Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Department of Computer Science

Software Engineering Orientation

Bachelor thesis

2024

Functional language compiler to WebAssembly

Technical documentation

Noah Godel

Supervisors: Jacques Supcik
Serge Ayer

Experts: Baptiste Wicht
Valentin Bourqui

Fribourg, 5 July 2024

Version 0.3

Hes·so

Table of versions

Version	Date	Modifications
0.3	5.7.2024	Implementation section
0.2	19.6.2024	Analysis section
0.1	12.6.2024	First version - introduction and chapter titles

Contents

Section 1
Introduction

This report documents the development of a functional language compiler to WebAssembly (Wasm). The project was conducted as part of the Bachelor's thesis at the Haute école d'ingénierie et d'architecture de Fribourg (HEIA-FR). The goal of the project was to design and implement a compiler for a functional language that targets Wasm. The project was supervised by Dr. Jacques Supcik and Dr. Serge Ayer, with Dr. Baptiste Wicht and Mr. Valentin Bourqui as experts. For further details, please refer to the requirement specification document [6]. The project repository can be found at the following URL.

https://gitlab.forge.hefr.ch/noah.godel/24-tb-wasm-compiler

1.1. Context

The functional programming paradigm offers advantages for certain types of problems like data transformations, parallel processing, and mathematical computations. However, it has limitations, and many use cases are better suited for imperative or object-oriented programming. Ideally, developers should be able to leverage the strengths of different paradigms within the same codebase. Unfortunately, integrating functional languages into existing codebases written in other languages can be challenging.

Wasm is a bytecode format designed to execute code at near-native speeds across different environments like web browsers and Wasm runtimes. By developing a compiler for a functional language, or in the context of this project, a subset of an existing one, that compiles to Wasm, we can combine functional programming benefits with Wasm's performance and portability. This enables seamless integration of high-performance functional code into codebases of different languages, allowing developers to utilize functional programming strengths for specific components.

The project aims to demonstrate embedding the new functional language compiled to Wasm into existing codebases, showcasing interoperability and the potential for combining paradigms within the same project. For more details on the context, refer to the requirements specification document [6].

Figure 1 illustrates the concept of embedding a Wasm module into a codebase.

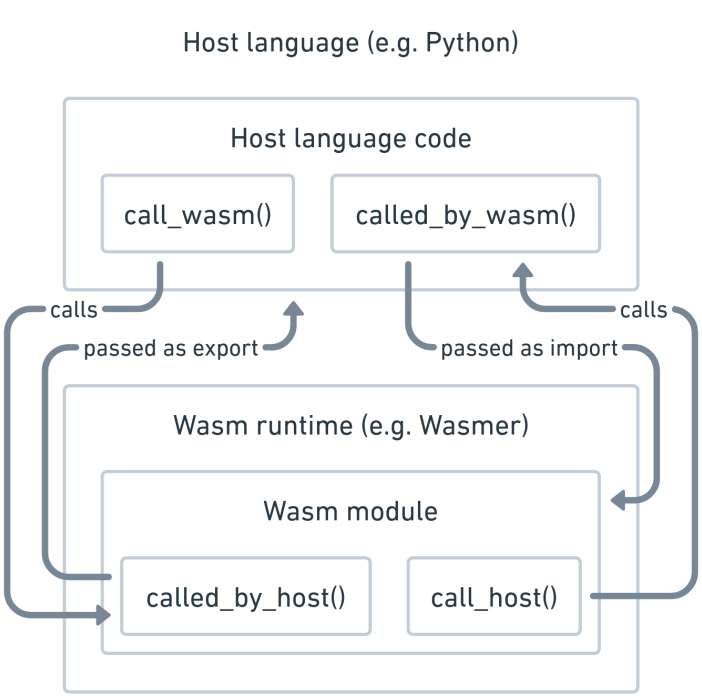


Figure 1: Illustration of embedding a Wasm module into a codebase.

1.2. Objectives

Upon completion of the project, the following key objectives will be achieved:

- **Functional Programming Language Specification**: Define a functional programming language that is a subset of an existing language, tailored for efficient Wasm compilation and seamless embedding into other codebases. A subset of the standard library supporting the language features will also be defined.
- **Functioning Compiler**: Develop a fully operational compiler capable of translating the defined functional language into efficient Wasm bytecode for high-performance execution across environments. The compiled code should interoperate with other languages it is embedded into.
- **Language Documentation**: Provide documentation detailing the usage and development of the new language, including examples, references, and demonstrations of embedding into different language codebases to facilitate learning and adoption.

While not production-ready after 7 weeks, the project will serve as a proof of concept and foundation for potential further development by delivering the defined language, compiler, documentation, and embedding examples. Refer to the requirements specification for more details on the objectives [6].

1.3. Document structure

This document is structured as follows.

- *Introduction*: Provides an overview of the project and its context.
- *Analysis*: Describes the context, objectives, and requirements of the project.
- *Design*: Details the design of the functional language, compiler, and standard library.
- *Implementation*: Explains the implementation of the compiler and standard library.
- *Evaluation*: Discusses the evaluation of the compiler and standard library.
- *Conclusion*: Summarizes the project, highlights achievements, and outlines future work.

Section 2
Analysis

This section presents the constraints, the exploration of different technologies and features and the technological choices.

2.1. UNESCO and Sustainable Development Goals

The project aligns with the United Nations Educational, Scientific and Cultural Organization (UNESCO) and the Sustainable Development Goals (SDGs) by promoting education, innovation, and sustainable development. By developing a functional language compiler to WebAssembly, the project contributes to the advancement of technology, education, and the digital economy. The project's outcomes can be used to enhance programming education, foster innovation, and support sustainable development initiatives. [9]

The following SDGs are relevant to the project.

2.1.1. Goal 4: Quality Education

The project aims to improve the quality of education by providing a new tool for teaching and learning functional programming concepts. By developing a functional language compiler to WebAssembly, the project enables educators to introduce students to functional programming paradigms and demonstrate the benefits of functional programming in real-world applications. The project's outcomes can be used to enhance programming education and prepare students for careers in software development and technology.

In addition, the project enhances educational opportunities in the field of compiler development and programming language design. By providing a hands-on experience in developing a compiler for a functional language, the project equips students with practical skills and knowledge that are valuable in the software industry.

2.1.2. Goal 8: Decent Work and Economic Growth

The project contributes to decent work and economic growth by developing a functional language compiler to WebAssembly. By enabling developers to compile functional code to a portable and efficient bytecode format, the project supports the development of new applications and services that can drive economic growth. The project's outcomes can be used to create job opportunities in the software development industry, foster entrepreneurship, and promote innovation in the digital economy.

2.1.3. Goal 17: Partnerships for the Goals

The project promotes partnerships for the goals by collaborating with academic institutions, industry experts, and stakeholders in the software development community. By working with supervisors, experts, and external partners, the project leverages diverse expertise and resources to achieve its objectives. The project's outcomes can be shared with the wider community to promote knowledge sharing, collaboration, and the advancement of technology and education.

2.2. Choice of language for the subset

In this project, the choice of the language subset is crucial. The language should be expressive enough to demonstrate the functional programming paradigm's benefits while being simple enough to implement within the project's timeframe. The language should also be a subset of an existing language to so that the task of having to specify the language's syntax and semantics is simplified.

The following languages were considered for the project.

2.2.1. OCaml

OCaml is a general-purpose, multi-paradigm programming language that extends the ML language with object-oriented features. It has a strong type system, automatic memory management, and supports functional, imperative, and object-oriented programming styles. OCaml is widely used in academia and industry, particularly in areas such as theorem proving, compiler development, and systems programming.

Advantages:
- Strong static type system can facilitate efficient compilation and optimization.
- Since OCaml is often used in compiler development, its compiler is well documented and can serve as a reference for the project.
- Already supports Wasm compilation, which can serve as a reference for the project.

Disadvantages:
- Multi-paradigm nature and complex syntax may complicate the task of creating a purely functional subset.
- OCaml is a vast language with many features, which may make it challenging to define a subset that is both expressive and manageable, given the author's limited experience with the language.

2.2.2. F#

F# is a multi-paradigm programming language that encompasses functional, imperative, and object-oriented styles. It is a part of the .NET ecosystem and can be seamlessly integrated with other .NET languages such as C# and Visual Basic. F# is particularly well-suited for data-oriented programming tasks, parallel programming, and domain-specific language development.

Advantages:
- Seamless integration with the .NET ecosystem and interoperability with other .NET languages.
- Already supports Wasm compilation through Bolero (which uses Blazor), providing a reference for the project.

Disadvantages:
- Limited adoption and smaller community compared to more popular languages like C#.
- Multi-paradigm nature may complicate the task of creating a purely functional subset.
- The author is not familiar with F# and would need to learn the language from scratch.

2.2.3. The Lisp languages (Common Lisp, Clojure)

Lisp (List Processing) is a family of programming languages with a long history and a distinctive syntax based on parentheses and lists. Common Lisp and Clojure are two prominent dialects of Lisp.

2.2.3.1. Common Lisp

Common Lisp is a multi-paradigm language that supports functional, imperative, and object-oriented programming styles. It is used in artificial intelligence, computer algebra, and symbolic computation applications.

Advantages:
- Very simple and consistent syntax, which makes it easy to define a subset.
- Established language with a rich ecosystem of libraries and tools.

Disadvantages:
- No built-in support for Wasm compilation, which means there is no reference implementation for the project.
- The author is not familiar with Common Lisp and would need to learn the language from scratch.

2.2.3.2. Clojure

Clojure is a modern Lisp dialect that runs on the Java Virtual Machine (JVM) and emphasizes immutable data structures and functional programming. It is designed for concurrent and parallel programming, and is often used in web development and data analysis applications.

Advantages:
- Functional programming paradigm aligned with the project's goals.
- Runs on the JVM, which has existing tooling and libraries for Wasm compilation.

Disadvantages:
- No *direct* support for Wasm compilation, which means there is no reference implementation for the project.
- The author has limited experience with Clojure and defining a subset may be challenging.

2.2.4. The BEAM languages (Erlang, Elixir)

The Beam languages, Elixir and Erlang, are functional programming languages that run on the Erlang Virtual Machine (BEAM). They are designed for building scalable, fault-tolerant, and distributed systems.

2.2.4.1. Erlang

Erlang is a general-purpose, concurrent programming language with built-in support for distributed computing. It is widely used in telecommunications, banking, and e-commerce systems that require high availability and fault tolerance.

Advantages:
- Functional programming paradigm aligned with the project's goals.
- There are alternative compilers for BEAM languages that target Wasm, which can serve as a reference for the project.

Disadvantages:
- The author has limited experience with Erlang, which may complicate the task of defining a subset.

2.2.4.2. Elixir

Elixir is a more recent functional language that builds upon the strengths of Erlang's VM and ecosystem. It aims to provide a more modern and productive syntax while maintaining the robustness and concurrency features of Erlang.

Advantages:
- Functional programming paradigm aligned with the project's goals.
- Elixir has a more modern syntax and tooling compared to Erlang.
- As with Erlang, there are alternative compilers for BEAM languages that target Wasm, which can serve as a reference for the project.

Disadvantages:
- The author has limited experience with Elixir, which may complicate the task of defining a subset.

2.2.5. Haskell

Haskell is a purely functional programming language with a strong static type system and lazy evaluation. It is known for its elegance, conciseness, and expressive type system, which facilitates safe and modular code development.

Haskell's functional paradigm and powerful abstraction mechanisms make it well-suited for a wide range of applications, including data analysis, concurrent and parallel programming, domain-specific language development, and cryptography.

Advantages:
- Purely functional programming paradigm, aligning perfectly with the project's goals.
- Advanced type system can facilitate efficient compilation and optimization.
- Existing tools and libraries for Wasm compilation, such as the Glasgow Haskell Compiler (GHC) and its support for various intermediate representations.
- Author's familiarity with the language can facilitate implementation and understanding of language intricacies.

Disadvantages:
- Lazy evaluation may introduce complexities in the compilation process and performance considerations.
- Haskell's advanced type system may require additional effort to define a subset that is both expressive and manageable within the project's timeframe.

Considering the project's goals of creating a functional language subset tailored for efficient compilation to WebAssembly (Wasm), Haskell stands out as the most suitable choice. Its purely functional nature, advanced type system, existing tooling for Wasm compilation, and the author's familiarity with the language make it an ideal foundation for this project. Since the project has a limited timeframe of 7 weeks, the choice of a language subset that the author is most comfortable with, is crucial.

Since Haskell is a purely functional language, defining a subset that is both expressive and manageable within the project's timeframe should be feasible. Additionally, the motivation behind the project is to be able to leverage the strengths of functional programming within existing codebases, and Haskell's functional paradigm aligns perfectly with this goal. Since November 2022, GHC has supported the compilation of Haskell code to WebAssembly. This means that the project can use GHC as a reference for the compilation process.

While other languages like OCaml, F#, Lisp dialects, and the Beam languages have their strengths, their multi-paradigm nature or limited direct support for Wasm compilation could introduce additional complexities or hinder the efficient realization of the project's objectives.

2.3. The functional paradigm

The functional programming paradigm is based on the concept of functions as first-class citizens, immutability, and the absence of side effects. Functional programming languages treat computation as the evaluation of mathematical functions and emphasize declarative programming styles. The functional paradigm offers several advantages, including:

- **Modularity**: Functions are modular and composable, enabling code reuse and maintainability.
- **Conciseness**: Functional languages often require less code to express complex operations compared to imperative languages.
- **Safety**: Immutability and strong type systems reduce the likelihood of runtime errors and make code easier to reason about.
- **Parallelism**: Functional programming encourages pure functions, which are inherently thread-safe and can be executed in parallel without side effects.
- **Higher-order functions**: Functions can take other functions as arguments or return functions as results, enabling powerful abstractions and expressive code.
- **Declarative style**: Functional programming focuses on what should be computed rather than how, leading to more readable and maintainable code.

For more details on the functional programming paradigm or more specifically on Haskell, refer to official Haskell wiki:

https://wiki.haskell.org/Introduction

In the following sections, some key features of the functional programming paradigm that are relevant to the project will be discussed.

2.3.1. Partial application and currying

Partial application and currying are common techniques in functional programming that involve creating new functions by applying a function to some of its arguments. Partial application involves supplying fewer arguments than the function expects, while currying involves transforming a function that takes multiple arguments into a series of functions that each take a single argument.

In Haskell, functions are curried by default, which means that all functions take exactly one argument and return a new function that takes the next argument. This allows for partial application and function composition, enabling powerful abstractions and expressive code. As we can see, in Listing 1, the type signature of the `add` function indicates that it takes an `Int` and returns a function that takes another `Int` and returns an `Int`. A good real-world example of currying is the `map` function in Haskell, which takes a function and a list and applies the function to each element of the list. If we want to increment each element of a list by 1, we can use partial application to pass a function that increments each element (see Listing 1).

As a side note, function application in Haskell is done by separating the function name from its arguments with whitespace, for example, `add 1 2`. This is different from most other programming languages, where parentheses are used for function application, for example, `add(1, 2)`.

```haskell
add :: Int -> Int -> Int
add x y = x + y

-- map :: (a -> b) -> [a] -> [b]
-- add 1 :: Int -> Int
incrementList :: [Int] -> [Int]
-- the list argument can be omitted because of partial application ->
-- the function returns a new function that corresponds to the function
signature
incrementList = map (add 1)
```

Listing 1: Example of currying in Haskell.

Haskell has some helper functions that can be used to create curried functions from uncurried functions and vice versa. For example, the `curry` function takes an uncurried function and returns a curried function, while the `uncurry` function takes a curried function and returns an uncurried function. Listing 2 shows an example of using the `curry` and `uncurry` functions in Haskell. Uncurried functions in Haskell are functions that take a tuple as an argument. Listing 2 shows an example of using the `curry` and `uncurry` functions.

```haskell
-- curry :: ((a, b) -> c) -> a -> b -> c
-- uncurry :: (a -> b -> c) -> (a, b) -> c

addUncurried :: (Int, Int) -> Int
-- pattern matching on a tuple
addUncurried (x, y) = x + y

addCurried :: Int -> Int -> Int
addCurried = curry addUncurried

addUncurried' :: (Int, Int) -> Int
addUncurried' = uncurry addCurried
```

Listing 2: Example of currying helpers in Haskell.

2.3.2. Algebraic data types

Algebraic data types are a fundamental concept in functional programming that allows developers to define complex data structures using simple building blocks. There are two main types of algebraic data types: sum types and product types. Sum types represent a choice between different alternatives, while product types represent a combination of different values.

In Haskell, algebraic data types are defined using the `data` keyword. For example, the `Boolean` type is a sum type that represents a value that may or may not be present. Listing 3 shows an example of the `Boolean` type definition in Haskell, which consists of two alternatives: `True` and `False`.

The `List` type is another example of an algebraic data type that represents a list of values. It is defined as a sum type with two alternatives: an empty list `Nil` and a cons cell `Cons a List`. This definition allows for recursive data structures, such as linked lists, trees, and graphs. This data structure also shows that, in Haskell, sum types and product types can be combined to create more complex data structures, the `Cons` constructor is a product type that combines a value `a` with another list `List a`. The `a` type variable is a type parameter that allows the `List` type to be polymorphic and store values of any type. Listing 3 shows an example of the `List` type definition in Haskell.

```haskell
data Boolean = True | False

data List a = Nil | Cons a (List a)
```

Listing 3: Example of an algebraic data type in Haskell.

