Laia Ortiga Coca

Jelly: Design of a new Programming Language and Compiler Implementation

Degree in Computer Engineering

Final Degree Project

Directed by
Dr. Sergio Gomez

UNIVERSITAT ROVIRA i VIRGILI

Tarragona, 2024

Contents

List of code snippets

List of Figures

List of Tables

Resum

Aquest projecte descriu Jelly, un nou llenguatge de programació, dissenyat com una millora del llenguatge C i amb alguns aspectes inspirats en el llenguatge Rust. És un llenguatge imperatiu, de propòsit general, de tipat estàtic i compilat. La filosofia de Jelly és simplicitat, velocitat, llegibilitat i coherència. Algunes de les seves característiques clau són els tipus afins i els tipus etiquetats. Els tipus afins prevenen una classe sencera d'errors relacionats amb l'ús de recursos computacionals. El recurs utilitzat més comunament és la memòria. El control manual de memòria és una font freqüent d'errors i, a conseqüència, és un problema difícil de tractar. Els tipus afins poden evitar alguns d'aquests errors limitant el nombre d'usos d'un objecte a un com a màxim. Si s'utilitza diverses vegades, es mostrarà un error de compilació, cosa que prevé vulnerabilitats com use-after-free i double-free. Els tipus etiquetats son tipus estructurals basats en entitats que es poden etiquetar amb qualsevol nombre d'etiquetes. Aquestes etiquetes poden ser qualsevol cosa i es poden fer servir per millorar la seguretat de tipus del llenguatge. S'implementa un compilador per a provar el llenguatge Jelly amb alguns projectes. El compilador utilitza un disseny basat en dades que el fa fàcil de paral·lelitzar.

Resumen

Este proyecto describe Jelly, un nuevo lenguaje de programación, diseñado como una mejora del lenguaje C y con algunos aspectos inspirados en el lenguaje Rust. Es un lenguaje imperativo, de propósito general, de tipado estático y compilado. La filosofía de Jelly es simplicidad, velocidad, legibilidad y coherencia. Algunas de sus características clave son los tipos afines y los tipos etiquetados. Los tipos afines previenen una clase entera de errores relacionados con el uso de recursos computacionales. El recurso usado más comúnmente es la memoria. El manejo manual de memoria es una fuente frecuente de errores y, a consecuencia, es un problema difícil de tratar. Los tipos afines pueden evitar algunos de estos problemas limitando el número de usos de un objeto a uno como máximo. Si se utilitza varias veces, se mostrará un error de compilación, cosa que previene vulnerabilidades como use-after-free y double-free. Los tipos etiquetados son tipos estructurales basados en entidades que se pueden etiquetar con cualquier número de etiquetas. Estas etiquetas pueden ser cualquier cosa y se pueden usar para mejorar la seguredad de tipos del lenguaje. Como estos tipos son estructurales y no nominales, pueden generar una infinidad de tipos nuevos. Se implementa un compilador para probar el lenguaje Jelly con algunos proyectos. El compilador utiliza un diseño basado en datos que lo hace fácil de paralelizar.

Abstract

This project describes Jelly, a new programming language, designed as an improvement upon the C language and with some constructs inspired by the Rust language. It is an imperative, general-purpose, statically-typed and compiled language. Jelly's philosophy is simplicity, speed, readability and consistency. A few of its key features are affine types and tagged types. Affine types prevent a whole class of bugs involving resource-like objects. The most commonly used resource is memory. Manual memory management is a frequent source of bugs, therefor it is a difficult problem to deal with. Affine types can avoid some of these problems by only allowing an object to be used at most once. Using it multiple times causes a compile time error which prevents use-after-free and double-free bugs. Tagged types are structural types based on entities that can be tagged with any number of tags. These tags could be anything, and they can be used to enhance type safety. Since these types are structural and not nominal, they can create an infinite amount of types. A compiler is implemented to showcase some projects written in Jelly. The compiler uses a data-oriented design that makes it easy to parallelize.

1 Introduction

Programming languages have become a very important part of software development. Their use allows software to be developed at a much faster pace and with fewer errors. But, although they are used extensively, most software is still developed in a reduced number of languages.

PYPL ranking September 2023	Stack Overflow's Developer Survey 2023
Python (1991)	JavaScript (1995)
Java (1995)	HTML/CSS (1993)
JavaScript (1995)	Python (1991)
C# (2000)	SQL (1974)
C/C++ (1972)	TypeScript (2012)
PHP (1995)	Bash/Shell (1989)
R (1993)	Java (1995)
TypeScript (2012)	C# (2000)
Swift (2014)	C++ (1985)
Objective-C (1984)	C (1972)

Table 1: Rankings of all programming languages in 2023 (in parentheses the year they first appeared)

[16] From all of these, Swift is the most recent, and it first appeared in 2014. The second most recent is Typescript which first appeared in 2012. The rest appeared more than twenty years ago. C is still used today, and it first appeared in 1972. From this list it can be noted that only C, C++ and Swift are system programming languages. System programming languages are those that are designed to write system software. Some examples of such software would be operating systems, game engines and industrial automation.

1.1 Objectives

My objective with this project is to learn how programming languages work internally, improve my programming experience and create a language that would satisfy my needs. Another objective is to make a compiler for this language that is quite fast compared to the competition like the C++ compilers. I also want to make a little project written in my new programming language that can show me if this could be usable for future projects of mine. Eventually it would also be nice if this language can get adopted by other people, and they can provide me feedback on things to improve while keeping the vision I have for it.

1.2 Motivation

There are multiple reasons that motivated the creation of this project. One of the reasons behind this project is programming a game engine from scratch. Game engines need to be optimized heavily because they must produce frames at a certain rhythm. Nowadays, it is very common to have games that run at 60 frames per second (FPS) or much more. This leaves very little CPU time to do the work required. Therefore, full control over code execution is really important to make this guarantee. Garbage collection is one of the many reasons why game engines are still programmed in C or C++. It is not deterministic because it usually has to stop all threads from moving forward and that can cause several frames to be skipped.

Many of the design flaws of C and C++ inspired the creation of Jelly. The objective of this language is to simplify memory management while not requiring garbage collection. It strives to be a simple language that can interact with already existing C code.

The entity component system (ECS) [11] architecture is a pattern that comprises entities that can have multiple data components with systems that operate on these components. This model can simplify data dependencies and improve CPU cache usage. One famous example of a game engine that includes an ECS is Unity [17]. I want Jelly to be a programming language that is easy to use with an ECS.

Figure 1: The Unity logo

2 State of the art

C++ is a programming language that is often used for game development since it has a wide variety of libraries for that purpose. C++ started as an evolution from C. This means it carries all of its baggage, and it tries to cover it by adding a lot of new features. Its promise for backwards compatibility means that most of these older problems will remain in the language forever. That has made the language very complex, and it has added a lot of corner cases. This adds a lot of cognitive overhead, and it can lead to cryptic bugs that can take very long to solve.

C is a very simple language in comparison to C++. This helps to make faster decisions and to overthink much less. But C is a very old language and some of its design decisions would be different had it been invented today. Commonly criticized design decisions include:

- Integer promotion
- Integer sizes depending on the target platform
- Undefined behavior

- No memory safety by default
- Preprocessor
- No type generics

Some of these can be improved with compiler warnings. Others are impossible to change without changing the language itself. Currently, there are newer alternatives like Rust and Zig.

2.1 Rust

Rust is a programming language that first appeared in 2015. It was created by Mozilla Research employee Graydon Hoare. One of its unique features is memory safety without garbage collection. It achieves this by setting rules that force the compiler to reject code that would not be memory safe. Because memory safety is an undecidable problem, this forces the compiler to reject some code that would be memory safe. These rules can make the learning curve very steep. It changes how people approach programming completely. But the restrictions can still feel overwhelming. The advantage in learning Rust is that this new mindset can help a lot in other languages like C.

Rust has been used in software across different domains. It was developed with the intention to be used for a parallel browser engine called Servo. The Linux kernel allows code written in Rust as of version 6.1 and Microsoft is also rewriting parts of the Windows operating system in Rust.

Game development in Rust can be tricky. Game engines may require heavy optimizations to run simulations with millions of entities with complex behaviors. These optimizations may require unsafe Rust code. These are sections of Rust code that allow programmers to perform tasks that would otherwise give a compiler error. If those are used often, most of the benefits of Rust are lost and the code becomes more verbose. At that point, using a language that doesn't require unsafe code sections in a lot of places might be more suitable.

Figure 2: The Rust logo

2.2 Zig

Zig is a programming language that appeared in 2016. It hasn't been released yet, but it makes a lot of promises. It is meant to be the successor to C and, as such, it is very simple, and it adds a lot of functionality.

A compiler for Zig requires a complete interpreter of the language because Zig allows for compile-time execution of regular Zig code. This is a very powerful tool, and it is mostly useful in library code. Its generic type system is also quite basic because it is implemented as compile-time duck typing. Duck typing is a type system with objects that satisfy the rule "If it walks like a duck, and it quacks like a duck, then it must be a duck".

Zig doesn't have a type system that guarantees memory safety. Instead, it tries to use different memory allocators to ease the use of manual memory management. Memory allocators are objects that can obtain a contiguous chunk of memory of a specific size, and they can release these chunks later. Depending on the use, different strategies can be used. One example is a region-based allocator that can easily return objects of different sizes. Then, all objects in the same region can be released in one function call. These allocators can help to reduce memory errors, but mistakes can still happen.

Since Zig hasn't been released yet as of 2024, there aren't many projects using it. One project is bun, which is a JavaScript and TypeScript runtime.

Figure 3: The Zig logo

3 Methodologies

A compiler is a computer program that translates computer code written in one programming language into another language. Usually the target language is a low-level language like assembly. Compilers are usually divided into stages which favors separation of concerns. This makes their design more modular and easier to test for correctness. Separation of concerns is a common design pattern for separating a program in different sections that don't depend much on each other.

3.1 Compiler stages

The stages of a compiler can be divided in different phases. The front end analyzes the source code and transforms it into an internal intermediate representation (IR). It comprises lexical analysis, syntax analysis and semantic analysis. The middle end optimizes the intermediate representation independently of the target platform. The back end generates optimized target code.

3.1.1 Lexical analysis

Lexical analysis is the first stage in many compilers. It converts the source code into meaningful tokens. A token is a string with an assigned type. These token types are defined by the programming language, and they are usually described by regular expressions. Usually, tokens can be identified using regular expressions which can be implemented using finite-state machines. The following example shows how the code is converted into tokens:

```
if x < 8 {
    print_str("x is small\n")
}
```

The code is converted into the following tokens:

(keyword, `if`), (identifier, `x`), (operator, `<`), (integer, `8`), (punctuation, `{`), (identifier, `print_str`), (punctuation, `(`), (string, `"x is small\n"`), (punctuation, `)`), (punctuation, `}`)

Finally, the tokens obtained from this analysis are fed to the parser which is part of the next stage.

3.1.2 Syntax analysis

Syntax analysis or parsing receives the tokens obtained from lexical analysis, and it generally produces a syntax tree. The leaves of a syntax tree are the tokens and the other nodes represent the structure of the syntax. These can be operators, statements, functions, etc.

A syntax tree can be abstract which means that it doesn't exactly represent the syntactic structure of the source code. A difference between an abstract syntax tree (AST) and a regular syntax tree might be that grouping parenthesis are left out from an AST because they are already represented by the tree itself.

An AST has some benefits over working directly with source code. One benefit is that source code can have a lot of punctuation or delimiters that are meaningless and implicit in an AST. Another benefit is that an AST can be easily modified and extended as compilation collects more information about a program.

