UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

21 | 22

Bachelor thesis

Formal design and implementation of a programming language based on facets

Ángel Alberto Carretero Ramos

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Bachelor as Doble Grado en Ingeniería Informática y Matemáticas

BACHELOR THESIS

Formal design and implementation of a programming language based on facets

Author: Ángel Alberto Carretero Ramos
Advisor: Juan de Lara Jaramillo

May 2022

Ángel Alberto Carretero Ramos
Formal design and implementation of a programming language based on facets

Ángel Alberto Carretero Ramos

PRINTED IN SPAIN

Acknowledgements

This project was done under the scholarship “Beca de Colaboración” 2021–2022 from “Ministerio de Educación y Formación Profesional”.

I want to explicitly thank my tutor, Juan de Lara, for his dedication to this project. Not only did he gave me lots of corrections, but most importantly he gave me complete freedom. Even when I veered into a new idea, even when I insisted on doing everything by hand despite of the time constraints, he was always helpful and supportive. Additionally, I want to also thank him for providing me with the opportunity of assisting the meetings of the Research Group here at UAM.

Abstract

Maintaining software is becoming increasingly difficult due to the constant stream of technological advances and the pressure to ship as fast as possible. Undoubtedly, a crucial and undervalued aspect is producing and maintaining good documentation. In this work, we will analyze the problem through the lens of a concrete example of a Machine Learning pipeline, where a set of operations have to be called in a predefined order to correctly transform the data; an order which is, precisely, poorly documented. The consequence is that we do not know if a function can be refactored, if we have to call it, or something as fundamental as how does the function change the data.

Throughout this text we will develop a generic solution starting from this simple example. A programming language will be designed and implemented that, thanks to the paradigm of facets and its static type system, can solve the example problem in an elegant and pragmatic way.

The second objective, motivated by the academic nature of this work, is learning. Because of it, the interpreter will be developed from scratch and, in the text, we will alternate between the theoretical development of the language and the challenges of the implementation. On top of that, we will use Rust which will allow us to justify several design decisions, and to delve into a less common programming paradigm.

The result is an interpreter that contains: an agnostic parser library, the grammar of the language, a static type system, the interpreter itself and various additional tools. Lastly, the validation will consist of a bespoke test framework with different test cases.

Keywords

Facets, interpreters, programming languages, parsing, typechecking, parser combinators, semantics

Resumen

Mantener software es una tarea cada vez más complicada debido a los constantes avances tecnológicos y al vertiginoso ritmo de desarrollo. Sin duda un aspecto crucial y a la vez menospreciado es producir buena documentación y actualizarla. En este trabajo analizaremos este problema a través del ejemplo concreto de una *pipeline* de Aprendizaje Automático en la que, para transformar correctamente los datos, tenemos que llamar a funciones en un orden determinado, un orden que está escasamente documentado. Esto lleva a que no sepamos si se puede refactorizar una función, si hace falta llamarla o algo tan fundamental como qué añade a los datos.

A lo largo del texto desarrollaremos una solución genérica partiendo de este sencillo ejemplo. Se diseñará e implementará un lenguaje de programación que, gracias al paradigma de las facetas y a un sistema de tipado estático, pueda resolver este problema de forma elegante y pragmática.

El segundo objetivo, motivado por la naturaleza académica de esta empresa, es el aprendizaje. Por ello, el intérprete se desarrollará desde cero y alternaremos entre el desarrollo teórico del lenguaje y los retos de la implementación. Además, usaremos Rust lo que nos permitirá ahondar en un paradigma de programación menos habitual y justificar varias decisiones de diseño.

El resultado es un intérprete que contiene: una librería para *parsear* la gramática del lenguaje, un sistema de tipado estático, el intérprete en sí y distintas herramientas auxiliares. Para finalizar, la validación consistirá en un sistema propio de pruebas con multitud de casos de uso.

Palabras clave

Facetas, intérpretes, lenguajes de programación, *parsear*, sistemas de tipado, *parser combinators*, semántica

Table of Contents

1 Introduction and Background **1**

1.1 Problem . . . 2

1.2 Facets . . . 3

1.2.1 Expressiveness and performance considerations . . . 4

1.2.2 How do facets solve the original problem? . . . 4

1.3 Related work . . . 5

1.4 Background: Programming Language Theory Concepts . . . 6

1.5 Objectives . . . 8

1.6 Document organization . . . 9

2 Design of the language syntax **11**

2.1 Syntax rules . . . 11

2.2 Design decision: Null . . . 13

2.3 Basic usages . . . 13

2.4 Original problem revisited . . . 14

3 Parsing **17**

3.1 Simple parsers . . . 18

3.2 Some combinators . . . 19

3.2.1 Advanced combinators: Macros over the AST . . . 20

3.3 Complete example: Parsing comments and whitespace . . . 21

4 Typechecking **23**

4.1 Rules . . . 23

4.1.1 Simple typechecking . . . 25

4.1.2 Facets . . . 26

4.2 Implementation . . . 27

4.2.1 Case study: Typechecking a facet initialization . . . 28

4.2.2 Case study: Return statements . . . 29

5 Interpreter **31**

5.1 Rules . . . 31

5.2 Implementation . . . 33

5.2.1 Case study: Environment . . . 33

5.2.2 Case study: Facet fields . . . 34

5.2.3 Case study: Return statements . . . 35

6 Implementation and validation **37**
6.1 Validation 38
7 Conclusions and future work **39**
7.1 Future work: generics 40
Bibliography **41**
Appendices **43**
A Formal grammar rules **45**
B Introduction to Rust **47**
B.1 References 47
B.1.1 Lifetimes 48
B.1.2 Borrow checker 48
B.1.3 Traits 49
B.2 Generics 50
C Advanced topics on parser implementation **51**
C.1 The Parser trait 51
C.1.1 What is really a closure? 51
C.1.2 Passing arguments to functions 53
C.1.3 The Fn trait vs the fn type 53
C.1.4 The Copy trait 54
C.1.5 The Parser trait 54
C.2 Grammar rules variations and effect on performance 55
D CLI usage **59**

Lists

List of codes

1.1 Example of getting recommendations for a user in pseudo-Java. 2
1.2 Employee facet 3
1.3 Example syntax to define a Person facet with automatic constraint based transformations. 4
1.4 Example program with nested closures. 6
1.5 Example code. 7

2.1 Syntax specification for our language. 12
2.6 Original problem implementation using facets. 15
2.7 Wrong call order with output from the interpreter. 15
2.8 Accessing a field that does not exist at that point in time. 16

3.1 `peek` function and the struct Information where we hold the state. 18
3.2 Function that parses a character that matches the supplied condition. 19
3.3 The and combinator takes two parsers and returns only if both parsers have matched the same input and produced the same output. 19
3.4 `tuple` implementation for only three elements. 20
3.5 `Tuple` trait and function that performs the dispatch. 21
3.6 Defining a combinator the consumes the whitespace and comments around the given parser. 22
3.7 Example of how “junk” is used in parsing a block. 22

4.1 Typechecking specification for our language. 24
4.2 Example program with type annotations. 26
4.3 Type enum. 27
4.4 Trait for typechecking. 28
4.5 Implementation for facet initialization in the type system. 28
4.6 When we encounter a function definition, we bind it. 30
4.7 In a return statement, we get back the function we are in. 30

5.1 Evaluation semantics specification for our language. 32
5.2 Executing a function call in the interpreter. 33
5.3 The closure stores a reference to n and, when n is changed, it is changed also for the closure. 33

5.4 Definition for our tree of environments. 34
5.5 Looking up a field in the environment. 35
5.6 Executing return statements and bubbling up the value. 36

7.1 Example of the usage of generics, and how the natural problem of adding facets to arbitrary types can be modeled. 40

A.1 Formal grammar rules in BNF. 45

B.1 Example where we specify that the output lifetime is maximum 'a. 48
B.2 Excerpt from the language prelude of the trait used to test for equality. 49
B.3 Implement trait A for all types that already implement B and C. 49
B.4 Example where lifetime parameters are used alongside generics. 50

C.1 Example program with nested closures. 52
C.2 Attempting to call a combinator with the same parser twice. 54
C.3 Parser trait and implementation for closures/functions. 54
C.4 Rules for binary expressions, left recursive. 55
C.5 Rules for binary expressions, non left recursive. 55
C.6 Final set of rules for binary expressions. 56

List of figures

1.1 Employment, employee and person facets and their relationship. 3
1.2 Nested list of scopes at line 8. 6
1.3 AST from Listing 1.5 created using the graphing tool. 8

4.1 Nested list of scopes at line 4. 26

C.1 Close up of the stack frame of a function (x86 convention). 52
C.2 The stack as of line 8 in Listing C.1. 52
C.3 The stack after returning from function b in Listing C.1. 53
C.4 Simplified stack frame for parsing `2 == 2` 56

List of tables

6.1 Lines of code (LOC) absolute and relative per project module. 38

1 Introduction and Background

The constant change of requirements, the pressure to ship as fast as possible and technological advances surfacing every month, are several of the reasons why software today is less maintainable than ever before. One of the, if not the most neglected task, is writing documentation and, even more so, keeping it up to date. Everyone knows it is crucial to do so and, yet, developers never find enough time to do it. For example, picture a Machine Learning pipeline where data is taken in and transformed many times until the result is obtained. What happens if we want to remove one transformation or, if we want to only call a subset of the methods. It happens that we have to rely on outdated comments and, probably, look manually in the code to see what each function expects, so that we can call them in the right order.

In this work, we set out to design a new programming language to prove that the problems in the example can be solved using the correct paradigm. We will use the information at the type level to both, document the code, and verify its correctness. As an added benefit, because the documentation is part of the code itself, it cannot get outdated as was the case with comments.

The idea is to design and develop a language slowly from first principles, starting from this concrete example and generalizing the solution. Along the way, we will also cover the necessary theory. In particular, because facets are more of a niche idea, we will detail the semantics of how they work precisely. Additionally, because designing and implementing a language is no easy feat, there will be countless discussions permeating the document justifying the rest of our choices.

Lastly, the principle that guides this whole endeavour is pragmatism. We want to create as little friction as possible for the programmer and a solution that is both performant and useful. The second main motivator is learning. Creating a compiler is a great exercise in itself and as a way to learn the host language by testing its limits. In that spirit, we have chosen Rust to delve into a less common programming model. In summary, in this work we are going to interleave programming language theory, learning about Rust, and solving the difficult programming challenges that the interpreter poses.

1.1 Problem

The inspiration for creating a new language came originally from getting recommendations for a given user in an online business where data is spread across different services. As stated before, it involved transforming the data through several steps that had to follow one of many possible orders. The issue is that this was documented using comments and not enforced at the type level. That, coupled with not up-to-date documentation, made it very challenging to work with the code base. For a small recreation in pseudo-Java, have a look at Listing 1.1.

```java
Data getUserRecommendations(String userid) {
    // Data is a container, a Map<String, Object> for all possible attributes.
    // For example the username would be key: "username" and the values will
    // be of type String.
    Data data = new Data();
    data.addColumn(USERID, userid);
    // Log the username. Notice that we have to cast and hope that the column
    // exists.
    System.out.println((String) data.getColumn("username"));
    // Queries the service to check if the id is valid and resolve the early
    // data such as username used by services down the line.
    userContext.featurize(data);
    // Adds some hints such as if the user is suspected to be a child.
    userHints.featurize(data);

    parallel {
        // Different services
        userRecommendationsFactory.get(ServiceAEndpoint).featurize(data);
        userRecommendationsFactory.get(ServiceBEndpoint).featurize(data);
    }
    // Gets the affinity to the shop featured items
    userTopItemsRecommendations.featurize(data);
    return data;
}
```

Listing 1.1: Example of getting recommendations for a user in pseudo-Java.

The code contains, at least, the following problems:

1. There is no type information on what gets added to the `Data` map at each step. That means that if, for example, we wanted to add a specific column, we would have to check all the code manually to see which function does it.
2. If we wanted to call a function that requires another three calls before it, we would have no way of knowing but trial and error.
3. Even though it is not shown in the short snippet, because there is no type information, we have to

check the constraints for each function manually using tests.