2.3.3. Pattern matching

Pattern matching is a powerful feature in functional programming that allows developers to destructure data structures and extract values based on their shape. It is often used in conjunction with algebraic data types to define functions that operate on different alternatives of a sum type.

In Haskell, pattern matching is achieved using the `case` expression or function definitions with pattern matching clauses. The `Boolean` type definition from Listing 3 can be used to define a function that negates a boolean value. The function `negateBoolean` pattern matches on the `Boolean` type and returns the opposite value. Listing 4 shows an example of the `negateBoolean` function in Haskell.

The `List` type definition from Listing 3 can be used to define a function that calculates the length of a list. The `length` function pattern matches on the `List` type and recursively calculates the length of the list. Listing 4 shows an example of the `length` function in Haskell. As we can see, pattern matching allows for capturing values from different alternatives of a sum type and defining functions that operate on these values like the `xs` in the `Cons` alternative of the `List` type (the `_` is a wildcard pattern that matches any values without binding them).

```haskell
negateBoolean :: Boolean -> Boolean
negateBoolean b = case b of
  True -> False
  False -> True

-- or using pattern matching in function definitions
negateBoolean' :: Boolean -> Boolean
negateBoolean' True = False
negateBoolean' False = True

length :: List a -> Int
length Nil = 0
length (Cons _ xs) = 1 + length xs
```

Listing 4: Example of pattern matching in Haskell.

2.3.4. Parametric polymorphism

Parametric polymorphism is a feature of functional programming languages that allows developers to write generic functions that operate on values of any type. It is achieved by introducing type variables that represent unknown types and can be instantiated with concrete types when the function is used. All type variables are universally quantified, meaning that they can represent any type. When defining a function with type variables, the function needs to be correct for all possible types that the type variables can represent (because the type variables are universally quantified). So a function that has 2 type variables but these type variables are always the same type, the function

won't compile. The `foo` function in Listing 5 is an example of a parametrically polymorphic function in Haskell that doesn't compile.

In Haskell, parametric polymorphism is achieved using type variables in function signatures. For example, the `id` function is a parametrically polymorphic function that takes a value of any type and returns the same value. The `id` function is defined as `id :: a -> a`, where `a` is a type variable that can represent any type. The `id` function is a common example of a parametrically polymorphic function that demonstrates the power of type variables in functional programming. Listing 5 shows an example of the `id` function in Haskell.

Listing 5 shows examples of parametric polymorphism in Haskell.

```haskell
-- doesn't compile because the type variables are different
foo :: a -> b -> a
foo x y = x

id :: a -> a
id x = x

const :: a -> b -> a
const x _ = x
```

Listing 5: Example of parametric polymorphism in Haskell.

2.3.5. Higher-order functions

Higher-order functions are functions that take other functions as arguments or return functions as results. They are a fundamental concept in functional programming that enables powerful abstractions and expressive code. Higher-order functions allow developers to write generic functions that can be customized with different behaviors by passing functions as arguments.

In Haskell, higher-order functions are used extensively to create composable and reusable code. For example, the `map` (used in Listing 1) and `filter` functions are higher-order functions that take a function and a list and apply the function to each element of the list or filter the list based on the function's result. The `map` function is a common example of a higher-order function that demonstrates the power of functional programming. Listing 6 shows an example of the `map` function in Haskell. The first argument of the `map` function is a function that takes an `a` and returns a `b`, and the second argument is a list of `a` values. The `map` function applies the function to each element of the list and returns a list of `b` values.

```haskell
map :: (a -> b) -> List a -> List b
map _ Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

Listing 6: Example of a higher-order function in Haskell.

2.3.6. Lazy evaluation

Lazy evaluation is a feature of functional programming languages that delays the evaluation of expressions until their results are actually needed. This can lead to more efficient use of resources and enable the creation of infinite data structures. In Haskell, all expressions are lazily evaluated by default, which means that functions only evaluate their arguments when the arguments are needed to produce a result.

The only way to force the evaluation of an expression in Haskell is through pattern matching or through the evaluation of the `main` function (there are other ways but they are not relevant for this project). This can lead to more efficient code execution and resource usage, as only the necessary parts of the program are evaluated when needed. However, lazy evaluation can also introduce complexities in reasoning about the order of evaluation and performance considerations.

Listing 7 shows an example of lazy evaluation in Haskell. The `repeat` function creates an infinite list of the same value by recursively consing the value to the rest of the list. The `take` function takes a number `n` and a list and returns the first `n` elements of the list. When we call `take 5 (repeat 1)`, Haskell only evaluates the first 5 elements of the infinite list, demonstrating the power of lazy evaluation.

```haskell
  repeat :: a -> List a
  repeat x = Cons x (repeat x)

  take :: Int -> List a -> List a
  take 0 _ = Nil
take n Nil = Nil
  take n (Cons x xs) = Cons x (take (n - 1) xs)

  list :: List Int
  list = take 5 (repeat 1)
  -- list = Cons 1 (Cons 1 (Cons 1 (Cons 1 (Cons 1 Nil))))
```

Listing 7: Example of lazy evaluation in Haskell.

Internally, the GHC (Glasgow Haskell Compiler) uses a technique called graph reduction to evaluate expressions lazily. This technique represents expressions as graphs of nodes and edges, where each node represents a value and each edge represents a reference to another node. When an expression is evaluated, the graph is traversed and nodes are evaluated as needed. This allows for sharing of common subexpressions and efficient memory usage. So as an added bonus, no expression is evaluated more than once.

Since this technique is not trivial to implement by hand in Wasm, a more naive approach will be used in the project. This approach will treat every function application as a new closure and will not share common subexpressions. This will lead to less efficient memory usage and potentially slower execution times, but it will simplify the implementation of the compiler. These closures will be stored evaluated in only when needed.

2.3.7. Added challenges of implementing a functional language

Implementing a functional language subset that targets Wasm presents several challenges that need to be addressed during the design and implementation phases of the project. Some of the key challenges compared to implementing an imperative language include:

- **Higher-order functions**: Supporting higher-order functions, which are functions that take other functions as arguments or return functions as results, requires careful handling of function closures and environments.
- **Lazy evaluation**: Implementing lazy evaluation, where expressions are only evaluated when their results are needed, can introduce complexities in the compilation process and runtime behavior.
- **Garbage collection**: Managing memory and resource cleanup in a functional language that supports immutable data structures and higher-order functions requires an efficient garbage collection mechanism.
- **Tail call optimization**: Ensuring that tail-recursive functions are optimized to avoid stack overflows and improve performance is essential for functional programming languages since recursion is a common pattern.
- **Parametric polymorphism**: Supporting parametric polymorphism, which allows developers to write generic functions that operate on values of any type, requires careful handling of type variables and type inference. This language feature is very important for the project because without it, the language would be very limited in its expressiveness.

2.4. Wasm extensions

Wasm is a stack-based virtual machine designed to execute code at near-native speeds across different environments. It is used in web browsers, server-side applications, and other environments where performance and portability are essential. Wasm bytecode is generated from high-level languages and can be executed on any platform that supports the Wasm runtime.

In its current form, Wasm provides a set of core features that are sufficient for executing code efficiently. However, there are several extensions and proposals that aim to enhance Wasm's capabilities and make it more versatile for different use cases. The following Wasm extensions were considered for the project.

2.4.1. Component model

One of the main limitations of Wasm (especially in the context of embedding it into existing codebases) is the small number of types it supports (essentially integers and floats). The component model proposal [3] aims to address this limitation by introducing a new language that allows developers to define custom types and interfaces and an ABI for interacting with Wasm modules. This extension could be beneficial for the

project as it would enable more seamless integration of the functional language subset into other codebases.

Using this new language, developers can define interfaces using the .wit file format and implement these interfaces in Wasm modules. To use the generated component, bindings need to be generated in the host codebase that match the interface, its types and functions that are defined in the .wit file. This allows the host codebase to interact with the Wasm component using the defined interface.

The problem with this extension is that it is still in the proposal stage and Wasm components can only be run in a few languages (Rust, JavaScript and partially Python) using the Wasmtime runtime. This could limit the project's ability to demonstrate embedding the functional language into different codebases.

Listing 8 shows an example of an interface using the .wit file format, and Listing 9 shows an example of the implementation of the interface in Wasm.

```
package example:add;

world root {
  export add: func(x: s32, y: s32) -> s32;
}
```

Listing 8: Example of a Interface using the .wit file format.

```
(module
  (func (export "example:add/root#add") (param i32) (param i32) (result i32)
    local.get 0
    local.get 1
    i32.add
  )
)
```

Listing 9: Example of the implementation of the Interface.

2.4.2. Reference types and function references

The reference types proposal [14] aims to allow for reference types (function references or external references) to be used as values. This extension could be beneficial for the project since this extension simplifies the implementation of functions as first-class citizens.

In core Wasm, function references are only used inside function tables (necessary for indirect calls). The reference types proposal extends this to allow function references to be used as values in the functions themselves and not only as indices into the function table. It also introduces new instructions to interact with the function table to dynamically add and remove functions from it.

The proposal is still in the proposal stage, but it is supported by the Wasmer, Wasmtime and WasmEdge runtimes and practically everywhere else. This means that the project could leverage these runtimes to demonstrate the embedding of the functional language into different codebases.

The function references proposal [15] is an extension of the reference types proposal that simply enables function references to be called directly. It also makes a distinction between nullable and non-nullable function references. This extension could be beneficial for the project as it simplifies the implementation of functions as first-class citizens even further.

The function references proposal is still in the proposal stage and is less supported than the reference types proposal. It is supported by the Wasmtime and WasmEdge runtimes and in the browser.

Listing 10 shows an example of reference types and function references in Wasm.

```
(module
    (table 1 funcref)
    (type $type0 (func (result i32)))
    (type $type1 (func (param i32) (result i32)))

    (func $foo (result i32) i32.const 42)

    ;; This function calls the function referenced in the table with
    ;; the index returned by "add_func_to_tabel"
    (func $ref_types_example (result i32)
        call $add_func_to_tabel
        call_indirect (type $type0)
    )

    ;; This function adds the function reference to the table and
    ;; returns the index
    (func $add_func_to_tabel (result i32)
        ref.func $foo
        i32.const 0
        table.set 0
        i32.const 0
    )

    ;; this function takes a int and returns it
    (func $bar (param i32) (result i32)
        local.get 0
    )

    ;; This function takes a int and calls "call_passed_func" with
    ;; it and the function reference
    (func $func_types_example (param i32) (result i32)
        ref.func $bar
        local.get 0
        call $call_passed_func
    )

    ;; This function takes a int and a function reference and calls
    ;; the function reference with the int
    (func $call_passed_func (param i32) (param (ref $t1)) (result i32)
        local.get 1
        local.get 0
        call_ref $type1
    )
)
```

Listing 10: Example of reference types and function references in Wasm.

2.4.3. Garbage collection

The garbage collection proposal [16] aims to introduce garbage collection support in Wasm. This extension could be beneficial for the project as it would simplify memory management and resource cleanup in the functional language subset. It is a quite complex proposal and is still in the proposal stage. Since the support for garbage collection in Wasm is as of now limited to the browser and node.js, this could limit the project's ability to demonstrate embedding the functional language into different codebases.

The proposal bases itself on the reference types and function references proposals and introduces new types (so-called heap types) like structs, arrays, and references to these types. It also introduces new instructions to allocate and modify these types on the heap.

2.4.4. Tail call optimization

The tail call optimization proposal [17] aims to introduce tail call optimization support in Wasm. This extension could be beneficial for the project as it would optimize the performance of recursive functions in the functional language subset. The proposal is still in the proposal stage and is supported by the Wasmtime and WasmEdge runtimes and practically everywhere else.

Listing 11 shows an example of tail call optimization in Wasm.

```
(module
    (func $factorial (param $x i64) (result i64)
        (return (call $factorial_aux (local.get $x) (i64.const 1)))
    )

    (func $factorial_aux (param $x i64) (param $acc i64) (result i64)
        (if (i64.eqz (local.get $x))
            (then (return (local.get $acc)))
            (else
                (return
                    (call $factorial_aux
                        (i64.sub (local.get $x) (i64.const 1))
                        (i64.mul (local.get $x) (local.get $acc))
                    )
                )
            )
        )
        unreachable
    )

    (func $factorial_tail (param $x i64) (result i64)
        (return_call $factorial_tail_aux (local.get $x) (i64.const 1))
    )

    (func $factorial_tail_aux (param $x i64) (param $acc i64) (result i64)
        (if (i64.eqz (local.get $x))
            (then (return (local.get $acc)))
            (else
                (return_call $factorial_tail_aux
                    (i64.sub (local.get $x) (i64.const 1))
                    (i64.mul (local.get $x) (local.get $acc))
                )
            )
        )
        unreachable
    )
    (export "factorial" (func $factorial))
    (export "factorial_tail" (func $factorial_tail))
)
```

Listing 11: Example of tail call optimization in Wasm.

Listing 12 shows a performance comparison between a factorial function with and without tail call optimization.