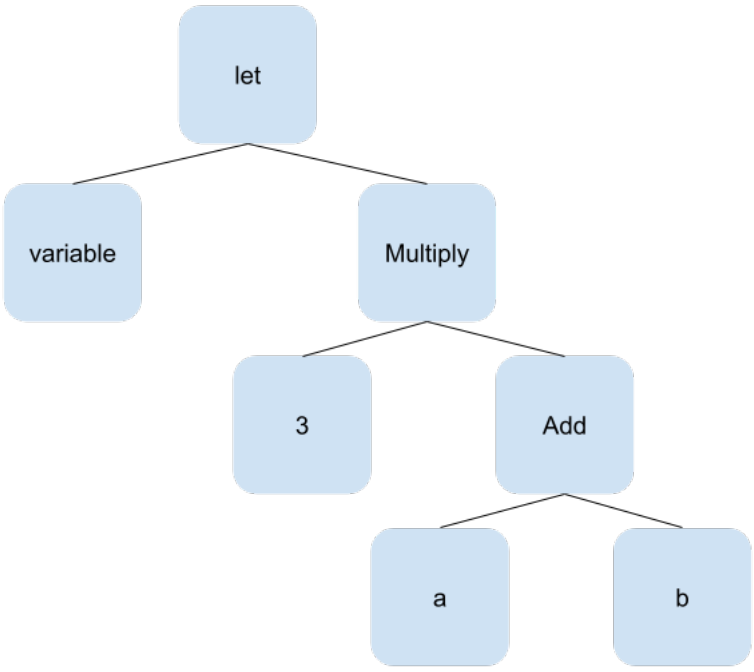


Figure 4: Syntax tree for the following statement: let variable = 3 * (a + b)

3.1.3 Semantic analysis

Once the syntax tree is formed, it can be traversed to perform semantic analysis. This stage finds all semantic errors which can vary from one language to another. Some examples of semantic errors include the use of undeclared identifiers, missing variable assignments, type checking and data-flow analysis [18]. The syntax tree is often traversed depth-first because type checking information is propagated bottom-up. Most warnings and errors from a compiler come from this phase. Therefore, it is important to issue diagnostics that are clear and meaningful.

```
51 |     int a = 3.4;
52 |

PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

/mnt/Files/Documents/Code/compiler/src/main.c: In function 'main':
/mnt/Files/Documents/Code/compiler/src/main.c:51:9: warning: unused variable 'a' [-Wunused-variable]
   51 |     int a = 3.4;
      |         ^
```

Figure 5: GCC (C compiler) warnings during semantic analysis

3.1.4 Code generation

Code generation is the last stage, and it will convert the previous intermediate representation to the target code. This can be machine code, assembly, byte code or a different programming language. LLVM [10] is a programming language that is

designed to be used as a backend for many other languages like C, C++, Rust, Swift and Zig. Its compiler has many optimization passes and generates very efficient code, which is why the compiler implemented in this project uses it as one of the backends for Jelly.

Figure 6: The LLVM logo

4 Design of the Jelly Programming Language

4.1 Source Code

To understand Jelly's design, first we need to understand how code is organized. Jelly code is written in UTF-8 [4] files that have the file extension ".jel" by convention. Then, these files are grouped into modules.

4.2 Module

In Jelly, modules are a set of source code files that share similar concepts and functionality. When I started designing Jelly, I thought each file would be a self-contained unit, but I never really liked this concept. Splitting similar functionality into multiple files can be helpful in keeping files smaller, but then files start with a very long list of import statements like it happens in Java. So I decided to separate the concept of module from the concept of source file. Each file indicates which module it belongs to and multiple files are allowed to be part of the same module.

4.3 Identifier

Identifiers are useful to give names to commonly used entities found in code. Having them allows code to be more readable, and they let programmers be more expressive. In Jelly, identifiers can be defined in different sections of a source file and their meaning might differ depending on how they are defined. These sections are called scopes.

4.4 Scope

The scope of an identifier is the part of the source code that can refer to that identifier. The scope is always determined at the point of definition. There are four types of scopes in Jelly: global scope, module scope, file scope and local scope.

When using names, the concept of variable shadowing arises, which is the definition of a name that is already present in the current scope. I decided that Jelly shouldn't allow variable shadowing because it can confuse the meaning of a name.

4.4.1 Global Scope

The global scope is an implicit scope that covers the entire source code. It can't be modified, and it contains the following names that are defined automatically by the compiler:

- Built-in types (`i8`, `i16`, `i32`, `i64`, `isize`, `f32`, `f64`, `bool`, `char`, `byte`, `'Size`, `'Alignment`)
- Type constructors (`'Affine`, `'ArrayLength`)
- Value constructors (`'Slice`)
- Functions (`'zero_extend`)

4.4.2 Module Scope

The scope of a module contains the names defined in said module which can be classified in two groups: public or private. By default, a module name is private, but it can be marked public by adding the keyword `public` at the beginning of its definition. A public name can be accessed in a different module by using the syntax `<module-name>.<name>`. Otherwise, a module name can be accessed anywhere in any of the files that are part of the module that contains the definition.

4.4.3 File Scope

The scope of a file contains the names that can be used in that file. The only names that can be part of this scope are module imports.

4.4.4 Local Scope

Each module definition contains an inner local scope that only covers the inside of its definition. Usually they refer to the scopes in a function body, but it could also refer to the scope of a struct definition or other entities. For functions, a new smaller local scope is created when a new name or code block is introduced. These blocks are syntactic structures surrounded by braces that contain any number of statements. The fact that a new scope is created for every new name means that they can't be used before they are defined unlike module names which can be referenced in any order as long as there's no cyclical references.

4.5 Type System

A type is a set of possible values and operations that can be performed on these values. Type systems can be classified in many ways. Jelly is a statically-typed language which means that all values have a known type at compile time. This prevents an entire class of errors from happening at runtime. For instance, a string can't be accidentally passed to a function that expects an integer.

One of my main criticisms of C is that it is weakly typed. A weakly-typed type system is one that allows using values of one type as a different type through implicit type conversions. C allows the programmer to do something like this:

```
int x = "hello world";
```

This code has two implicit conversions. The first one is that the string is decaying into a pointer. The second one is that the pointer is being converted into an integer. In a bigger project, the result could be a disaster because C doesn't add any runtime checks to prevent memory corruption. Jelly is strongly-typed and code like this would be a compile time error.

4.5.1 Safety

Jelly is not a memory safe language. Memory can be manipulated in any way as long as the type system doesn't complain. Regardless, Jelly has enough abstractions to allow programmers to create safe interfaces that are impossible to be used in a way that can corrupt memory if pointers are not used. That is all thanks to its substructural

type system and tagged types.

4.5.2 Parametric Polymorphism

Parametric polymorphism allows code to be written ignoring which types of data will be used with that code. Then, the missing types can be specified later when that code needs to be used with specific types. This allows code to be written more generally, and it prevents duplication. Often, this concept is also called generic programming or just generics.

Different parts of code can be polymorphic in this way. The general idea is that these parts take some type parameters which are a similar concept to function parameters. Then, the programmer can pass types as if they were regular function arguments. Sometimes languages provide a concept called type inference that allows these type arguments to provided implicitly if the compiler can infer them from context.

In Jelly, I decided to make a very simple and basic implementation of parametric polymorphism. I wanted something that felt like an extension from the C programming language that has no such concept. I decided to only allow types that would have the same memory layout to be polymorphic. The reason behind this choice is to prevent a process called monomorphization. Monomorphization is the process in a compiler that duplicates polymorphic code and specializes it into monomorphic instances. This can increase compiler complexity, and it can result into code bloat [15] and slow compilation if it is not implemented properly. If the operations that can be performed on polymorphic types are limited and their sizes are fixed, the need for monomorphization vanishes. This is what I decided to go with. This limits the use of polymorphic types, but it is still an advantage over a language like C that does not have them.

For now, type parameters can only be added on functions and certain types, but I plan on adding generic structs.

4.5.3 Properties

In Jelly, most types have two basic properties: alignment and size. These properties are needed to allocate space for them in the program's stack and they remain the same for the whole duration of the program's execution.

The alignment of a type is the number of bytes between successive addresses at which objects of this type can be allocated.

The size of a type is the number of bytes a type needs to be able to be store all of its possible values. The value might be bigger than the minimum required to satisfy the alignment requirements.

These properties can be queried in code with the built-in functions `‘align_of` and `‘size_of`. Some types do not have these properties because they can't be known at compile time such as generic type parameters. This means that these types can't be allocated on the stack or used normally like all other types.

4.5.4 Primitive Type

A primitive type is one that doesn't depend on any other types. Jelly has the following primitive types:

- Fixed-size signed integers (`i8`, `i16`, `i32`, `i64`)
- `isize`: pointer-sized signed integer
- Floating-point numbers defined by IEEE 754 [19] (`f32`, `f64`)
- `bool`: a byte that can represent the values false and true
- `char`: a single byte integer
- `byte`: a generic byte without any other meaning

I was inspired by Java to avoid unsigned integers. My reasoning is that, even though they are common in system programming languages, they introduce complexity. This avoids common unsigned bugs such as doing a backwards for loop from a specific integer down to zero.

```
for (unsigned int i = 7; i >= 0; i--) {
    ...
}
```

Figure 7: Infinite loop bug in C code

Other bugs involve the conversion from a signed integer to an unsigned integer.

```
unsigned int i = -3;
... = malloc(i);
```

Figure 8: Unsigned integer overflow bug in C code

Another decision I made is that `isize` is not an alias to one of the other integer types, but its own type. This is so that the same program can compile in different platforms without changing the meaning of code. Since there are no implicit conversions involving this type, an explicit type cast is necessary.

I added the type `byte` to improve interoperability with void pointers from C.

4.5.5 Derived Type

A derived type is one that is based on other types. Jelly has different derived types, some of them found commonly in other programming languages:

- Array - `[:N]T` or `[I -> T]`
- Array length - `'ArrayLength[N]`
- Function - `function (Args...) -> R`
- Pointer - `*T` `*mut T`

- Slice - `@T @mut T`
- Tagged type - `T[Args...]`
- Affine type - `'Affine[T]`

4.5.5.1 Array and Array Length

An array type is a contiguously allocated sequence of values of a particular element type. The number of values is static, and it is part of the type.

Initially arrays had the syntax `[:N]T` where N is an integer constant and T is the element type. Later, while thinking about adding generic programming, I added a new type `'ArrayLength`. Since it is a type and not a value, it can be inferred when calling generic functions without having to add value generics. This forced me to add a new array type syntax `[I -> T]` that can use an index type. The nice property about this new syntax is that I can add arrays indexed by enums or other types later in the future, which is a feature that other languages, like Ada [1], have.

4.5.5.2 Function

A function type is a pointer to a segment of code that can be executed with a certain amount of parameters and an optional return value. It is equivalent to a function pointer in C. Later, when I added generics, I enhanced the type to also contain the number of type parameters.

4.5.5.3 Pointer

A pointer type is a value that can refer to another value in memory. In Jelly there are two types of pointers: constant and mutable pointers. Constant pointers are the default, and they can't be used to modify the value they point to. On the other hand, mutable pointers have both read and write access. A mutable pointer can be used anywhere a constant pointer is expected, but not the other way around.

Initially, pointers had the same properties as in C. They could be null, and they could be indexed like arrays. This was just to quickly test the language, but it was not my intention. Later, I removed the ability to index pointers because this can lead to many memory errors. They still retain the ability to point to invalid addresses. One such example would be the null pointer. I also want to remove that in the future, but I haven't had time. My plan would be to add some way to indicate that a pointer can be null. One way other languages, like Rust, achieve this is through sum types.

4.5.5.4 Slice

A slice type is a special pointer that points to a contiguous range of elements. The length of this range is part of the value, thus it can be used at runtime. Similarly to regular pointers there are constant and mutable slices.

4.5.5.5 Tagged Type

Tagged types are types that have the same memory layout as their inner type, but they are tagged with a list of types. This allows creating new types that are not compatible with each other as long as the list of types is different. This property can be useful to give programmers the ability to add safe generic programming in Jelly if they choose to.

The idea behind these types stemmed from the frustration that C can't represent a generic type like a dynamically growing array, even though it is capable of representing pointers of different types. I didn't want something like C++ templates that expand the code for every unique combination of types because I knew that most data structures are stored in the heap and can be represented with one single memory layout. For instance:

```
struct list {
    void *data;
    size_t length;
    size_t capacity;
};
```

It doesn't matter what the list contains. It will always have the same memory layout. If we want to push an element to the list, we can just pass the size of the element as an argument. The problem with this is that C doesn't have a way to prevent the programmer from passing the wrong size or casting the data pointer to the wrong pointer type. So in Jelly, I added tagged types to be able to share the same exact memory layout as a base type without losing any type information.