1.2 Facets

The plan is to add some dynamism, namely facets, to the otherwise "stiff" type systems, such as Java's in this case. We take the term facet from the paper [1]. Informally, it is a way of creating more dynamic objects by letting them get and drop facets which lets us model the real world more closely. Individually, each facet contributes with a type and fields to the objects which holds it. The canonical example is the relationship between being a `Person` and being an `Employee`. We are always people but we are sometimes employed. Instead of having to deal with two types of objects and association or composition, we have an `Employee` facet added to the `Person`. Later, we can drop it and still have the same `Person`.

```
addFacet homer
    dayJob: Employment.Employee with {
        name = fullName [equality]
        salary = 15000
        ssNumber = 12345
        active = true
    }
    nightJob: Employment.Employee with {
        name = fullName [equality]
        salary = 16400
        ssNumber = dayJob.ssNumber [equality]
        active = dayJob.active [equality]
    }
```

Listing 1.2: Employee facet. *Reproduced from listing 5 in [1].*

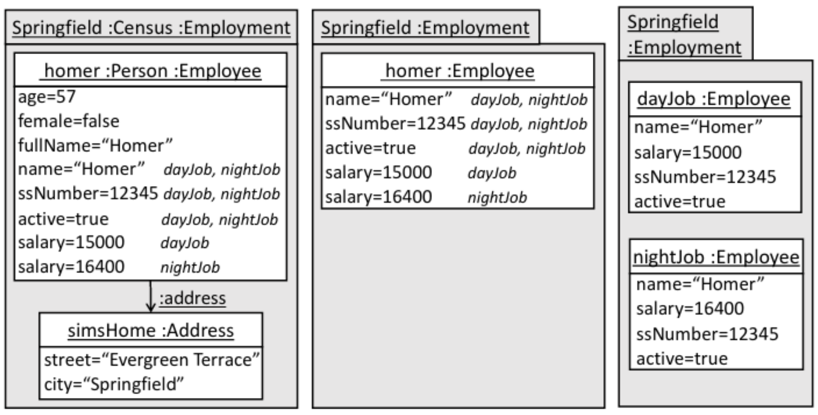

Figure 1.1: Employment, employee and person facets and their relationship. Visual result of Listing 1.2. *Reproduced from figure 5 in [1].*

In the paper they present several techniques that allow for more expressiveness such as having one facet derive fields from another (see Figure 1.1). For example, we might derive automatically the age till retirement for the `Employee` facet by looking at the `Person` age. The other fascinating technique is adding these facets automatically based on conditions. For example, if a person's age is greater than 17, we make them an `Employee` (see Listing 1.3) and when their age is greater than 67, we remove the facet because they retire.

```
var p : Person := new Person;
p.age := 23; // implicitly creates an Employee facet (as p.age > 17)
p.salary := 15100; // OK, as p has now an Employee facet
p.age := 16; // p loses its Employee facet (as p.age <= 17)
p.salary := 21000; // Error! p has no Employee facet
```

Listing 1.3: Example syntax to define a Person facet with automatic constraint based transformations. *Reproduced from listing 12 in [1].*

1.2.1 Expressiveness and performance considerations

Facets the way we have presented them are very expressive and lets the programmer model a lot of situations with ease and elegance. The problem is that there is usually a balance between being expressive and being performant. For example, adding and removing facets automatically based on constraints is very expressive but it would put a big toll on performance. First, we need to add a runtime check to know when the constraints are met, which means possibly checking them after every modification to the object. Secondly, if we were to create a concurrent programming language, how would we deal with these automatic modifications in the context of different threads? And, how would we deal with locks?

Performance and explicitness are the main reasons we have chosen to implement only a subset of [1]. Namely, we would like to add or remove facets explicitly. We will also not add various other advanced features such as deriving fields because they are not necessary for our use case. What will be crucial is that we implement a static type system that helps the programmer deal with the intricacies of changing objects.

1.2.2 How do facets solve the original problem?

The idea is to make the functions add facets, tags in this case, to the `Data` container. That way, we can say that a function needs facets `A+B+C` and produces an item with the extra facet `D`. For example, we may identify `A` with having the `USER_ID` column and `D` as having the `username`. Then, the first call receives `A` and produces `A+D`.

Under this paradigm, we now have all the information about the calling order at the type level. It also serves as documentation because it clearly states the form of the data coming in and out. Furthermore, the code is very similar to the original except for the new types, which make the transition trivial, and the fact that we no longer need tests makes the time investment worth it.

1.3 Related work

We have already studied how our solution is based on facets as presented in [1] trading expressiveness for performance. In this section we set out to find other languages to further compare our approach. Most of the languages that employ a paradigm similar to facets use the role terminology. In the parlance of role-oriented programming, a facet is a role and the object that contains the facet is said to be a *player*. Normally, there is also a notion of an *institution* that grants roles, for example a school may grant the role teacher.

Even though there are differences, roles are pretty similar to facets and much more common in the literature. For its abundance we will select and compare several role programming languages. The first ones are: powerJava ([2]) and OT/J ([3]). In both languages, the focus is in modelling the relation between role and institution. Instead, in our language, we do not have a notion of institution, but focus on acquiring and dropping facets dynamically, a use case not supported by neither of them. Additionally, there is again the tradeoff between expressiveness and performance, the former being pursued by both, OT/J and powerJava, and the latter by us. In its defense, powerJava, but not OT/J, does favour explicitness by requiring that the role is mentioned and not performing type conversions, sharing at least one of our goals.

The third contender is a full-blown programming language: Raku, formerly known as Perl 6. Raku implements most of OOP concepts such as interfaces, traits and mixins using roles as explained in [4]. Their definition of role is strikingly similar to ours where a role is a combination of fields and methods that can be added to objects, and the change is reflected in the type system with a notation close to ours. Apart from being a mature programming language with a complete ecosystem, our main difference resides in how facets/roles are intended to be used. For example, in Raku you may not drop a facet from an object, while it is essential for our use cases. Fundamentally, their objectives are to model OOP concepts using roles while we try to leverage them to add dynamism to the type system so, even though, syntax-wise both languages are pretty similar, the incentives are different.

There are many more projects such as SCROLL ([5]), JavaStage([6]) or EpsilonJ ([7]), but their approaches are a combination of the above ideas. One in particular deserves a mention, SCROLL, because it achieves role programming by creating a library in Scala. Meaning it does not need to bootstrap a lexer/parser nor an interpreter because it works by rewriting Scala constructs directly.

1.4 Background: Programming Language Theory Concepts

In this section, we will present the usual jargon alongside the most important programming language theory concepts, which uncoincidentally, will be ubiquitous in the rest of this work.

- **Functional language**: A language is said to be functional if functions are first class objects. That is, they can be used in the same way that all the other types; for example, by passing a function as an argument to another function.
- **Closures**: A closure is a function that *closes* over its environment, it captures it. For example, in Listing 1.4 `||` is used to create a closure with 0 arguments (line 4). In the case of the first `fun` (line 4), it is capturing the variables `a` and `b` in the topmost environment (Figure 1.2) while the second `fun` (line 6) does not capture anything.

  It is important to note that not all languages have closures because of the complexity they bring to the language and to the memory allocation (for more information check subsection C.1.1).
- **Scoping**: A scope or environment is an abstract identification between all defined identifiers and their values at any point in the code. Normally we like to visualize the scope as nested environments [1]. See Figure 1.2 for an example program and its environment representation. For instance, note how `a` and `b` are defined in the outer function, lines 2, lines 3, while `c` is defined in the inner closure, line 5. Also observe that there are two identifiers called `fun` in the environment and that we will always return the first one starting from the bottom. This is called shadowing and for more information check [8].

```
 1  fn main() {
 2      let a = 2;
 3      let b = 3;
 4      let fun = || {
 5          let c = 4;
 6          let fun = || {
 7              let d = 5;
 8              println!("{}", d);
 9          };
10          fun();
11      }
12      fun();
13  }
```

Listing 1.4: Example program with nested closures.

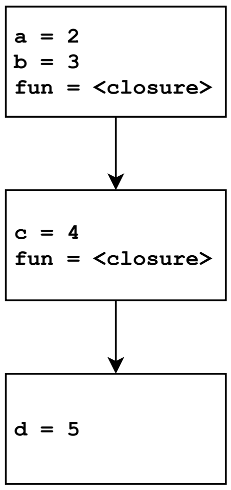

Figure 1.2: Nested list of scopes at line 8 for program Listing 1.4.

[1] See [9] for a more in depth explanation.

Inadvertently, when we think of scoping, we think of lexical scoping because that is what we are used to [2]. However, we still need to define it more precisely. Lexical scoping means that a variable can only be referred to in the scope it was defined or deeper, and only after it has indeed been defined. For example, in Listing 1.4 we can only refer to `d` in line 8.

- **Typing**: We can colloquially classify type systems into the following categories:
    - **Strongly typed**: Variables have one type and one type only. Examples: Rust, Java, C [3], Lisp, Python, etc.
    - **Weakly typed**: A variable can be coerced into a different type depending on the context. A good example is PHP where we can pass a string into a function that requires an int and the language will try to perform the conversion. For example, the string "1" would be coerced into the integer 1.
    - **Static type-checking**: Every variable must have a known type at compile time. The type safety of a program is thus verified at compile time. Roughly speaking, type safety checks that our program satisfies a set of rules such as only summing [4]numbers to numbers. Examples: Rust, Java, C, Go, etc.
    - **Dynamic type-checking**: The type safety is checked at runtime. For example trying to sum a string and an int will provoke a runtime error. Examples: Python, Ruby, Lisp, JavaScript, etc.

- **Abstract Syntax Tree (AST)**: is a tree representation of the syntactic structure of a program. Let's take an example program written in our language and generate an AST representation using the graphing tool we created (Appendix D). For example, Figure 1.3 is the representation of Listing 1.5.

```
let x: int = 3;
let y: int = x + 2;
print(y);
```

Listing 1.5: Example code.

- **Garbage Collector (GC)**: A garbage collector is a form of memory management employed in languages such as Python, Java or Go. It is an *automatic* technique unlike manually fiddling with allocs in C. We usually say that the language ships a runtime which is the overhead that we have to pay for "forgetting" about memory management. That is, even if our user program is compiled, it has extra functionality that is running in the background cleaning the unused memory. For example: every variable could have a counter that increases or decreases each time it is shared,

[2] Other types such as dynamic or mixed scoping are niche and/or have fallen into disuse.

[3] It can be argued that being able to cast pointers into numbers to perform arithmetic and bypass the type system is a form of weak typing.

[4] Once again, the sum operator (+) may be overloaded to work to concatenate strings. This is only an illustrative example and the rules obviously depend on the language.

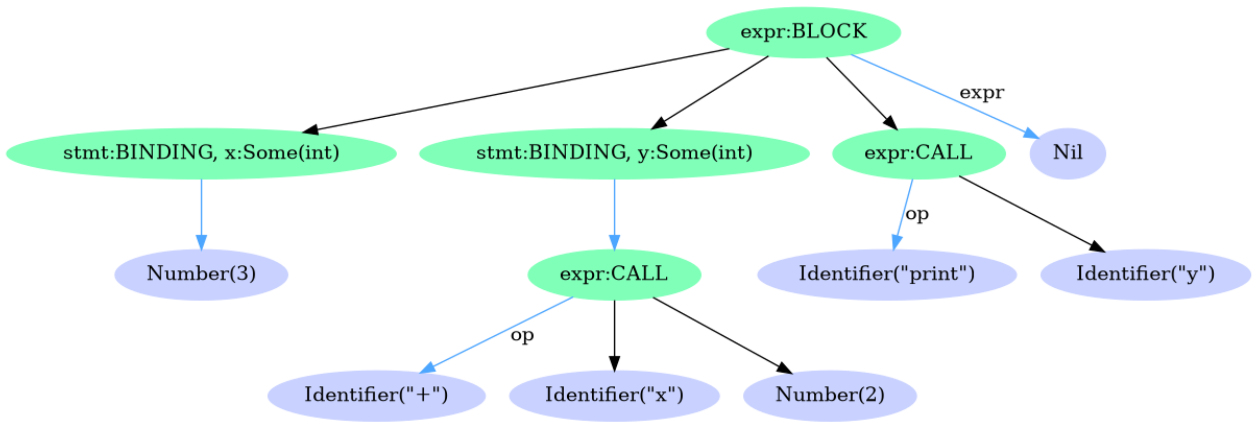

Figure 1.3: AST from Listing 1.5 created using the graphing tool.

and when it reaches 0, the variable is freed (Reference Counting [10]).

In the case of Rust, there is no overhead and the memory management is automatic [11, sec. 4.1]. It is thanks to the strict semantic of the borrow checker that we are able to know where a variable has to be freed at compile time.

1.5 Objectives

As stated in the motivation of the document, one of the objectives is to learn as much as possible which is why we decided to implement the interpreter from zero without the aid of external libraries. The broken down objectives considering design and implementation are:

- Formal syntax, typing, and semantic specifications for our language. The three of them together completely determine how our language is written and interpreted. In the text, natural language explanations will accompany the formal verbiage.
- A library of general parser combinators [12]. By creating a standalone module we get better separation of concerns and encapsulation.
- A parser for our language's grammar using our agnostic parser combinator library.
- A simple static type-checker for the rules we laid out.
- A "tree-walk" interpreter where the AST is not transformed nor compiled but the interpreter traverses it.
- To help aid debugging, we will develop a graphing backend where the AST is translated into the DOT [5]format and rendered using Graphviz [6].
- To further aid development, a small and simple testing framework will be created to check for regressions automatically.

[5] https://en.wikipedia.org/wiki/DOT_(graph_description_language)
[6] https://graphviz.org/

All of the programming will be done in Rust. For a brief introduction see Appendix B, and for more in-depth information see [11].

1.6 Document organization

In the beginning of the chapter we gave a brief overview of the narrative structure of the document. The objective of this section is to present the reader with a schematic description followed by a detailed account.

Roughly speaking, the work is divided into 4 parts excluding this one where each of them is concerned with one of the areas of creating an interpreter: design, parsing, typechecking and interpreting. While the design follows a standard structure, the rest of the chapters are further subdivided into two parts: theory and implementation.

For the more detailed account, we will follow the order of the document. Starting from the design chapter (chapter 2), where, as the name indicates, the language that solves our quintessential problem will be designed. First, the syntax will be laid down in section 2.1 and, then, we will justify our design decisions in section 2.2. Next, we will be able to revisit the problem for the last time and give a complete implementation in section 2.4, where we will further emphasize the advantages in our solution.

In the rest of the text, we will implement an interpreter in three distinct stages: parsing (chapter 3), typechecking (chapter 4) and interpreting (chapter 5). When discussing parsing, we will first review the different approaches and, then, the advantages and disadvantages of each one to finally settle on parser combinators. In the next section, we explain the how they work and motivate it with a few examples in section 3.2. We will close the section with an illustrative excerpt from the implementation in section 3.3.

After parsing comes typechecking (chapter 4). We will start by laying down the most important rules for our language (section 4.1). To further explain them and to aid the reader in understanding the new formalism, we will also explain the rules in plain text. We will end the chapter with a high level overview of the main challenges of the implementation through a few case studies (section 4.2).

In the last chapter, we will tackle the final challenge: the interpreter itself. Because the formalism was already explained in the previous chapter, we will plainly state the rules in section 5.1 with no further explanation. For the remainder we will, once again, explain lightly the implementation using a few examples (section 5.2).