```
factorial(20): 2432902008176640000 in 12.41µs
factorial_tail(20): 2432902008176640000 in 1.319µs
```

Listing 12: Example of tail call optimization performance comparison.

2.5. Embedding the Wasm module into a codebase

The embedding of the Wasm module into a codebase is a crucial aspect of the project. The Wasm module should be able to interact with the host codebase seamlessly.

Since Wasm is originally designed to run in web browsers, the embedding of Wasm modules into web applications is well supported. However, embedding Wasm modules into other codebases, such as server-side applications or desktop applications, can be more challenging. To be able to interact with the Wasm module, the host codebase needs a runtime that can load and execute the Wasm module (see Figure 1). The runtime should also provide mechanisms for passing data between the host codebase and the Wasm module.

The following technologies were considered for the project.

2.5.1. Wasmer

Wasmer s a standalone Wasm runtime that supports running Wasm modules outside the browser. It provides a set of APIs for loading and executing Wasm modules, as well as mechanisms for interacting with the host codebase. Wasmer also has a registry of Wasm modules that can be used to share and distribute Wasm modules. These modules can also be deployed one the cloud using Wasmer's cloud service.

However, Wasmer does supports the least amount of Wasm proposals out of the three runtimes (see Table 1), which could limit the project's ability to demonstrate seamless embedding the functional language into different codebases.

On the other hand, Wasmer provides a large set of SDKs for different programming languages (see Table 2 [12]). Compared to the other runtimes Wasmer has the most extensive support for different programming languages. Which is important for the project since the functional language subset should be able to be embedded into different codebases.

2.5.2. Wasmtime

As with Wasmer, Wasmtime is a standalone Wasm runtime that supports running Wasm modules outside the browser. It is developed by the Bytecode Alliance, a group of companies and individuals working on WebAssembly and related technologies. To see the full list supported Wasm proposals see Table 1 [13] and for the supported programming languages see Table 2.

Wasmtime is the only runtime that supports the component model proposal, which could be beneficial for greatly simplifying the embedding of the functional language into different codebases.

2.5.3. WasmEdge

As with the other runtimes, WasmEdge is a standalone Wasm runtime that supports running Wasm modules outside the browser. It is developed by the Second State, a company that provides a platform for building and deploying Wasm applications. WasmEdge supports almost all Wasm proposals (see Table 1 [10]). It achieves this by running JavaScript code in a sandboxed environment and can so support the browser's implementation of Wasm.

However, WasmEdge supportes fewer programming languages than Wasmer and Wasmtime (see Table 2 [11]). This could limit the project's ability to demonstrate embedding the functional language into different codebases.

2.5.4. Wasm proposal compatibility and language support

Table 1 shows a summary of the compatibility of the Wasm proposals with the different runtimes [18].

Proposal	**Wasmer**	**Wasmtime**	**WasmEdge**	**Browser**
Reference types	✓	✓	✓	✓
Function references	×	✓	✓	✓
Garbage collection	×	×	✓	✓
Tail call optimization	×	✓	✓	✓

Table 1: Summary of Wasm proposal compatibility with different runtimes.

Table 2 shows a summary of the language support of the different runtimes [18].

Language	Wasmer	Wasmtime	WasmEdge
Rust	✓	✓	✓
C/C++	✓	✓	✓
.NET (C#, F#, VB)	✓	✓	×
D	✓	×	×
Python	✓	✓	✓
JavaScript	✓	×	×
Go	✓	✓	✓
PHP	✓	×	×
Ruby	✓	✓	×
Java	✓	×	✓
R	✓	×	×
Postgres	✓	×	×
Swift	✓	×	×
Zig	✓	×	×
Dart	✓	×	×
Crystal	✓	×	×
Common Lisp	✓	×	×
Julia	✓	×	×
V	✓	×	×
OCaml	✓	×	×
Elixir	×	✓	×
Perl	×	✓	×

Table 2: Summary of language support of different runtimes.

2.6. Choice of compiler technology

The choice of compiler technology is essential for the project's success. The compiler should be able to translate the functional language subset into efficient Wasm bytecode. The following technologies were considered for the project.

2.6.1. LLVM

LLVM is a collection of modular and reusable compiler and toolchain technologies. It is widely used in industry and academia for developing compilers, static analysis tools, and runtime environments. LLVM provides a set of libraries and tools for building com-

pilers, including a compiler front-end (Clang), a compiler back-end (LLVM Core), and a set of optimization passes.

By making a compiler front-end that translates the functional language subset into LLVM intermediate representation (IR), the project could leverage LLVM's existing infrastructure for optimizing and generating efficient machine code. The LLVM IR can then be translated into Wasm bytecode using the Binaryen toolchain. Additionally targeting LLVM IR would allow the project to compile the functional language to other targets like x86, ARM, or RISC-V.

However, LLVM's complexity and the learning curve associated with it could make it challenging to implement within the project's timeframe. The project would also need to define a subset of the functional language that can be efficiently translated into LLVM IR.

Advantages:
- Efficient optimization and code generation capabilities.
- Support for multiple targets and architectures.
- Existing infrastructure for building compilers and toolchains.

Disadvantages:
- Complexity and learning curve associated with LLVM.
- Need to define a subset of the functional language that can be efficiently translated into LLVM IR.

2.6.2. Manual translation

Manual translation refers to the process of writing a custom compiler that directly translates the functional language subset into Wasm bytecode without using an intermediate representation like LLVM IR. This approach would involve defining a custom compiler architecture that parses the functional language syntax, performs semantic analysis, and generates Wasm bytecode.

While manual translation provides full control over the compilation process and allows for tailoring the compiler to the project's specific requirements, it can be time-consuming and error-prone. The project would need to implement lexing, parsing, type checking, and code generation from scratch.

Advantages:
- Full control over the compilation process (e.g., Wasm proposal compatibility, embedded runtime support, etc.).
- Tailoring the compiler to the project's specific requirements.

Disadvantages:
- Time-consuming and error-prone implementation.
- No existing infrastructure for optimization and code generation.

This approach was chosen for the project due to the limited timeframe and the need for a simple and manageable compiler architecture. The manual translation approach

allows for a more straightforward implementation of the compiler while focusing on the functional language subset's core features and efficient Wasm compilation. To circumvent the lack of optimization and code generation infrastructure, the project will use the "wasm-opt" tool of the Binaryen toolchain to optimize the generated Wasm bytecode.

2.7. Possible approaches to the compiler architecture

The project could use different approaches to the compiler architecture, depending on the choice of compiler technology.

The project could use the GHC Haskell compiler as a frontend or backend for the functional language subset. The GHC compiler provides a robust infrastructure for parsing, type checking, and optimizing Haskell code, which could be leveraged to translate the functional language subset into efficient Wasm bytecode. By using GHC as a backend, the project could focus on defining the functional language subset and leveraging GHC's existing infrastructure for compilation and optimization. If GHC is used as a frontend, the project would need to define a custom compiler backend that translates the GHC IR into Wasm bytecode.

However, using GHC as a backend or frontend for the project could introduce complexities and dependencies that may not be necessary for the project's scope. It is difficult to extract the necessary parts of GHC for the project and integrate them into the compiler architecture given the size and complexity of the GHC codebase.

Considering the limited timeframe and the need for a simple and manageable compiler architecture, the project will use a manual translation approach to implement the compiler for the functional language subset. This approach allows for a more straightforward implementation of the compiler while focusing on the core features of the functional language subset and efficient Wasm compilation.

The architecture of GHC can still be used as a reference for the project's compiler architecture. The following section describes the compilation process in GHC and how the Haskell compiler works.

2.8. How the GHC Haskell compiler works

This chapter is inspired by notes from a lecture on the GHC compiler at Stanford University [2].

The Glasgow Haskell Compiler (GHC) is the most widely used Haskell compiler and provides a reference implementation for the Haskell language. GHC translates Haskell source code into intermediate representations (IRs) and eventually into machine code. The compilation process in GHC involves several stages, each performing specific tasks to optimize and generate efficient code.

In summary, the compilation process in GHC consists of the following stages (see Figure 2):

1. First the Haskell source code typed checked and desugared into a simplified intermediate representation (Core). This representation is very similar to the original Haskell code but all syntactical constructs are removed or transformed into only let and case statement. All pattern matching definitions for functions are also reduced to a lambda abstraction with a case statement. This enables to simplify the code and make it easier to optimize. The places where allocations take place (let bindings) and the place expressions get evaluated (case statements) are more clear to see and it is easier to reason about the execution order of the program.
2. The Core representation is then optimized using a set of optimization passes. These passes include inlining, constant folding, dead code elimination, and other optimizations that aim to improve the performance of the code. The optimizations are applied in a sequence of passes, each pass transforming the Core representation to a more optimized version.
3. The optimized Core representation is then translated into a lower-level intermediate representation called STG (Spineless Tagless G-machine). The STG representation is closer to the actual execution model of the Haskell runtime system and provides a more detailed view of the program's evaluation. In this representation the allocations only take place in the let bindings and evaluation only takes place in the case statements. The difference to the Core representation is that the STG representation is more detailed and provides more information about the evaluation order of the program, e.g. function application are represented as a thunk (a closure that takes no arguments) that gets evaluated when needed.
4. The STG representation is further optimized using a set of machine-independent optimizations. The optimizations are applied in a sequence of passes, each pass transforming the STG representation to a more optimized version.
5. The optimized STG representation is then translated into a lower-level intermediate representation called Cmm (C minus minus). The Cmm representation is a low-level imperative language that closely resembles the actual machine code that will be generated. In this representation, the program is represented as a sequence of instructions that manipulate memory and perform computations.
6. The Cmm representation is further optimized using a set of machine-dependent optimizations. Compiler backends can then generate efficient machine code for the target architecture, C code, or LLVM IR.

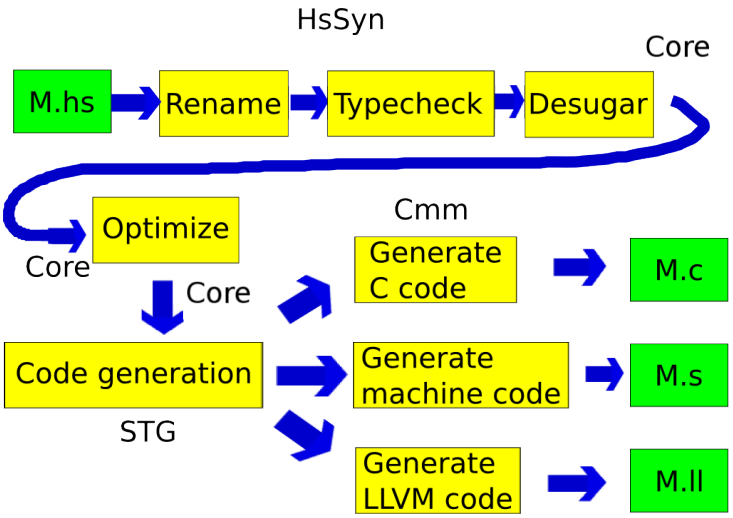


Figure 2: Compilation process in the GHC Haskell compiler (taken form the lecture [2]).

2.8.1. Other similar projects

The following projects are similar to the project and provide insights into the design and implementation of compilers for functional languages. These projects illustrate that this kind of project is feasible and can be implemented within a reasonable timeframe.

2.8.1.1. Asterius

Asterius is a Haskell-to-WebAssembly compiler that translates Haskell source code into WebAssembly bytecode. The project has been archived since 2022 and the project is no longer maintained because Wasm compilation is now supported by GHC. The project was used as a reference for the project's compiler architecture. [8]

Asterius supports FFI (Foreign Function Interface) for interacting with JavaScript code and provides mechanisms for passing data between Haskell and JavaScript. It can be run in the Browser and Node.js and also supports Promises.

Since the project is a compiler for Haskell and not a subset of Haskell, it is more complex than the project's compiler needs to be. The project will use a simpler architecture that directly translates the functional language subset into Wasm bytecode, so it can be more easily understood and implemented within the project's timeframe.

2.8.1.2. Wisp

Wisp is a Lisp-like language that compiles to WebAssembly implemented in Zig. This project is also no longer maintained. The language is similar to Common Lisp and provides a set of features for interactive development and debugging. [4]

2.9. Other technological choices

The following technological choices were made for the project.

2.9.1. Gitlab

Gitlab is a web-based DevOps lifecycle tool that provides a Git repository manager providing wiki, issue-tracking, and CI/CD pipeline features. It is widely used in industry and academia for managing software projects and collaborating on code development. In the context of the project, Gitlab will be used to host the project's source code, issue-tracking, documentation, and CI/CD pipelines. The CI/CD pipelines in Gitlab will be used to automate the linting, testing, and deployment of the build artifacts.

2.9.2. Typst

The documentation for the project will be written in Typst, a typesetting system that allows for the creation of structured documents designed to be a modern alternative

to LaTeX. Typst provides a simple and intuitive syntax for writing documents, including support for figures, tables, code blocks, a package ecosystem, and mathematical expressions.

Mr. Supcik provided a template for the project's documentation, which includes a title page, table of contents, list of figures, list of tables, a header and footer, and a bibliography. The author introduced a glossary, a syntax file for the Wasm text format to enable syntax highlighting, and some styling changes to the template.

2.9.3. Language for the compiler

Rust is a systems programming language that focuses on safety, speed, and concurrency. It is widely used in industry and academia for developing high-performance software, operating systems, and embedded systems. Rust's memory safety features, zero-cost abstractions, and modern tooling make it an ideal choice for implementing the compiler for the functional language subset.

Rust also has great support for WebAssembly and all its runtime environments. The Rust compiler can target WebAssembly directly and the Rust ecosystem provides tools and libraries for working with WebAssembly.

Section 3
Design

This section describes the design of the functional language, compiler, and standard library. It includes the lexical and context-free syntax of the language, the compiler's architecture, and the standard library's design.

3.1. Language specification

The syntax of the functional language (named "Waskell") is as already mentioned a subset of Haskell. There is a language specification document that defines the syntax and semantics of Haskell made by the Haskell team [7]. The Lexical syntax and context-free syntax of the functional language are based on this document. The lexical syntax refers to the rules that define how the characters in the source code are grouped into tokens. The context-free syntax refers to the rules that define how the tokens are grouped into expressions, declarations, and other constructs.

The syntax diagrams in this section are based on the Haskell report [7]. The conventions used in the syntax diagrams are as follows:

- Terminal symbols are shown in a monospaced font and are inclosed in single or double quotes (e.g., `'let'`, `"in"`).
- Non-terminal symbols are shown in math font (e.g., program, lexeme).
- Repetition (one or more occurrences) is shown using braces (e.g., $\{\ a\ \}$).
- Alternatives are shown using vertical bars (e.g., $a \mid b$).
- Optional elements are shown using square brackets (e.g., $[a]$).
- Set subtraction is shown using brackets (e.g., $\text{symbol}_{\langle \text{special} \mid \_ \mid \text{"} \mid \text{'} \rangle}$).
- Annotations to provide additional information about the non-terminal symbols are shown in sans-serif font (e.g., digit ::= any digit).
- The name of the token used in the compiler is shown after the production rule in italics (e.g., integer ::= digit {digit} *Integer*).

3.1.1. Lexical syntax

The lexical syntax of Waskell is identical to Haskell. EBNF 1 shows the lexical syntax of the functional language (the text after the annotations is the token's name used in the compiler).

program	::=	{lexeme | whitespace}	
lexeme	::=	var_id | con_id | var_sym | con_sym	
	|	literal | special | reserved_id | reserved_op	
whitespace	::=	whitechar | line_comment | block_comment	
var_id	::=	(small {small | large | digit | '_'})$_{\langle \text{reserved\_id} \rangle}$	*VariableIdent*
con_id	::=	large {small | large | digit | '_'}	*ConstructorIdent*
var_sym	::=	$\left(\text{symbol}_{\langle ':' \rangle}\{\text{symbol}\}\right)_{\langle \text{reserved\_op} \rangle}$	*VariableSym*
con_sym	::=	(':' {symbol})$_{\langle \text{reserved\_op} \rangle}$	*ConstructorSym*
literal	::=	integer | char | string	
special	::=	'(' | ')' | ',' | ';' | '[' | ']' | '`')	
	|	'{' | '}'	*Special*
reserved_id	::=	'case' | 'class' | 'data' | 'default' | 'deriving'	
	|	'do' | 'else' | 'foreign' | 'if' | 'import' | 'in'	
	|	'infix' | 'infixl' | 'infixr' | 'instance' | 'let'	
	|	'module' | 'newtype' | 'of' | 'then' | 'type'	
	|	'where' | '_'	*ReservedId*
reserved_op	::=	'..' | ':' | '::' | '=' | '\' | '|' | '<-'	
	|	'->' | '@' | '~' | '=>'	*ReservedOp*
whitechar	::=	' ' | '\t' | '\n' | '\r' | '\f' | '\v'	
line_comment	::=	'--'	
	|	'--' {any character} '\n'	*LineComment*
block_comment	::=	'{-' {any character} '-}'	*BlockComment*
small	::=	any lowercase letter	
large	::=	any uppercase letter	
digit	::=	any digit	
symbol	::=	any symbol character$_{\langle \text{special} \mid \text{'_'} \mid \text{'"'} \mid \text{"'"} \rangle}$	
integer	::=	digit {digit}	*Integer*
char	::=	"'" graphic$_{\langle \text{"'"} \rangle}$ "'"	*Char*
string	::=	'"' $\left\{\text{graphic}_{\langle \text{'"'} \rangle}\right\}$ '"'	*String*
graphic	::=	digit | small | large | symbol | special	
	|	'_' | "'" | '"'	

EBNF 1: Lexical syntax of the functional language.

One of the main differences between the lexical syntax of Haskell and the one of Waskell is the lack of the so-called layout rule in the latter. In Haskell, the layout rule allows the programmer to omit braces and semicolons in the source code by using indentation to indicate the structure of the program. In Waskell, the layout rule is not supported, and the programmer must use braces and semicolons to delimit blocks of code. Every declaration must be followed by a semicolon.

3.1.2. Context-free syntax

The context-free syntax of Waskell is a subset of Haskell (defined in report [7]). The context-free syntax of Waskell is based on this document. EBNF 2 shows the context-free syntax of the functional language (the text after the annotations is the non-terminal's name used in the compiler).

```
body          ::= declaration {';' declaration [';']}   TopDeclarations
declaration   ::= fun_bind | data_decl | foreign_decl   TopDeclaration
fun_bind      ::= fun_lhs '=' exp                       FunctionDeclaration
                | fun_sign                              TypeSig
data_decl     ::= 'data' simple_type ['=' constr]       DataDeclaration
foreign_decl  ::= 'foreign import wasm' ['"lib"'] fun_sign
                | 'foreign export wasm' ['"unevaluated"'] fun_sign
fun_sign      ::= (var_id | '(' var_sym ')') '::' ftype
simple_type   ::= con_id {var_id}
constr        ::= con_id {'|' type}
ftype         ::= type {'->' type}                      FunctionType
type          ::= type_elem {type_elem}                 Type
type_elem     ::= var_id                                TypeVariable
                | '(' ftype ')'                         ParenthesizedType
                | '(' ftype ',' ftype {',' ftype} ')'   TupleType
                | type_con
type_con      ::= con_id                                TypeConstructor
                | '()'                                  Unit
                | '(,' { ',' } ')'                      TupleConstructor
fun_lhs       ::= (var_id | '(' var_sym ')') {apat}
apat          ::= var_id ['@' apat]                     AsPattern
                | pat_type_con
                | integer                               IntegerLiteral
                | char                                  CharLiteral
                | string                                StringLiteral
                | '_'                                   Wildcard
                | '(' pat ')'                           ParenthesizedPattern
                | '(' pat ',' pat {',' pat} ')'         TuplePattern
pat           ::= con_id apat {apat}                    ConstructorPattern
                | '-' integer                           NegatedIntegerLiteral
                | apat                                  FunctionParameterPattern
pat_type_con  ::= con_id                                ConstructorPattern
                | '()'                                  UnitPattern
                | '(,' { ',' } ')'                      EmptyTuplePattern
exp           ::= lexp (con_sym | '`' var_id '`') exp   InfixedApplications
                | '-' exp                               NegatedExpression
                | lexp                                  LeftHandSideExpression
lexp          ::= fexp {fexp}                           FunctionApplication
fexp          ::= exp_type_con
                | (var_id | '(' var_sym ')')            Variable
                | con_id                                Constructor
                | integer                               IntegerLiteral
                | char                                  CharLiteral
                | string                                StringLiteral
                | '(' exp ')'                           ParenthesizedExpr
                | '(' exp ',' exp {',' exp} ')'         TupleExpr
exp_type_con  ::= con_id                                Constructor
                | '()'                                  Unit
                | '(,' { ',' } ')'                      Empty
```

EBNF 2: Context-free syntax of the functional language.

3.1.3. Function declarations

All function declarations have to be preceded by a type signature (there is no automatic type inference for function declarations). The type signature specifies the function's name, the types of its parameters, and the return type. The type signature is followed by the function's definition, which consists of a series of equations with pattern matching. The function's definition can include multiple equations with different patterns to handle different cases (more on pattern matching in Section 3.1.4).

The definition of the function can have less arguments than the type signature specifies as long as the expression on the right-hand side of the equation is a function that takes the remaining arguments (more on partial application in Section 3.1.5).

Listing 13 shows the syntax for function declarations in the functional language.