4.5.5.6 Substructural Type

Substructural types [20] are types that have added constraints on how they can be used. There are different type systems that can be created depending on which constraints are added: Adding restrictions on types can allow the programmer to model certain

	Exchange	Weakening	Contraction	Use
Ordered	-	-	-	Exactly once in order
Linear	Allowed	-	-	Exactly once
Affine	Allowed	Allowed	-	At most once
Relevant	Allowed	-	Allowed	At least once
Normal	Allowed	Allowed	Allowed	Arbitrarily

Table 2: Substructural type systems

behaviors and to prevent errors at compile time. Rust is a famous example of this. For Jelly, I decided to add affine types. At the beginning, I thought about adding linear types, but when I thought about generic programming, I came to the conclusion that affine types are less cumbersome to use.

Linear types can be used to guarantee that a type will always end up being consumed in a certain function call. For example, one could make a File linear type and define

a function that consumes the file and closes it. That would guarantee at compile time that the programmer doesn't forget to close a resource. But it doesn't work as well with containers and memory. An example that is not ideal is how a list of lists would have to be freed. The creator of the list library would need to take into account linear types and add a function that can consume the list and all of its elements. And the user would have to pass a function to free the elements. If we take generics into account, it is worse because now the library developer has to always assume that types are linear which can be a big restriction. I could have added some way of specifying when a type parameter accepts linear types or when it doesn't, but that seemed overcomplicated. So I decided to go with affine types only.

Affine types can't guarantee that they are consumed, and they don't prevent as many types of errors as linear types, but they are enough to prevent bugs like use after free or double frees [13] which is already much better than what C can do. Affine types allow memory leaks, but I don't consider those as dangerous because they don't corrupt memory.

4.5.6 User-defined Type

Jelly lets programmers define new types with the following constructs:

- struct
- enum
- newtype

4.5.6.1 Struct

Structs are types that have named fields and each field has its type. It represents the product of all its fields types. The alignment of a struct is the biggest alignment of all of its fields types. The size of a struct is the total size of its fields and any padding bytes added so that their alignment requirements are met.

The struct's fields are laid in memory in the same order as defined using the syntax. This is for compatibility with C code, but I might change it in the future and add an attribute to change the memory layout at compile time.

4.5.6.2 Enum

Enums are special integer types that will have the same memory representation as one of the primitive integer types. They can contain any number of members. Each member is assigned a unique integer from 0 to N-1 where N is the number of members defined. Unlike C enums, Jelly treats enums as different unique types and no implicit conversions exist between them and integers.

4.5.6.3 Newtype

Newtypes are generic types based on an already existing type. Tagged types are only allowed to have newtypes as their inner type. An example would be the built-in type `'Size` which is the base of the tagged type returned by calling `'size_of` on a type T.

4.6 Implicit Type Conversion

Implicit type conversions are type conversions added by the compiler to expressions that expect a value of a specific type. These conversions are performed in function call arguments, return statements, assignments, etc. They can be useful when two types are very similar, and they are often used in the same ways as the two types of pointers defined in Jelly.

Jelly adds a very small number of implicit conversions so that programmers are more aware of what the code means. There's no implicit conversion that loses information because I believe losing information should be an explicit operation. At the beginning I had conversions from integers and floating-point numbers to types with a wider range, but I decided against it to let the compiler catch more errors. The following table shows all implicit conversions:

Actual type	Target type
`*mut T`	`*T`
`@mut T`	`@T`
`*mut T[N]`	`@mut T`
`*(mut) T[N]`	`@T`
`*mut T`	`*mut byte`
`*(mut) T`	`*byte`
`T[Tags...]`	`T`

Table 3: Implicit conversions in Jelly

4.7 Value Category

Values in Jelly can have different categories:

- Temporary or rvalue [5]
- Constant lvalue or place
- Mutable lvalue or place

Temporaries represent values that have no memory address. They can be thought of as if the values were stored in CPU registers.

Places represent objects with a memory address. They can be constant or mutable. Only mutable places can be modified. Having both types is beneficial because it can prevent writing to read-only memory.

4.8 Constant Value

A constant value is an integer or floating-point number that is known at compile time. In Jelly, operators that work on constant values are computed at compile time which allows them to be used in array types.

4.9 Role

I decided very early on that I wanted types and values to share the same syntax. This would allow to have simpler syntax for the language, and it could allow some interesting metaprogramming properties like using types as values. So I created the concept of roles. In Jelly any expression can have different roles:

- Module
- Built-in
- Type
- Tag type
- Value
- Multivalue

Values and multivalues have additional properties like value category and type.

Tag types are types that can be tagged to form a tagged type. They describe the type parameters necessary to create a tagged type, and they can also be used as regular types. In the latter case, they act as a type-erased [2] generic type which would be the inner type. This property is useful to have when creating type-erased data structures. For instance, one might want to have a list of lists of any type. In that case we could do `List[List]`. Notice how we have a tagged type and its type argument is a type-erased generic type. An example would be `'Size`.

Multivalues were a late addition and their motivation stems from the ability to show clearly when an expression is creating a slice pointer or not. Jelly uses the operator `&` to obtain a pointer to a place and the indexing syntax `array[i]` often returns a place value. So if a pointer to an element from an array is required, the syntax to do that is `&array[i]`. That shows that the programmer is creating a pointer by using an ampersand. Previous to the addition of multivalues, the slicing syntax `array[low:high]` returned a temporary slice, but then it was not clear that a pointer was being created. The concept of multivalues was added to fix this issue.

Multivalues represent a contiguous range of elements of unknown length. This means that they can't be used normally because they have an unknown compile time size. But then you can apply the operator `&` to a multivalue, and it will create a temporary slice. That makes the new syntax `&array[low:high]` more obvious. It clearly shows that we are working with a pointer.

4.10 Token

Based on the compiler stages explained earlier, Jelly has been designed with the following token types:

- Identifier
- Built-in identifier
- Keyword
- Integer literal
- Floating-point literal
- String literal
- Character literal
- Operator
- Punctuation

Not every character in the source code is meaningful. For instance, whitespace is not meaningful in Jelly. Whitespace and new lines are meaningful in some languages like Python, but I decided to only make new lines meaningful. The reason is that automatic formatting can be a harder problem otherwise.

Comments are an important part of code that is not given any meaning in the language. They are sections of code that are completely ignored by the compiler so that programmers can add useful information that is not conveyed in the actual code. In Jelly, comments start with the character "#" and end at the end of a line. Multiline comments can be useful to quickly test code without executing an entire section. These can also be used for documentation purposes and longer explanations. I haven't added them to Jelly yet, but I plan to in the future.

4.10.1 Identifier

An identifier is any sequence of characters that follows the regular expression `[a-zA-Z_][a-zA-Z0-9_]*`. This is based on how most existing programming languages already work.

An interesting addition in Jelly is built-in identifiers. These follow the same rules as regular identifiers, but they start with a back tick character "`". The fact the regular identifiers can't contain this special character allows me to add as many built-in symbols without interfering with user defined names. One example would be the built-in function `` `size_of `` which is treated as a keyword in other languages like C. The choice of the back tick character "`" was quite arbitrary. It needed to be a character that isn't really used anywhere else, but one that is not very noisy like the backslash. I might decide to change it in the future.

4.10.2 Keyword

A keyword is a word that has a predefined meaning. They are used to aiding in readability and the parsing of the syntax. The keywords in Jelly are:

- and
- as
- break
- const
- continue
- else
- enum
- extern
- false
- for
- function
- if
- import
- let
- module
- mut
- newtype
- null
- or
- public
- return
- struct
- switch
- true
- while

These were selected so that they are intuitive and similar to other languages. Most of the keywords come from C or C++.

The logical operators use `and` and `or` from Python because I think they are more readable than the usual "&&" and "||".

The use of `function` to declare a function comes from JavaScript.

The keywords `as`, `let` and `mut` come from Rust because of their similar semantics in Jelly.

4.10.3 Integer Literal

Integer literals are tokens that represent an integer value. They can take two forms. The first form is base 10, and it is expected to be the most common one. It can be described with the following regular expression: `[0-9]+`. A lot of languages add a way to separate digits to improve readability for big numbers. Rust is one of these languages [6], and it uses underscores for that purpose. I haven't added this feature to Jelly yet, but I plan to add it in the future. An example of integer literal could be `1428190385`. The second form is written in a hexadecimal base, and it follows the regular expression: `0[xX][0-9A-Fa-f]+`.

The type of integer literals is context-dependent, and it is decided during semantic analysis. The decided type must have a big enough range to represent the integer. Otherwise and in case that there's no context, the type chosen will be `i64`.

4.10.4 Floating-point Literal

Floating-point literals are tokens that represent a floating-point value. They have different parts. The first part is the whole number. The fractional part comes after the decimal point. Finally, an optional part can be added at the end that represents an exponent. Adding this part multiplies the value by a power of ten with the specified exponent. The exponent can be negative if the minus character is used. For instance, 1.7e-2 represents the value 0.017.

Just like integer literals, the type of floating-point literals is context-dependent, and

it is also decided during semantic analysis. If there's no context, the type chosen will be `f64`.

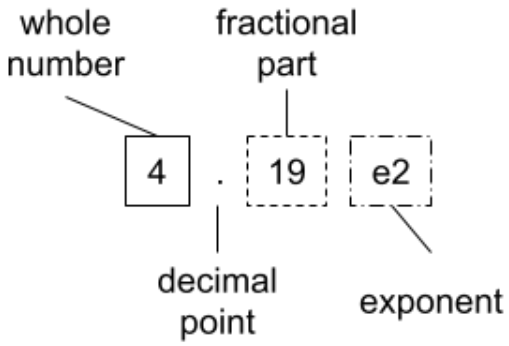


Figure 9: Parts of a floating-point literal

4.10.5 String and Character Literals

A character literal is a token that represents a single byte. During semantic analysis, it is a `char` temporary.

String literal represent a sequence of characters using the Unicode UTF-8 encoding. During semantic analysis, it is a constant place that holds an array of `char`. The length of the array is determined by the length of the string in bytes. During execution, strings are stored in a read-only section of the program's memory. This is why they are places, and it is possible to obtain a reference to them.

Strings are compatible with C's null terminated strings. A null character is added to every string, but it is not part of the type. This means that the length of its array type doesn't include the null character, but it can be passed to C functions safely.

Both types of literals can contain escape sequences. These are sequences of characters that have a special meaning. They are useful to insert characters that can't normally appear in a string or character literal. The escape sequences are:

- `\\` - backslash
- `\t` - tab
- `\n` - newline
- `\r` - return
- `\'` - single quote
- `\"` - double quote
- `\xDD` - hexadecimal byte DD

The last escape sequence can represent any byte possible which covers any character that would be impossible to insert otherwise. For instance, `\x0A` represents the same character as `\n`.

4.10.6 Boolean Literal

There are two boolean literal tokens: true and false. They are considered keywords by the language. During semantic analysis, a boolean literal is a `bool` temporary.

4.10.7 Null Literal

The null literal is represented by the null keyword, and it represents a pointer that is invalid. During semantic analysis, the type of a null literal is inferred from context. Otherwise, its type is `*mut byte`.

4.10.8 Operator

Operators are tokens that might represent an operation during semantic analysis. Jelly has the following operator tokens:

- plus (`+`)
- minus (`-`)
- star (`*`)
- division (`/`)
- modulo (`%`)
- not (`!`)
- at (`@`)
- bitwise (`&`, `|`, `^`)
- bit shift (`«`, `»`)
- comparison (`==`, `!=`, `<`, `>`, `<=`, `>=`)
- assignment (`=`)
- compound assignment (`+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`)

4.10.9 Punctuation

Punctuation tokens are tokens needed to delimit or separate expressions, statements or other syntactic structures. Jelly has the following punctuation tokens:

- round parenthesis (`(`, `)`)
- square parenthesis (`[`, `]`)
- curly parenthesis (`{`, `}`)
- comma (`,`)
- period (`.`)

- colon (`:`)
- semicolon (`;`)
- arrow (`->`)

4.11 Syntax

I chose Jelly's syntax carefully so that it is context-free, and it can be easily parsed by a recursive descent parser. I really like languages that do not require the use of semicolons to separate statements, and so I decided not to include them in Jelly. That means that new lines can be significant, but they are only required for parsing statements that start with an expression like assignments of function calls. Everywhere else, new lines are ignored.