To finalize, a brief overview into how the project is validated and tested is presented in chapter 6, and the general conclusions are given in chapter 7.

2

Design of the language syntax

Now that the jargon has been explained, we can state the most important characteristics for our language:

- It is functional.
- It is strongly typed and has static type checking, albeit a really simple one.
- It has closures.

In this chapter, we will design the syntax for a language that satisfies these characteristics and addresses the problem described in section 1.1. First, we will specify the syntax and give several examples of valid programs that each illustrates a feature of our language. Lastly, we will discuss some of the design decisions that conditioned our choices for the syntax.

2.1 Syntax rules

To closely model a collection of facets, we are going to represent the types and objects of our language as sets. For example, an object having facets `A` and `B` would be $t = \{A, B\}$ with type $T = \{A, B\}$. Even though we use the same symbols for both cases, the distinction will be clear based on the context.

We include the most important constructs in our language below. For the full grammar rules, consult Appendix A. The grammar contains broadly three categories: types, terms and values. The difference between the last two is that terms are interpreted into values which cannot further be reduced. We will see it in detail in chapter 5. The rest of the formalism such as closures will be further explained in chapter 4.

Syntactic forms

$$T ::= \qquad \text{(types)} \qquad (2.1)$$

$$\texttt{int}, \texttt{bool}, \texttt{nil}, \texttt{str} \qquad \text{(primitive types)} \qquad (2.2)$$

$$F_1 + ... + F_n \qquad \text{(derived type } \{F_1, ..., F_n\}\text{)} \tag{2.3}$$

$$\texttt{facet}\{s_1 : T_1, ..., s_n : T_n\} \qquad \text{(facet type)} \tag{2.4}$$

$$(\Gamma_c, \texttt{fn}(x_1 : T_1, ..., x_n : T_n)\texttt{->}R) \qquad \text{(closure type)} \tag{2.5}$$

$$v ::= \qquad \text{(values)} \tag{2.6}$$

$$\texttt{true, false, nil, <number>} \qquad \text{(constants)} \tag{2.7}$$

$$(\Gamma_c, \texttt{fun}\,(x_1, ..., x_n)\ b) \qquad \text{(closure)} \tag{2.8}$$

$$\Sigma_i F_i\{s^i_j : v^i_j\}_{1 \leq j \leq n_i} \qquad \text{(derived value)} \tag{2.9}$$

$$t ::= \qquad \text{(terms)} \tag{2.10}$$

$$v \qquad \text{(values)} \tag{2.11}$$

$$f(t_1, ..., t_n) \qquad \text{(function application)} \tag{2.12}$$

$$\texttt{if}\ b_1\ b_2\ \texttt{else}\ t_3 \qquad \text{(if else)} \tag{2.13}$$

$$\texttt{if}\ t_1\ t_2 \qquad \text{(if)} \tag{2.14}$$

$$\{t_1\texttt{;}\ ...\texttt{;}\ t_n\} \qquad \text{(block)} \tag{2.15}$$

$$\texttt{fun}\,(x_1\texttt{:}T_1, ..., x_n\texttt{:}T_n)\ b \qquad \text{(function)} \tag{2.16}$$

$$\texttt{facet}\{s_1 : t_1, ..., s_n : t_n\} \qquad \text{(facet initialization)} \tag{2.17}$$

$$\texttt{let}\ x\texttt{:}\ T = t\texttt{;} \qquad \text{(let binding)} \tag{2.18}$$

Listing 2.1: Syntax specification for our language.

Observe a few key points:

- *Rule 2.2*. The primitive or pre-defined types are: `int` , `str` , `bool` , and `nil` .
- *Rules 2.3 and 2.4*. The type of a facet contains the names and types of its fields:

  $$\texttt{facet}\{\ s_1 : T_1, ..., s_n : T_n\ \}\ ,$$

  and a derived type is a combination of these facets.
- *Rule 2.5*. The type of a function (without its environment) is:

  ```
  fn(TypeArg1, ..., TypeArgN) -> RetType
  ```
- *Rules 2.12 and 2.17*. Roughly speaking, every time an item has to be evaluated, a term is used. For example, when initializing a facet each field is set to a term, or when a function is called, each argument is a term. Later, in the interpreter, we will see that terms are eagerly evaluated into values which are then stored or passed on.

2.2 Design decision: Null

As it can be seen in the specification (section 2.1), we have both a null type and a null value that we call `nil`. Deciding whether to include null in your language is a difficult choice. The issue is serious enough that Tony Hoare, inventor of null in 1965, recently [1]called null his “billion dollar mistake”.

Why then is null prevalent across so many languages? The answer is convenience. What makes null special is that it can pass as any type. For example, in C it is standard that when an operation fails it returns a null pointer instead of a valid one. The problem is that the programmer is not forced to check if the return value is null because it **is** the same type; and this creates a whole class of bugs [2].

Creating a language without null is no easy task and usually revolves around providing a version of Haskell’s `Either` [3]type. Basically, the idea is to handle errors using monads; in this case, using, arguably, the simplest one. Modeling errors this way is more of an advanced topic and we will not delve into the details in this work. We will be satisfied knowing that it is possible to replace null and evade its pitfalls.

Returning to our task, why then do we have null in our language? It is because null conflates two ideas: the one we described, and expressing that no value is produced. For example, under this point of view, statements “return” null, and the same thing happens for functions with no return type, while loops, etc. The technique of not having null *per se*, but what is called a unit type, is widely used in functional programming languages: for example Haskell and Rust both use the `()` [4]to model it.

2.3 Basic usages

In this section, four short programs will be presented to exemplify the abstract syntax rules we have seen. In the first example, Listing 2.2, we can see the different control structures: if, while, return, and function calling. Further notice that a function is defined by a standard let expression (line 1). In the second example, Listing 2.3, we can see how facets are defined (lines 1 to 5), and how objects take and drop them (lines 9 and 10 respectively). In the next one, Listing 2.4, we can see how `@` combines objects by adding all the facets (line 10), and how fields are accessed (lines 11, 12). More important is the syntax used for disambiguating when two facets have a field with the same name, `::<facet>`. Lastly, in Listing 2.5, we can see how field assignment works.

[1]https://web.archive.org/web/20220307130953/https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

[2]https://owasp.org/www-community/vulnerabilities/Null_Dereference

[3]https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Either.html

[4]https://doc.rust-lang.org/std/primitive.unit.html

```
let print_n = function(n: int) {
    if n < 0 {
        return;
    };
    let i = 0;
    while i < n {
        print(i);
        i = i + 1;
    }
};
print_n(5); // 0,1,2,3,4
```

Listing (2.2) Flow control structures: while, return, if, and function calling.

```
facet A {
    a: int,
}

facet B {
    a: int,
}

let x = A { a: 2} @ B { a: 3};
let y = x.drop::A();
print(y); // {B {a:3}}
```

Listing (2.3) Droping a facet returns a copy of the rest of the object.

```
facet A {
    a: int,
}

facet B {
    a: int,
}

let x = A { a: 2};
let y = x @ B { a: 3};
print(y.a::A); // Facet A: 2
print(y.a::B); // Facet B: 3
```

Listing (2.4) Combining the facets of two objects, accessing the field by disambiguating the name, initializing facets.

```
facet A {
    a: int,
}

facet B {
    a: A,
}

let x = B { a: A{ a: 3 } };
x.a::B = A { a:-1 };
print(x.a::B.a::A); // -1
print(x.a.a); // -1
```

Listing (2.5) Accessing a nested field with facets, initializing facets.

The `tests` directory contains many more edge cases and uses. For more documentation on how to run the tests and interpreting the results, see Appendix D and chapter 6.

2.4 Original problem revisited

We can implement a subset of the original problem to see how we might solve the problems laid down on section 1.1.

```
// Types:
facet Data {
}
facet U {
    username: str
}
facet UC {
    context: str
}
// Functions:
let addUserId = function(data: Data, id: str, val: str) -> Data+U {
    data @ U { username:val }
};
let addUserContext = function(data: Data+U) -> Data+U+UC {
    data @ UC { context:"mock call" }
};

let getUserRecommendations = function(userid: str) -> Data+U+UC {
    let data: Data = Data {};
    let data: Data+U = addUserId(data, "USERID", userid);
    // Log the username. Notice that we dont need to cast because of the new
    // facet and we can just get the field safely.
    print(data.username);
    let data: Data+U+UC = addUserContext(data);
    data
};
getUserRecommendations("test");
```

Listing 2.6: Original problem implementation using facets.

If we recapitulate over the pain points again: now there is enough information at every step of the program to determine if a call is valid. Furthermore, we know which fields are available at every step. All of this using the type system that guarantees all constraints at compile time.

What happens if we were to call the functions in the wrong order in our implementation?

```
    // ...
    let data: Data = Data {};
    let data: Data+U+UC = addUserContext(data);
    let data: Data+U = addUserId(data, "USERID", userid);
    // ...
```

Listing 2.7: Wrong call order with output from the interpreter.

```
    >>>  [TYPE ERROR] Expected arg types to be: (Data+U), got Data.
```

If we were to try to instead access the field at the wrong time a similar error would appear.

```
    // ...
    let data: Data = Data {};
    print(data.username);
    let data: Data+U = addUserId(data, "USERID", userid);
    // ...
```

Listing 2.8: Accessing a field that does not exist at that point in time.

```
    >>>  [TYPE ERROR] Field data.username does not exist.
```

These errors serve two purposes. First warning the programmer that the program is not correct at compile time, and, second, serving as live documentation that needs to be forcefully updated with the code. It is after all the most powerful incentive against documentation rot.

3 Parsing

Parsing is taking free-form text, a text file representing our program, and translate it into a syntactic structure that encodes the relations and the semantic meaning, an AST. In plain words, taking a text file and transforming it into a well-formed data structure or throwing an error.

In this chapter we will first give a brief overview about parsing in general. Then, we will explain a concrete technique: parser combinators. To better understand the theory, we will interleave examples from our own library. Lastly, we will present the complete implementation of a moderately difficult parser.

There are several techniques that can be used to develop parsers:

- Handmade. Most serious languages [1]employ a bespoke parser that can be customized for their needs: expressiveness, error checking, speed, etc. The most common technique and the one proposed on [8] is a recursive descent parser, because they are both performant, easy to code, and maintain.
- Flex, Bison, ANTLR [2]et al. These tools take a formal grammar directly and produce the parser. There are two problems: it is hard to tweak to your liking and, from an academic point of view, there is not much to learn from using these tools.
- Parsing combinators [3]. The brilliant idea is to use function composition as the fundamental operation and defer the execution (see [9] and [13]). Each parser is a function and combinators take several functions and combine them. There are two advantages: they are more elegant as they promote reusability and composability, and are written in the host programming language (not like Bison/ANTLR). They also have disadvantages: working with a higher level of abstraction can lead to complex code, error reporting is more difficult, debugging is also harder and performance is usually worse.

For the interpreter we will choose parser combinators because we can learn a lot by creating our own

[1] See `https://gcc.gnu.org/wiki/New_C_Parser` for GCC, `https://clang.llvm.org/docs/InternalsManual.html#the-parser-library` for Clang.

[2] `https://www.antlr.org/`

[3] See [14] for a good introduction.

little library from scratch. It is also a superb programming exercise because, working within functional code in Rust, forces the programmer to understand how expressiveness is hindered by performance and explicitness.

3.1 Simple parsers

In this section we will see how to create some simple parsers in Rust. The most basic function is `peek`. It looks at the input and returns the next character without consuming it or error if there is no more input (see Listing 3.1). We also create a struct called `Information` that holds the state: the reference to the input, current index and line (used for error reporting purposes). We avoid shared state and make each parser stateless.

```rust
pub struct Information<'a>
{
    pub input: &'a [char], // input array of character
    pub index: usize,      // index of the input where the parser starts
    pub line:  usize,      // line we are in, for debugging purposes
}

pub fn peek(info: Information) -> Result<char, PError> {
    // Get the character at the current position without advancing the index.
    // If we have ran out of input, throw error.
    info.input.get(info.index)
        .ok_or(PError { index: info.index, line: info.line })
        .map(|c| *c)
}
```

Listing 3.1: `peek` function and the struct Information where we hold the state.

Note: In Listing 3.1 there is one "generic parameter" called `'a`. It is not really a parameter but a lifetime specifier. Even though a deeper understanding is not necessary, readers might refer to section B.1 for more information.

See Listing 3.2 for an example of a simple parser that returns a character if it satisfies a condition. Let's break it down step by step:

1. We take a function or a closure as the condition. It takes a char and returns a boolean: true if we match the condition and false if not. Note: we will see later in our deep dive into closures why we need to create a generic parameter, in this case `F` (line 2).

2. In line 4 we create a closure using `move |info: Information|`; namely, the parser that we will return. Note that the return type is omitted because it is automatically inferred.

3. In line 5 we `peek` and get a char (if error, propagate `?`). We then check the condition in line 8 and, either return the char and advance the index, or return an error.

```rust
 1  pub fn parse_char_cond<'a, F>(cond: F) -> impl Parser<'a, char>
 2      where F: Copy + Fn(char) -> bool
 3  {
 4      move |info: Information<'a>| { // Return a closure to defer execution
 5          let c = peek(info)?;
 6          // Do we have to increment the line count?
 7          let line_inc = if c == '\n' { 1 } else { 0 };
 8          if !cond(c) { // Does the character not match the condition, throw error
 9              Err(PError {
10                  index: info.index,
11                  line:  info.line,
12              })
13          } else { // Return the character and advance the input's index for the
14              // next parser.
15              Ok((Information {
16                  input: info.input,
17                  index: info.index + 1,
18                  line:  info.line + line_inc,
19              }, c))
20          }
21      }
22  }
```

Listing 3.2: Function that parses a character that matches the supplied condition.

