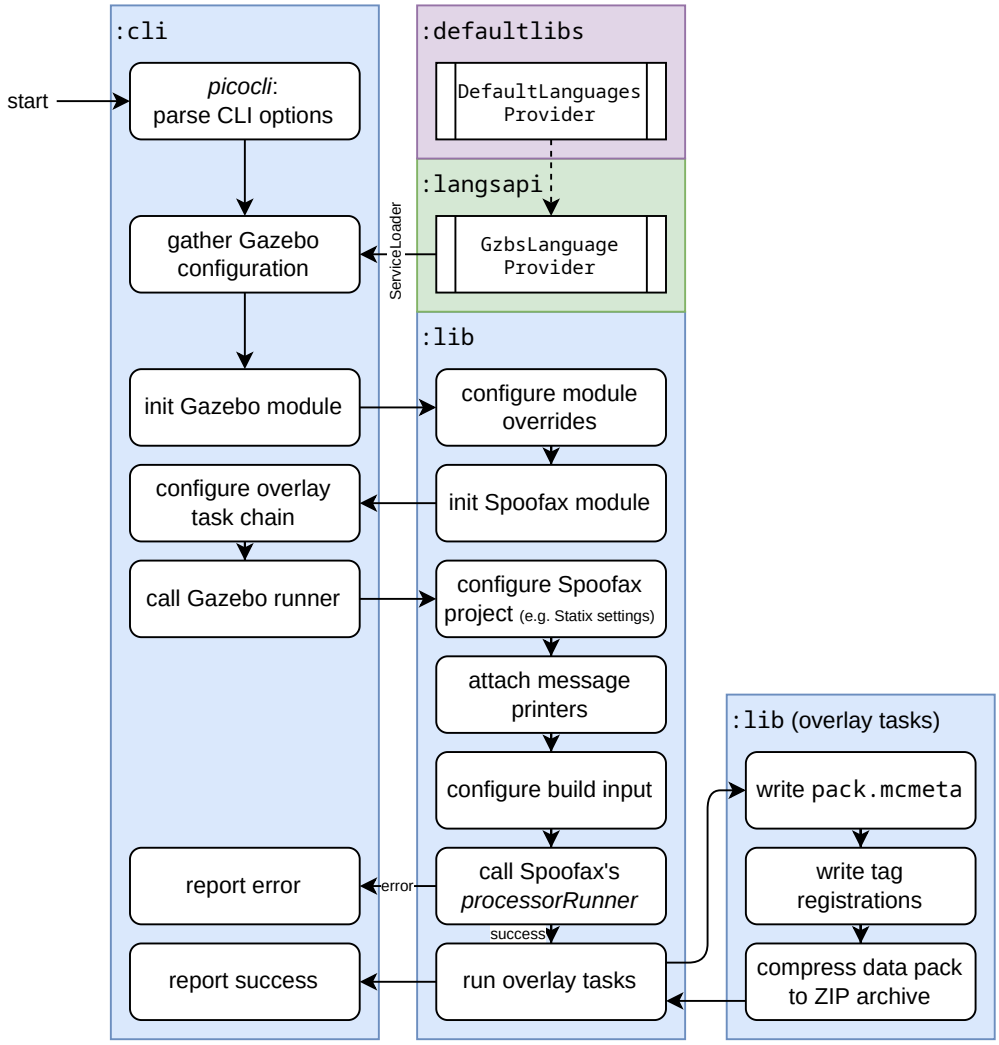


Figure B.1: Schematic of the GZBS architecture, most relevant parts

B.2 Message Printing and Logging

Message printing in Spoofax Core is mostly a solved problem, by providing an implementation of the `IMessagePrinter` interface. The default implementation is only capable of writing to a single `OutputStream`. This is the reason that some custom message wiring still was necessary to be implemented.

One implementation is the `AggregateMessagePrinter` which allows wiring a broadcast set of other message printers. Additionally, as a workaround to the fact that some Spoofax meta-languages emit HTML-formatted messages, we added an unescape filter to transform the HTML content to (terminal-suitable) plaintext.

Logging of Spoofax and Gazebo internals is fully handled by the commonly used *SLF4J*[1] API and the default `SimpleLogger`[2] back-end. It prints all log entries above or equal to a configurable severity level to the standard error stream, which is usually the user's terminal.

[1] `https://www.slf4j.org/`

[2] `https://www.slf4j.org/api/org/slf4j/simple/SimpleLogger.html`

```
val runnerConfig = /* ... */
val compress = true
GazeboRunner(runnerConfig)
  .withOverlayTask(
    EmitDataPackTask(EmitDataPackTask.PackFormat.VERSION_9)
      .runIf(compress) {
        chain { dataPackLoc →
          CompressDataPackTask(dataPackLoc)
        }
      }
  )
  .run(spoofax)
```

Listing B.1: Usage example of GZBS overlay task chaining

Although Spoofax does include several logger management classes, they all dispatch to *SLF4J* in the end.