```
-- Function declaration with a single parameter.
f :: Int -> Int;
f x = x + 1;

-- Function declaration with multiple parameters.
g :: Int -> Int -> Int;
g x y = x * y;

-- Function declaration with pattern matching.
fib :: Int -> Int;
fib 0 = 0;
fib 1 = 1;
fib n = fib (n - 1) + fib (n - 2);

-- Function declaration with partial application.
add :: Int -> Int -> Int;
add x = (+) x;

-- Function declaration of a higher-order function.
-- first argument is a function that takes any type and returns a value of
that type (a -> a)
-- second argument is a value of that type (a)
-- the return value is a value of that type (a)
applyTwice :: (a -> a) -> a -> a;
applyTwice f x = f (f x);
```

Listing 13: Syntax for function declarations.

3.1.4. Pattern matching

Pattern matching is a fundamental feature of functional programming languages that allows functions to be defined by cases. Each case consists of a pattern and an expression. When a function is applied to an argument, the patterns are matched against

the argument to determine which case applies. If a match is found, the corresponding expression is evaluated. If no match is found, an error is raised.

Pattern matching can be used in function declarations (as shown in Listing 13) to define functions that behave differently based on the input arguments. Patterns can include literals, variables, constructors (more on constructors and data structures in Section 3.1.6), wildcards, and tuples. Patterns can also be nested to match complex data structures.

Listing 14 shows examples of pattern matching in the functional language.

```
-- Pattern matching with literals.
isZero :: Int -> Bool;
isZero 0 = True;
isZero _ = False;

-- Pattern matching with variables.
factorial :: Int -> Int;
factorial 0 = 1;
factorial n = n * factorial (n - 1);

-- Pattern matching with constructors.
data List a = Nil | Cons a (List a);
length :: List a -> Int;
length Nil = 0;
length (Cons _ xs) = 1 + length xs;

-- Pattern matching with tuples.
fst :: (a, b) -> a;
fst (x, _) = x;

-- Pattern matching with nested patterns.
data Maybe a = Just a | Nothing;
maybeLength :: Maybe (List a) -> Int;
maybeLength Nothing = 0;
maybeLength (Just Nil) = 0;
maybeLength (Just (Cons _ xs)) = 1 + maybeLength (Just xs);
```

Listing 14: Examples of pattern matching.

3.1.5. Function application

Function application is the process of applying a function to its arguments. In the functional language, functions can be applied to one or more arguments, and the arguments can be expressions, variables, literals, or other functions. Function application can be used to create new functions by partially applying (see Section 2.3.1) existing functions.

As seen in Listing 13, function declarations can have less arguments than the type signature specifies. This is because the expression on the right-hand side of the equation is an expression that returns a function that takes the remaining arguments.

Listing 15 shows examples of function application in the functional language.

```
-- Function application with literals.
add :: Int -> Int -> Int;
add x y = x + y;

result1 :: Int;
result1 = add 1 2;

-- Function application with variables.
square :: Int -> Int;
square x = x * x;

result2 :: Int;
result2 = square result1;

-- Function application with expressions.
result3 :: Int;
result3 = add (square 3) (square 4);

-- Function application with partial application.
addOne :: Int -> Int;
addOne = add 1;

result4 :: Int;
result4 = addOne 5;

-- Function application with higher-order functions.
applyTwice :: (a -> a) -> a -> a;
applyTwice f x = f (f x);

result5 :: Int;
result5 = applyTwice square 2;

result6 :: Int;
result6 = applyTwice (add 1) 2;
```

Listing 15: Examples of function application.

3.1.6. Simple types

Waskell supports the definition of simple algebraic data types. It includes some built-in types like tuples, integers, characters and strings (which is an alias for lists of characters). The definition of custom data types is done using the `data` keyword. A data type can have one or more data constructors, each of which can have zero or more arguments. The data constructors can be used in pattern matching to create and deconstruct values of the data type.

Each type can also have its own type variables (more on type variables in Section 3.1.7). The name given to a type can also be called a type constructor since it constructs

a new type from other types. For example, the type `List` is a type constructor that constructs a new type from the type variable `a`, a list of integers would be `List Int` (a type application). In Waskell only concrete types can be used in a type application, so `List` is not a type but a type constructor that needs to be applied to a concrete type to become a new concrete type itself. A good analogy is that a type constructor is like a function that operates on types instead of values, with the difference that a type constructor can't recieve another type constructor as an argument (see the `Foo` type in Listing 16).

If a data constructor has type variables, the type variables must be declared in the data type definition. The type variables are used to make the data type generic over types. For example, the `List a` type is generic over the element type `a`, so it can represent lists of integers, characters, or any other type (more on polymorphism in Section 3.1.7).

Listing 16 shows examples of simple types in the functional language.

```
-- Simple type definition.
data Bool = True | False;
bool :: Bool;
bool = True;

-- Simple type definition with data constructor arguments.
data Ratio = Ratio Int Int;
ratio :: Ratio;
ratio = Ratio 1 2;

-- Custom data type with multiple constructors.
data Maybe a = Just a | Nothing;
maybe1 :: Maybe Int;
maybe1 = Just 42;

maybe2 :: Maybe Int;
maybe2 = Nothing;

-- Custom data type with type variables.
data List a = Nil | Cons a (List a);
list :: List Int;
list = Cons 1 (Cons 2 Nil);

-- Custom data type with multiple type variables.
data Either a b = Left a | Right b;
either1 :: Either Int Bool;
either1 = Left 42;

either2 :: Either Int Bool;
either2 = Right True;

-- Having a type variable that takes a type constructors which takes another
type is not supported.
-- This code does not compile in Waskell.
data Foo a b = Foo (b a);
foo :: Foo Int List;
foo = Foo (Cons 1 Nil);
```

Listing 16: Examples of simple types.

A last note on data constructors is that, if they are used in an expression, they act as a function that constructs a value of the data type. For example, the expression `Just 42` constructs a value of the `Maybe Int` type using the `Just` data constructor. The data constructor can also be used in pattern matching to deconstruct values of the data type. For example, the pattern `Just x` matches values of the `Maybe Int` type constructed with the `Just` data constructor and binds the value `x` to the inner value.

3.1.7. Polymorphism

Polymorphism (or in this case parametric polymorphic) is a feature that allows functions and data types to be generic over types. In the functional language, polymorphism is achieved through type variables. Type variables are placeholders for concrete types that can be instantiated with different types. Functions and data types that use type variables are polymorphic and can work with a wide range of types.

The functional language supports simple type polymorphism, where type variables can be used to define functions and data types that are generic over types. Type variables are introduced using lowercase variable identifiers (by convention, single-letter identifiers are used). Type variables can be used in type signatures to specify the types of function parameters, return values, and data constructors.

Haskell also supports ad-hoc polymorphism (or function overloading) through type classes, but this feature is not included in Waskell.

Listing 17 shows examples of polymorphic functions and data types in the functional language.

```
-- Polymorphic function with type variables.
id :: a -> a;
id x = x;

-- Polymorphic function with multiple type variables.
const :: a -> b -> a;
const x _ = x;

-- Polymorphic function with type variables and type constructors.
map :: (a -> b) -> List a -> List b;
map f Nil = Nil;
map f (Cons x xs) = Cons (f x) (map f xs);
```

Listing 17: Examples of polymorphism.

3.1.8. Operators

Operators are functions that can be used in infix notation. In the functional language, operators are defined using symbols (e.g., `+`, `-`, `*`) or a combination of symbols (e.g., `&&`, `||`). Operators can also be used in prefix notation by enclosing them in parentheses (e.g., `(+)`, `(&&)`). Normal functions can also be used in infix notation by enclosing them in backticks (e.g., `x `add` y`).

Waskell only supports infix notation for binary operators. The definition of precedence, associativity, fixity, and arity $\neq 2$ operators is not supported in Waskell. Since there the different operators don't have a precedence, the order of evaluation is determined by the order of the operators in the expression. If the programmer wants to enforce a

specific order of evaluation, parentheses can be used to group expressions (e.g., `(x + y) * z` and `x + y * z` are the same in Waskell).

Sections and constructor symbols are not supported in Waskell. Sections are a feature of Haskell that allows the programmer to partially apply infix operators by fixing one of the arguments (e.g., `(/2)` or `(2/)`). Constructor symbols are a feature of Haskell that allows the programmer to define custom operators using symbols (e.g., `(:)`, the cons operator for lists).

To define an operator in Waskell, the operator must be enclosed in parentheses and used as a function name. The infix syntax to define an operator is not supported in Waskell.

Listing 18 shows examples of operators in the functional language.

```
-- Operator definition using symbols.
(++) :: List a -> List a -> List a;
(++) Nil ys = ys;
(++) (Cons x xs) ys = Cons x (xs ++ ys);

-- Operator usage in infix notation.
list1 :: List Int;
list1 = (Cons 1 (Cons 2 Nil)) ++ Cons 3 Nil;

-- Operator usage in prefix notation.
list2 :: List Int;
list2 = (++) (Cons 1 (Cons 2 Nil)) (Cons 3 Nil);


div :: Int -> Int -> Int;
div x y = x / y;

-- Function usage in infix notation.
val :: Int;
val = 4 `div` 2;
```

Listing 18: Examples of operators.

3.1.9. Lazy evaluation

Waskell uses lazy evaluation to evaluate expressions. Lazy evaluation is a strategy where expressions are only evaluated when their values are needed. This allows for more efficient evaluation of expressions and can prevent unnecessary computations. In lazy evaluation, expressions are represented as thunks, which are unevaluated computations that can be forced to produce a value.

Lazy evaluation allows for the creation of infinite data structures and the use of higher-order functions like `map`, `filter`, and `foldr`. Lazy evaluation also allows for the use of recursion without the risk of stack overflow, as only the necessary parts of the computation are evaluated.

In Waskell (as in Haskell), the entry point of the program is the `main` function. The `main` function is a special function that is called when the program is executed (it is exported by default). As with any other exported function, the evaluation of the expressions in the `main` function is forced when the function is called.

The only other way to force the (partial not complete) evaluation of an expression in Waskell is during pattern matching. When a pattern is matched, the expression on the right-hand side of the equation is evaluated until a data constructor is reached. This is done to determine which case applies and to bind the variables in the pattern to the values in the data constructor.

The reason why pattern matching only "partially" evaluates the expression is that the expression is only evaluated until a data constructor is reached. An infinite data structure can be pattern matched without causing an infinite loop, as only the necessary parts of the data structure are evaluated (e.g., `take 5 (repeat 1)` will only evaluate the first 5 elements of the infinite list).

Because of lazy evaluation, imported functions that return void are mapped to functions that return unit (a type with a single value, `()`). This is done to prevent the evaluation of the imported function when it is called from the host language. The evaluation of the imported function is deferred until the value is needed.

Listing 19 shows examples of lazy evaluation in Waskell (for more examples see Listing 7).

```
foreign export wasm square :: Int -> Int;
square x = x * x;

foreign export wasm sq_print :: Int -> ();
sq_print x = printInt (square x);

foreign import wasm printInt :: Int -> ();

const :: a -> b -> a;
-- The second argument is not evaluated because the first argument is returned.
const x _ = x;

main :: ();
main = const (sq_print 3) (sq_print 4);
-- if the printInt function is implemented to print the value and return it,
the output will be: 9
-- 16 is not printed because the second printInt is not evaluated
```

Listing 19: Examples of lazy evaluation.

3.1.10. Embedding

The embedding of Waskell code in other languages works by using the import and export features of Wasm. Any Waskell function can be exported to be used in other languages and any Wasm function can be imported to be used in Waskell. The import and export features are used to define foreign function interfaces (FFI) that allow functions written in different languages to interact with each other.

Wasmtime and Wasmer allow for functions in the host language to be injected into one of the Wasm module's imports. This allows for the Wasm module to call functions in the host language. The host language can also call functions in the Wasm module by using the Wasm module's exports.

To export a function from Waskell, the function must be declared with the `foreign export wasm` keyword. The function can then be called from other languages by importing the Wasm module and calling the exported function. To import a function into Waskell, the function must be declared with the `foreign import wasm` keyword. The function can then be called from Waskell by using the imported function name.

For the implementation of the compiler, some Wasm functions will be imported to provide basic functionality like memory allocation, deallocation, and other low-level operations. For these functions, the implementation will use the `foreign import wasm "lib"` syntax to import the functions from a predefined Wasm library. The added `"lib"` keyword is used to indicate that the function is part of a library and not a custom function since it has a different calling convention.

When exporting functions from Waskell, some times the function will be marked as `"unevaluated"`. This is used to indicate that the function should not be evaluated when called from the host language. This is useful when exporting functions that create recursive data structures that will be later used in Waskell. This feature exists to fix a issue with the current implementation of the compiler (more on this in Section 4.8.4).

Listing 20 shows examples of embedding Waskell code in other languages.

```
-- Exporting a function from Waskell.
foreign export wasm fib :: Int -> Int;
fib :: Int -> Int;
fib 0 = 0;
fib 1 = 1;
fib n = fib (n - 1) + fib (n - 2);

-- Importing a function into Waskell.
foreign import wasm printInt :: Int -> ();

-- Using the imported function in Waskell.
foreign export wasm fibPrint :: Int -> ();
fibPrint n = printInt (fib n);

-- Importing a function into Waskell from the library.
foreign import wasm "lib" (+) :: Int -> Int -> Int;

-- Exporting a function from Waskell that is unevaluated.
data List a = Nil | Cons a (List a);
foreign export wasm cons "unevaluated" :: a -> List a -> List a;
cons = Cons;
```

Listing 20: Examples of embedding Waskell code in other languages.

Listing 21 shows examples of using embedded functions with the wasmtime runtime in Python.

```python
from wasmtime import FuncType, Store, Module, Linker, ValType, WasiConfig,
Engine

engine = Engine()
linker = Linker(engine)
linker.define_wasi()

store = Store(engine)
wasi = WasiConfig()
wasi.inherit_stdout()
wasi.inherit_stdin()
wasi.inherit_stderr()
wasi.inherit_env()
wasi.inherit_argv()
store.set_wasi(wasi)

def printInt(value):
    print(f'Printing from host: {value}')

linker.define_func("foreign", "printInt", FuncType([ValType.i32()], []),
printInt)

module = Module.from_file(engine, 'out.wasm')
instance = linker.instantiate(store, module)

fib_print = instance.exports(store)["fibPrint"]
fib = instance.exports(store)["fib"]

fib_print(store, 7)
print("fib(7) =", fib(store, 7))
# Output:
# Printing from host: 13
# fib(7) = 13
```

Listing 21: Examples of using embedded functions with the wasmtime runtime in Python.

Listing 22 and Listing 23 shows examples of using embedded functions with the wasmtime runtime in a more complex example. This code interacts with the code of the `waskellc/examples/types.wsk` file.

```
def bytes_to_int32(memory, store, ptr):
    return memory.data_ptr(store)[ptr + 3] * 2**24 +
       memory.data_ptr(store)[ptr + 2] * 2**16 +
       memory.data_ptr(store)[ptr + 1] * 2**8 +
       memory.data_ptr(store)[ptr]

def write_int32(memory, store, ptr, value):
    memory.data_ptr(store)[ptr] = value & 0xFF
    memory.data_ptr(store)[ptr + 1] = (value >> 8) & 0xFF
    memory.data_ptr(store)[ptr + 2] = (value >> 16) & 0xFF
    memory.data_ptr(store)[ptr + 3] = (value >> 24) & 0xFF

def parse_waskell_list(memory, store, list_ptr):
    result = []
    while True:
        data_constr = bytes_to_int32(memory, store, list_ptr + 4)
        if data_constr == 0:
            break

        data = bytes_to_int32(memory, store, list_ptr + 8)
        result.append(data)
        list_ptr = bytes_to_int32(memory, store, list_ptr + 12)
    return result

def create_waskell_tree(memory, store, empty_fn, node_fn, make_val_fn, data):
    def recurse(data):
        if data is None:
            return empty_fn()

        l_ptr = recurse(data.get("l"))
        r_ptr = recurse(data.get("r"))
        val_ptr = make_val_fn(0, data.get("e"))
        return node_fn(l_ptr, val_ptr, r_ptr)

    res = recurse(data)
    res = bytes_to_int32(memory, store, res + 1)
    return res

flattenDfs = tree.flattenDfs
exampleTreeFlattened = tree.exampleTreeFlattened
empty = tree.empty
node = tree.node
# get the make_val function from the tree module (not a usual name for python)
make_val = list(
  filter(lambda x: x[0] == ":make_val", inspect.getmembers(tree))
)[0][1]
memory = tree.memory
store = wasmtime.loader.store
```

Listing 22: Example of using embedded functions with the wasmtime runtime in a more complex example part 1.