3.2 Some combinators

We will take the `and` combinator as an example (Listing 3.3).

1. In line 1 we take two parsers: `parser1` and `parser2`. Once again, note that we are using a generic type parameter to define what a parser is.
2. In line 6 we then create a closure in the same manner as in Listing 3.2 to defer execution.
3. Once we have to evaluate the closure, we will call both parsers (lines 9 and 10) and, if both match the exact same part of the input (we check both `Information` and the output, line 11), then and only then do we return success.

```rust
1  pub fn and<'a, O, F, G>(parser1: F, parser2: G) -> impl Parser<'a, O>
2      where F: Parser<'a, O>,
3            G: Parser<'a, O>,
          O: PartialEq // We need equality to compare the output.
{
    move |info: Information<'a>| {
        // Only if the have matched the exact same part of the input and they
        // have the same output.
        let (i1, o1) = parser1.parse(info)?; // Run the first parser.
        let (i2, o2) = parser2.parse(info)?; // Run the second parser.
        if i1 == i2 && o1 == o2 { // Output and finished positions match.
            return Ok((i1, o1))
        } else {
            return Err(PError {
                index: info.index,
                line:  info.line,
            })
        }
    }
}
```

Listing 3.3: The and combinator takes two parsers and returns only if both parsers have matched the same input and produced the same output.

3.2.1 Advanced combinators: Macros over the AST

The bread and butter of our combinator library are the `alt` and the `tuple` combinators [4]. Both have similar implementations, so we will only look at `tuple`.

We want `tuple` to take **any number** of parsers as a tuple and execute them sequentially on the input. If all match, we return a tuple of all the outputs and if one of them fails, we return an error. We have two challenges:

- Each parser can return a different output type and we have to reflect that in the signature.
- We want to implement it for arbitrary arity but we want it to be typechecked. That is why we want it to be a function and not a macro, or exploit varargs like "printf" in C.

Let's tackle the first problem by fixing a tuple with 3 elements. We can model the output types using generic parameters, see Listing 3.4.

```
fn tuple_3<'a, O0, I0, O1, I1, O2, I2>(self: &(I0, I1, I2), info: Information<'a>)
    -> PResult<'a, (O0, O1, O2)>
where I0: Parser<'a, O0>,
      I1: Parser<'a, O1>,
      I2: Parser<'a, O2>
{
```

[4]The API is modelled after the nom library [15] that provides parser combinators in Rust.

```rust
    let (i, o) = tuple_2((self.0, self.1)).parse(info)?;
    let (ii, oo) = self.2.parse(i)?;
    return Ok((ii, (o.0, o.1, oo)));
}
```

Listing 3.4: `tuple` implementation for only three elements.

If we followed this example, we would have n different `tuple_i` implementations. We would like to have one function that accepts an arbitrary tuple, so on we go to solve the second problem. We can define a trait and use it to dispatch, see Listing 3.5.

```rust
pub trait Tuple<'a, O>: Copy {
    // We cannot return a parser and we have to add the thunk inside tuple
    // function because of non existential types.
    fn fold(&self, info: Information<'a>) -> PResult<'a, O>;
}
pub fn tuple<'a, I, O>(parsers: I) -> impl Parser<'a, O>
    where I: Tuple<'a, O>
{
    move |info: Information<'a>| {
        // Dispatch.
        parsers.fold(info)
    }
}
```

Listing 3.5: `Tuple` trait and function that performs the dispatch.

Lastly, to create all the implementations we can use a macro. Rust has two kinds: textual macros and macros over the AST. We will use the second kind because, even though they are slower, they are typechecked and written in Rust and not in a DSL (like textual macros). For the complete implementation see chapter 6.

3.3 Complete example: Parsing comments and whitespace

In a normal lexer-parser architecture, comments are discarded directly in the lexer. In the case of parser combinators it is not that simple because of the following reasons:

1. If we changed `peek` we would be changing *every* parser and that would defeat the whole purpose of building an independent parsing library. For example, if now we were to parse a string like `"//not a comment"` we would have to pass down the state so that `peek` does not consume the comment.

2. If we chose to include it in the language parsers, we would have to change *every* parser because

we allow comments practically everywhere.

As usual, we do not have to invent anything new. We can use the technique outlined on [12], that is: create a combinator the consumes whitespace and comments around other parsers and decorate those.

```
/// It discards the whitespace around the given parser, left and right.
/// It also discards any line comments.
pub fn junk<'a, F, O>(parser: F) -> impl Parser<'a, O>
    where F: Parser<'a, O>
{
    fn comment(info: Information) -> PResult<()>{
        tuple((
            parse_literal("//"),
            many0(parse_char_cond(|c| c != '\n')),
            parse_char('\n')
        )).parse(info).map(|(i, _)| (i, ()))
    }
    move |info: Information<'a>| {
        let (i, (_, o, _)) = tuple((
            w(many0(comment)),
            parser,
            w(many0(comment)),
        )).parse(info)?;
        Ok((i, o))
    }
}
```

Listing 3.6: Defining a combinator the consumes the whitespace and comments around the given parser.

```
pub fn parse_block(info: Information) -> PResult<Expression> {
    // A block is a series of statements that ends with an optional expression
    // that produces a value. If no expression if found, the block returns nil.
    let (i, (_, stmts, expr, _)) = tuple((
        junk(parse_char('{')),
        many0(stmt::parse_statement),
        optional(parse_expression),
        junk(parse_char('}')))).parse(info)?;
    Ok((i, Expression::Block(Box::new(Block {
        stmts: stmts,
        expr: expr.unwrap_or(Expression::Value(Value::Nil)),
    }))))
}
```

Listing 3.7: Example of how “junk” is used in parsing a block.

4 Typechecking

Typecheking refers to the action of validating the program against some constraints at compile time **before** interpreting anything.

There is a lot of formalism behind type theory but, following the general spirit of this work, we will try to keep things simple because we have a lot of concepts to cover. We are going to mostly follow [16] and, for a much rigorous treatment, check it directly.

In this chapter, we will first present the rules for typechecking our language in the usual rigour. We will then explain the most important rules in natural language to further familiarize the reader with the notation. Lastly, we will explain some design decisions of our implementation with illustrative examples of what the code looks like.

4.1 Rules

We are going to employ mostly standard notation ([16]) except when dealing with imperative constructs/side effects. To model actions like defining a variable in a scope, we are going to use the following symbols:

- Γ: An environment with the set of correspondences of variable↔type or variable↔value, for typing and execution respectively.
- Rules produce tuples: (v, Γ'), corresponding to the result, and the possibly altered environment in the cases where we have introduced a variable binding.
- The program is not represented as a big expression like in Lisp or Lambda Calculus, but as a vector of terms $\{t_0; \ldots; t_n\}$. We will introduce the necessary rules so side effects such as binding are preserved only in their scope.
- When a result is not going to be used, we will name it "_".

Each rule will be presented alongside an explanation; see the correspondence between the equation numbers and the enumerations in the text below.

Typing

$$\frac{\Gamma \vdash t_1 :: (\texttt{bool, }\_), t_2 :: (T, \_), t_3 :: (T, \_)}{\texttt{if}\ t_1\ t_2\ \texttt{else}\ t_3\ ::\ (T, \Gamma)} \tag{4.1}$$

$$\frac{\Gamma \vdash t :: (T, \_)}{\texttt{let}\ x\texttt{:}\ T = t\texttt{;}\ ::\ (\texttt{nil}, \Gamma \cup \{x \mapsto T\})} \tag{4.2}$$

$$\frac{\Gamma \vdash x :: (T, \_) \quad \Gamma \vdash t :: (T, \_)}{x = t\texttt{;}\ ::\ (\texttt{nil}, \Gamma)} \tag{4.3}$$

$$\frac{\Gamma \vdash t_1 :: (T_1, \_), ..., t_n :: (T_n, \_) \quad \Gamma \vdash f :: (\texttt{fn}(x_1 : T_1, ..., x_n : T_n)\texttt{->}R, \_)}{\Gamma \vdash f(t_1, ..., t_n) :: (R, \Gamma)} \tag{4.4}$$

$$\frac{\Gamma \vdash b :: R}{\Gamma \vdash \texttt{fun}(x_1\texttt{:}T_1, ..., x_n\texttt{:}T_n)\texttt{->}R \quad b :: (\texttt{fn}(x_1 : T_1, ..., x_n : T_N)\texttt{->}R, \Gamma)} \tag{4.5}$$

$$\frac{\Gamma^0 \vdash t_1 :: (\_, \Gamma^1) \quad ... \quad \Gamma^i \vdash t_{i+1} :: (\_, \Gamma^{i+1}) \quad ... \quad \Gamma^{n-1} \vdash t_n :: (T, \Gamma^n)}{\texttt{\{}t_1\texttt{;}...\texttt{;}t_n\texttt{\}}\ ::\ (T, \Gamma^n)} \tag{4.6}$$

$$\frac{}{\texttt{facet}\ F\texttt{\{}s_1 : T_1, ..., s_n : T_n\texttt{\}}\ ::\ (\texttt{nil}, \Gamma \cup \{F \mapsto \texttt{facet}\texttt{\{}s_1 : T_1, ..., s_n : T_n\texttt{\}}\})} \tag{4.7}$$

$$\frac{\Gamma \vdash t_1 :: (T_1, \_), t_2 :: (T_2, \_), T_1 \cap T_2 = \emptyset}{t_1\texttt{@}t_2 :: (T_1 \cup T_2, \Gamma)} \tag{4.8}$$

$$\frac{\Gamma \vdash F :: (\texttt{facet}\texttt{\{}s_1 : T_1, ..., s_n : T_n\texttt{\}}, \_) \quad \forall i\, \Gamma \vdash t_i :: (T_i, \_)}{\Gamma \vdash F\texttt{\{}s_1 : t_1, ..., s_n : t_n\texttt{\}}\ ::\ (F, \Gamma)} \tag{4.9}$$

$$\frac{\Gamma \vdash F :: (\texttt{facet}\texttt{\{}s_1 : T_1, ..., s_n : T_n\texttt{\}}, \_) \quad \Gamma \vdash o :: (T, \_) \quad F \in T}{\Gamma \vdash o\texttt{.}s_i :: (T_i, \Gamma)} \tag{4.10}$$

$$\frac{\Gamma \vdash F :: (\texttt{facet}\texttt{\{}s_1 : T_1, ..., s_n : T_n\texttt{\}}, \_) \quad \Gamma \vdash o :: (T, \_) \quad F \in T \quad \Gamma \vdash t :: (T_i, \_)}{\Gamma \vdash o\texttt{.}s_i = t\ \texttt{;}\ \ :: (T_i, \Gamma)} \tag{4.11}$$

Listing 4.1: Typechecking specification for our language.

We are going to define a type as the *shape* of the data. Thus, if two elements are of the same type, the underlying data has the same shape. It is after all a design decision because having the same type can mean different things in different languages (for more information see subsection C.1.1). Our first set of rules for typechecking will be:

- *Rule 4.1*. All the branches of an `if` statement should have the same type and the condition should be `bool` .
- *Rule 4.2*. When assigning a value to a variable, the expression type should match the type hint:

```
let var: <type hint> = <expression>
```

Additionally, the type of a bound variable is the one given in its definition in the `let` expression.

- *Rule 4.3.* When modifying a variable, the new value must have the variable type as defined in their let expression.
- *Rule 4.4.* When calling a function, each argument has the type that matches the signature of the function. Example: `sum("12", 1)` does not typecheck because the type of “12” is `str` and not `int`.
- *Rule 4.5.* A closure stores the current environment when it is defined. For a closure to be created successfully, the type of the body must match the signature's return type.
- *Rule 4.6.* A block evaluates to its last expression. Notice how the environments are passed down for each expression evaluation. If this were not the case, variable declarations and modifications would be lost even inside the same scope.

4.1.1 Simple typechecking

We are going to follow a very simple strategy of resolving the types recursively. Let's take as an example an `if` expression. We would first resolve the condition expression and see that the **result** is indeed `bool`, then resolve both branches and check that the types match. The rest of the rules are very similar except for modification of a variable.

If we think about it, we need a way to keep track of the variable definitions. We also have to take into account the scope and if a variable is *shadowed* [1]. It is strikingly similar to our first discussion in section 1.4 in that we need to keep track of the correspondence identifier↔type at any point in the program. We will adopt the same solution of nested environments.

Let's take Listing 4.2 as an example:

When we encounter the `if` expression in line 4, we try to apply the type checking rule 4.1. For the condition, the steps would be roughly the following:

1. It is a function call of `>` with arguments `n` and `0`. We check the environment [2]for the function signature and see that it is:

```
fn(int, int) -> bool
```

[1] Recall that in the examples we presented, we were able to create a variable in the inner scope with the same name as a variable in the outer one, and that the former had precedence over the latter when resolving. Formally, this is called “variable shadowing”. For more information see [11, sec. 3.1].

[2] In our example picture we have not included the definition of functions from the prelude to not clutter the picture. The prelude is a set of names which are imported automatically for every program.

```
 1 let a = 42;
 2 let b: str = "42";
 3 let sum_a = function(n: int) -> int {
 4     if (n > 0) {
 5         n + a
 6     } else {
 7         n
 8     }
 9 };
10 sum1(2);
```

Listing 4.2: Example program with type annotations.

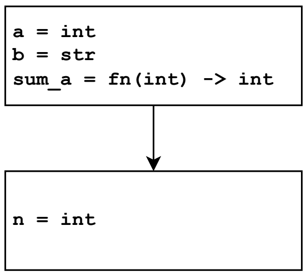

Figure 4.1: Nested list of scopes at line 4 for program Listing 4.2.