B.3 Overlay Task Structure

Some responsibilities of GZBS, as described in the following sections, are not necessarily Spoofax-concerned, but still need to be executed in relation to inputs or outputs. It is impossible to extend Spoofax Core API natively with additional build steps, as the entire build sequence is hard-coded, hence we implemented a simple task mechanism built around the Spoofax builder. A usage example is given in listing B.1.

Gazebo Runner accepts any number of overlay tasks. Once the results from the Spoofax builder are available, all these tasks are executed sequentially, in order of registration.

This does not allow simple sharing of task results between tasks. The 'emit data pack' and 'compress data pack' tasks are examples of tasks that *do* depend on each other, sequentially. By implementing a helper task, called `ChainedTask`, any task can be sequenced after any other task, passing the result of the first to the second.

The overlay tasks that are available are the following three:

- Emit data pack: create directory structure, write `pack.mcmeta` and write tag registrations.
- Compress data pack: apply ZIP compression to a data pack directory structure.
- Emit StxLib: make a *Statix* project library, applying necessary patches[3].

B.4 Language Archive Loading

The language archives that make up Gazebo as a whole, as mentioned in the overview, need to be made available to Spoofax Core in order for them to be loaded and wired up.

From the perspective of Spoofax Core it is not important where these archives reside. The API provides a *language discovery service*, which can be instructed to look for language archives at a particular location. The locations where to load from, either a file or a directory, are provided automatically by GZBS.

[3] At the time of writing, *Statix* requires applying some patches to the `.stxlib` file before it can acutally be used. See: `https://www.spoofax.dev/howtos/statix/migrating-to-concurrent-solver/#using-libraries`.

This is in clear contrast to Sunshine which requires the locations to language archives to be configured via command-line arguments. GZBS implements automatic discovery and loading of language archives by leveraging Java's SPI framework.

The SPI interface is defined in the `:langsapi` module, which is referenced by `:lib` and implemented by `:defaultlibs`. The `:defaultlibs` module returns paths to all the language archives. A custom Gradle task makes sure that these paths are included in the JAR of `:defaultlibs`, as to make them available on the class path.

While in practice the collection of language archives does not change within a particular build of GZBS, the SPI architecture lends itself more to future extensibility. The main advantage is that this architecture theoretically allows for this, without having to modify any of the current code. Supplying a JAR on the class path that implements the SPI interface is sufficient, as the discovery is fully handled by the Java runtime.

B.5 Intermediate Passing Optimization

The Gazebo architecture works on the basis of having three distinct languages, which are separated in isolated projects and transformed between by the extension projects (see chapter 2).

Spoofax Core writes the result of a transformation to disk. It does not care about the exact contents of this output: it is an ATerm (Brand et al. 2000) that is merely converted to a string representation. The only exception is that, if it is a string at top-level, no quoting is applied.

The *syntax service* from Spoofax Core is responsible for reading and parsing source files, in order to be further processed by the build system via analysis and transformation stages. This service however, does require the source file it loads to be formed according to the syntax associated with the respective extension, as it is fed though a dynamically selected parser. In practice, this turns out to only be overhead for very specific occasions.

When executing the transformations from within the *Eclipse* environment, it is to be expected that the current file on which the transformation action is executed is properly formatted. However, this is not necessarily true for a fully stand-alone pipeline. Invoking an integrated pipeline, to the user, only appears as one big black box which accepts files formatted according to the surface syntax and emits files formatted according to the target syntax. Only for human inspection it might be beneficial to read the intermediate files in a pretty-printed form.

The pretty-printing by the transformation followed by immediate parsing by the next build step is wasteful in terms of processor cycles. GZBS mitigates this overhead by emitting the raw AST from a transformation and overriding the *syntax service* such that it skips parsing altogether if it notices such a raw AST.

A raw AST is indicated by the file of interest having the format `.+\.aterm-speed\.[^.]+`, expressed as a regular expression. In all other cases, the default `SpoofaxSyntaxService` is called, allowing for full backwards compatibility.

Overriding the *syntax service* is achieved by overriding the binding to Spoofax's `ISpoofaxSyntaxService` with the new `ATermSpeedSyntaxService`. By default this is not possible, as the *syntax service* interface is already bound to `SpoofaxSyntaxService` in the `SpoofaxModule`. Fortunately, Guice allows module overriding during initialization, used in the `GazeboSpoofaxFactory`.

B.6 Result Data Pack Packaging

The last stage of the compiler emits the *MCFunction* files, already in the structure that is expected by the game. Still, it is not possible for the game to load these files directly. Some

additional processing is still necessary: writing meta data, writing tags and optionally applying ZIP compression.