```
import inspect

import wasmtime.loader
import tree

list_ptr = exampleTreeFlattened()
print(parse_waskell_list(memory, store, list_ptr))

example_tree = {
    "e": 2,
    "l": {
        "e": 1,
        "l": None,
        "r": None,
    },
    "r": {
        "e": 3,
        "l": None,
        "r": None,
    },
}

tree_ptr = create_waskell_tree(memory, store, empty, node, make_val,
example_tree)
flattened_ptr = flattenDfs(tree_ptr)
print(parse_waskell_list(memory, store, flattened_ptr))
# Output:
# [3, 6, 9, 12, 15]
# [1, 2, 3]
```

Listing 23: Example of using embedded functions with the wasmtime runtime in a more complex example part 2.

3.2. Standard library

The standard library of the functional language is a subset (with some differences to account the lack of some language features) of the Haskell standard library (or the Prelude). The standard library provides a set of functions and types that are commonly used in functional programming. The standard library includes functions for working with lists, tuples, numbers, and other data types. The design of the standard library is based on the Haskell standard library documentation [1].

3.2.1. Basic types

Listing 24 shows the list of basic types in the standard library.

```
-- Boolean type with values True and False.
data Bool = True | False
-- Character type representing Unicode characters.
data Char = ...
-- Integer type with fixed precision.
data Int = ...
-- String type representing lists of characters (alias for [Char]).
type String = [Char]
-- List of elements of type a (two constructors: [] and :).
data List a = Nil | Cons a (List a)
-- Tuple type with n elements of types a, b, ..., z.
data (a, b, ..., z) = ...
-- Unit type with a single value denoted by ().
data () = ...
-- Maybe type representing optional values.
data Maybe a = Just a | Nothing
-- Either type representing disjoint unions.
data Either a b = Left a | Right b
-- Ratio type representing fractions.
data Ratio = Ratio Int Int
```

Listing 24: The list of basic types in the standard library.

3.2.2. Boolean functions

Listing 25 shows the list of functions for working with booleans in the standard library.

```
boolToString :: Bool -> String -- Converts a boolean to a string.
boolEq :: Bool -> Bool -> Bool -- Equality comparison for booleans.
not :: Bool -> Bool -- Negates a boolean value.
(&&), (||) :: Bool -> Bool -> Bool -- Logical AND and OR operations.
if' :: Bool -> a -> a -> a -- Conditional expression.
```

Listing 25: The list of functions for working with booleans.

3.2.3. Numeric functions

Listing 26 shows the list of functions for working with numbers in the standard library.

```
intToString :: Int -> String -- Converts an integer to a string.
data Ordering = LT | EQ | GT -- Ordering type for comparison results.
compare :: Int -> Int -> Ordering -- Compares two values.
(+), (-), (*) :: Int -> Int -> Int -- Addition, subtraction, and
multiplication.
negate, abs :: Int -> Int -- Sign negation and absolute value.
(==), (/=), (<), (<=), (>), (>=) :: Int -> Int -> Bool -- Comparison
operations.
min, max :: Int -> Int -> Int -- Minimum and maximum of two values.
minBound, maxBound :: Int -- Smallest and largest value of a type.
quot, rem :: Int -> Int -> Int -- Quotient and remainder operations.
quotRem :: Int -> Int -> (Int, Int) -- Quotient and remainder as a pair.
even, odd :: Int -> Bool -- Checks for even and odd numbers.
(^) :: Int -> Int -> Int -- Exponentiation operation.
```

Listing 26: The list of functions for working with numbers.

3.2.4. List functions

Listing 27 and Listing 28 shows the list of functions for working with lists in the standard library.

```
listToString :: (a -> String) -> List a -> String -- Converts a list to a
string.
listEq :: (a -> a -> Bool) -> List a -> List a -> Bool -- Equality comparison
for lists.
map :: (a -> b) -> List a -> List b -- Applies a function to each element of
a list.
(++), concat :: List a -> List a -> List a -- Concatenates two lists.
filter :: (a -> Bool) -> List a -> List a -- Filters a list based on a
predicate.
head :: List a -> Maybe a -- Returns the first element of a list.
last :: List a -> Maybe a -- Returns the last element of a list.
tail :: List a -> Maybe (List a) -- Returns the list without the first
element.
init :: List a -> Maybe (List a) -- Returns the list without the last element.
null :: List a -> Bool -- Checks if a list is empty.
length :: List a -> Int -- Returns the length of a list.
(!!): List a -> Int -> Maybe a -- Returns the element at a specific index.
reverse :: List a -> List a -- Reverses a list.

-- Reduction functions
foldl :: (b -> a -> b) -> b -> List a -> b -- Folds a list from the left.
foldr :: (a -> b -> b) -> b -> List a -> b -- Folds a list from the right.
and :: List Bool -> Bool -- Checks if all elements are true.
or :: List Bool -> Bool -- Checks if any element is true.
any :: (a -> Bool) -> List a -> Bool -- Checks if any element satisfies a
predicate.
all :: (a -> Bool) -> List a -> Bool -- Checks if all elements satisfy a
predicate.
sum :: List Int -> Int -- Sums the elements of a list.
product :: List Int -> Int -- Multiplies the elements of a list.
concat :: List (List a) -> List a -- Flattens a list of lists.
concatMap :: (a -> List b) -> List a -> List b -- Maps and concatenates a
list.
maximum :: List a -> a -- Returns the maximum element of a list.
minimum :: List a -> a -- Returns the minimum element of a list.

-- Building functions
scanr :: (a -> b -> b) -> b -> List a -> List b -- Scans a list from the
right.
scanl :: (b -> a -> b) -> b -> List a -> List b -- Scans a list from the
left.
iterate :: (a -> a) -> a -> List a -- Generates an infinite list by repeatedly
applying a function.
repeat :: a -> List a -- Generates an infinite list with a single element.
replicate :: Int -> a -> List a -- Generates a list with a repeated element.
cycle :: List a -> List a -- Generates an infinite list by cycling a list.
```

Listing 27: The list of functions for working with lists part 1.

```
-- Sublist functions
take :: Int -> List a -> List a -- Takes the first n elements of a list.
drop :: Int -> List a -> List a -- Drops the first n elements of a list.
splitAt :: Int -> List a -> (List a, List a) -- Splits a list at a specific
index.
takeWhile :: (a -> Bool) -> List a -> List a -- Takes elements from a list
while a predicate is true.
dropWhile :: (a -> Bool) -> List a -> List a -- Drops elements from a list
while a predicate is true.
span :: (a -> Bool) -> List a -> (List a, List a) -- Splits a list into two
parts based on a predicate.
break :: (a -> Bool) -> List a -> (List a, List a) -- Splits a list into two
parts based on a predicate.

-- Zipping functions
zip :: List a -> List b -> List (a, b) -- Zips two lists together.
zipWith :: (a -> b -> c) -> List a -> List b -> List c -- Zips two lists with
a function.
unzip :: List (a, b) -> (List a, List b) -- Unzips a list of pairs.
```

Listing 28: The list of functions for working with lists part 2.

3.2.5. Tuple functions

Listing 29 shows the list of functions for working with tuples in the standard library.

```
fst :: (a, b) -> a -- Returns the first element of a tuple.
snd :: (a, b) -> b -- Returns the second element of a tuple.
curry :: ((a, b) -> c) -> a -> b -> c -- Curries a function.
uncurry :: (a -> b -> c) -> (a, b) -> c -- Uncurries a function.
```

Listing 29: The list of functions for working with tuples.

3.2.6. Ratio functions

Listing 30 shows the list of functions for working with ratios in the standard library.

```
ratioToString :: Ratio -> String -- Converts a ratio to a string.
ratioEq :: Ratio -> Ratio -> Bool -- Equality comparison for ratios.
(%) :: Int -> Int -> Ratio -- Constructs a ratio from two integers.
numerator, denominator :: Ratio -> Int -- Extracts the numerator and
denominator of a ratio.
ratioFromInt :: Int -> Ratio -- Converts an integer to a ratio.
addRatio, subRatio, mulRatio, divRatio :: Ratio -> Ratio -> Ratio --
Arithmetic operations on ratios.
recipRatio, negateRatio :: Ratio -> Ratio -- Reciprocal and negation of a
ratio.
evalToInt :: Ratio -> Int -- Evaluates a ratio
```

Listing 30: The list of functions for working with ratios.

3.2.7. Miscellaneous functions

Listing 31 shows the list of miscellaneous functions in the standard library.

```
maybe :: b -> (a -> b) -> Maybe a -> b -- Handles optional values.
either :: (a -> c) -> (b -> c) -> Either a b -> c -- Handles disjoint unions.

id :: a -> a -- Identity function.
const :: a -> b -> a -- Constant function.
flip :: (a -> b -> c) -> b -> a -> c -- Flips the arguments of a function.
($) :: (a -> b) -> a -> b -- Function application operator.
(.) :: (b -> c) -> (a -> b) -> a -> c -- Function composition operator.
until :: (a -> Bool) -> (a -> a) -> a -> a -- Repeatedly applies a function
until a predicate is true.
error :: String -> a -- Throws an error with a message.
undefined :: a -- Throws an undefined error.
```

Listing 31: The list of miscellaneous functions.

3.3. Compiler architecture

The Waskell compiler is implemented in Rust and consists of several components that work together to parse, type-check, and compile Waskell code to WebAssembly. The architecture of the Waskell compiler is shown in Figure 3.

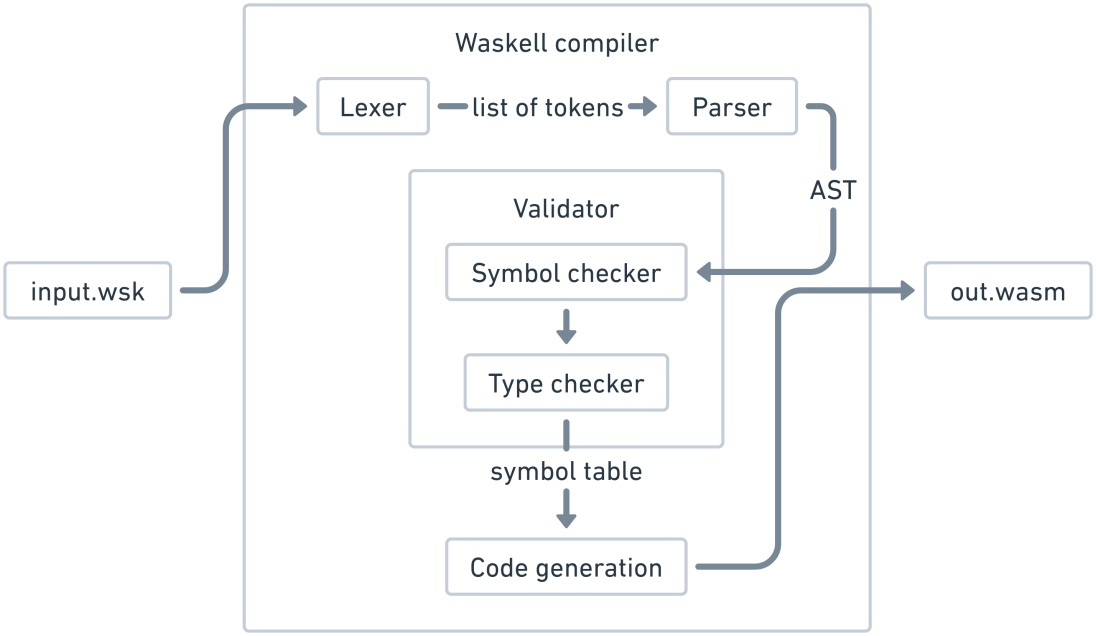


Figure 3: The architecture of the Waskell compiler.

The main components of the Waskell compiler are:

- **Lexer**: The lexer reads the input source code and tokenizes it into a stream of tokens. The lexer recognizes keywords, identifiers, literals, and other syntactic elements in the source code. The lexer outputs a stream of tokens that are consumed by the parser.
- **Parser**: The parser reads the stream of tokens produced by the lexer and constructs an abstract syntax tree (AST) that represents the structure of the source code. The parser enforces the grammar rules of the functional language and reports syntax errors if the source code is not well-formed. The parser outputs the AST that is consumed by the compiler.
- **Validator**: The validator checks the AST for semantic errors, such as type errors, undefined variables, and invalid expressions. The validator ensures that the source code is semantically correct before proceeding to the next stage. The validator outputs a symbol table that is used by the code generator.
  - **Symbol Checker**: The symbol checker builds a symbol table that maps identifiers to scopes. The symbol table is used to resolve variable references and enforce scoping rules.
  - **Type Checker**: The type checker checks the types of expressions in the source code. The type checker ensures that expressions are used in a type-safe manner and reports type errors if the source code is not well-typed.

- **Code Generator**: The code generator reads the symbol table produced by the validator and generates WebAssembly code that implements the functionality of the source code. The code generator translates the functional language constructs into WebAssembly instructions and emits a WebAssembly module that can be executed by a WebAssembly runtime.

Section 4
Implementation

The implementation of the Waskell compiler is divided into several modules that correspond to the components of the compiler architecture. Each module is responsible for a specific task, such as lexing, parsing, symbol checking, type checking, and code generation. The modules work together to transform the source code into a WebAssembly module that can be executed by a WebAssembly runtime.

All the code for the Waskell compiler can be found in the `waskellc` directory of the repository. The `src` directory contains the Rust source code for the compiler, and the `lib` directory contains the standard library and wasm lib file used by the compiler.

The `waskellc` crate is made up of several modules (see Listing 32 for the directory of the source code):

- `ast_gen`: Contains the lexer, parser, and AST generation code.
- `validator`: Contains the symbol checker and type checker code.
- `code_gen`: Contains the code generator code.

```
src
├── ast_gen
│   ├── lexer.rs
│   ├── mod.rs
│   └── parser.rs
├── code_gen
│   ├── encoder_wrapper.rs
│   ├── mod.rs
│   └── wasm_generation.rs
├── lib.rs
├── main.rs
└── validator
    ├── mod.rs
    ├── symbol_check.rs
    └── type_check.rs
```

Listing 32: The directory structure of the source code for the `waskellc` crate.

4.1. Compiler entry point

The entry point of the Waskell compiler is the `main` function in the `main.rs` file. In order for usage of the compiler to be more user-friendly, the `main` uses the `clap` crate to parse command-line arguments and display help messages. The `main` function reads the input source code from a file, passes it to the lexer, parser, validator, and code generator, and writes the output WebAssembly module to a file.

Listing 33 shows the help message displayed by the Waskell compiler when the `--help` flag is used.

```
A compiler for the Waskell programming language (subset of Haskell) that
targets WebAssembly.

Usage: waskellc [OPTIONS] <INPUT>

Arguments:
  <INPUT>  Path to input file

Options:
  -o, --output <OUTPUT>                Path to output file
  -l, --wasm-lib-path <WASM_LIB_PATH>  Path to wasm lib file [default: lib/
lib.wasm]
  -p, --prelude-path <PRELUDE_PATH>    Path to prelude file [default: lib/
prelude.wsk]
  -d, --debug                          Debug mode
  -L, --debug-lexer                    Print lexer output
  -A, --debug-ast                      Print AST
  -S, --debug-symbols                  Print symbol table
  -D, --debug-desugar                  Print desugared symbol table
  -W, --debug-wasm                     Print WAT output of wasm module
  -s, --show-wasm-offsets              Show offsets in WAT output
      --no-merge                      Do not merge wasm module with wasm lib
  -h, --help                           Print help
  -V, --version                        Print version
```

Listing 33: Output of the `--help` flag for the Waskell compiler.

As we can see in Listing 33, the Waskell compiler supports several command-line options, such as specifying the input and output files, enabling debug mode, and printing intermediate results like the lexer output, AST, symbol table, desugared symbol table, and WAT output of the WebAssembly module. The debug mode flag enables allows for the enabling of the debug flags for the lexer, AST, symbol table, desugared symbol table, or WAT output. The `--no-merge` flag disables the merging of the generated WebAssembly module with the wasm lib file.

After the generation of the WebAssembly module, the `main` function writes the output to a file specified by the user or to a default file name based on the input file name. The `main` proceeds to merge the generated WebAssembly module with the wasm lib file (more on this in Section 4.5.1) if the `--no-merge` flag is not used. The merged WebAssembly module is then written to the output file.

The tool used for the merging of the WebAssembly module with the wasm lib file is the `wasm-merge` tool included in binaryen. To use the `wasm-merge` tool, the `main` function calls the `Command` struct from the `std::process` module to execute the `wasm-merge` command with the generated WebAssembly module and the wasm lib file as arguments. The output of the `wasm-merge` command is then written to the output file. An additional

benefit of using the `wasm-merge` tool is that it, since it is part of the binaryen repository, it performs some basic optimizations on the WebAssembly module like dead code elimination and function inlining.

In order to work on all platforms, a bash script to install the wasm-merge tool from the binaryen repository is included in the `waskellc` directory. The script downloads the binaryen repository and copies it to the `waskellc/binaryen-tools` directory. The main function then checks the platform and uses the appropriate binaryen tool for the merging of the WebAssembly module with the wasm lib file.

4.2. Lexer

The lexer is implemented in the `src/ast_gen/lexer.rs` file (part of the `ast_gen` module. The lexer reads the input source code character by character and tokenizes it into a stream of tokens. The lexer recognizes keywords, identifiers, literals, and other syntactic elements in the source code. The lexer outputs a stream of tokens that are consumed by the parser.

To simplify the implementation of the lexer, the `logos` crate is used to generate the lexer code from a lexer specification. To define a lexer, the programmer defines an enum with the token types and a `#[derive(Logos)]` attribute. The `Logos` derive macro generates the lexer code based on the token types and the lexer specification. Each token type is annotated with a regular expression that matches the token in the source code.

The token types defined in the lexer are based on the syntax defined in EBNF 1. The lexer recognizes keywords, identifiers, literals, and other syntactic elements in the source code. The lexer outputs a stream of tokens that are consumed by the parser.

Listing 34 shows a shortened version of the lexer implementation. The full implementation can be found in the `src/ast_gen/lexer.rs` file.