2. We typecheck `n`. It is a variable so we get the type from the environment (Figure 4.1): `int`.

3. We typecheck `0`: It is a number literal and we assign it the type `int`.

4. We can see that the arguments match the signature so the condition typechecks to `bool` without errors.

4.1.2 Facets

In our language we have the ability to define facets: new types that consist of several fields of previously defined types. We also need to keep track of their fields if we want to typecheck their access. For this problem, we have to define a `struct` that stores the map of field↔type. Facets are declared following *Rule 4.7*.

Lastly, we can also combine many facets into one type using the `@` operator to create a Derived Type. Just like we saw on the design (section 1.2), we can represent any combination of facets, including one, as a set.

With both considerations, the rules are as follows:

1. *Rule 4.8*. When combining types with `@`, they cannot share a facet. It would be ambiguous because the result would only have one, it is a set after all. If successful, the result is the union of both sets.

2. *Rule 4.9*. Instantiating a facet creates a derived type with that one facet. And, implicitly: a derived type cannot contain primitive type as facets.

3. *Rule 4.10*. Accessing field requires that the type is derived and that the field itself exists in the chosen facet.

4. *Rule 4.11*. Modifying a field is similar to modifying a variable in that the type of the definition of the facet must be used.

4.2 Implementation

In the implementation, by far, most of the code consists of error checking and error reporting. If we think about it, our typechecker is proving that the program that we are going to interpret is syntactically correct. It not only has to do that, but it also has to present the user with readable error messages in case something goes wrong.

In a real compiler, error reporting with pretty output, line numbers and suggestions, takes up even more code. The rust compiler is notorious for doing this correctly, look at the following example:

```
    error: expected `;`, found keyword `if`
  --> dyntypes/src/typecheck/mod.rs:68:130
   |
68 |                  let facet = it.next().unwrap_or_else(||
        panic!("Attempting to access a field without specifying facet, {}!", id))
   |
69 |                  // If we have asked for a facet that our object does not
70 |                  // have, error.
71 |                  if !fs.contains(facet) {
   |                  -- unexpected token
```

Our starting point is the environment from last section (subsection 4.1.1), this time as a collection of correspondences between identifiers and types. For now, we are going to keep it as a black box because the implementation is tricky and is best explained in the context of the interpreter (subsection 5.2.1). We just have to know that we can bind and retrieve values while the rules for scoping are respected.

We have to keep the following values in the environment:

```rust
#[derive(Clone, Debug, Hash, Eq)]
pub enum Type {
    // A primite type: int, str, bool, etc.
    Primitive(Facet),
    // Derived type is a set of facets.
    Derived(BTreeSet<Facet>),
    // A function contains its signature, type of the args that it receives and
    // output type.
    Function(Box<Signature>, Option<Box<Signature>>),
    // Struct is a correspondence name<->field. We use this to internally
    // represent facet types.
    Struct(BTreeMap<Identifier, Type>),
    // Macros, @, and, or, etc. that dont abide by the normal evaluation rules.
    SpecialForm(Identifier),
}
```

Listing 4.3: Type enum.

Lastly, we define a trait that we are going to implement for each data structure that can be type-checked (Listing 4.4). The trait only contains one function that, if successful, returns the type of the checked structure and, if not, returns an error with an appropriate message.

```
pub trait TypeCheck {
    fn type_check(&self, env: &TEnv) -> Result<Type, TError>;
}
```

Listing 4.4: Trait for typechecking.

4.2.1 Case study: Typechecking a facet initialization

Instead of something simple like an `if` expression, we have chosen a more prototypical example of what the code looks like: full of error checking and reporting.

```
// Get the struct type to see the fields and arity.
let sty = env.borrow().lookup_value(name)
    .unwrap_or_else(|| panic!("Struct {} not defined", name));
// Check that it is indeed an struct type.
if let Type::Struct(ref tmembers) = sty {
    // Check that we are using the same number of fields per the definition.
    if tmembers.len() != members.len() {
        return Err(TError {
            reason: format!("Missing fields in facet {} initialization", name)
        });
    }
    // For each member, typecheck it and see if it exists on the definition and
    // if it matches the type.
    for (mid, mexp) in members {
        let t_mexp = mexp.type_check(env)?;
        let t_expected = tmembers.get(mid)
            .ok_or(TError {
                reason: format!("Field {} not found in struct {}", mid, name)
            })?;
        if &t_mexp != t_expected {
            return Err(TError {
                reason: format!("Assignment to facet field {}.{} is of wrong
                                type. Got {}, expected {}",
                                name, mid, t_mexp, t_expected)
25             });
26         }
27     }
28 } else {
29     return Err(TError {
30         reason: format!("Expected facet, got {}", name)
31     });
32 }
33 Ok(Type::new_derived().insert(name))
```

Listing 4.5: Implementation for facet initialization in the type system.

Observe that we have to check and report four different kinds of errors in this short snippet:

- In line 3 we check that the struct type has been defined before trying to instantiate it.
- In line 30 we check that the given type is indeed a facet and can be initialized.
- In line 9 we check that the number of fields supplied and the expected number match.
- In line 18 we check that the field names supplied are correct.
- In line 24 we check the that the type of the field in the facet matches the value the programmer is assigning to it.

Observe that line 15 is of special significance. The crux of the typechecker is the `type_check` function declared in Listing 4.4. It is a recursive call that bubbles up the previous result to typecheck bigger structures. In this case, we are using it pretty straightforwardly to get the type of the expression whose value is to be assigned to the field and see if it matches the expected type. Also note that at the end of the function (line 33) we return the type of the facet initialization expression: a derived set containing only said facet.

4.2.2 Case study: Return statements

This is a good example of a not so straight forward implementation. In this case, we want to type-check return statement that are *nested* arbitrarily deep. That means that, in the middle of typechecking another rules, we have to see if the return matches the outer function. How could we do that?

We could pass the function down for every call but that is very inelegant considering we have the environment at hand. Our solution is to choose a reserved identifier, in this case `"."` and bind the current function to it and the environment will take care of the rest. See line 5 in Listing 4.6 for an example of how we bind the function using the environment, and line 1 in Listing 4.7 for an example of how we retrieve it to later use it.

```rust
// Create a new scope for the function.
let scope = Environment::from_parent(env);
// Bind the function because we have to typecheck all the nested return
// statements. When we encounter one, we resolve "." and we get the function.
scope.borrow_mut().bind_value(Identifier::from("."), &fun_typ);
```

Listing 4.6: When we encounter a function definition, we bind it.

```rust
let ffun = env.borrow().lookup_value(&Identifier::from("."))
    .ok_or(TError {
        reason: "Return outside of a function".to_string()
    })?;
if let Type::Function(ref sig, _) = ffun {
    let ret = e.type_check(env)?;
    if ret != sig.ret {
        return Err(TError {
            reason: format!("Return value {} does not match signature {}",
                            ret, sig.ret)
        });
    }
    return Ok(ret);
} else {
    panic!("Error: expected '.' to be bound to a function but it is bounded to \
    something else");
}
```

Listing 4.7: In a return statement, we get back the function we are in.

5 Interpreter

Interpreting the code is the last step of our architecture. At this point we have an AST that was "proven" by the typechecker to be correct, which means that we only have to concern ourselves with the execution part. For most constructs it is pretty straightforward so, in this chapter, we will mostly concern ourselves with the details of the implementation. But first we will present the semantic specification based on the formalism described on section 4.1. Then, we will introduce the implementation generally and, lastly, walk through a few examples.

5.1 Rules

We build on top of the framework for imperative constructs in section 4.1. The main difference is that, when dealing with semantics, we will not choose small-steps semantics like in [16], but big-step semantics ([17]) that more closely model actions such as declaring a variable or modifying the environment. There is a new symbol, $\Downarrow$, that, informally, means evaluate a term until it becomes a value and cannot be further evaluated. Once again, we are not going to explain rigorously the formalism and, instead, readers unfamiliar to big-step semantics who want to deepen their understanding should check the references directly.

Semantics

$$\frac{\Gamma \vdash t_1 \Downarrow (\texttt{true}, \_), t_2 \Downarrow (v, \_)}{\texttt{if}\ t_1\ t_2\ \texttt{else}\ t_3\ \Downarrow\ (v, \Gamma)} \quad \frac{\Gamma \vdash t_1 \Downarrow (\texttt{false}, \_), t_3 \Downarrow (v, \_)}{\texttt{if}\ t_1\ t_2\ \texttt{else}\ t_3\ \Downarrow\ (v, \Gamma)} \tag{5.1}$$

$$\frac{}{\Gamma \vdash (v, \Gamma) \Downarrow (v, \Gamma)} \tag{5.2}$$

$$\frac{\Gamma \vdash t_1 \Downarrow (v_1, \_), t_2 \Downarrow (v_2, \_)}{t_1 \texttt{@} t_2 \Downarrow (v_1 \cup v_2, \Gamma)} \tag{5.3}$$

$$\frac{\Gamma \vdash t \Downarrow (v, \Gamma')}{\texttt{let}\ x\texttt{:}\ T = t\texttt{;}\ :: \ (\texttt{nil}, \Gamma \cup \{x \mapsto (v, \Gamma')\})} \tag{5.4}$$

$$\frac{\Gamma \vdash t \Downarrow (v, \_)}{x = t\texttt{;}\ \Downarrow \ (\texttt{nil}, (\Gamma \setminus \{x \mapsto \_\}) \cup \{x \mapsto v\})} \tag{5.5}$$

$$\frac{}{\Gamma \vdash \texttt{fun}(x_1, ..., x_n)\ b \Downarrow ((\Gamma, \texttt{fun}(x_1, ..., x_n)\ b), \Gamma)} \tag{5.6}$$

$$\frac{\Gamma^0 \vdash t_1 \Downarrow (\_, \Gamma^1) \quad ... \quad \Gamma^i \vdash t_{i+1} \Downarrow (\_, \Gamma^{i+1}) \quad ... \quad \Gamma^{n-1} \vdash t_n \Downarrow (v_n, \Gamma^n) T}{\Gamma \vdash \texttt{\{} t_1\texttt{;} ...\texttt{;} t_n \texttt{\}} \Downarrow (v_n, \Gamma^n)} \tag{5.7}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 \Downarrow (v_1, \_), ..., t_n \Downarrow (v_n, \_) \quad \Gamma \vdash f \Downarrow ((\Gamma_c, \texttt{fun}(x_1, ..., x_n)\ b), \_) \\ \Gamma_c \cup \{x_i \mapsto v_i\}_{1 \leq i \leq n} \vdash b \Downarrow (v, \_)\end{array}}{\Gamma \vdash f(t_1, ..., t_n) \Downarrow (v, \Gamma)} \tag{5.8}$$

$$\frac{\forall i\ \Gamma \vdash t_i \Downarrow (v_i, \_)}{\Gamma \vdash F \texttt{\{} s_1 : t_1, ..., s_n : t_n \texttt{\}} \Downarrow (F \texttt{\{} s_i : v_i \texttt{\}}_{1 \leq i \leq n}, \Gamma)} \tag{5.9}$$

$$\frac{\Gamma \vdash o \Downarrow (T, \_) \quad F \texttt{\{} s_i : v_i \texttt{\}}_{1 \leq i \leq n} \in T}{\Gamma \vdash o\texttt{.}s_i \Downarrow (v_i, \Gamma)} \tag{5.10}$$

$$\frac{\Gamma \vdash o \Downarrow (T, \_) \quad F \texttt{\{} s_i : v_i \texttt{\}}_{1 \leq i \leq n} \in T \quad \Gamma \vdash t \Downarrow v}{\Gamma \vdash o\texttt{.}s_j = t\ \texttt{;}\ \Downarrow (\texttt{nil}, (\Gamma \setminus \{o \mapsto F\{s_i : v_i\}_i\}) \cup \{o \mapsto F\{s_i : v_i\}_{i \neq j} \cup \{s_j : v\}\})} \tag{5.11}$$

Listing 5.1: Evaluation semantics specification for our language.

After explaining each rule individually in chapter 4, readers should be already familiar to the formalism. Additionally, semantics are more intuitive because most languages employ the same constructs. For these reasons, we will not explain every rule and, instead, focus on a few examples.

The first rule, 5.1, splits the if into two cases, if the condition is true we return the result of evaluating the first branch, otherwise return the result of evaluating the other branch. As usual, the devil is in the details. For instance, both branches are not evaluated, only the one that is selected by the predicate. This means that side effects such as altering a variable or printing to the screen only happen for the evaluated branch, as expected.

Another important rule is 5.2 that says that values evaluate to themselves, for example, a string evaluates to itself. Seemingly simple, it is crucial because it serves as the base case of the recursion. Lastly, rule 5.6 says that when we create a function, we evaluate it into a closure storing the environment at that point in time.

The rest are concerned with function application (rule 5.8), facet instantiation(rule 5.9), facet combining (rule 5.3), let binding (rule 5.4), and running blocks of statements (5.7). These rules are very similar to their homonyms in chapter 4.

5.2 Implementation

Once the rules have been set, or even from common sense, we can “transcribe” them to implement the interpreter. The code is far simpler than that of the typechecker because we do not care about error reporting or correction. If things go wrong we can crash (safely) the program because that means we committed a mistake in the type checker. Even if we look at the most contrived example, see Listing 5.2, there is only one control flow, and it is indeed much sorter than similar implementations in the typechecker.

```
// Create a new scope.
let scope = Environment::from_parent(&env);
let Function{ arg_names, body, ..} = fun.as_ref();
// Evaluate all the args.
for (name, arg) in arg_names.iter().zip(args) {
    (*scope).borrow_mut().bind_value(name.to_owned(), &arg)
}
// Evaluate the function.
let res = body.eval(&scope);
```

Listing 5.2: Executing a function call in the interpreter.

There were only two difficult design decisions: return statements and modeling the environment.