A Jelly source file should always start with the module declaration syntax. The syntax can be divided in top-level definitions, statements and expressions. A top-level definition is a syntactic structure that can appear in any order on the source code. They include the following types:

- Import
- Function
- Struct
- Enum
- Newtype
- Constant
- Extern

A statement is any of the elements that can appear inside a function body. They are classified in:

- Variable/constant definition
- If/else statement
- While loop statement
- For loop statement
- Break/continue statement
- Return statement
- Expression statement

An expression is a part of the syntax that can represent any of the Jelly roles. These are composed of the following items:

- Atomic expression
- Unary operator
- Binary operator
- Field access
- Function call
- Subscript
- Slice
- List
- Switch expression
- Array type
- Function type

In the following sections, the syntax for Jelly is defined using Backus-Naur form (BNF) [14].

4.11.1 File

```
<file> ::= <module-declaration> <imports> <definitions>
<imports> ::= "" | <import> <imports>
<definitions> ::= "" | <definition> <definitions>
<definition> ::= <function> | <struct> | <enum> | <constant> |
    <extern-function> | <extern-mut>
```

This is the syntax that each source file must follow.

4.11.2 Module

```
<module-declaration> ::= "module" <id>
```

Each source file must declare what module it belongs to using the syntax above.

4.11.3 Import

```
<import> ::= "import" <id>
```

An import statement lets a file use names from other modules with the access syntax. The name of the imported module will be usable as an expression that has module role.

4.11.4 Expression

```
<atom> ::= <id>
    | <built-in-id>
    | <int>
    | <float>
    | <character>
    | <string>
    | "true"
    | "false"
    | "null"
    | "(" <expr> ")"
<expr> ::= <atom>
    | <unary>
    | <binary>
    | <field-access>
    | <inferred-field-access>
    | <call>
    | <subscript>
    | <slice>
    | <list>
    | <array-type>
    | <function-type>
```

Each expression will be given a role during semantic analysis.

4.11.5 Unary Operator

```
<unary> ::= <operator> <expr>
```

Operator	Example	Meaning
`+`	`+a`	the value of a
`-`	`-a`	the negative of a
`*`	`*a`	dereference the pointer a to access the value it refers to
`*`	`*T`	constant pointer to type T
`*mut`	`*mut T`	mutable pointer to type T
`&`	`&a`	create a pointer that refers to the value a
`@`	`@T`	constant slice to type T
`@mut`	`@mut T`	mutable slice to type T
`!`	`!a`	the bitwise negative of a

Table 4: Unary operators and their meaning

All unary operators have the same precedence.

In C, the unary operator `&` that creates a pointer, can only be used on values that have a memory address. In Rust, the same operator can be used on any value, but if it is not a memory address, a new temporary is stored on the stack and its address is returned. This is very helpful when working with generic code or when a function takes a parameter of pointer type and the programmer just wants to pass a temporary object to it. For these reasons, I decided to add the same functionality that Rust provides.

The same syntax is used for operating on values and types. At the beginning I thought about choosing a different syntax for types. Some languages like Odin [3] have syntax that works in two directions. Types go from the right to the left. Values are used from left to right. This has very nice properties and makes the syntax very intuitive:

```
x: ^int     // a pointer to an integer
x^          // a dereference operator on a pointer
```

Because of this rule, parenthesis are not necessary for types. The problem is that, in Jelly, any expression can be a type. This makes it a lot harder to have this syntax because it would mean having an operator that works as prefix and postfix. This would complicate parsing, so I decided against it even if I really like it.

4.11.6 Binary Operator

```
<binary> ::= <expr> <operator> <expr>
```

Operator	Example	Meaning
`+`	`a + b`	the addition of a and b
`-`	`a - b`	the subtraction of a and b
`*`	`a * b`	the product of a and b
`/`	`a / b`	the division of a by b
`%`	`a % b`	the remainder of a divided by b
`&`	`a & b`	the bitwise AND of a and b
`|`	`a | b`	the bitwise OR of a and b
`^`	`a ^ b`	the bitwise XOR of a and b
`«`	`a « b`	a left shifted by b
`»`	`a » b`	a right shifted by b

Table 5: Arithmetic operators and their meaning

Binary operators can work on tagged types whose inner types are numeric types. The result will get rid of any tags.

Operator	Example	Meaning
`and`	`a and b`	the logical AND of a and b
`or`	`a or b`	the logical OR of a and b

Table 6: Logical binary operators and their meaning

The following list shows the order of precedence of operators, "1" being the highest.

4.11.7 List

```
<list> ::= "[" <list-elements> "]"
<list-elements> ::= "" | <expr> | <expr> "," <list-elements>
```

The list syntax provides a way to create arrays that contain a sequence of elements. The result of this expression is a new array value that contains the values in the

Operator	Example	Meaning
`==`	`a == b`	a is equal to b
`!=`	`a != b`	a is not equal to b
`<`	`a < b`	a is less than b
`>`	`a > b`	a is greater than b
`<=`	`a <= b`	a is less than or equal to b
`>=`	`a >= b`	a is greater than or equal to b

Table 7: Comparison operators and their meaning

Operator	Example	Meaning
`=`	`a = b`	a becomes equal to b
`+=`	`a += b`	a becomes equal to the addition of a and b
`-=`	`a -= b`	a becomes equal to the subtraction of a and b
`*=`	`a *= b`	a becomes equal to the product of a and b
`/=`	`a /= b`	a becomes equal to the division of a by b
`%=`	`a %= b`	a becomes equal to the remainder of a division by b
`&=`	`a &= b`	a becomes equal to the bitwise AND of a and b
`|=`	`a |= b`	a becomes equal to the bitwise OR of a and b
`^=`	`a ^= b`	a becomes equal to the bitwise XOR of a and b

Table 8: Assignment operators and their meaning

expression list. The element type of the array can be inferred from context. If there's no context, the element type is the type of the first expression. The value category of the result is a temporary. Empty arrays are treated as a semantic error.

The syntax is designed so that a trailing comma is allowed. This is something I really care about because it makes it easier to reorder elements in a long list or adding elements at the end. This feature is consistent in all syntactic structures that are separated by commas. But in C, this is not possible when calling functions:

```
int x = call_function(
    a,
    b,
    c,      // syntax error
);
```

The following code shows how to create a list containing the five consecutive integers starting from zero:

```
[0, 1, 2, 3, 4]

# error
[]
```

4.11.8 Field Access

```
<field-access> ::= <expr> "." <id>
<inferred-field-access> ::= "." <id>
```

Precedence	Operators	Associativity
1	()	Left-to-right
	[]	Left-to-right
	.	Left-to-right
2	unary operators	Right-to-left
3	as	Left-to-right
4	* / %	Left-to-right
5	+ -	Left-to-right
6	&	Left-to-right
	|	Left-to-right
	^	Left-to-right
	« »	Left-to-right
7	== != < > <= >=	Left-to-right
8	and or	Left-to-right
9	=	Left-to-right
	+= -= *= /= %=	Left-to-right
	&= |= ^=	Left-to-right

Table 9: Jelly's precedence rules

- If the expression is an enum type and the identifier is one of the enumerators defined in the enum, then the result is the value of the enumerator. If the expression is missing, then the enum type must be inferred from context. Otherwise, it is an error.
- If the expression is a module and the identifier is a public definition in the module, then the result is the value of that definition.
- If the expression is a value of struct type and the identifier is one of the fields defined in the struct, then the result is the value of that field. The value category is the same as the value category of the expression.
- If the expression is an array or slice and the identifier is length, then the result is its length.
- If the expression is a slice and the identifier is data, then the result is a pointer to the beginning of the slice.
- If the expression is a pointer, then it is dereferenced implicitly and the previous rule is applied. This rule is not recursive.

Examples:

```
mut p = Point(0.0, 1.0)

# assigns the value of the field y to the field x
p.x = p.y

# evaluates to 3
let length = [3, 4, 5].length
```

4.11.9 Call

```
<call> ::= <expr> "(" <call-arguments> ")"
<call-arguments> ::= "" | <expr> | <expr> "," <call-arguments>
```

- If the expression is a struct, then the result is a new value constructed from the arguments provided. The number and order of arguments must match the number and order of struct fields. The types of the arguments must be implicitly convertible to the fields types. The resulting value category is a temporary.
- If the expression is an affine type, then the result is a new value constructed from the arguments provided. Only one argument must be provided, and it must have a type compatible with the non-affine type. The resulting value category is a temporary.
- If the expression is a function, then the result is the value that results from calling the function with the specified arguments. If the function has no return value, the expression is considered a statement, and it can't be used as a value. The number of arguments must match the number of the function type parameters. The types of the arguments must be implicitly convertible to the types of the function type parameters. If the function has type parameters, those will be inferred by the arguments provided. The resulting value category is a temporary.

Examples:

```
# struct constructor
Point(4.1, -9.9)

# affine constructor
AffineInteger(3)

# function call
f(7, [0, 0])
```

4.11.10 Subscript

```
<subscript> ::= <expr> "[" <subscript-arguments> "]"
<subscript-arguments> ::= "" | <expr> | <expr> "," <subscript-arguments>
```

- If the expression is an array or slice, then the argument list must have one value of isize type. The resulting value category is the same as the one of the expression used.
- If the expression is a tag type, then the result is a tagged type whose inner type is the type of the expression and the tags are the types provided in the argument list.

- If the expression is a built-in, then the built-in function is invoked with the arguments provided.
- If the expression is a value of pointer type, then the pointer is dereferenced implicitly and the previous rules are applied without recursion.

Examples:

```
# array indexing
[4, 5, 6][1]

# tagged type
`Size[T]

# built-in call
`size_of[i32]
```

The following table shows what Jelly's built-in functions do:

Built-in	Parameters	Return type	Result
`size_of	type T	`Size[T]	The size of the type T in bytes
`align_of	type T	`Alignment[T]	The alignment of the type T in bytes
`zero_extend	integer	Inferred integer	Casts an integer with no sign extension
`slice	isize, *(mut) T	@(mut) T	Constructs a slice with the given length and pointer to first element
`Affine	type T	`Affine[T]	Creates a new affine type
`ArrayLength	isize N	`ArrayLength[N]	Creates a new `ArrayLength type

Table 10: Jelly's built-in functions

4.11.11 Slice

```
<slice> ::= <expr> "[" <slice-arguments> "]"
<slice-arguments> ::= ":" | <expr> ":" | <expr> ":" <expr>
```

The slicing operator slices an array with the given range. The first argument is the lower bound and the second argument is the upper bound. If the lower bound is not provided, it is implicitly treated as the beginning of the array or slice. If the upper bound is not provided, it is implicitly treated as the end of the array or slice. The upper bound is not included in the resulting range.

The bounds must be values of isize type and the expression must be a value of array or slice type. The result is a multivalue of slice type.

Examples:

```
let a = [1, 2, 3, 4, 5, 6]

# the resulting slice will refer to the elements [2, 3, 4]
print_integers(&a[1:4])

# the resulting slice will refer to the elements [4, 5, 6]
print_integers(&a[3:])

# the resulting slice will refer to the elements [1, 2, 3, 4, 5, 6]
print_integers(&a[:])
```

4.11.12 Array Type

```
<array-type> ::= "[" ":" <expr> "]" <expr>
    | "[" <expr> "->" <expr> "]"
```

The result of this expression is an array type with the specified index and element types. If the first syntax is used, the index type is an `‘ArrayLength` type with the integer value provided in the first expression.