```rust
#[derive(Logos)]
#[logos(skip r"\s+")]
pub enum Token {
    #[regex(r"[[[:lower:]]_][[[:word:]]']*", |lex| lex.slice().to_owned())]
    VariableIdent(String),

    // shortened regex for report
    #[regex( r"case|class|data|...",
        |lex| lex.slice().to_owned(),
        priority = 100)]
    ReservedIdent(String),

    // ...

    #[regex(r"--[^\n]*", logos::skip)]
    LineComment,

    // ...
}
```

Listing 34: A shortened version of the lexer implementation.

The `logos(skip r"\s+")` attribute is used to skip whitespace characters in the input source code.

The `VariableIdent` token type recognizes identifiers that start with a lowercase letter or underscore and are followed by any word character (a-z, A-Z, 0-9, _) or an apostrophe. The identifier is captured as a string and stored in the token variant.

The `ReservedIdent` token type recognizes reserved keywords in the functional language, such as `case`, `class`, `data`, and others. The keyword is captured as a string and stored in the token variant. The `priority = 100` attribute is used to give the token type a higher priority (default is 2) to ensure that reserved keywords are recognized before variable identifiers.

The `LineComment` token type recognizes line comments that start with `--` and continue until the end of the line. The line comment is ignored by the lexer and not included in the stream of tokens (because of the `logos::skip` attribute).

4.3. Parser

The parser is implemented in the `src/ast_gen/parser.rs` file (part of the `ast_gen` module). The parser reads the stream of tokens produced by the lexer and constructs an abstract syntax tree (AST) that represents the structure of the source code. The parser enforces the grammar rules of the functional language and reports syntax errors if the source code is not well-formed. The parser outputs the AST that is consumed by the validator.

The parsing rules are defined using a recursive descent parser that follows the grammar defined in EBNF 2. The parser uses pattern matching to match the tokens produced by the lexer and construct the AST nodes. The parser uses the `Token` enum defined by the lexer to match the token types and construct the AST nodes.

4.3.1. Syntax diagram translation

The method used to implement the parser is inspired by the method used by Dr. Niklaus Wirth in his book "Compiler Construction" [5]. The method involves translating the syntax diagrams of the language into a set of mutually recursive functions that parse the input source code. The translation of the syntax diagrams into parsing functions is done manually by the programmer based on the grammar rules of the language.

For every non-terminal symbol in the grammar, there is a corresponding parsing function that constructs the AST node for that non-terminal symbol. In the case of Rust and the Waskell compiler, the parsing functions are defined as methods of a struct or enum that represents the respective AST node.

Lets take a look at an example from the Waskell parser. Listing 35 and Listing 28 shows a simplified version of the parsing function for the function type in the Waskell language (see EBNF 2 for the grammar rule).

```rust
pub struct FunctionType(pub Vec<Type>); // a function type is a list of types
(the last type is the return type)

impl FunctionType {
    fn parse(input: &mut TokenIter) -> Result<Self, String> {
        let mut types = vec![];
        loop {
            types.push(Type::parse(input)?);
             // this helper function takes the next token without consuming
it
           // the second argument determines if the token should be consumed
            match next_token(input, true)? {
                Token::ReservedOperator(op) if op == "->" => {
                    input.next(); // consume the '->' token
                    continue;
                }
                _ => {
                    break;
                }
            }
        }

        Ok(FunctionType(types))
    }
}
pub struct Type(pub Vec<TypeApplicationElement>); // a type is a list of type
application elements

impl Type {
    fn parse(input: &mut TokenIter) -> Result<Self, String> {
        let mut types = vec![];
        loop {
            types.push(TypeApplicationElement::parse(input)?);
            match next_token(input, true)? {
                // check if the type continues of if this is the end of the
type signature
                Token::ReservedOperator(op) if op == "->" => {
                    break;
                }
                _ => {}
            }
        }

        Ok(Type(types))
    }
}
```

Listing 35: The parsing function for the function type in the Waskell language part 1.

```rust
pub enum TypeApplicationElement {
    /// Represents a type variable.
    TypeVariable(String),
    /// Represents a type constructor for a custom type (e.g. `Maybe`).
    TypeConstructor(String),

    // ... skipped for brevity
}

impl TypeApplicationElement {
    /// Parse a type application element from the input token iterator.
    fn parse(input: &mut TokenIter) -> Result<Self, String> {
        match next_token(input, false)? {
            Token::ConstructorIdent(ident) =>
                Ok(TypeApplicationElement::TypeConstructor(ident)),
            Token::VariableIdent(ident) =>
                Ok(TypeApplicationElement::TypeVariable(ident)),

            // ... skipped for brevity
            t => Err(format!("Unexpected token: {:?}", t)),
        }
    }
}
```

Listing 36: The parsing function for the function type in the Waskell language part 2.

It is sometimes necessary to use helper functions to parse more complex constructs. The `next_token` function is used to get the next token from the input stream without consuming it. The function takes a mutable reference to the token iterator and a boolean flag that determines if the token should be consumed. The function returns the next token from the input stream or an error message if the end of the input stream is reached.

4.3.2. Abstract syntax tree

The AST nodes are defined as structs or enums that represent the different syntactic elements of the language. The AST nodes are used to represent expressions, statements, declarations, and other language constructs. The AST nodes are constructed by the parsing functions and form a tree structure that represents the structure of the source code.

Listing 37, Listing 38, Listing 39, and Listing 40 show the types of AST nodes for the top-level declarations, function and data declarations, types, and patterns in the Waskell language.

```
/// Represents a list of top-level declarations in a Haskell module.
struct TopDeclarations(Vec<TopDeclaration>);

// helper type for the type signature - not an AST node directly
/// Represents the foreign import/export annotations for a function type
signature.
enum IsForeign {
    /// The function is imported from the WASM library.
    LibImported,
    /// The function is imported from a foreign module.
    ForeignImported,
    /// The function is exported in the WASM module.
    Exported,
    /// The function is exported but the parameters and return value are
    /// unevaluated.
    UnevaluatedExported,
    /// The function is not foreign.
    NotForeign,
}

enum TopDeclaration {
    /// Represents a data declaration in a Haskell module.
    DataDecl(DataDeclaration),
    /// Represents a type signature or function declaration in a Haskell
    /// module.
    TypeSig {
        /// name of the function
        name: String,
        /// type signature of the function
        ty: FunctionType,
        /// foreign import/export annotation
        is_foreign: IsForeign,
    },
    /// Represents a function declaration in a Haskell module.
    FunctionDecl(FunctionDeclaration),
}
```

Listing 37: The types of AST nodes for the top level declarations.

```
/// Represents a function declaration in a Haskell module.
struct FunctionDeclaration {
    /// The name of the function.
    name: String,
    /// The pattern matching for the function left-hand side.
    lhs: Vec<FunctionParameterPattern>,
    /// The right-hand side expression for the function.
    rhs: Expression,
}

/// Represents a data declaration in a Haskell module.
struct DataDeclaration {
    /// The name of the type constructor for the data declaration.
    ty_constructor: String,
    /// The type variables for the data declaration (can be empty if the data
    /// declaration is a simple type).
    ty_vars: Vec<String>,
    /// The data constructors for the data declaration.
    data_constructors: Vec<DataConstructor>,
}

/// Represents a data constructor in a Haskell module.
struct DataConstructor {
    /// The name of the data constructor.
    name: String,
    /// The fields of the data constructor.
    fields: Vec<TypeApplicationElement>,
}
```

Listing 38: The types of AST nodes for the function and data declarations.

```rust
/// Represents a type that can be a function type (e.g. `Int -> Int`, a
/// function that takes an `Int` and returns an `Int`) or a simple
/// type (e.g. `Int`).
struct FunctionType(Vec<Type>);

/// Represents a type application. It can be a simple type (e.g. `Int`) or
/// a type constructor (e.g. `Maybe Int`).
struct Type(Vec<TypeApplicationElement>);

/// Represents a type application element.
enum TypeApplicationElement {
    /// Represents a unit type.
    Unit,
    /// Represents an unapplied tuple constructor.
    TupleConstructor(i32),
    /// Represents a tuple type.
    TupleType(Vec<FunctionType>),
    /// Represents a type variable.
    TypeVariable(String),
    /// Represents a parenthesized type (e.g. if an element of the type
    /// signature is a function type).
    ParenthesizedType(Box<FunctionType>),
    /// Represents a type constructor for a custom type (e.g. `Maybe`).
    TypeConstructor(String),
}
```

Listing 39: The types of AST nodes for the types and type applications.

```rust
/// Represents a pattern for a function parameter in a function declaration.
enum FunctionParameterPattern {
    /// Represents a variable pattern. It can be an `as` pattern (e.g. `x@p`)
    /// or a simple variable
    AsPattern(String, Option<Box<FunctionParameterPattern>>),
    /// Represents a constructor pattern. In this case its matching against
    /// a data constructor without any fields.
    ConstructorPattern(String),
    /// Represents a unit pattern. It matches against the unit type `()`.
    UnitPattern,
    /// Represents an empty tuple pattern. It matches against a tuple with
    /// no elements.
    EmptyTuplePattern(i32),
    /// Represents a string literal pattern.
    StringLiteral(String),
    /// Represents an integer literal pattern.
    IntegerLiteral(i32),
    /// Represents a character literal pattern.
    CharLiteral(char),
    /// Represents a wildcard pattern. It matches against any value.
    Wildcard,
    /// Represents a parenthesized pattern. It is required for more complex
    /// patterns.
    ParenthesizedPattern(Box<Pattern>), // boxed because of indirect
                                        // recursion with Pattern
    /// Represents a tuple pattern. It matches against a tuple with one or
    /// more elements.
    TuplePattern(Vec<Pattern>),
}

/// Represents a more general pattern for negated integer literals and data
/// constructors with fields.
enum Pattern {
    /// Represents a function parameter pattern.
    FunctionParameterPattern(FunctionParameterPattern),
    /// Represents a constructor pattern. It matches against a data
    /// constructor with fields.
    ConstructorPattern(String, Vec<FunctionParameterPattern>),
    /// Represents a negated integer literal pattern.
    NegatedIntegerLiteral(i32),
}
```

Listing 40: The types of AST nodes for the function parameter patterns.

```rust
/// Represents a top level expression in a Haskell module.
enum Expression {
    /// Represents an infixed application of an operator to two expressions
    /// (an operator can be a variable identifier with backticks:
    /// ``a `op` b`` or variable symbols: `a + b`).
   InfixedApplication(Box<LeftHandSideExpression>, String, Box<Expression>),
    /// Represents a negated expression (e.g. `-a`).
    NegatedExpr(Box<Expression>),
    /// Represents a left-hand side expression.
    LeftHandSideExpression(Box<LeftHandSideExpression>),
}

/// Represents a left-hand side expression in a Haskell module (in this case
/// left-hand side means either the left-hand side of a infix expression or
/// just an expression that can't be represented in the [`Expression`] enum).
enum LeftHandSideExpression {
    /// Represents a function application.
    FunctionApplication(Vec<FunctionParameterExpression>),
    // could be extended with other expressions like let, case, if, etc. in
the future
}

/// Represents a parameter of a function application in a Haskell module.
enum FunctionParameterExpression {
    /// Represents a string literal.
    StringLiteral(String),
    /// Represents an integer literal.
    IntegerLiteral(i32),
    /// Represents a character literal.
    CharLiteral(char),
    /// Represents a variable.
    Variable(String),
    /// Represents a constructor.
    Constructor(String),
    /// Represents an empty tuple.
    EmptyTuple(i32),
    /// Represents a unit value.
    Unit,
    /// Represents a parenthesized expression.
    ParenthesizedExpr(Box<Expression>),
    /// Represents a tuple expression.
    TupleExpr(Vec<Expression>),
}
```

Listing 41: The types of AST nodes for the expressions.

Since Rust has to know the size of the types at compile time, the AST nodes that contain recursive references to themselves (or other types that reference each other) are wrapped in a `Box` to make them indirect. The `Box` type is a smart pointer that allows for the allocation of heap memory for the type, but the ownership of the memory is

transferred to the `Box` type. This allows for recursive references to be resolved at runtime and for the size of the type to be known at compile time.

Each AST node has a corresponding parsing function that constructs the AST node based on the tokens produced by the lexer. The parsing functions are mutually recursive and follow the grammar rules of the language. The parsing functions construct the AST nodes by recursively calling each other and matching the tokens produced by the lexer.

The value that gets returned from the parsing functions is a `Result` type that contains either a `TopDeclarations` value or an error message. The `Result` type is used to handle errors that occur during parsing, such as syntax errors or unexpected tokens. If an error occurs during parsing, the error message is returned to the caller, and the parsing process is stopped.

4.3.3. AST example

Listing 42 shows a simple Waskell program that defines an `id` function and a `not` function. The program consists of a type signature for the `id` function, a function declaration for the `id` function, a data declaration for the `Bool` type, a type signature for the `not` function, and two function declarations for the `not` function.

```
id :: a -> a
id x = x

data Bool = True | False

not :: Bool -> Bool
not True = False
not False = True
```

Listing 42: A simple Waskell program.

Listing 43 shows the formatted (to fit in the report) version of the debug output of the AST for the simple Waskell program. The AST consists of a list of top-level declarations that represent the type signatures, function declarations, and data declarations in the source code.

```
TopDeclarations([
    TypeSig {
        name: "id",
        ty: FunctionType([
          Type([TypeVariable("a")]), Type([TypeVariable("a")])
        ]),
        is_foreign: NotForeign,
    },
    FunctionDecl{
        name: "id",
        lhs: [AsPattern("x", None)],
        rhs: LeftHandSideExpression(FunctionApplication([Variable("x")])),
    },
    TypeSig {
        name: "print",
       ty: FunctionType([Type([TypeConstructor("String")]), Type([Unit])]),
        is_foreign: LibImported,
    },
    DataDecl {
        ty_constructor: "Bool",
        ty_vars: [],
        data_constructors: [
            DataConstructor { name: "True", fields: [] },
            DataConstructor { name: "False", fields: [] },
        ],
    },
    TypeSig {
        name: "not",
        ty: FunctionType( [
            Type([TypeConstructor("Bool")]),
            Type([TypeConstructor("Bool")]),
        ]),
        is_foreign: NotForeign,
    },
    FunctionDecl {
        name: "not",
        lhs: [ConstructorPattern("True")],
        rhs: LeftHandSideExpression(
          FunctionApplication([Constructor("False")])
        ),
    },
    FunctionDecl {
        name: "not",
        lhs: [ConstructorPattern("False")],
        rhs: LeftHandSideExpression(
          FunctionApplication([Constructor("True")])
        ),
    },
])
```

Listing 43: The debug output of the AST for a simple Waskell program.

4.4. Validator

The validator is the module that checks the AST for semantic errors, such as type errors, undefined variables, and invalid expressions. The validator ensures that the source code is semantically correct before proceeding to the next stage. The validator outputs a symbol table that is used by the code generator.

This symbol table is a reduced form of the AST that contains only the necessary information for the code generator. The symbol table maps identifiers to scopes and is used to resolve variable references and enforce scoping rules. The symbol table is built by the symbol checker and is used by the type checker to check the types of expressions in the source code.

4.4.1. Symbol Table

The symbol table is a data structure that maps identifiers to scopes. The symbol table is used to resolve variable references and enforce scoping rules. The symbol table is built by the symbol checker and is used by the type checker to check the types of expressions in the source code.

Since Rust uses a borrow checker to enforce memory safety, recursive data structures where the elements reference each other in a cycle can be difficult to implement. In a data structure where expressions can have references to symbols in any part of the program (essentially a graph), the use of simple references would lead to borrowing issues since the lifetime of the references would be difficult to determine.

To solve this issue, Rust provides the `Rc` (reference-counted) smart pointer that allows for multiple ownership (read-only) of a value. The `Rc` type keeps track of the number of references to a value and deallocates the value when the number of references drops to zero. The `Rc` type is used to create a reference-counted symbol table that allows for multiple references to the same symbol across the AST.

Since we do have situations where the symbols are modified, the `RefCell` type is used to allow for runtime borrow checking (the compiler is not capable of checking the borrow rules at compile time in this case). This means however that the programmer has to ensure that only one mutable reference to the symbol table is present at a time.

The symbol table is implemented as a `HashMap` that maps identifiers to scopes. The `HashMap` is used to efficiently look up symbols by their identifiers and to enforce scoping rules. The symbol table is built by the symbol checker and is used by the type checker to check the types of expressions in the source code.

The types and the expressions in the AST are simplified to allow for easier type checking and code generation. The types are checked against the symbol table to ensure that they exist and are used correctly. The expressions are desugared to simplify the AST and make it easier to generate WebAssembly code.

As we will see later in the chapter, there also exists a type constructor table that maps type constructor names to their definitions. This table is used to check that the types used in the AST are defined and to resolve type constructor references.

Listing 44, Listing 45, Listing 46, and Listing 47 show the types used to represent the symbol table, types, and expressions in the Waskell compiler.