5.2.1 Case study: Environment

We need to build a tree of environments where some of them are shared. For example, when a closure captures its environment, we want to store a reference and not a copy because it might be changed later on (see Listing 5.3) and that change has to be reflected.

```
let n = 3;
let sum_n = function(i: int) -> int {
    n + i
};
print(sum_n(1)); // 3 + 1 = 4
n = 2;
print(sum_n(1)); // 2 + 1 = 3
```

Listing 5.3: The closure stores a reference to n and, when n is changed, it is changed also for the closure.

It is clear that we need shared ownership because both, the outer context and the closure *own* the environment. Any of them could outlive the other and we have to ensure that the environment is not freed; that is why references do not suffice.

Having multiple owners is directly against the semantics of Rust that, by forcing the programmer to explicitly think about ownership, can have compile time memory management. It does not forbid it though, but we have to resort to what all languages use: a GC, in our case, reference counting (recall section 1.4). It is already provided as a wrapper type in Rust `Rc<T>` that does everything automatically. The only caveat is that we cannot mutate the inner `T` because that would violate the "max one mutable reference" rule. Rust also provides another type `RefCell` that lets us mutate internally [1]by checking at runtime whether we have more than one mutable reference (and panicking if we do).

```rust
#[derive(Debug, Clone)]
pub struct Environment<T>
    where T: Clone
{
    env: Env<T>,
    // Option because the topmost node does not have a parent.
    parent: Option<Rc<RefCell<Environment<T>>>>
}
```

Listing 5.4: Definition for our tree of environments.

5.2.2 Case study: Facet fields

Resolving the facet fields proved very tricky. We could have changed what an `Identifier` is and represent it using a vector of fields. However, that is not a very flexible implementation. For instance, we want to be able to omit the explicit facet annotation in the field if there is no ambiguity (recall the syntax from Listing 2.5). Plus, now we have to think in terms of two types of identifiers because fields are not allowed in let expressions; which means that have to create a new datatype.

Because facet fields were added when most of the scaffolding for the interpreter was already there, I decided to keep everything as is and perform checks instead of changing `Identifier`. This means that when resolving a field against the environment, it is not as simple as looking for the key in the HashMap. Apart from the error handling, for each field we have to do the following (also see Listing 5.5):

1. Get the type associated to the first identifier and check that it is derived (i.e. has facets). We perform this step by querying the whole environment, not just the current scope.
2. Get the next field and facet delimited by "." and "::" respectively.
3. Check that such a facet exists in the current derived type and that inside said facet there exist a field with the given name.
4. Get the type. If there are more fields, go to step 1. If this is the last field return the value successfully.

[1] Mixing Rc and RefCell is such a common pattern that there is a chapter in [11] dedicated to it.

```rust
pub fn lookup_field(&self, id: &Identifier) -> Option<IValue> {
    // We try to recursively resolve identifiers with dots as
    // <struct1>.<struct2>.<structN>.field. Note: if there is no dot, main_id
    // is directly the variable: "abc".split(".") == ["abc"]
    let mut it = id.split('.');
    // It is never gonna fail because it is the first one.
    let main_id = it.next().unwrap();

    // Get the type to start traversing for fields.
    let mut v = self.lookup_value(&main_id.to_string())?;
    for field in it {
        // We are now looking at main.{field} where {field} can have nested fields.
        // First of all we have to make sure it is a derived type.
        if let IValue::Derived(ref fs) = v {
            // Then we have to extract the "turbofish" name in {field} i.e.
            // a::A that is field a of struct A.
            let mut it = field.split("::");
            let field_alone = it.next().unwrap();
            let facet = it.next().unwrap();

            // We can unwrap because it has been type checked.
            v = fs.get(facet).unwrap().get(field_alone).cloned().unwrap();
        } else {
            panic!()
        }
    }
    Some(v)
}
```

Listing 5.5: Looking up a field in the environment.

Binding fields is pretty similar.

5.2.3 Case study: Return statements

We saw in subsection 4.2.2 that return statements are complicated because they can appear arbitrary deep inside the control flow and they have to be “propagated” to the outside function. In [8], the author proposes the use of Java exceptions because of their similar semantics. Once an exception is thrown inside the interpreter, the control flow is unwound until the function call catches it. Unfortunately for us, there are no such things as exceptions in Rust because of the extra cost. We can nevertheless model returns as a new data type and make expressions that contain a return evaluate to that. For statements, we just have to be mindful and check the type, if it is a return use it and, if not, discard it (statements do not produce values). Lastly, we have to check for returns inside loops to exit them early. The resulting code is very simple, see Listing 5.6.

```
Statement::Expression(e) => {
    // Statements return empty (): in our language, null except if we are
    // bubbling up a return statement.
    if let i @ IValue::Return(_) = e.eval(env) {
        return i;
    } else {
        return IValue::Nil;
    }
}
Statement::While { condition, body } => {
    ...
    while condition.eval(&scope).get_boolean().unwrap_or(false) {
        // If we get a return, break from the while early.
        if let i @ IValue::Return(_) = body.eval(&scope) {
            return i
        }
    }
}
```

Listing 5.6: Executing return statements and bubbling up the value.

6 Implementation and validation

The purpose of this chapter is to explain the structure of the implementation in itself. In the previous chapters, we have seen a few of the technical challenges in isolation, but we have never talked about the big picture.

The complete project consists of 2669 lines of Rust code divided into several modules [1]:

- **lib.rs**. Some definitions that will be used across modules extensively, for instance the definition of a parser.
- **parser**. It is the library of general parser combinators, the primitives that we use to create our language parser. It is further subdivided into parsers and combinators of increasing complexity.
- **lang**. This module builds upon our agnostic library of parser combinators to create bespoke parsers for our language. There are also numerous primitives and utility functions but, in general, it is structured in a hierarchical manner where one set of parsers uses another. Lastly, it also contains the **graphing** submodule that creates a visual representation of the AST.
- **typecheck**. It defines the traits, structures and functions necessary for typechecking along with its implementation.
- **eval**. It contains all the necessary definitions and implementation for evaluation. Its structure is very similar to that of the typecheck module.
- **impl/lib**. It exports two macros that are the core of our parser module, `alt` and `tuple` that let us combine parsers.

A common heuristic used to gauge the relative complexity of each module is to calculate its percentage of lines of code (LOC) in relation to the whole project. Said numbers can be found in Table 6.1

The complete source code is available in a public git repository: https://gitlab.com/letFunny/dyntypes.

[1] For technical reasons we had to create another "crate" for the macros. A Rust crate is a standalone library that can be distributed in itself.

module	LOC (absolute)	LOC (relative)
parser	297	11%
lang	1079	40%
typecheck	514	19%
eval	470	17%
impl/lib	192	7%
rest	117	4%

Table 6.1: Lines of code (LOC) absolute and relative per project module.

6.1 Validation

A project of this complexity, where modules depend on each other, needs testing. This is especially important when writing an interpreter because, when a program fails, it is difficult to debug the issue. For example, when a program produces the wrong result, it may not be the interpreter that is wrong, it may be the fault of the parser that misunderstood the syntax. Furthermore, debugging parsers combinators is inherently difficult because you are dealing with function recursion and huge stack traces. Among other reasons, this is why we created a graphing backend. When a program fails, we are able to quickly check the AST visually to see if it was a fault of the parser.

Returning to the testing aspect, end-to-end tests were preferred over unit testing each module. One of the reasons is that, at the end of the day, what we care about is that the interpreter as a whole works, and produces either the correct result or the relevant error. The other one is that building the project was an iterative and learning process, thus unit tests would have to be constantly rewritten.

Validation results: A bespoke testing framework was created in Python where currently we have **70** test programs. For more information on how the testing works see Appendix D.

7 Conclusions and future work

Let's recall where we started to see where we were headed. Through an example, we first discussed the pattern of not using the type system because of its stiffness. This, we found, led to several problems such as documentation rot or replacing types with a mixture of casts and ad hoc tests.

We then explained the framework under which we will attempt to solve the previous issues: facets. Using them, we designed a new language that had two objectives. The first and the obvious one is to showcase how facets can solve the problem by bringing more flexibility to the type system. The second, due to the nature of this work, is to learn as much as possible and convey, in this text, the main challenges of creating an interpreter from the ground up. This proved very enriching because, by knowing about the full picture, we could tweak design decisions that affected seemingly unrelated areas. For instance, when designing the syntax we often only think about expressiveness, however we learnt that speed and explicitness of the interpreter also have to be a consideration from day one. We finalized the chapter by giving a full implementation that corresponded to the original problem in our language.

From here on, the rest of the text is devoted to explaining the three areas of our interpreter: parsing, typechecking and executing. Each filled with theory, a formal specification of our language, and some of the most important implementation details. Even though we could not cover a lot of topics related to the implementation itself, we saw a glimpse of the difficulty that the interpreter entails. Maybe the most illustrative example is the parsing chapter. If we take the usual route and use a program to translate our grammar we would be done, but what if we wanted to do parsing ourselves like most mature languages? We saw the even under one of the most simple paradigms: parser combinators, things were very difficult. It is nevertheless a very rewarding task, and one that everyone should undertake at some point because, after all, there is no better way to learn something than to implement it.

The future work is to build upon the scaffolding we have created. Many areas could use improvement and novel techniques. On the one hand, if we focus on the design and expressiveness, we discussed in the text that generics fit naturally. We could then upgrade the type system to a more complex type theory. On the other hand, if we were to focus on the interpreter, we could upgrade the implementation's speed and usability considerably. For the former, we would have to create a new more performant interpreter that is not tree-walk but probably a VM à la Java. For the latter, we could improve parsing by having better error messages by implementing more advanced techniques such as the one described in [18].

7.1 Future work: generics

If we had to choose one among the many improvements we have discussed, it would clearly be generics. There is a very natural statement that we would like to write in our language: take any type that has facet `A` and add facet `B`. Without generics, it is simply not possible. That is why, in the remainder of this section, we will lay down the design for future work to build upon.

The syntax will be modeled after Rust. See Listing 7.1 for an example program that showcases it, and see Appendix A for the complete syntactic rules.

```
 1  facet A { a : str }
 2  facet B {}
 3  facet Base {}
 4
 5  let fun = function<T: A>(t: T) -> T+B {
 6      print(t.a);
 7      // Result is T+B
 8      a @ B {}
 9  };
10  let obj: Base+A = Base {} @ A { a : "working" };
11  let obj_2: Base+A+B = fun(obj);
```

Listing 7.1: Example of the usage of generics, and how the natural problem of adding facets to arbitrary types can be modeled.

Observe that in line 5 we are basically accepting any type T that has facet A. More formally, it involves what is called a subtyping relationship; we would say that T is such that $T < \{A\}$. In our language this relationship is easily modelled after the subset operation ($\subset$) on sets. As a result, $T < \{A\}$ means $\{A\} \subset T$ or, in this case, $A \in T$. Another important point is that when we call the function at line 11, `T` is set to `Base+A` and the function signature is coerced into `Base+A -> Base+A+B`. In layman terms, it is as if the function was defined `Base+A -> Base+A+B` for that particular function call.

In the above paragraph, in that glimpse into how generics work, we already see that the theory is indeed quite complex. In addition to the difficulty of the implementation, that is one of the reasons that generics were not included in the interpreter, even tough they are the preferred improvement. The other reason is that we have many topics to cover in this work, and we prefer to leave it as a future enhancement than to do a poor job at explaining all the different ideas, lacking depth in each one.

Bibliography

[1] Juan De Lara, Esther Guerra, and Jörg Kienzle. “Facet-oriented Modelling”. In: *ACM Transactions on Software Engineering and Methodology* 30.3 (May 2021). ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3428076. URL: https://dl.acm.org/doi/10.1145/3428076 (visited on 02/09/2022).

[2] Matteo Baldoni, Guido Boella, and Leendert van der Torre. “powerJava: ontologically founded roles in object oriented programming languages”. In: *Proceedings of the 2006 ACM symposium on Applied computing - SAC ’06*. the 2006 ACM symposium. Dijon, France: ACM Press, 2006. ISBN: 978-1-59593-108-5. DOI: 10.1145/1141277.1141606. URL: http://portal.acm.org/citation.cfm?doid=1141277.1141606 (visited on 04/25/2022).

[3] Stephan Herrmann. “A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java”. In: (2007).

[4] Raku. *Raku documentation: syntax role*. URL: https://docs.raku.org/syntax/role (visited on 04/25/2022).

[5] Max Leuthäuser and Uwe Aßmann. “Enabling View-based Programming with SCROLL: Using roles and dynamic dispatch for etablishing view-based programming”. In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*. MORSE/VAO ’15: Joint 2015 MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering. L’Aquila Italy: ACM, July 21, 2015. ISBN: 978-1-4503-3614-7. DOI: 10.1145/2802059.2802062. URL: https://dl.acm.org/doi/10.1145/2802059.2802062 (visited on 04/26/2022).

[6] Fernando Sérgio Barbosa and Ademar Aguiar. “Modeling and Programming with Roles: Introducing JavaStage”. In: (2012).

[7] Supasit Monpratarnchai and Tamai Tetsuo. “The Implementation and Execution Framework of a Role Model Based Language, EpsilonJ”. In: *2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*. 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing. Phuket, Thailand: IEEE, 2008. ISBN: 978-0-7695-3263-9. DOI: 10.1109/SNPD.2008.103. URL: http://ieeexplore.ieee.org/document/4617382/ (visited on 04/26/2022).

[8] Robert Nystrom. *Crafting interpreters*. Daryaganj Delhi: Genever Benning, 2021. ISBN: 978-0-9905829-3-9.

[9] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. 2nd ed. Cambridge, Mass. : New York: MIT Press ; McGraw-Hill, 1996. ISBN: 978-0-262-01153-2 978-0-07-000484-9.