Examples:

```
# array of three i32
[:3]i32

# same as the above, but more explicit and useful for generics
[`ArrayLength(3) -> i32]
```

4.11.13 Function Type

```
<function-type> ::= "function" "(" <parameters> ")" <return-type>
```

The result of this expression is a function type with the specified parameter types and return type if one is provided. The following function type represents any function that takes two i32 parameters and returns an i64:

```
function(x i32, y i32) -> i64
```

4.11.14 Switch Expression

```
<switch> ::= "switch" <switch-condition> "{" <cases> "}"
<switch-condition> ::= "" | <expr>
<cases> ::= "" | <case> | <case> "," <cases>
<case> ::= <expr> "->" <expr> | "else" "->" <expr>
```

A switch expression returns one of the values that can be computed from the list of cases. There are two types of switch expressions depending on whether they have a condition expression. When there is a condition expression, each case pattern is

compared to the condition and the first one to compare equal is returned. If there is no condition, the case patterns are values of bool type and the first one to evaluate to true is executed. This one can serve as a replacement to the typical if-else-if chain in a lot of programming languages. This use case was inspired by the Go programming language [8]. The else case is always successful, in case any other cases were not. Switches are expressions, which means their cases must be exhaustive, otherwise the expression could have undefined values.

Example of switch on condition:

```
let value = switch color {
    .red -> 0,
    .green -> 1,
    .blue -> 2,
    else -> -1,
}
```

Example of bool switch:

```
let value = switch {
    x < 2 -> 1.0,
    x > 2 -> -1.0,
    else -> 0.0,
}
```

4.11.15 Variable Definition

```
<variable> ::= <var-keyword> <id> "=" <expr>
<var-keyword> ::= "let" | "mut"
```

A variable definition is a statement that introduces a new identifier in the current local scope. The variable is a place that is constant or mutable depending on weather the keyword let or mut is used respectively. The type of the variable is inferred from the type of the expression that initializes the variable. The type must have a known size at compile time, which means that it can't be used with types that depend on generics.

4.11.16 Constant Definition

```
<constant> ::= "const" <id> "=" <expr>
```

A constant is a top-level definition or statement that represents an expression that is known at compile time. The expression must have a value, type or built-in role. If the expression is a value, it can only be a constant value.

Here we declare a constant that holds the value of pi:

```
const pi = 3.14159265358979323846
```

4.11.17 If/else Condition

```
<if> ::= "if" <expr> "{" <statements> "}" <else>
<else> ::= "" | "else" "{" <statements> "}" | "else" "if" <expr> "{"
    <statements> "}" <else>
```

An if statement introduces a conditional branch of code execution. The condition must be a value of bool type. If the condition is true, the first block of code is executed. Otherwise, if an else branch exists, the second block of code is executed.

Example:

```
if value > 0 {
    print_str("positive integer")
} else {
    print_str("non-positive integer")
}
```

4.11.18 While Loop

```
<while> ::= "while" <expr> "{" <statements> "}"
```

A while statement executes a list of statements repeatedly while the condition evaluates to true. The condition is evaluated at the beginning of every iteration.

Example:

```
mut i = 0

while i < 3 {
    print_int(i)
    i += 1
}
```

4.11.19 For Loop

```
<for> ::= "for" <id> "=" <expr> ";" <expr> ";" <expr> "{" <statements> "}"
```

A for statement initializes a new variable and executes a list of statements repeatedly while the condition evaluates to true. The condition is evaluated at the beginning of every iteration. At the end of every iteration, the next expression is guaranteed to be executed unless a break statement is executed. In the future I would like to make the variable immutable inside the block because I believe it can make the code less clear.

Example:

```
for i = 0; i < 100; i += 1 {
    if should_skip(i) {
```

```
        continue
    }

    print_int(i)
}
```

The for loop above will print the integers from 0 to 99 only when the function should_skip returns false.

4.11.20 Return Statement

```
<return> ::= "return" <return-value>
<return-value> ::= "" <expr>
```

By default, the last statement of a function is its return value if it has a return type. Otherwise, a return statement must be used to return early from a function. The type of the value must be implicitly convertible to the return type of the function.

Example of return statement:

```
function abs(x i64) -> i64 {
    if x < 0 {
        return -x
    }

    x
}
```

4.11.21 Function

```
<function> ::= "function" <id> <optional-type-parameters> "("
   <parameters> ")" <return-type> "{" <statements> "}"
<optional-type-parameters> ::= "" | "[" <type-parameters> "]"
<type-parameters> ::= "" | <id> <type-parameters>
<parameters> ::= "" | <parameter> | <parameter> "," <parameters>
<parameter> ::= <id> <expr>
<return-type> ::= "" | "->" <expr>
<statements> ::= "" | <statement> <statements>
<statement> ::= <variable>
    | <constant>
    | <if>
    | <while>
    | <for>
    | <return>
    | <expr>
```

Functions are segments of code that can be called from anywhere in the code. They allow composability and they reduce code repetition. Functions may have any number of parameters and a return value. They can also provide a list of type parameters for generic type programming.

Parameters in Jelly are passed by value as C does. This means that a function call cannot modify its arguments, but reference semantics can still be achieved with pointers and dereferencing. C treats arrays as a special case, and it doesn't allow to have them as return types. They can be used as parameters, but they are treated exactly like a pointer. In Jelly, arrays are treated the same as any other type, and they would be passed by value.

Jelly does an optimization to return types of bigger sizes. Copying big types can be inefficient, so Jelly returns them by pointer transparently. A new implicit pointer parameter is added to the function and all return statements will implicitly write to this parameter.

Functions that return a value will use the last statement as the return value. This statement has to be a value that can be converted to the return type.

Example of function definition:

```
function clamp(x f64, min f64, max f64) -> f64 {
    if x < min {
        return min
    }

    if x > max {
        return max
    }

    x
}
```

4.11.22 Struct

```
<struct> ::= "struct" <id> "{" <fields> "}"
<fields> ::= "" | <field> | <field> "," <fields>
<field> ::= <id> <expr>
```

Structs in Jelly are not allowed to be empty, but that is checked during semantic analysis so that parsing can succeed.

Example of struct definition:

```
struct Point {
    x f32,
    y f32,
}
```

4.11.23 Enum

```
<enum> ::= "enum" <id> <expr> "{" <enumerators> "}"
<enumerators> ::= "" | <id> | <id> "," <enumerators>
```

4.11.24 Newtype

```
<enum> ::= "newtype" <id> <optional-type-parameters> "=" <expr>
```

Example of how the built-in `‘Size` newtype could be defined:

```
newtype Size[T] = isize
```

This introduces a new tag type that can be tagged with one type argument, and it will always have the memory layout of an isize. Newtypes can't be defined using any of the type parameters.

4.11.25 Extern

```
<extern-function> ::= "extern" "function" <id> "(" <parameters> ")"
    <return-type>
<extern-mut> ::= "extern" "mut" <id> <expr>
```

An extern definition is a symbol that will be provided externally by a linker. These are necessary to interact with libraries written in C. They come in two forms: extern functions and extern variables.

An extern variable represents a place of a specified type. This type of variable is a compatibility feature with C that allows programmers to read global variables defined in C code. The variable can't be modified from within Jelly code.

The following example imports a function from the C standard library:

```
extern function putchar(ch i32) -> i32
```

5 Compiler Implementation for Jelly

5.1 Language Choice

A Jelly compiler can be implemented in any programming language because it doesn't require any special libraries. Since compilers are very complex pieces of software, I decided it was best to use a programming language I was already comfortable using. I had several choices: Java, C++, C, Python or Rust. From those, I picked C because it can be fast and simple. If I ever wanted to switch to C++, it would be relatively easy because it is mostly a superset of C with some exceptions. Another advantage of C is that it has a multiprocessing library called OpenMP that can speed up compilation since some compiler stages are embarrassingly parallel.

Because Jelly is a similar language to C, writing the compiler in C would make the potential transition to a self-hosting compiler much easier. A self-hosting compiler is a compiler that can compile itself. In this case it would be a Jelly compiler that can compile Jelly code.

5.2 Design

The compiler [12] has been redesigned multiple times. At first, it only parsed the AST, and it did everything on the same data structure. This created code duplication when doing code generation that had to be updated twice every time I changed something. This is when I decided add more intermediate representations as more analysis was done.

At one point I also changed completely the design of the whole compiler to add support for multithreaded compilation. This was easy for parsing because each file can be parsed separately, but it proved to be very difficult for semantic analysis because names can refer to entities defined in other files. So what I did was to split semantic analysis in two steps. The first step would only analyze all the top level definitions. This step can't be done in multiple threads. The second step is to analyze all the function bodies. Because function bodies can't interact between each other (they are self-contained), they can be analyzed in parallel.

One issue that I came across was that types and other data is stored in big arrays, and function analysis needs to add new types. That would lead to a data race if two functions try to add new types at the same time. One solution would be to use a mutex, but creating new data is a common operation during semantic analysis. So I opted for a different way. I split all data into two arrays: a global array that is only mutated during the first non-parallel step and an array per thread. This approach doesn't require any mutex, but it requires special indices to access to these arrays. Since we have two arrays, we can't use a simple integer to know where the data is stored. The solution I came up with was to reserve the range (0, N - 1) to data in the global array and (N, M) to data in the thread array. Since the global array can't be modified once we start adding data to the local arrays, these ranges are safe to use. Two pieces of data could share the same index if they are created in different functions, but since they can't refer to each other, it is easy to identify the local array that contains the data.

Unfortunately, the examples that I have for testing the compiler show that the

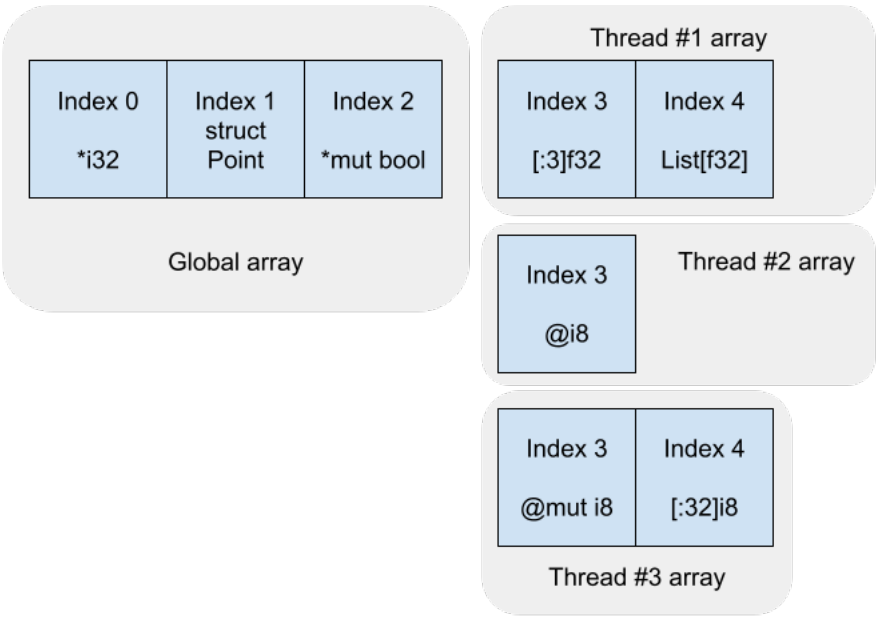


Figure 10: Simple graphic for type arrays during type analysis

performance is worse when using more than one thread. I believe this is mostly because the examples are not big enough and that performance would improve if the project was very big.

5.3 Arguments

The compiler is a program that can take a different amount of arguments and options. There are currently three options:

Option	Values	Meaning
-backend	c, llvm	Selects the output of the compiler
-print-debug	-	Prints debug information of intermediate representations
-help	-	Prints how to use the compiler program

Table 11: Jelly's compiler options

5.4 Diagnostic

Compiler diagnostics are messages providing useful information to the user. Compilers don't necessarily need to do this, but it is a nice feature that can help a lot when diagnosing coding bugs. Diagnostics can have several purposes which include errors, warnings, notes, fix-it's, hints and others. The Jelly compiler shows diagnostics with context information so that users can quickly spot the mistakes in the source code.