First of all, we describe the meta data file. This is a file in the root directory of the data pack, named 'pack.mcmeta'. Its contents must be formatted in JSON format. The file contains a compatibility identifier and a descriptive string which may be shown to the user. The current implementation of GZBS statically emits this file, i.e. it is not possible to give a custom data pack description.

Secondly, tag registrations are emitted, if applicable. This is a general post-processing task which scans all output files, collects all tag registrations, and writes these to the appropriate location in the data pack. All files are scanned for IMP headers, specifically for the `@context` annotation, which, according to the IMP specification (Arcensoth 2020), indicate in which tag contexts a function is expected to be run. These tag registration files are JSON-formatted, containing a list of function identifiers.

These two tasks create a well-formed data pack, but leave everything in a plain directory structure. For ease of use, the game cannot only read from such a directory, but additionally supports ZIP archives. GZBS will archive and compress the directory structure if the `--compress` CLI flag is specified, simplifying the development process.

Appendix C

Standard Libraries

One of our promises is static typing for almost everything. In order to achieve this goal, we need to provide all necessary typing information to the type checker. Precisely this information is what standard libraries aim to provide.

This chapter describes what is contained in the standard libraries. Additionally, we outline the construction procedure by which these libraries are built. Finally, we discuss some practically applied performance tweaks.

C.1 Standard Library Contents

Without any libraries, Gazebo works perfectly fine, except that it requires a lot of boilerplate code to start writing useful programs. The standard library aims to provide most, if not all, of this boilerplate code. On a high level, this can be viewed as three distinguished aspects:

Entity Data Structures All entities in the game can be read and often also modified. The standard library defines the data structures that are accompanied by each entity.

Registries The game categorizes many *things* into registries: collections of items with some meta-data associated. Entity types are an example of this, which are always accompanied by a data structure. Another important registry is the block type registry, which lists all legal states for each block.

All registries and registrations include protocol IDs. This is intended to be used for compile-time reflection operations, such as creating a function that produces the protocol ID given the coordinates of a block at runtime.

Command Wrappers The game provides many useful commands, which are directly available using the raw notation. To prevent usage of this unsafe mechanism, the standard library wraps most commands in a fully type-safe way.

C.2 Automated Construction

The standard library is constructed from several sources. Only a small part is hand-written, whereas the majority is generated[1]. As most of the contents are repetitive and following the same general structure, auto-generation lends itself very well. Moreover, both the game's developers and the community provide high-quality structured data sources.

[1] At time of writing, approximately 20-30 kSLOC are generated

We strongly rely on two of such sources: Yurihaia (2019)'s `mc-nbtdoc` and Arcensoth (2018)'s `mcdata` repository. The former contains partially automatically generated and manually annotated data structure definitions, covering most of the game data that can be accessed via commands. The latter only contains data generated by the game itself, but made more accessible. Additionally, some minor post-processing is already done on that data.

These sources are combined by the Python script[2], processed, and written to disk. Since this does not yet cover all our promises for the standard library contents, we finalize the source code emission by copying the 'overlay' directory structure[3] onto the output from the first stage. These overlay files contain hand-written GZB source code. To prevent overlay files from overriding automatically generated files, we concatenate the file contents in case the destination file already exists.

In short, the standard library source text in GZB is constructed in the following sequence:

1. Create data structure definitions according to `mc-nbtdoc` (Yurihaia 2019).
2. Create registries according to `mcdata` (Arcensoth 2018).
3. Apply static overlay files.

C.3 Performance Considerations

Without any further processing of the raw source text, the standard library can be packaged into a language archive, which in turn is configured as a source dependency of the user project. By Spoofax's include-file discovery mechanism, the files are automatically taken into account while building a project.

Unfortunately, this comes at a relatively high cost: most compilation time is spent on static analysis of the standard library, even if the user project does not reference anything from it. Statix offers a feature called 'project libraries'[4], which, for convenience, we abbreviate to StxLib. This allows us to analyze the standard library, store and package these results, and use them in the user project *instead of* the includes.

Although it might sound counterintuitive, no complex bootstrapping procedure is required to achieve this. After we emit the source code of the standard library, GZBS (see appendix B) is invoked in a special way which does *not* require the standard library. Compared to the default compilation pipeline, some steps are skipped, and some other steps are added, such as the creation of a StxLib file. Finally, the StxLib and original sources are then packaged into the conventional language archive.

For ease of further integration in a later stage, we also create standard library language archives for the two other intermediate languages, by applying the source transformations. In practice, the size of these intermediate standard libraries is small, as there are few function definitions; most of the standard library is dedicated to type definitions. As the intermediate languages do not require static analysis, there is no need to create any other StxLibs.

[2] `https://github.com/MetaBorgCube/gazebo/blob/master/gen/gen.py`

[3] `https://github.com/MetaBorgCube/gazebo/tree/master/gen/overlay/`

[4] `https://www.spoofax.dev/howtos/statix/migrating-to-concurrent-solver/#using-libraries`