[10] Microsoft. *Back To Basics: Reference Counting Garbage Collection*. URL: https://web.archive.org/web/20210224190617/https://docs.microsoft.com/en-us/archive/blogs/abhinaba/back-to-basics-reference-counting-garbage-collection (visited on 03/29/2022).

[11] Steve Klabnik and Carol Nichols. *The Rust programming language*. San Francisco: No Starch Press, 2019. ISBN: 978-1-71850-044-0.

[12] Graham Hutton and Erik Meijer. “Monadic Parser Combinators”. In: (1996).

[13] John Hughes. “Why Functional Programming Matters”. Functional Conf 2016. Bengaluru, India, Oct. 13, 2016. URL: https://confengine.com/conferences/functional-conf-2016/proposal/2965/why-functional-programming-matters.

[14] James Coglan. *Introduction to parser combinators*. The If Works. URL: https://web.archive.org/web/20201109041458/https://blog.jcoglan.com/2017/07/06/introduction-to-parser-combinators/ (visited on 03/29/2022).

[15] Geoffroy Couprie. *nom*. Version 7.1.1. URL: https://docs.rs/nom/7.1.1/nom/index.html.

[16] Benjamin C. Pierce. *Types and programming languages*. Cambridge, Mass: MIT Press, 2002. ISBN: 978-0-262-16209-8.

[17] Jonathan Aldrich. “Lecture Notes: Big Step Environment Semantics”. In: (2020).

[18] Sérgio Medeiros and Fabio Mascarenhas. “Syntax error recovery in parsing expression grammars”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC 2018: Symposium on Applied Computing. Pau France: ACM, Apr. 9, 2018. ISBN: 978-1-4503-5191-1. DOI: 10.1145/3167132.3167261. URL: https://dl.acm.org/doi/10.1145/3167132.3167261 (visited on 03/19/2022).

APPENDICES

A

Formal grammar rules

```
program ::= ( stmt )+ ;
stmt    ::= var_decl | assignment | while | return | facet_decl | expression_stmt ;
expr    ::= function | block | if | tag ;

// Statements
var_decl        ::= "let" IDENTIFIER ( ":" TYPE )? ( "=" expr)? ";" ;
while           ::= "while" expr block ;
assignment      ::= IDENTIFIER_DOT "=" expr ";" ;
return          ::= return ( expr )? ";" ;
facet_decl      ::= "facet" IDENTIFIER "{"
                        ( IDENTIFIER ":" TYPE "," )*
                        ( IDENTIFIER ":" TYPE )?
                    "}" ;
expression_stmt ::= expr ";" ;

// Expressions
function    ::= "function" ( generics )? "(" ( args )? ")" ( "->" TYPE )? block ;
block       ::= "{" ( stmt )* ( expr )? "}" ;
if          ::= "if" expr block ( "else" block )? ;

// Ladder
tag         ::= logic_or ( "@" logic_or )* ;
logic_or    ::= logic_and ( "or" logic_and )* ;
logic_and   ::= equality ( "and" equality )* ;
equality    ::= comparison ( ( "==" | "!=" ) comparison )* ;
comparison  ::= term ( ( "<=" | ">=" | "<" | ">" ) term )* ;
term        ::= factor ( ( "+" | "-" ) factor )* ;
factor      ::= unary ( ( "*" | "/" ) unary )* ;
unary       ::= "!" expr | call ;
call        ::= ( IDENTIFIER_DOT | grouping ) "(" ( ( expr "," )* expr )? ")"
              | facet ;
facet       ::= IDENTIFIER "{"
                    ( IDENTIFIER ":" expr "," )*
                    ( IDENTIFIER ":" expr )?
                "}"
              | primary ;
```

```

primary     ::= "true" | "false" | "nil" | NUMBER | STRING
              | IDENTIFIER_DOT | grouping ;
grouping    ::= "(" expr ")" ;


// Utility
TYPE           ::= function_type | derived_type ;
IDENTIFIER     ::= ( "a" ... "z" | "_" ) ( "a" ... "z" | "_" | "0" ... "9" )* ;
IDENTIFIER_DOT ::= IDENTIFIER ( "." IDENTIFIER "::" IDENTIFIER )* ;
NUMBER         ::= ( "0" ... "9" )+ ;
STRING         ::= ( "\"" ( ^"\"" )* "\"" )+ ;

// Aux
args          ::= ( IDENTIFIER ":" TYPE "," )* IDENTIFIER ":" TYPE ;
generics      ::= "<" IDENTIFIER ( ":" TYPE )? "," )* IDENTIFIER ":" TYPE ">" ;
function_type ::= "fn" generics "(" ( TYPE )* ")" ( "->" TYPE )? ;
derived_type  ::= IDENTIFIER ( "+" IDENTIFIER )* ;
```

Listing A.1: Formal grammar rules in BNF.

B Introduction to Rust

Rust is best described as a hybrid between a high level and low level programming language. It is functional and has a powerful type system, but you can also work with raw pointers and interact with C code fairly easily. The cherry on top of this mix is the infamous borrow checker. We are going to delve into the exact rules later but, for now, it suffices to give a high level overview: basically Rust has restricted its semantics so that each variable has clear ownership. This lets the compiler figure out exactly when each variable has to be freed at **compile time**, sparing the need for a Garbage Collector and preventing whole classes of bugs such as Use-After-Free.

Another of the major goals of Rust is fearless concurrency which means that, thanks to clearly defined ownership, we can program concurrently and safely. We cannot share a value between threads because it must have one owner only. We are forced to wrap it in a safe container that protects it with a mutex; there is no undefined behaviour [1].

The last major goal is zero cost abstractions. Even though Rust is a high level language with a lot of tools such as generics or collections, these do not come at a runtime cost but rather at a compile time cost. The claim is that the compiler will write the final binary as if we never had those luxuries.

All of these points combined result in a language with a great deal of expressiveness that is as fast as C [2]but safe and concurrent by design.

Throughout the next few sections we will present several aspects of the language that are needed to understand the design and implementation of the interpreter. The syntax can be understood just by looking at the examples.

B.1 References

By far my biggest mistake in the implementation was to think that references were pointers when they are not. For the rest of the explanation refrain from thinking that they are sort of a generalization; we

[1] That is if you only consider safe Rust code. We will not explain it here, but you can drop into unsafe Rust at any moment and the strict rules do not apply anymore. However, doing so is heavily discouraged and only done for performance critical code.

[2] See for example the benchmark for web servers at https://www.techempower.com/benchmarks/ where Rust is among the best.

have to start from a clean slate.

A reference in Rust comes in two flavours:

- **Immutable**. Example: `&variable`. As the names indicates, an immutable borrow means that you do not have ownership but you can *refer* to the variable, for example print it or use it to do calculations. What you cannot do is mutate the variable.
- **Mutable**. Example: `&mut variable`. We also do not have ownership and we can do everything [3]that an immutable reference can. Additionally, we also are able to mutate the inner value.

B.1.1 Lifetimes

In simple Rust code, we can use `&` to create references because their lifetimes are elided. Nevertheless, all of them have a lifetime and, when dealing with slightly more complex scenarios we have to specify it manually.

A lifetime just marks how long we expect the reference to last. For example, if we take a reference from an owned value, we expect it to last less than or equal to the life of the owned value (if not, we have the analogue of a dangling pointer). See a more contrived example in Listing B.1.

```rust
fn<'a, 'b> output_lifetime_s1(s1: &'a str, s2: &'b str) -> &'a str {
    if s1 == s2 {
        s1
    } else {
        ""
    }
}
```

Listing B.1: Example where we specify that the output lifetime is maximum 'a.

B.1.2 Borrow checker

Rules for references:

- Either have several immutable borrows or one mutable borrow.
- The reference must not outlive the value. That is: the lifetime is the reference is less than or equal to the lifetime of the value.

Rules for ownership:

[3]This is not strictly true. For example, we may clone an immutable reference to produce another one, but we cannot clone a mutable reference because that violates the rules of the borrow checker.

- Data only has one owner.

B.1.3 Traits

We could say that traits are like really powerful interfaces. That is, we specify a contract, a set of functions that implementors have to define to have the interface's type. See Listing B.2 for an example.

```
pub trait PartialEq<Rhs: ?Sized = Self> {
    /// This method tests for `self` and `other` values to be equal,
    /// and is used by `==`.
    fn eq(&self, other: &Rhs) -> bool;

    /// This method tests for `!=`.
    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}
```

Listing B.2: Excerpt from the language prelude of the trait used to test for equality.

Their power stems from the fact that we can say things like: "Implement trait A for all types that implement traits B and C" (see Listing B.3).

```
impl<T> A for T
    where T: B+C
{ ... }
```

Listing B.3: Implement trait A for all types that already implement B and C.

Important traits

There are many crucial and specialized traits in the standard library. Here, we will present the three indispensable ones a newcomer is likely going to come across:

- **Copy**: When a type implements this trait it does not change ownership, it is copied for the new owner. The usual mantra is that every object that can implement copy should.
- **Clone**: Used to signal that you can copy the type but it is not a cheap operation; thus, it requires and explicit call.
- **Display**: Trait used to format an object to present it to the user.

Additionally, most of the important traits and can be automatically implemented for a type using a `#derive` macro. For more information, check out [11].

Ownership of captured variables in closures (Rust)

As you might have guessed, capturing variables introduces a change in ownership and, in Rust, that has very strict semantics. By default, the compiler will try to help by capturing the variable in the most lax way possible. For example, if a reference is enough, only that will get captured. Another rule is that if the value can be copied the compiler will copy it. In the cases when neither of the fixes are possible, we have to specify that we want to move ownership of every value to the closure by defining it like: `move || .`

B.2 Generics

Generics works like in any other programming language. The main difference is that lifetimes (subsection B.1.1) and type aliases are mixed together in the generic parameter list. See Listing B.4 for an example.

```rust
pub type PResult<'a, O> = Result<(Information<'a>, O), PError>;

pub trait Parser<'a, O>: Copy {
    fn parse(&self, info: Information<'a>) -> PResult<'a, O>;
}

impl<'a, O, T> Parser<'a, O> for T
    where T: Copy + Fn(Information<'a>) -> PResult<'a, O>
{
    fn parse(&self, info: Information<'a>) -> PResult<'a, O> {
        self(info)
    }
}
```

Listing B.4: Example where lifetime parameters are used alongside generics.

Both when declaring, implementing and naming types, both lifetimes and generic parameters have to be provided. However, the Rust compiler is intelligent enough to guess them in most of the situations.

The last important point is that generics are included in the so called “zero-cost abstractions”. It means that, even though, they are a higher level construct which usually translates into a performance penalty, Rust’s generics work at the same speed as if they were not used.

C

Advanced topics on parser implementation

This section will cover advanced topics on the parser implementation. The discussions will dive deep into some niche topics which are not essential but, nevertheless, provide interesting insights. We will discuss the two main conceptual problems we encountered in the implementation. The first section, section C.1, answers the question of why parser combinators use generic parameters, and simultaneously unveils the `Parser` trait. In the second section, section C.2, we will explain how the syntactic rules were chosen alongside several variations and the troubles they entail.

C.1 The Parser trait

In order to understand the parser trait, first we have to digress to really understand what closures really are.

C.1.1 What is really a closure?

We explained what a closure was in section 1.4 and even touched on some points such as ownership of the captured values. In this chapter we are interested on how to represent a closure in memory and how that affects its type.

First we need to know how function calls are represented in memory. In an overly simplistic approach we could say that memory is divided into the heap and the stack. Each time you call a function a stack frame is created and “pushed” onto the stack and, once you return, the memory is freed and the previous frame is used. Technically, frames are logical units and run continuously on the stack, have a look at Figure C.1 for a close up of one. The complete stack is composed of several frames on top of each other.

Let’s take an example program (Listing C.1) with nested functions and imagine that the execution has reached line 8 and we are about to return the closure. Then, the stack would look like Figure C.2.

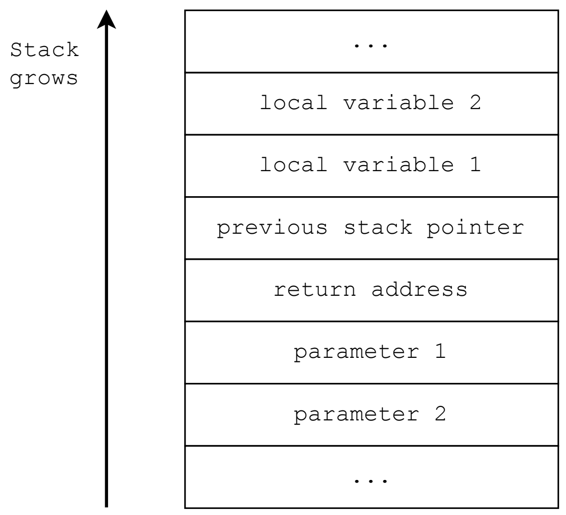

Figure C.1: Close up of the stack frame of a function (x86 convention).

```
let a = function() {
    let b = function() {
        let local_var_1 = 41;
        let local_var_2 = 1;
        let closure = function() {
            print(local_var_1 + local_var_2);
        };
        closure
    };
};
```

Listing C.1: Example program with nested closures.