At first, diagnostics were implemented via a function that took a message string as a parameter and the function printed it. Later, I found a lot of messages being repeated, and some other messages contained information such as types so the logic for generating these messages was all over the code. Then I decided to switch to a struct that would contain the necessary information to generate the error message. This allowed me to localize all messages into one single place.

```
1   module main
2
3   function main() {
4       let constant_variable = 3
5       constant_variable = 19
6   }
test/selfhost/main2.jel:5:5: error: expected mutable lvalue
5 |     constant_variable = 19
      ~                 ^
test/selfhost/main2.jel:4:9: note: consider replacing `let` with `mut`
4 |     let constant_variable = 3
          ^~~~~~~~~~~~~~~~~
```

Figure 11: Compiler diagnostics

5.5 Arena Allocator

An arena allocator is a memory allocator that is very simple as it disallows freeing any of its objects. This makes its implementation very simple, and it is very similar to the concept of function stack, but on the heap. An arena only requires a block of memory and two pointers: the top and the end. The top represents the current position in the block of memory. The end is the end of this block. Whenever an object allocation is requested, the top pointer is aligned to the alignment of the object, and we return it. Then, the pointer is increased by the size of the object. Before that, enough space left to perform the allocation is asserted. If there is no enough space, the arena implementation could be improved by adding new memory blocks.

I've decided to use arenas for most memory allocations in the Jelly compiler. This simplifies lifetime management [7] because only a single big object needs to be freed instead of many smaller ones. Because they are very limited, their implementation is quite simple.

Arenas come with their downsides. During development, I've used a tool called valgrind to debug memory errors that are common in C. The tool works well with the existing malloc and free functions, but doesn't work automatically with other allocators that you might implement. This has lead to some bugs that have been difficult to diagnose. Usually, the problem stemmed from duplicating an arena and allocating pointers from both duplicates. The second allocation would overwrite the memory of the first. Something like affine types would have prevented these bugs.

5.6 Hash Table

Hash tables are a very important data structure for compilers, and they are used extensively. Every time an identifier is found, a table look-up needs to be executed. That means that a good implementation can make the compiler a lot faster than a non-optimal implementation.

Hash tables can be implemented in many ways. Some of these use what's called separate chaining, in which every bucket of the hash table is a linked list and, if collisions happen, elements will be stored in these lists. Other types use open addressing, in which only a contiguous array is used and collisions are resolved differently. Open addressing is a very efficient implementation for a compiler because it results in less cache misses. Compilers usually only need insertion and look-up, and they rarely need to remove

```
extern function sinf(x f32) -> f32
extern function cosf(x f32) -> f32
extern function fminf(x f32, y f32) -> f32
extern function fmaxf(x f32, y f32) -> f32
```

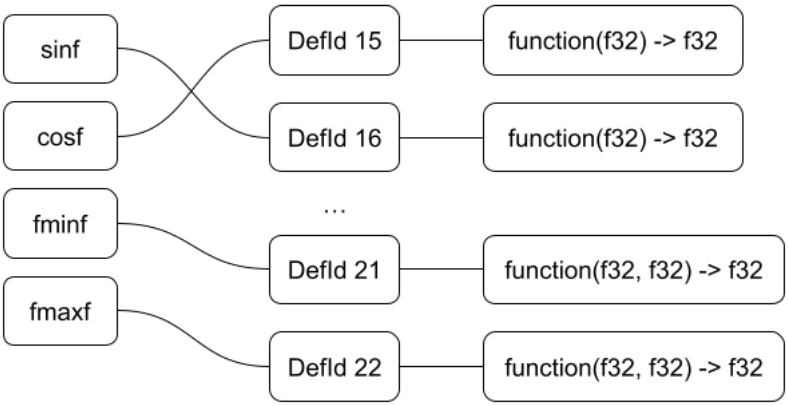


Figure 12: Hash table for a module scope

elements from tables. This quirk allows the biggest problem with open addressing, which is removal, to be avoided completely. Finally, most of the time is spent reading the hash table keys. If separate arrays are allocated for the keys and the values, the cache can be used more efficiently because more keys will fit in. So this is the approach I used to implement hash tables in the compiler.

5.7 Intermediate Representation

Intermediate representations (IR) are different forms to represent the original source code that the programmer wrote. A new IR is usually added to a compiler so that the next stage can be easier to implement. IR's can take many forms depending on their use. Some of them can form trees like the AST and others are just lists like a three-address code. In my case, I started with the AST and nothing else, but semantic analysis become a very big stage that did a lot of things and the code generation stage repeated some of those things. So I decided to split it into multiple stages and IR's after I wanted to add substructural types. In the current state, the compiler has the following IR's: AST, RIR, TIR and MIR. Each one is represented in code by a different set of C structs and enums, so the type checker can help prevent bugs. They will be explained individually in their own sections.

5.8 Lexical Analysis

The implementation for lexical analysis is usually very simple. There can be many ways to do it and the options that were contemplated are: lexical analyzer generators, state machines and plain loops. Every alternative has its advantages and disadvantages.

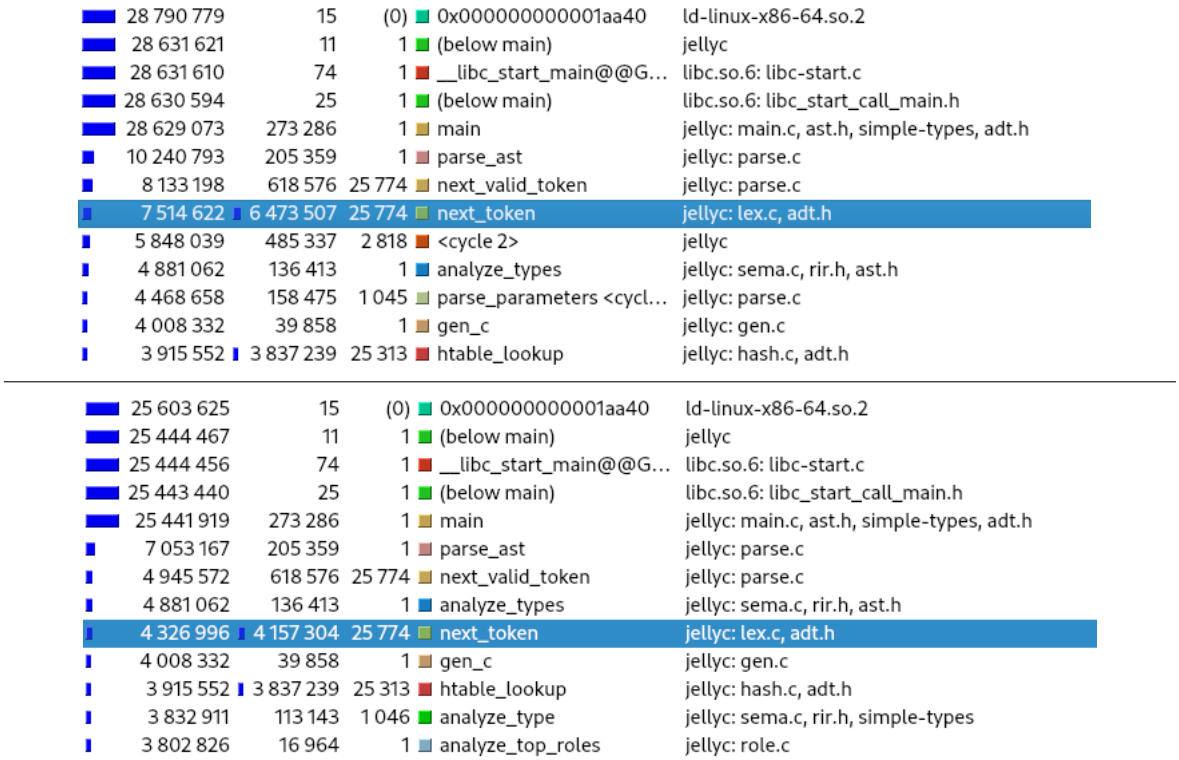
Generators are very easy to use because they do the hard work for you. They require you to provide the language's grammar, and it will generate a program that can transform source code into all of its tokens. The problem is that the added complexity might not be worth it, considering that making your own lexical is not that hard.

State machines can be hard to implement, and they can become hard to maintain if you decide to change the grammar.

Finally, plain loops are very easy to implement, and the grammar can be modified later without much of a problem. They behave like state machines, but the machine is given by the control flow of the program. For ease of use, the option I used for the compiler was plain loops.

The lexical analyzer interface is designed so that tokens don't need to be stored in memory. Instead, a function will return the next token, and it can be used directly by the parser. In case of error, the analyzer skips one character and returns an invalid token. This way, the parser can keep going and attempt to display more errors.

When measuring performance for a very long file generated from the OpenGL API, a lot of the compilation time was spent trying to find if an identifier was any of the keywords. An optimization that worked really well was to store the keywords in different arrays depending on their initial. This method is called radix sort, and it doesn't require any comparisons. Once these arrays are set, only a few keywords have to be checked per identifier.



28 790 779	15	(0)	0x000000000001aa40	ld-linux-x86-64.so.2
28 631 621	11	1	(below main)	jellyc
28 631 610	74	1	__libc_start_main@@G...	libc.so.6: libc-start.c
28 630 594	25	1	(below main)	libc.so.6: libc_start_call_main.h
28 629 073	273 286	1	main	jellyc: main.c, ast.h, simple-types, adt.h
10 240 793	205 359	1	parse_ast	jellyc: parse.c
8 133 198	618 576	25 774	next_valid_token	jellyc: parse.c
7 514 622	6 473 507	25 774	next_token	jellyc: lex.c, adt.h
5 848 039	485 337	2 818	<cycle 2>	jellyc
4 881 062	136 413	1	analyze_types	jellyc: sema.c, rir.h, ast.h
4 468 658	158 475	1 045	parse_parameters <cycl...	jellyc: parse.c
4 008 332	39 858	1	gen_c	jellyc: gen.c
3 915 552	3 837 239	25 313	htable_lookup	jellyc: hash.c, adt.h

25 603 625	15	(0)	0x000000000001aa40	ld-linux-x86-64.so.2
25 444 467	11	1	(below main)	jellyc
25 444 456	74	1	__libc_start_main@@G...	libc.so.6: libc-start.c
25 443 440	25	1	(below main)	libc.so.6: libc_start_call_main.h
25 441 919	273 286	1	main	jellyc: main.c, ast.h, simple-types, adt.h
7 053 167	205 359	1	parse_ast	jellyc: parse.c
4 945 572	618 576	25 774	next_valid_token	jellyc: parse.c
4 881 062	136 413	1	analyze_types	jellyc: sema.c, rir.h, ast.h
4 326 996	4 157 304	25 774	next_token	jellyc: lex.c, adt.h
4 008 332	39 858	1	gen_c	jellyc: gen.c
3 915 552	3 837 239	25 313	htable_lookup	jellyc: hash.c, adt.h
3 832 911	113 143	1 046	analyze_type	jellyc: sema.c, rir.h, simple-types
3 802 826	16 964	1	analyze_top_roles	jellyc: role.c

Figure 13: At the top, cycle estimation without the optimization. At the bottom, added optimization.

5.9 Syntax Analysis

The implementation of a syntax analyzer or parser often depends on the type of grammar the language has. In this case, I decided that a recursive descent parser would be best because of its simplicity. A recursive descent parser is a kind of top-down parser built from a set of mutually recursive functions where each such function implements one of the nonterminals of the grammar. Thus, the structure of the resulting program closely mirrors that of the grammar it recognizes.

One of the challenges in parsing is error handling. In C, errors can be handled in

different ways:

- A function can return an error code.
- A global variable can be set.
- The standard C library `setjmp` can be used.

Returning an error code is easy, but it can get verbose because all function calls have to be adjusted to check the return value. Setting a global variable has the same problems as returning an error code does and, in addition, the function is no longer thread-safe. Finally, using `setjmp` works similarly to C++ exceptions, and it results in very clean code, but it is also not thread-safe. For this project, I came up with a new solution. Whenever a syntax error is encountered, the lexical analyzer is changed so that it only returns invalid tokens. This way, all error handling in the parser is done automatically and most parsing code can remain unchanged. This could also have the added benefit of continuing parsing if an expected token was not found. The only thing that has to be done is to replace the look ahead token by the expected token, but I haven't done this for the compiler.