```rust
/// A type alias for the symbol table.
type SymbolTable = HashMap<String, Rc<RefCell<Symbol>>>;
/// A type alias for the type constructor table.
type TypeConstructorTable = HashMap<String, Rc<RefCell<TypeConstructor>>>;

/// A struct representing a symbol in the symbol table.
struct Symbol {
    /// The name of the symbol.
    name: String,
    /// The type of the symbol.
    ty: Type,
    /// The expression of the symbol. This can be `None` if the symbol is a
    /// data constructor for example.
    expr: Option<Expression>,
    /// The index of the data constructor in the type constructor. This can
    /// be `None` if the symbol is not a data constructor.
    data_constructor_idx: Option<usize>,
    /// The foreign annotations of the symbol.
    is_foreign: ast_gen::IsForeign,
}

/// A struct representing a type constructor in the type constructor table.
struct TypeConstructor {
    /// The name of the type constructor.
    name: String,
    /// The type variables of the type constructor. Each type variable is a
tuple of the type
    /// variable name and the type constructor name.
    type_vars: Vec<(String, String)>,
   /// The data constructors of the type constructor as a vector of symbols.
    data_constructors: Vec<Rc<RefCell<Symbol>>>,
}
```

Listing 44: The definition of the symbol table and type constructor table types.

```rust
/// An enum representing a type in the symbol table.
pub enum Type {
    /// The `Int` type.
    Int,
    /// The `Char` type.
    Char,
    /// The function type with a vector of types. The last type is the
    /// return type.
    Function(Vec<Type>),
    /// The `List` type with the type of the elements.
    List(Box<Type>),
    /// The `Tuple` type with a vector of types.
    Tuple(Vec<Type>),
    /// The `Unit` type.
    Unit,
    /// The type variable with the name of the variable and the name of the
    /// symbol in which the variable is defined.
    TypeVar {
        /// The name of the type variable.
        var_name: String,
        /// The name of the symbol in which the type variable is defined.
        ctx_symbol_name: String,
    },
    /// The custom type with the name of the type constructor and a vector
    /// of types for the type application.
    CustomType(String, Vec<Type>),
}
```

Listing 45: The definition of the types used to represent types of symbols in the symbol table.

```rust
/// An enum representing an expression in the symbol table.
pub enum Expression {
    /// The integer literal expression.
    IntLiteral(i32),
    /// The string literal expression.
    StringLiteral(String),
    /// The character literal expression.
    CharLiteral(char),
    /// The unit value expression.
    UnitValue,
    /// The symbol expression with a reference to the symbol in the symbol
    /// table.
    Symbol(Rc<RefCell<Symbol>>),
    /// The scope symbol expression with the name of the symbol. This is
    /// used for lambda abstractions and symbols defined in case
    /// expressions.
    ScopeSymbol(String),
    /// The function application expression with a vector of expressions and
    /// a boolean indicating whether the function is partially applied.
    FunctionApplication {
        /// The vector of expressions.
        params: Vec<Expression>,
        /// The boolean indicating whether the function is partially
        /// applied.
        is_partial: bool,
    },
    /// The tuple expression with a vector of expressions.
    Tuple(Vec<Expression>),
    /// The lambda abstraction expression with a vector of parameter names
    /// and an expression.
    LambdaAbstraction(Vec<String>, Box<Expression>),
    /// The case expression with a [`CaseExpression`].
    CaseExpression(CaseExpression),
}
```

Listing 46: The definition of the types used to represent expressions in the symbol table.

```rust
/// A struct representing a case expression in the symbol table.
struct CaseExpression {
    /// The input expression of the case expression.
    input_expr: Box<Expression>,
    /// The type of the input expression.
    input_ty: Box<Type>,
    /// The branches of the case expression.
    branches: Vec<CaseBranch>,
}

/// A struct representing a case branch in the symbol table.
struct CaseBranch {
    /// The pattern of the case branch.
    pattern: CaseBranchPattern,
    /// The expression of the case branch.
    branch_expr: Expression,
}

/// An enum representing a case branch pattern in the symbol table.
enum CaseBranchPattern {
    /// The integer literal pattern.
    IntLiteral(i32),
    /// The as pattern with the name of the parameter and an optional
    /// pattern.
    AsPattern(String, Option<Box<CaseBranchPattern>>),
    /// The constructor pattern with the data constructor and a vector of
    /// patterns.
    Constructor {
        /// A reference to the data constructor in the symbol table.
        data_constructor: Rc<RefCell<Symbol>>,
        /// The vector of patterns for the fields of the data constructor.
        fields: Vec<CaseBranchPattern>,
    },
    /// The tuple pattern with a vector of patterns.
    Tuple(Vec<CaseBranchPattern>),
    /// The unit pattern.
    Unit,
    /// The wildcard pattern.
    Wildcard,
}
```

Listing 47: The definition of the types used to represent case expressions in the symbol table.

4.4.2. Symbol checking

The symbol checker is responsible for building the symbol table that maps identifiers to scopes. The symbol table is used to resolve variable references and enforce scoping rules. The symbol checker traverses the AST and collects information about the identi-

fiers used in the source code. The symbol checker ensures that each identifier is defined before it is used and that the scoping rules of the language are enforced.

It works by first collecting all the data declarations and function declarations in the source code and adding them to a table of types. This table is later used by the symbol checker to check that the types used in signatures and data constructors exist. Since data constructors act as functions, when used in an expression, they are added to the symbol table as well.

The symbols for the data constructors have a field that contains the id of which data constructor they are within their type. This is used by the code generator to identify which data constructor it is in the memory representation.

The second step is to collect all the function signatures and add them to the symbol table but without the function bodies for now. This is done to allow for recursive functions. The function bodies are added to the symbol table after all the function signatures have been added.

The types used in the AST are checked against the table of types to ensure that they exist and in the case of type applications, that the type constructor is used correctly. During this step the string type is transformed into a list of characters to allow for pattern matching on strings.

The last step is to check the function bodies. The function bodies are checked against the symbol table to ensure that all the variables used in the function body are defined in the function signature. The symbol checker also does desugaring of the AST to further simplify the AST for the type checker and code generator.

The following steps are taken to parse a function body:

1. If the function body has no arguments, the expression is just transformed the simplified expression.
2. If the function body has arguments, only one declaration, and the arguments are just variables (no patterns), the expression is transformed into a lambda expression. The reason for this is that the lambda expression is the only context (for now) in which non-top-level symbols can be defined. What this means is that only the expression of the lambda can use theses symbols.
3. In all other cases, a lambda expression is created with n arguments (with generated names) where n is the arity of the function. The body of the lambda is a case expression with the pattern matching the arguments of the function. The case branches are the declarations of the function. To group the arguments (if there are more than one) a tuple is used.

4.4.3. Type checking

The type checker is responsible for checking the types of expressions in the source code. The type checker uses the symbol table to resolve variable references and enforce type rules. The type checker traverses the symbol table to check the types of expressions and ensure that they are used correctly.

The type checker iterates over all symbols in the symbol table and compares the types of the expression in the symbol with the type of the symbol itself. If the types do not match, the type checker reports a type error. If the symbol being checked has no expression, it is skipped.

To evaluate the type of an expression the type checker first does the following things on the top level expression of the symbol being checked:

1. It flattens all the function applications and types in the expression (e.g. `(f x) y` becomes `f x y`, `Int -> (Int -> Int)` becomes `Int -> Int -> Int`) and updates the expression in the symbol.
2. If the expression is a function application (the function was defined without any arguments), the expression is transformed into a lambda expression with arguments for each of the arguments in the function application and pushes these new arguments at the end of the function application.

   This is done to remove any function definition that is a partial application (e.g. `f = (+) 1` becomes `f x = (+) 1 x`).
3. If the expression is a lambda expression, the type of the symbol is checked to verify if the right amount of arguments are present in the lambda expression.
   3.1. If the lambda expression has more arguments than the function signature, the type checker reports an error.
   3.2. If the lambda expression has less arguments than the function signature, the lambda expression is modified to include the missing arguments. The expression inside of the lambda is then transformed into a function application with the arguments of the lambda expression (e.g. `f x = (+) x` becomes `f x y = ((+) x) y`, which gets flattened to `f x y = (+) x y`).

   Here again, the reason for this is to allow for partial application of top-level functions.
   3.3. All the arguments of the lambda expression are given the type from the function signature and passed to the function that checks the type of the expression inside of the lambda so that the type of the arguments is known when checking the expression.
4. If the expression is just a reference to another symbol, the type of the symbol simply gets returned. This would be a case where we define a function alias (e.g. `add = (+)`).

The rest of the type checking is done by the type checker recursively traversing the expression and checking the types of the subexpressions. The type checker ensures that the types of the subexpressions match the expected types and reports type errors if

the types do not match. The literals return the type of the literal (e.g. `IntLiteral(1)` returns `Int`), a lambda argument returns the type of the argument, a symbol returns the type of the symbol, etc.

When type checking for a function application, the type checker checks that the function being applied is a function type and that the arguments match the expected types. It then returns the return type of the function (another function if the function is partially applied).

When the type checker encounters a case expression, it checks if the pattern of the branches corresponds to the type of the input expression. If the pattern introduces new symbols, these symbols are added the scope of the type checking function. It finally also checks that the return types of all the branches are the same.

4.4.4. Parameterized types

The type checker also supports parameterized types. The type checker checks that the type variables used in the type signature are defined and that the type constructor is used correctly. The type checker also checks that the type variables are used consistently throughout the type signature.

In the type checker, the type variables are resolved during the type checking process. A `HashMap` is used to map the type variables to their types. The goal of this map is to track the assignments of the type variables and to ensure that the type variables are used consistently throughout the type signature.

When defining a type signature for a generic function like `map`, there are a few things to consider. When the generic arguments get used inside of the function body, the type variables can't take on a concrete type since it would limit the function to only work with that type.

Another thing to look out for is the fact that type variables can be assigned to another type variable, belonging to a different function. In such a case it is important to check that no two type variables from the same function are assigned to the same type variable because it would limit the function to only work with that type and not two different types.

Listing 48 shows an example of a type signature that would cause an error in the type checker.

The function `f` has two type variables `a` and `b`, which means this function can be used with any two types (e.g. `Int -> Char`, `Char -> Bool`, `Char -> Char`, etc.) but since the type variables are assigned to the same type variable, the function can only be used with one type. This would cause an error in the type checker.

The function `g` has a type variable `a` that is assigned to a concrete type `Int`. This would also cause an error in the type checker since the function can only be used with the type `Int`.

```
f :: a -> b
f x = x

g :: a -> a
g _ = 1
```

Listing 48: An example of a type signature that would cause an error in the type checker.

The algorithm for checking the type variables uses the `HashMap` mentioned earlier. The key of the map is the name of the type variable and the name of the symbol in which the type variable is defined. The value of the map is a union find data node (a node of a disjoint set data structure), from the `disjoint_sets` crate, that contains the type of the type variable and the concrete type the variable took on (can be empty if not assigned). The union find data structure is used to track the assignments of the type variables and to ensure that the type variables are used consistently throughout the type signature.

The disjoint set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It allows for two main operations: finding the representative of the set that an element belongs to and merging two sets together.

The reason for using the union find data structure is that it allows for the type variables to be assigned to other type variables and to track the assignments of the type variables. The union find data structure is used to find the root of the type variable and to ensure that the type variables are used consistently throughout the type signature. A group of type variables that are assigned to the same type variable is in this case a set. When one of the type variables in the set is assigned to a concrete type, the type of the concrete type is assigned to all the type variables in the set.

The algorithm for checking the type variables is as follows:

1. While checking the top-level expression of the symbol, the type checker initializes the map of type variables and passes it to the function that checks the type of the expression.
2. When the type checker compares two types, it checks:
   2.1. If one of the types is a type variable and the other not and if it is, it tries to assign the type of the type variable to the other type.
   2.2. If both types are type variables, it tries to merge the sets of the type variables.
   2.3. If both types are not type variables, it checks if they are the same type.
   2.4. If the types are nested (tuple, function, list, etc.), it recursively checks the types inside of the nested types.

When the type of the expression is returned to the top-level expression check, the type checker checks that all the type variables present in the type of the symbol are in different sets. If they are not, the type checker reports an error. It also checks that no type variable is assigned to a concrete type.

4.4.4.1. Generic function example

Figure 4 shows an example of the assignments that happen during the type checking of the generic function `map` that takes a function `f` and a list `xs` and applies `f` to each element of the list (see Listing 51). The type signature of the `map` function is `map :: (a -> b) -> List a -> List b`. The type variables `a` and `b` are used to represent the types of the elements in the input list and the output list.

During the type checking process, the type variable `b` is unified with the type variable of the `Nil` and `Cons` constructors (`Nil` and `Cons` have signatures `Nil :: List a` and `Cons :: a -> List a -> List a`). In Figure 4, the type variables are represented by the function they occur in and their name joined by a colon.

The blue circles represent the sets in which the type variables are assigned. The type variable `b` is assigned to the type variable of the `Nil` and `Cons` constructors and the type variable `a` is assigned to nothing.

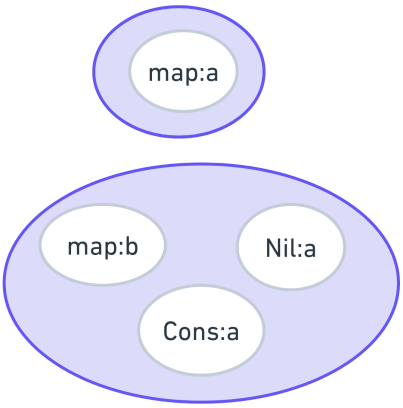


Figure 4: An example of the assignments during the type checking of `map`.

4.4.4.2. Example of using a generic function

Listing 49 shows an example of using the `map` function to double the elements of a list. The `doubleList` function takes a list of integers and doubles each element using the `map` function.

```
doubleList :: List Int -> List Int
doubleList = map ((*) 2)
```

Listing 49: An example of using the `map` function to double the elements of a list.

In this case the type variables of the `map` function are assigned to the types of the `doubleList` function. The type variables `a` and `b` are assigned to the type `Int`. Since the return type of the `map` function is `List a`, the type variable `a` is assigned to the type `Int` and the return type of the `doubleList` function is `List Int`.

Since the `doubleList` function has no type variables, there is no rule about the type variables being in the same set.

Figure 5 shows the assignments that happen during the type checking of the `doubleList` function. The concrete types are represented by a gray circle.

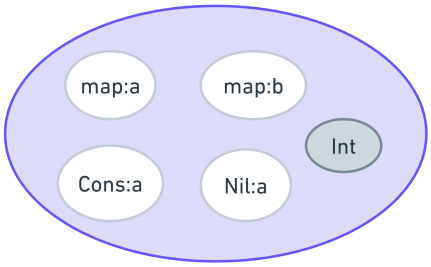


Figure 5: An example of the assignments during the type checking of `doubleList`.

4.4.5. Example output

Listing 50 shows the debug output of the symbol for the `map` function in the `prelude.wsk` file. The symbol table contains the type signature of the `map` function, the foreign annotation, and the expression of the function.

```
"map": Symbol {
    name: "map",
    ty: Function([
        Function([
            TypeVar { var_name: "a", ctx_symbol_name: "map" },
            TypeVar { var_name: "b", ctx_symbol_name: "map" },
        ]),
        CustomType("List", [
            TypeVar { var_name: "a", ctx_symbol_name: "map" }
        ]),
        CustomType("List", [
            TypeVar { var_name: "b", ctx_symbol_name: "map" }
        ]),
    ]),
    expr: Some(
        LambdaAbstraction([":map_0", ":map_1"], CaseExpression {
            input_expr: Tuple([
                FunctionParameter(:map_0),
                FunctionParameter(:map_1)
            ]),
            input_ty: Tuple([
                Function([
                    TypeVar { var_name: "a", ctx_symbol_name: "map" },
                    TypeVar { var_name: "b", ctx_symbol_name: "map" },
                ]),
                CustomType("List", [
                    TypeVar { var_name: "a", ctx_symbol_name: "map" },
                ]),
            ]),
            branches: [
                CaseBranch {
                    pattern: Tuple([Wildcard, Constructor(Nil, [])]),
                    branch_expr: Symbol(Nil),
                },
                CaseBranch {
                    pattern: Tuple([
                        AsPattern(f, None),
                        Constructor(Cons, [
                            AsPattern(x, None),
                            AsPattern(xs, None),
                        ]),
                    ]),
                    branch_expr: FunctionApplication([
                        Symbol(Cons),
                        FunctionApplication([
                            FunctionParameter(f),
                            FunctionParameter(x),
                        ], is_partial: false),
                        FunctionApplication([
                            Symbol(map),
                            FunctionParameter(f),
                            FunctionParameter(xs),
                        ], is_partial: false),
                    ], is_partial: false),
                },
            ],
        }),
    ),
    data_constructor_idx: None,
    is_foreign: NotForeign,
}
```

Listing 50: The debug output of the symbol for the `map` function.

For reference, Listing 51 shows the definition of the `map` function in the `lib/prelude.wsk` file.