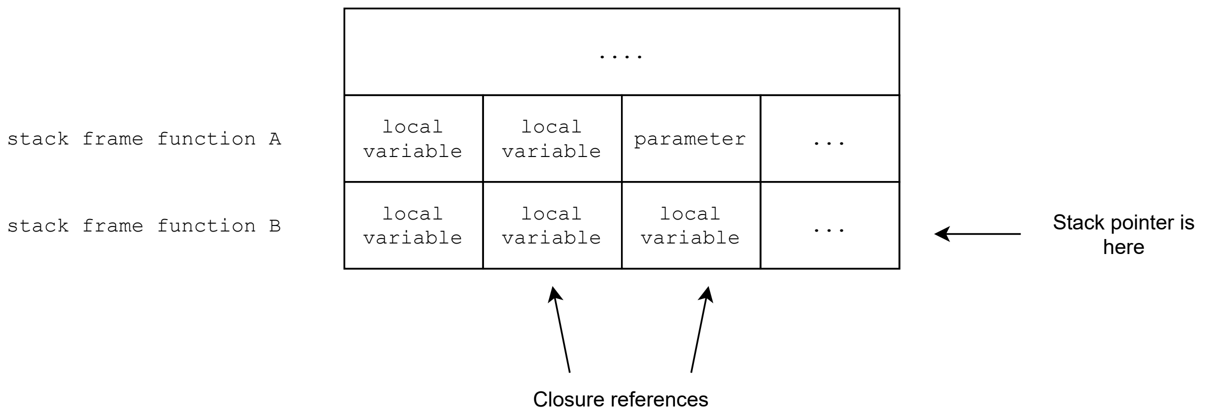

Figure C.2: The stack as of line 8 in Listing C.1.

Take note that the closures references two variables in the **current** stack frame of function `b`. We are going to return the closure and it might be called anywhere on the program, so those values have to persist. However, as we explained, when we return, we pop the trace from the stack, see Figure C.3.

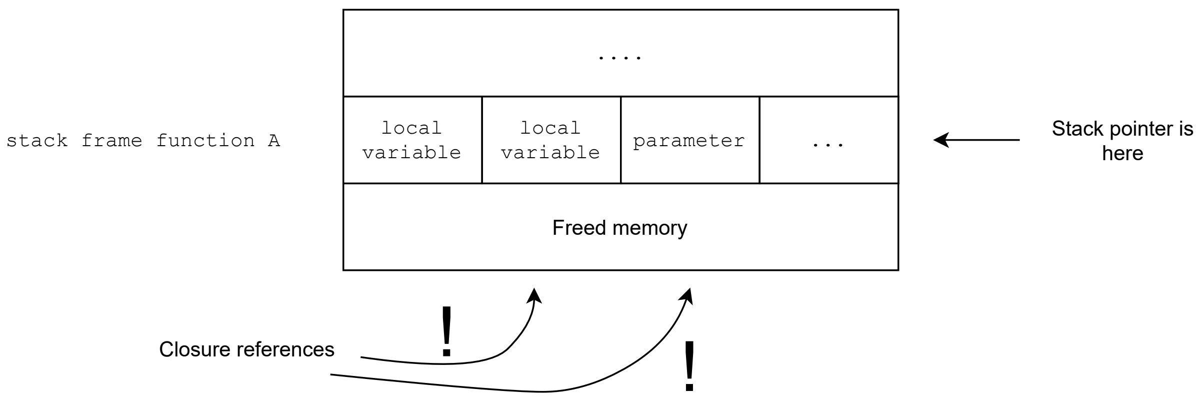

Figure C.3: The stack after returning from function b in Listing C.1.

The solution and the insight is that a closure is **not** a function in memory because it stores the values internally alongside the pointer to its code; it is a more complex data structure.

C.1.2 Passing arguments to functions

We are ready for the last piece of the puzzle. In the last section we saw what happens when you call a function. We missed one piece and that is that the stack traces' offsets are the same on every invocation; for example, `local_var_1` is always at position 8. Think that the function is compiled once so the offsets cannot change from invocation to invocation.

The implications of this are huge. For us, the most important thing is that, in Rust, all the arguments must have a concrete size at compile time. Contrast that with Java, where if you ask for a class A, you can get any subclass with potentially more fields and different size. The reason is that Java uses a GC (section 1.4), so every value [1]lives on the heap and we are just passing pointers around, which are always the same size. In Rust, if you ask for a type, you can only get that one type because we are passing it directly through the stack, so it has to have the same size because the offsets have to match.

It is utterly fascinating that when creating a language, the chosen technique for memory management has ramifications on the apparently unrelated type system.

C.1.3 The Fn trait vs the fn type

In Rust, functions have a type and closures do not. Closures implement a Trait called `Fn` [2]. For example, `Fn(usize, String) -> String` is a Trait while `fn(usize, String) -> String` is a type. Why is that so? It is because closures have different size depending on the environment they capture while

[1] That is not exactly right. For example, there are primitive types such as int that are not boxed, they are used directly.

functions occupy the same always, mystery unveiled.

C.1.4 The Copy trait

Lastly, we have to explain one more thing regarding `Copy` trait (section B.1.3). There is a massive limitation in using closures as parsers in Rust and that is that we cannot do things like:

```rust
pub fn my_extrange_combinator<'a, O, F>(parser: F) -> impl Parser<'a, O>
    where F: Parser<'a, O>,
          O: PartialEq
{
    and(parser, parser)
}
```

Listing C.2: Attempting to call a combinator with the same parser twice.

Because closures might not be `Copy` [3]which means that they cannot be cloned transparently, see section B.1.3 for more details.

C.1.5 The Parser trait

We are now ready to understand the Parser trait:

```rust
pub type PResult<'a, O> = Result<(Information<'a>, O), PError>;

pub trait Parser<'a, O>: Copy {
    fn parse(&self, info: Information<'a>) -> PResult<'a, O>;
}

impl<'a, O, T> Parser<'a, O> for T
    where T: Copy + Fn(Information<'a>) -> PResult<'a, O>
{
    fn parse(&self, info: Information<'a>) -> PResult<'a, O> {
        self(info)
    }
}
```

Listing C.3: Parser trait and implementation for closures/functions.

[2]This is an oversimplification. To be precise, there are three traits: Fn, FnOnce and FnMut. For more information, check [11] (chapter on Advanced Closures).

[3]Incidentally, this was a big limitation for me using the nom library. It is an act of balancing because if you make Parser implement Copy you have to restrict yourself to only a subset of closures. For my use case, it was clearly advantageous, but nom chooses to support the bigger set at this expense.

We are saying that we want `Parser` to always be `Copy`. Then, we implement it for all the closures that are themselves `Copy`.

Why do we even have a trait? Because a trait brings us the flexibility of changing it in one place without having to change the signature of every function. And, even if we did not have a trait, we would have to still use generic parameters but with `Fn` instead.

C.2 Grammar rules variations and effect on performance

The first thing we have to take into account when designing grammar rules is that parser combinators are not left recursive. In layman terms, it means that a rule cannot start with the rule itself. If we think about function recursion, it makes sense because the parser, which is a function, would call itself indefinitely crashing the program. The downside is that left recursion is often the most natural way of codifying rules, and we have to circumvent this limitation. For example, if we were to code the rules for binary expressions, we would produce something like this:

```
// Ladder
tag          ::= tag "@" tag | logic_or ;
logic_or     ::= logic_or "or" logic_or | logic_and ;
logic_and    ::= logic_and "and" logic_and | equality ;
equality     ::= equality ( "==" | "!=" ) equality | comparison ;
comparison   ::= comparison ( "<=" | ">=" | "<" | ">" ) comparison | term ;
term         ::= term ( "+" | "-" ) term | factor ;
factor       ::= factor ( "*" | "/" ) factor | unary ;
```

Listing C.4: Rules for binary expressions, left recursive.

Observe that we are codifying the precedence of the different operators in the rules themselves. We are building a “ladder” in the sense that if a rule fails, it tries to match the next one going down. That means that the highest precedence is the lowest rule.

If we were to translate Listing C.4 into a non left recursive form, we would produce something like this:

```
// Ladder
tag          ::= logic_or ( "@" logic_or )+ | logic_or ;
logic_or     ::= logic_and ( "or" logic_and )+ | logic_and ;
logic_and    ::= equality ( "and" equality )+ | equality ;
equality     ::= comparison ( ( "==" | "!=" ) comparison )+ | comparison ;
comparison   ::= term ( ( "<=" | ">=" | "<" | ">" ) term )+ | term ;
term         ::= factor ( ( "+" | "-" ) factor )+ | factor ;
factor       ::= unary ( ( "*" | "/" ) unary )+ | unary ;
```

Listing C.5: Rules for binary expressions, non left recursive.

Even though this set of rules works, during the development it was the ultimate culprit of long execution times. To put that into perspective, it took about 3 minutes to run the test suite while it currently takes only a few milliseconds. Even though it is an obvious mistake, it took a lot of debugging to locate it. The purpose of the rest of this section, as the title indicates, is to explain that although two set of rules can produce the same grammar, they can have vastly different performance.

The problem with Listing C.5 is better seen through an example. If we tried to parse a very simple expression, `2 == 2`, we would get the stack trace in Figure C.4, where the number of stack frames, or function calls, growths exponentially.

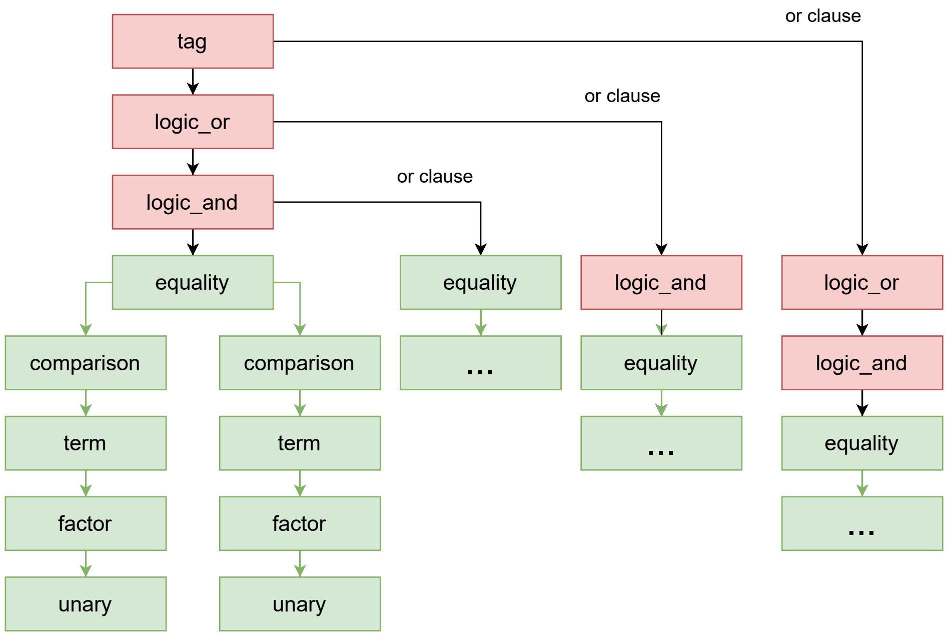

Figure C.4: Simplified stack frame for parsing `2 == 2` using Listing C.5. Green: parser returned successfuly in the main clause. Red: parser failed the main clause, tries the alternative.

Fortunately, once the culprit was found, a new set of rules was easily produced:

```
// Ladder
tag          ::= logic_or ( "@" logic_or )* ;
logic_or     ::= logic_and ( "or" logic_and )* ;
logic_and    ::= equality ( "and" equality )* ;
equality     ::= comparison ( ( "==" | "!=" ) comparison )* ;
comparison   ::= term ( ( "<=" | ">=" | "<" | ">" ) term )* ;
term         ::= factor ( ( "+" | "-" ) factor )* ;
factor       ::= unary ( ( "*" | "/" ) unary )* ;
```

Listing C.6: Final set of rules for binary expressions.

In Listing C.6, if we match a rule down the ladder but not the current one, we simply return the result instead of nothing. This happens because the glob (*) matches 0 or more occurrences which means that a single operand is a valid result. In contrast, before, we used the plus (+) operator that required at least one match. When it failed, we had to start all over in the topmost level by going into the other side of the rule.

D CLI USAGE

The prerequisites are having rust nightly and cargo installed. The interpreter is built like any cargo project:

```
cargo build --release
```

From here on, either continue to use cargo to run the interpreter, or locate the executable. Running it with the `--help` flag will return the usage (listed below).

```
# Both are equivalent
cargo run --release -- --help # Notice the extra -- that delimits the flags
target/release/dyntypes --help
```

```
dyntypes

USAGE:
    dyntypes [FLAGS] <INPUT>

FLAGS:
        --graph      Outputs the AST to ast.dot
    -h, --help       Prints help information
    -V, --version    Prints version information

ARGS:
    <INPUT>    Sets the input file to use
```

Per the usage, in order to run code you only have to pass the filename,

```
# Both are equivalent
cargo run --release -- ./program.in
target/release/dyntypes ./program.in
```

```
    >>> <Program output>
```

Additionally, there is one available flag: `--graph`. Its purpose is to aid debugging by checking if the parsing was correct. Its does so by graphing the program's AST. The usage is as follows:

```
# Both are equivalent
cargo run --release -- --graph ./program.in
target/release/dyntypes --graph ./program.in
```

A file named `ast.dot` is created on the current directory. A number of DOT programs can be used to render the AST; we show the simplest method for creating a png image:

```
dot -Tpng ast.dot -o ast.png
```

Lastly, you can find the test framework in `test.py`. The test cases and expected results can be found under the directory `tests`. To run them, you must also have python installed in your system to invoke the script:

```
python test.py
```

A typical output may look something like this:

```
Compiling the program...
100%|██████████████████████████████| 35/35 [00:00<00:00, 1312.87it/s]
--------------------------------------------------
Error 101: Probably panic.
Error 121: Probably parse error.
--------------------------------------------------
Case tests/generics
Return code 1, expected 0
>>>>>>>>>>>
b'working\n'
<<<<<<<<<<<
b''
-----------
Failed 1 of 35.
```

where we can see that a test failed because the neither the output nor the return code matched the expected. For documentation on how to define new tests or how the framework works, consult the `test.py` script directly.