The AST is represented with a big dynamic array that can grow to allocate more nodes. Each node is represented with a homogenous struct which means that there is no need for virtual dispatching and double indirection. One problem with this approach is that some nodes have an undetermined amount of children at compile time such as function calls. These are implemented using an extra dynamic array that stores these lists and the main array only needs to store an index to this extra array. Because each node is very small, more of them can fit in a single cache line and iteration can be a lot faster. The inspiration for such an implementation comes from the creator of the Zig programming language [9].

Previously, the AST used pointers, and it allocated static chunks of memory using malloc when needed. This design is very simple and basic, but freeing the whole AST has time complexity O(n), where n is the number of block allocated. Eventually, I changed the design to the one mentioned above which can be freed in O(1).

The compiler has the ability to print the AST if you pass the option `-print-debug`. An example can be found in Appendix A.

5.10 Role Analysis

The Role Intermediate Representation (RIR) is an IR created during role analysis in which all ASTs are processed into modules and name resolution is performed. The word role comes from the fact that an expression in the AST could have different roles depending on context. For example, x is an identifier, but it is impossible to know whether it is a type or a value without looking up its definition. If this was done during parsing, the grammar wouldn't be context free. In order to keep things simple, role analysis is the stage that will classify each expression in a role.

This is considered the first step of semantic analysis. Before it starts, there's a previous step that will prepare the data that the semantic analyzer needs. This step

will generate all the hash tables needed for all the modules, and it will insert the top-level definitions to these tables. This is necessary because top-level definitions can be defined in any order, and they could include identifiers that refer to later definitions. Role analysis finds any recursive definitions because they are not allowed in Jelly. Names are assigned unique IDs that can be associated with any type of information.

Data for this analysis is stored as side tables to the main AST tables. Each AST table has an associated RIR table and the same index can be used to access AST information or RIR information. RIR only ends up storing two side tables: one that can identify an AST node with the kind of RIR node it is and another that can associate identifiers with their definition IDs.

Once role analysis is done, we obtain the RIR for each AST and type checking can be performed.

5.11 Type Analysis

The Typed Intermediate Representation (TIR) is generated during type checking from the previously obtained RIR. It includes type information for all expressions, and it represents much better the actual semantics of a Jelly program. Its purpose is to add explicitly all implicit conversions and simplify some control flow statements like for and while loops into one single loop type. This is done to facilitate substructural type checking which doesn't really see a difference between types of loops or between short circuit logical operators and switch expressions. Compile-time constant evaluation is also done during type checking to allow any user defined constants in array types.

The first step in semantic analysis will only analyze the top-level definitions and ignore all function bodies. The second step will analyze all function bodies and convert them into an intermediate representation. This separation is done because function bodies can be analyzed in parallel.

TIR introduces a lot of different data types. It adds types, values, interned strings and instructions. The following are the different types of values:

- Function
- Extern function
- Extern variable
- Constant integer
- Constant floating-point number
- Constant null pointer
- String
- Variable
- Temporary

All of these can be used as instruction operands.

Interned strings are the raw bytes that will be added to the final program's read-only memory to represent string literals. This way, only an index to this pool of strings is required to identify one string.

Similarly to how the compiler can print the AST, it can also print TIR for each function. An example can be found in Appendix A.

5.12 Substructural Type Analysis

This analysis doesn't actually produce any new IR. It is just there to check that substructural types are used properly. The compiler will assign every variable with a state. The initial state is not consumed. Whenever a variable is read, it becomes consumed. Then the variable can't be read anymore. The rules are the following:

- A variable can't be used after it has been consumed.
- If a variable is consumed in one branch of an if or switch, it must be consumed in all others.
- A variable defined outside a loop can't be consumed inside.

These rules prevent the variable from being used multiple times. When an affine type is referenced, but only a part of it that is not affine is read, it remains not consumed. Example:

```
struct MixedType {
    a i32,
    b SomeAffineType,
}

let m = new_mixed_type()
let b = m.a  # m is not consumed because m.a is not an affine type
```

5.13 Middle Intermediate Representation (MIR)

MIR is the last IR used by the compiler, and it simplifies the semantics a lot so that the backend only has to work with a very small set of operations. MIR is the only IR that is not a tree. It is just a list of instructions and operands.

MIR is composed of instructions and values. MIR instructions are based on the concept of three-address code which consists in instructions that can have at most three operands, but MIR can contain any number of operands. Instructions can generate new temporary values or perform a certain action. This compiler defines the following instructions:

- param
- alloc

- minus
- not
- address
- deref
- add, sub, mul, div, mod
- and, or, xor, shl, shr
- eq, ne, lt, gt, le, ge
- assign
- itof, itrunc, sext, zext, ftoi, ftrunc, fext
- nop
- call
- index
- const_index
- access
- new_slice
- br
- br_if, br_if_not
- ret_void, ret

The following rules are applied to convert TIR to MIR:

- Structured control flow is replaced with goto statements and labels.
- Compound assignments get expanded into the operation and the assignment.
- Variable definitions become an allocation instruction and an assignment.
- Unary plus operators are removed.
- Address operators to temporaries are transformed into address of temporary allocations.
- Struct constructors are converted into allocations and field assignments.
- Lists are converted into allocations and array element assignments.
- Explicit return statements are added when the last expression in a function is implicitly returned.

Example:

```
mut a = 6
let b = -2
a = a * b * b
```

The Jelly code above gets transformed into the following MIR code:

```
t0: i32 = alloc i32
t1: i32 = alloc i32
t0 = 6
t2: i32 = not 2
t1 = t2
t3: i32 = mul t0 t1
t4: i32 = mul t3 t1
t0 = t4
```

The compiler cannot print the MIR of a function because it is very similar to the resulting generated code in C or LLVM.

5.14 Code Generation

Code generation requires a very simple implementation because of the intermediate representation generated by semantic analysis. There are two backends: C and LLVM.

5.14.1 C Backend

The backend for C generates code that is not human-friendly, uses goto statements and many identifiers are lost. The reason why it works like this is that C has many quirks and its syntax is different enough to Jelly's syntax that generating code that works in all cases can be quite challenging. An example would be complex expressions in `while` conditions. C requires an expression in `while` conditions. But some expressions in Jelly can't be converted in C expressions. Some of them require C statements. But that would not be allowed in the `while` syntax that C has. That would be a corner case that has to be fixed by adding an `if` statement inside the `while` with a break statement. Eventually it was decided that it would be easier to generate code that always works even if it is not human-friendly. This backend generates a single ".c" file that can be compiled with GCC or Clang.

5.14.2 LLVM Backend

The backend for LLVM is meant as an intermediate step to produce code that should be compiled into assembly or machine code. This backend is provided to bypass the need of an entire compiler that has to go through a lot of stages that will eventually produce an intermediate representation similar to LLVM code. The Clang compiler uses LLVM as a backend, so using this backend for Jelly would remove a lot of unnecessary extra work. This backend generates a single .ll file that can be compiled using llc which is the LLVM compiler. The result is an object file that needs to be linked using a linker. A disadvantage of this method is that three programs are needed to compile a Jelly project: the Jelly compiler, llc and a linker.

6 Results

To be able to test that the compiler keeps working as I add new features, I made some small projects. Finally, I made a bigger project to showcase what one could do with Jelly.

6.1 Hello, World!

The first small project I did was the famous "Hello, World!" program. Since Jelly doesn't have any standard library yet, all the examples rely on the C standard library for I/O and memory allocation.

hello_world.jel M

test > hello_world.jel

```
1  module main
2
3  extern function putchar(ch i32) -> i32
4
5  function print_str(s @i8) {
6      for i = 0 as isize; i < s.length; i += 1 {
7          putchar(`zero_extend[s[i]])
8      }
9  }
10
11 function main() {
12     print_str(&"Hello, World!\n")
13 }
14
```

Figure 14: Hello world program written in Jelly

This example already tests a lot of features from Jelly. First, it uses an extern function to be able to print a character to the terminal. Then, it defines a function that iterates over a slice of characters and prints them. Finally, it defines the main function that converts a static string into a slice and calls the previous function. The compiler is invoked, and it outputs a C source file which must be compiled with a C compiler like GCC. You can find the result in C in Appendix A.

This is the result:

```
> jellyc test/hello_world.jel
> gcc a.c && ./a.out
Hello, World!
```

6.2 Basic Lexer

A more interesting program is a lexer. This one implements a subset of Jelly and prints to the terminal all the tokens found. The program is compiled as follows:

```
> jellyc test/basic_lexer.jel lib/std.jel lib/libc.jel
> gcc a.c && ./a.out
```

The C output is very big, so it is not provided in the document, but it can be compiled using the GitHub repository [12]. You can find the output of the program in Appendix B.

6.3 OpenGL Window

Last, but not least, I tested a program that uses OpenGL, and it can be interacted with. The program generates 3D terrain procedurally on the GPU, and it displays it on a window using shaders. The user can move through the world and change their point of view.

The code uses a lot of features in Jelly like slices, generics and modules. This is the true test that shows what this language is capable of. The compilation of this program is a bit more complicated, but it is provided as a bash script on the GitHub repository [12]. It requires two libraries: GLFW3 and an OpenGL loader called glad. Once run, the result can be seen below:

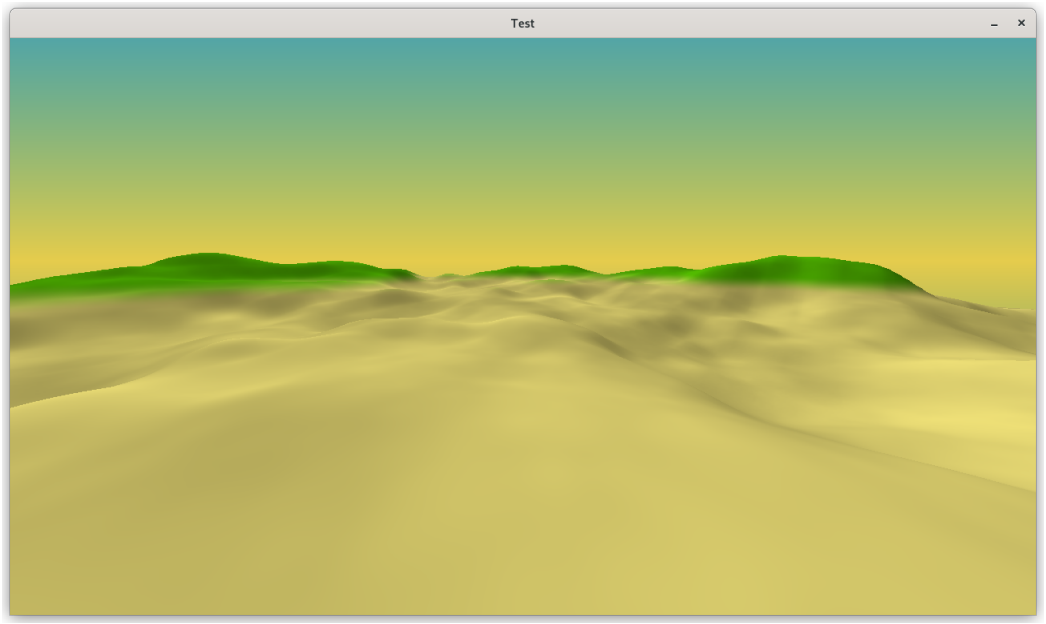


Figure 15: Window generated with OpenGL from Jelly code

7 Conclusions and future work

Designing a programming language and implementing a compiler is no easy task. It has made me learn a lot about new programming concepts and ideas. Some of these might be purely theoretical, and they might not be easy to implement in a real programming language, but others like uniqueness types and affine types show a lot of promise to help catch more bugs at compile-time. This project has solidified my understanding of some computer concepts like cache memory and parallel programming. This will help me a lot to optimize hot paths for future projects in case it is needed.

I have learned that type systems can help a lot more than I thought to make sure a program is sound and safe. I have also learned that abstraction comes at a cost, but this cost might outweigh the disadvantages. Furthermore, I have determined that no programming language is perfect. All of them have their faults and each one has its own use case. They are tools to accomplish an end goal.

In the following sections, I will mention features that I would like to add, change or remove from Jelly, now that I have acquired all this knowledge.

7.1 Macros

Macros were a feature of Jelly that transformed expressions based on certain rules. These worked on the AST level, unlike the C preprocessor which works on the source code level. Macros couldn't access the environment outside them which prevented them from accidentally using names that were defined in the current scope. I decided to remove them because they suffer from the fact that they expose their contents to whoever is using them. Then, the macro expansion could result in code with errors that the programmer can't understand because they come from a different module. That is similar to the types of errors that C++ templates can sometimes give and, in my experience, they are not very helpful. A good idea to add them back could be to restrict their visibility so that only the module where they are defined in can invoke them. Then, the programmer must know about the macro's internal details.

One thing that I would change about how macros used to work in Jelly is to make them have a type signature so that the compiler can type check them and give regular error messages. This makes macros a bit less powerful, but more user-friendly. Instead of doing this:

```
macro alloc[T, count] {
    malloc(`size\_of[T] * count) as *mut T
}
```

Which doesn't check that T is a type and that count is an integer, we could have:

```
macro alloc[T](count isize) {
    malloc(`size\_of[T] * count) as *mut T
}
```

Which could be type checked before being expanded, and therefore providing more meaningful diagnostics.

Macros were expanded before constant values were computed which means that they allowed computation of more complex expressions. They were also allowed to be called recursively. If switch expressions and arrays could be computed at compile time, constant computations would become Turing complete [21].

7.2 Arrays

I wanted to add out of bounds runtime checks for array indexing, but I didn't have time to do it even if it is a simple feature. Another feature I would like to add is arrays indexed with distinct integer types. Having a new type that is not compatible with integers can be useful to provide more type safety when using indices as IDs for some array of objects. In my compiler, I use an array of AST nodes and I use a type AstId to index that array which is just a wrapper to an integer. The problem with this is that you have to use it like this every time: `nodes[node_id.id]`. Which defeats the purpose of type safety if you can just access the internal integer. Ada has something like this [1] and I think it is a very nice idea.

7.3 Unsafe

I would like to delve into disallowing unsafe code (like using raw pointers) everywhere unless you mark it with some keyword like "unchecked". This is inspired by Rust, but what I don't like about it is that you might need to spam it everywhere if you are doing a lot of unsafe programming (for example, to interface with C code). Instead, I would make it on a module basis or file basis so that it is less verbose, but it is still clear when code is unchecked or not. Finally, I would like to see if there could be some way to make safe automatic freeing of memory like Rust does it, but allowing concepts like arenas and memory pools to exist.

7.4 Type Casts

Something I might want to change are type casts. I would like to have more specialized cast functions instead of the simple `as` operator that can be used in a lot of different ways. This would be similar to the already existing `'zero_extend` function. The possible implementation could add new functions like "int_cast", "float_cast", "int_to_float", etc. These would provide more errors at compile time, especially when a programmer decides to change the return type a function and then, suddenly, a type cast that was somewhere else has a new meaning.

7.5 Warnings

Compiler warnings are also on my list of things to add. They can be very helpful in showing the programming some properties about a program. Some types of warnings could be:

- Showing which names have been defined, but they aren't used anywhere.
- Revealing dead code.
- Telling the programmer when a variable is only modified, but never read.
- Warning about usage of deprecated features when the language starts evolving.

7.6 Sum types

Sum types or tagged unions are types that can hold the value of one of several variants determined at compile time. These are very helpful in representing entities that can have different shapes like tree nodes. Other use cases include types that can contain a value or nothing and types that can contain a value or an error. These are usually known as optional and result types. They are very useful when a function can fail and no value or an error must be returned instead. In C, the convention is to return an error code and return the result through a pointer. The problem with this approach is that, since both values can be used independently, it is not possible to guarantee that the programmer doesn't access the result when an error has occurred. Tying both the information about whether a function is successful and its result value leads to better and more readable code.

7.7 Generics

One thing I couldn't finish was generic structs which would allow creating things like generic list types and other data structures. Generics are one of the hardest problems I had to solve when designing Jelly. I came up with a lot of different solutions. Eventually, due to how the compiler was already structured and my goal for Jelly to be a simple language similar to C, I decided to avoid monomorphization fully even though I wanted it at some point. I might still add it in the future when I have more time because it makes generic programming easier and safer for the developer.

7.8 Final Conclusion

In conclusion, I have a lot of ideas and things that I would change, and I'm excited to try them and see which ones work and which ones don't. I think I have been successful on achieving most of my objectives. This project has been very fun to make, and I hope I continue to be passionate about this in the foreseeable future.

References

[1] AdaCore: *Introduction to Ada/Arrays*. – https://learn.adacore.com/courses/intro-to-ada/chapters/arrays.html

[2] Baeldung: *Type Erasure in Java Explained*. – https://www.baeldung.com/java-type-erasure

[3] Bill, Ginger: *Overview*. – http://odin-lang.org/docs/overview/

[4] Consortium, Unicode: *Unicode*. – https://home.unicode.org/

[5] cppreference: *Value categories*. – https://en.cppreference.com/w/c/language/value_category

[6] Example, Rust B.: *Literals and operators*. – https://doc.rust-lang.org/rust-by-example/primitives/literals.html

[7] Fleury, Ryan: *Untangling Lifetimes: The Arena Allocator*. – https://www.rfleury.com/p/untangling-lifetimes-the-arena-allocator

[8] Go, A T. of: *Switch with no condition*. – https://go.dev/tour/flowcontrol/11

[9] Kelley, Andrew: *Practical DOD*. – https://vimeo.com/649009599

[10] LLVM: *LLVM Language Reference Manual*. – https://llvm.org/docs/LangRef.html

[11] Morlan, Austin: *A Simple Entity Component System (ECS) [C++]*. – https://austinmorlan.com/posts/entity_component_system/

[12] Ortiga, Laia: *jellyc*. – https://github.com/Laia-Ortiga/jellyc

[13] OWASP: *Doubly freeing memory*. – https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory

[14] Ramos, Leodanis P.: *BNF Notation: Dive Deeper Into Python's Grammar*. – https://realpython.com/python-bnf-notation/

[15] Sola, Phil: *What is code-bloat and how to avoid it*. – https://cariadmarketing.com/insights/what-is-code-bloat-and-how-to-avoid-it/

[16] Stackscale: *Most popular programming languages in 2023*. – https://www.stackscale.com/blog/most-popular-programming-languages/

[17] Technologies, Unity: *Unity*. – https://unity.com/

[18] Wikipedia: *Data-flow analysis*. – https://en.wikipedia.org/wiki/Data-flow_analysis

[19] Wikipedia: *IEEE 754*. – https://en.wikipedia.org/wiki/IEEE_754

[20] Wikipedia: *Substructural type system.* – https://en.wikipedia.org/wiki/Substructural_type_system

[21] Wikipedia: *Turing completeness.* – https://en.wikipedia.org/wiki/Turing_completeness

Appendix A:
Output from Jelly Compiler

AST

This is one of the ASTs printed by the Jelly compiler when compiling the Fibonacci test file provided in the GitHub repository. The compiler was executed with the following command:

```
> jellyc -print-debug test/fibonacci.jel lib/std.jel lib/libc.jel
```

```
Ast(test/fibonacci.jel) {
    Import
    Function(
        TypeParameters(
        )
        Parameters(
            Param(
                n
                Id(i64)
            )
        )
        Id(i64)
        Block(
            If(
                <(
                    Id(n)
                    Int(0)
                )
                Block(
                    Return(
                        Int(0)
                    )
                )
                Null
            )
            Switch(
                Id(n)
                Case(
                    Int(0)
                    Int(0)
                )
                Case(
                    Int(1)
                    Int(1)
                )
                Case(
                    Null
```

```
                    Add(
                        Call(
                            Id(fibonacci)
                            Sub(
                                Id(n)
                                Int(1)
                            )
                        )
                        Call(
                            Id(fibonacci)
                            Sub(
                                Id(n)
                                Int(2)
                            )
                        )
                    )
                )
            )
        )
    )
    Function(
        TypeParameters(
        )
        Parameters(
        )
        Null
        Block(
            Let(
                f17
                Call(
                    Id(fibonacci)
                    Int(17)
                )
            )
            Call(
                Access(
                    Id(std)
                )
                Id(f17)
            )
        )
    )
}
```

Fibonacci's function TIR

This is the TIR printed by the Jelly compiler for the "fibonacci" function compiling the Fibonacci test. The names of the variables are erased, but functions have a mangled name. The types of each expression are printed after a colon. This information can be

very helpful while debugging the compiler.

```
Tir(file0_fibonacci) {
    if(
        lt(
            variable_0: i64
            0
        ): bool
        block(
            return(
                0
            )
        )
        block(
        )
    )
    return(
        switch(
            variable_0: i64
            case(
                0
                0
            )
            case(
                1
                1
            )
            case(
                else
                add(
                    call(
                        file0_fibonacci
                        sub(
                            variable_0: i64
                            1
                        ): i64
                    ): i64
                    call(
                        file0_fibonacci
                        sub(
                            variable_0: i64
                            2
                        ): i64
                    ): i64
                ): i64
            )
        ): i64
    )
```

```
}
```

Hello World

This is the C code generated by the compiler when compiling the hello world test mentioned in section 6.1. The first two non-empty lines are always added by the compiler. These are necessary for representing all Jelly types.

```
#include <stdint.h>

struct Slice { int64_t _0; char *_1; };

int32_t putchar(int32_t t0);
static void file0_print_str(struct Slice t0);
int main(void);
static void file0_print_str(struct Slice t0) {
    int64_t t1;
    t1 = 0;
    goto L1;
L1:
    ;
    int64_t *t5 = &t0._0;
    unsigned char t6 = t1 < (*t5);
    if (!t6) goto L4;
L2:
    ;
    char *t9 = &((char *)t0._1)[t1];
    int32_t t10 = (int32_t ) (*t9) & 0xFF;
    int32_t t11 = putchar(t10);
    goto L3;
L3:
    ;
    int64_t t14 = t1 + 1;
    t1 = t14;
    goto L1;
L4:
    ;
    return;
}
int main(void) {
    char (*t3)[14] = (char (*)[14]) "Hello, World!\x0A";
    struct Slice t4 = {14, (char *) t3};
    file0_print_str(t4);
    return 0;
}
```

Appendix B: Output from Test Programs

Basic Lexer

This is the output of the basic lexer written in Jelly after execution mentioned in section 6.2. It prints all the tokens for the Jelly file itself. Only a subset of the tokens are recognized for simplicity.

```
Tokens[
  `module`: id
  `main`: id
  `import`: id
  `std`: id
  `function`: function
  `fibonacci`: id
  `(`: (
  `n`: id
  `i64`: id
  `)`: )
  `->`: ->
  `i64`: id
  `{`: {
  `if`: id
  `n`: id
  `<`: <
  `0`: int
  `{`: {
  `return`: return
  `0`: int
  `}`: }
  `switch`: id
  `n`: id
  `{`: {
  `0`: int
  `->`: ->
  `0`: int
  `,`: ,
  `1`: int
  `->`: ->
  `1`: int
  `,`: ,
  `else`: id
  `->`: ->
  `fibonacci`: id
  `(`: (
  `n`: id
  `-`: -
  `1`: int
```

```
  `)`: )
  `+`: +
  `fibonacci`: id
  `(`: (
  `n`: id
  `-`: -
  `2`: int
  `)`: )
  `,`: ,
  `}`: }
  `}`: }
  `function`: function
  `main`: id
  `(`: (
  `)`: )
  `{`: {
  `let`: let
  `f17`: id
  `=`: =
  `fibonacci`: id
  `(`: (
  `17`: int
  `)`: )
  `std`: id
  `.`: .
  `print_int`: id
  `(`: (
  `f17`: id
  `)`: )
  `}`: }
]
```