```
map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

Listing 51: The definition of the `map` function.

4.5. Code Generator

The code generator is responsible for generating WebAssembly code from the AST. The code generator uses the symbol table to resolve variable references and enforce scoping rules. The code generator traverses the AST and generates WebAssembly instructions based on the expressions in the source code.

4.5.1. Wasm library

The Wasm library is a set of functions that are used during runtime to interact with the WebAssembly module. The WebAssembly library provides functions for printing strings, integers, and characters to the console, reading input from the console, and interacting with the memory of the WebAssembly module. It also contains helper functions to allocate memory, deallocate memory, create and execute thunk functions, partial applications, and encode data structures.

The Wasm library is implemented in the `lib/lib.wat` file. It is written in WebAssembly text format (WAT) and is compiled to WebAssembly binary format (Wasm) using the Binaryen `wasm-merge` tool. This Wasm module is also merged with the `lib/rust_lib.rs` file (which is compiled to Wasm). The rust library contain the functions for allocating memory, deallocating memory, and printing strings to the console. This part of the library is implemented in Rust because it is easier to write and maintain the code in Rust than in WebAssembly text format.

The allocation and deallocation of memory are done by creating a Rust vector with the specified size and then returning a raw pointer to the memory. Some unsafe Rust code is used to tell the borrow checker to forget about the vector so that the memory can be accessed by the WebAssembly module. This prevents the borrow checker from deallocating the memory when the vector goes out of scope. The deallocation of memory is done by converting the raw pointer back to a vector and then dropping the vector.

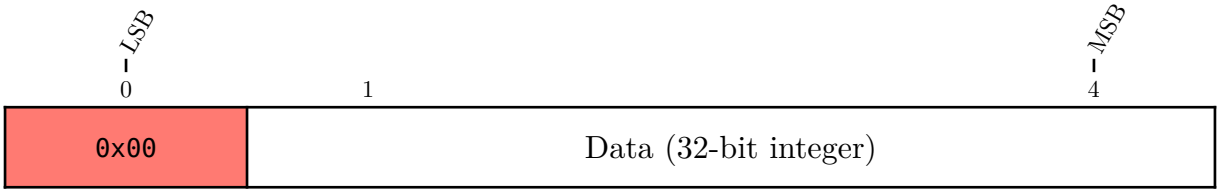
The print function is implemented by converting a pointer to a Waskell string to a Rust string and then printing the Rust string to the console (more about the representation of strings in Section 4.5.1.1).

All the functions in the Wasm library are exported with names that have a colon in front of them (e.g. `:make_val`, `:make_thunk`, etc.). This is done to prevent name collisions with the user-defined functions and types in the WebAssembly module.

4.5.1.1. Memory representation

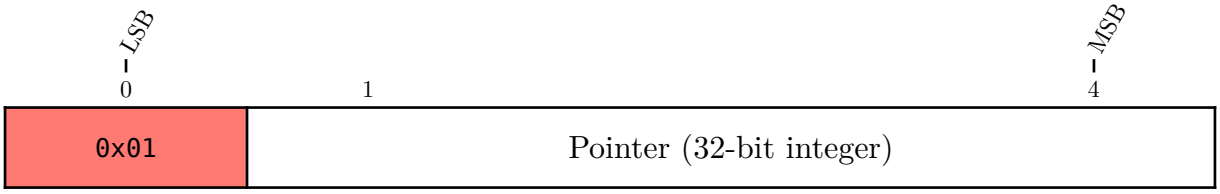
Wasm only supports integers and floats as primitive types. This means that strings, characters, and other data structures need to be encoded into integers and floats before being passed to the Wasm module. The memory representation of strings, characters, and data structures is done using a tagged union representation.

Byte Field 1 shows the memory representation of a literal value in the Wasm module. The first byte of the memory representation is used as a tag to indicate that the value is a literal. The tag is followed by the data of the value as a 32-bit integer (the numbers on top of the diagram represent the bytes in the memory representation).



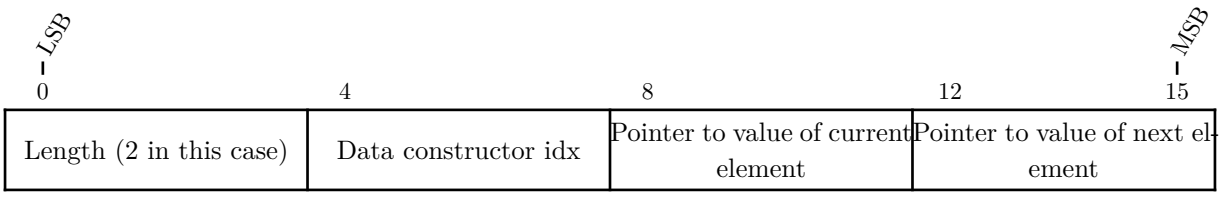
Byte Field 1: The memory representation of a literal value.

Byte Field 2 shows the memory representation of a Waskell data structure in the Wasm module. The first byte of the memory representation is used as a tag to indicate that the value is a data structure. The tag is followed by a pointer to the data structure in the memory.



Byte Field 2: The memory representation of a Waskell data structure.

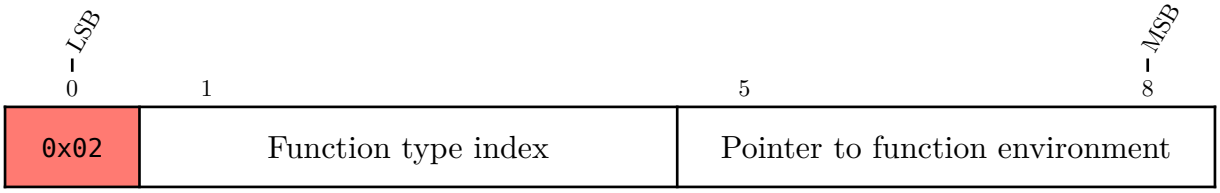
Behind this pointer lies a structure that contains the number of fields the current data constructor has, the tag to identify the data constructor (e.g. 0 for `Nil` and 1 for `Cons`), and the pointers to the value for each field of the data constructor. Byte Field 3 shows the memory representation of the `Cons` data constructor in the Wasm module.



Byte Field 3: The memory representation of the `Cons` data constructor.

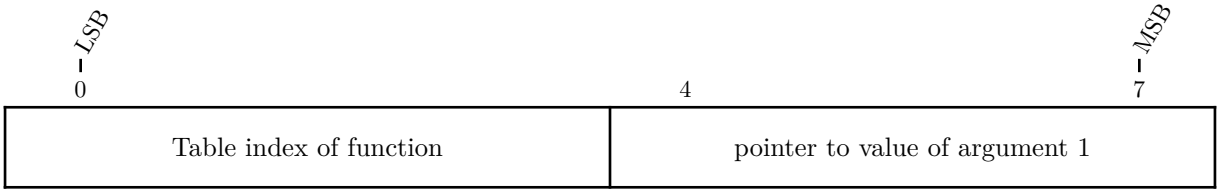
The thunk is a structure used to represent a function that has not been evaluated yet. The thunk is used to implement lazy evaluation in Waskell. The memory representation for a thunk is as follows: the first byte is a tag to indicate that the value is a thunk, the next 4 bytes are the index of the type of the function that the thunk represents (necessary for the apply function), and the last 4 bytes are the pointer to the function environment (the pointer to the function and the arguments).

Byte Field 4 shows the memory representation of a thunk in the Wasm module.



Byte Field 4: The memory representation of a thunk.

The environment of a thunk is a structure that contains the pointer to the function and the arguments of the function. To be able to do an indirect call in Wasm, the module uses a table of functions that are called by index. The first field of the environment is the table index of the function that the thunk represents and the rest of the fields are the arguments of the function. Byte Field 5 shows the memory representation of the environment (the function only has one argument in this case) of a thunk in the Wasm module.



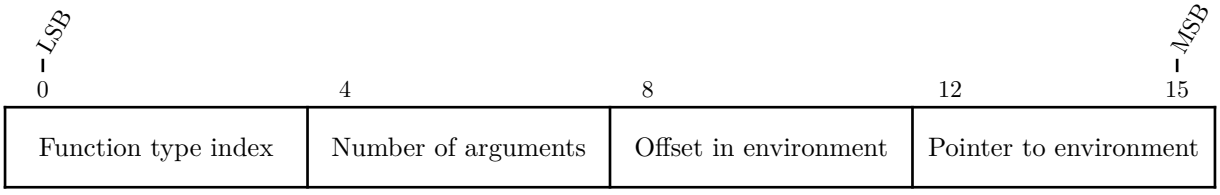
Byte Field 5: The memory representation of the environment of a thunk.

The memory representation for a partially applied function is a little different than the others. We know for a fact that when a value is evaluated it will never be a partially applied function. This means that there is no need to have a tag to indicate that the value is a partially applied function. Partially applied functions are temporary value that store the function and the arguments that have been applied to it to later be transformed into a thunk.

The structure of a partially applied function is as follows:

- The first 4 bytes are the type index of the function that the partially applied function represents.
- The next 4 bytes are the number of arguments that still need to be applied to the function.
- The next 4 bytes represent the offset in the environment where the next argument should be stored.
- The rest of the bytes are a pointer to the environment of the function (the environment is the same as the one used for thunks but not filled completely).

Byte Field 6 shows the memory representation of a partially applied function in the Wasm module.



Byte Field 6: The memory representation of a partially applied function.

4.5.1.2. Runtime functions

The runtime functions are Wasm functions that are used during the execution of the WebAssembly module to interact with data structures and perform operations on them.

Here is a list of the runtime functions:

- `:make_val` - Takes a tag (0 for a literal, 1 for a data structure) and a either a 32-bit integer (for literals) or a pointer (for data structures), allocates memory for the value, and returns a pointer to the value.
- `:make_thunk` - Takes a function type index and a pointer to the function environment, allocates memory for the thunk, and returns a pointer to the thunk.
- `:make_env` - Takes a size, allocates memory for the environment, and returns a pointer to the environment (it is up to the caller to fill the environment with the correct values).
- `:make_pap` - Takes a function type index, the number of arguments the function needs in total, the number of arguments that have been applied and an environment pointer, allocates memory for the partially applied function, and returns a pointer to the partially applied function.
- `:add_to_pap` - Takes a pointer to a partially applied function, and a pointer to a value and adds the value to the environment of the partially applied function (note that the pap gets copied to a new location in memory to avoid mutating the original pap).
- `:make_thunk_from_pap` - Takes a pointer to a partially applied function, and a pointer to an environment (containing the rest of the arguments) and transforms the partially applied function into a thunk.

- `:full_eval` - Takes a pointer any value and evaluates it fully. If the value is a thunk, it evaluates the thunk and replaces the thunk with the result of the evaluation. If the value is a data structure, it evaluates all the fields of the data structure recursively. If the value is a literal, it does nothing.
- `:eval` - Takes a pointer to a value and evaluates it partially. If the value is a thunk, it evaluates the thunk and replaces the thunk with the result of the evaluation. If the value is a data structure or a literal, it does nothing.
- `:apply` - Takes the index of the type of the function that the thunk represents, a pointer to the function environment and evaluates the function with the arguments in the environment.

  The type index is necessary since Wasm needs to know the type of the function to be able to call it, even when doing an indirect call. The `:apply` has a big switch statement that makes different calls to the functions in the runtime depending on the type of the function. This function is generated by the code generator because it needs to know all the types of the functions that are used in the program.
- The last set of functions are artithmetic functions that are used to perform operations on integers. They wrap the WebAssembly instructions for adding, subtracting, multiplying, etc. integers so they can be called from the Waskell code. The functions are `+`, `-`, `*`, `quot`, `rem`, `==`, `/=`, `<`, `<=`, `>`, `>=`, `compare` and `intToChar`.

4.5.2. Wasm Encoder

The `wasm_encoder` crate is used to encode the WebAssembly instructions into a binary format. It provides a set of functions for encoding the WebAssembly instructions and writing them to a buffer.

The crate works by having a type for every section of the WebAssembly module. The needed information for the section is stored and all the sections are combined into a `Module` type. The `Module` type is then encoded into a binary format using the `finish` function.

The sections of the WebAssembly module are (in order):

- The custom section - This section is used to store information that is not part of the WebAssembly standard (e.g. names of functions, debug information, etc.).
- The type section - This section is used to define the types of every entity (functions, tables, memories, etc.).
- The import section - This section is used to define the imports.
- The function section - This section is used to store the function signatures of the functions.
- The table section - This section is used to define the tables.
- The memory section - This section is used to define the memory.
- The global section - This section is used to define the global variables.
- The export section - This section is used to define the exports.
- The start section - This section is used to define the start function.

- The element section - This section is used to define the elements of the tables.
- The code section - This section is used to define the function bodies of the functions.
- The data section - This section is used to define the data of the memory.
- The data count section - This section is used to define the number of data segments in the data section.

In this project, only the type section, import section, function section, table section, export section, element section, and code section are used. The other sections are not needed.

Something to note is that the declaration of the function signature and the function body are separated in the WebAssembly module. This means that the order in which the functions signatures and the function bodies are defined in the WebAssembly module is important. The `wasm_encoder` crate relies on the order of the functions in the module to generate the correct WebAssembly module. Since this can make the code generation more complex, wrapping parts the `wasm_encoder` crate in a more user-friendly API would be beneficial.

Two different wrappers (located in file `src/code_gen/encoder_wrapper.rs`) are used to generate the WebAssembly module. The first wrapper is used for the type section. A `HashMap` is used to store the types of the functions and their index in the type section. At the end of the code generation, the type section is generated by iterating over the `HashMap` and adding the types to the type section in order of their index. A benefit of using the `HashMap` is that it allows for easy lookup of the type of a function when used in the code generation to avoid duplication of types.

The second wrapper is used to keep track of the indices of the functions in the function section and in the table section. The order of the functions in the table section is important because the indices are used to call the functions in the WebAssembly module. The wrapper keeps track of the indices of the functions and their place in the table section. At the end of the code generation, this wrapper creates the table section, function section, import section, and element section in the correct order.

The indices of imported functions are always the first indices in Wasm since the import section comes before the function section. To make sure that there is no issue with a function getting imported after other functions have been defined (this would shift the indices of the functions), the second wrapper works in 2 stages. In the first stage, it can only be used to import functions. In the second stage, it can only be used to define functions. This way the indices of the imported functions are always the first indices.

4.5.3. Translation of the Symbol Table

Figure 6 shows an overview of the code generation process. The code for the code generation is located in the `src/code_gen/wasm_generation.rs` file.

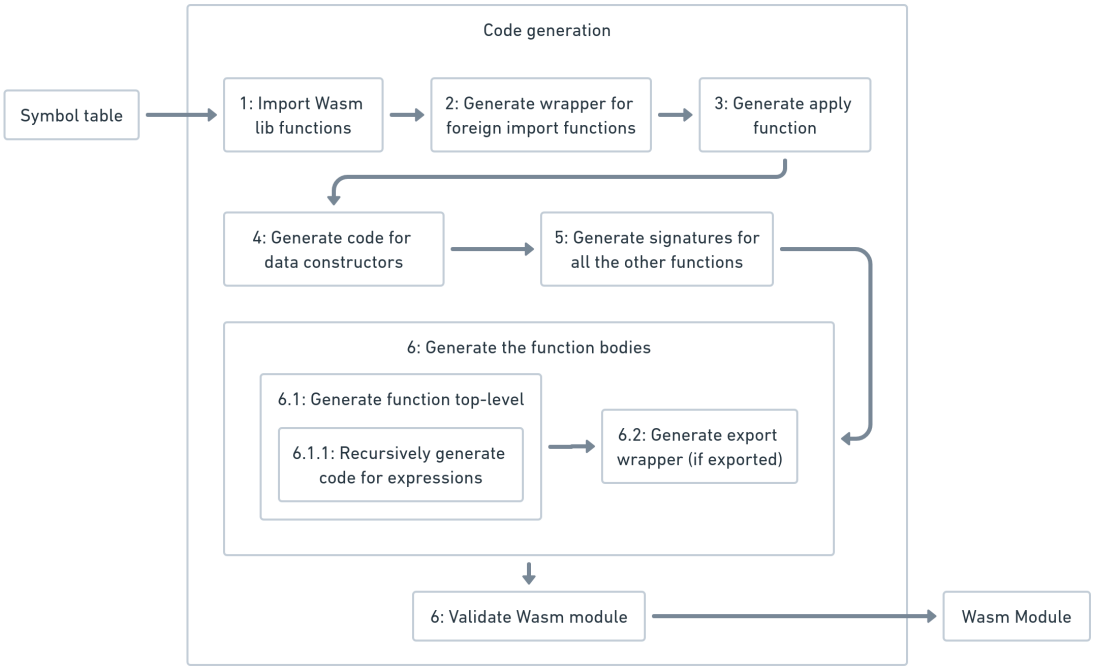


Figure 6: An overview of the code generation process.

The symbol table is translated into the WebAssembly module by iterating over all the symbols in the symbol table and generating the WebAssembly instructions for each symbol. The symbol table is used to resolve variable references and enforce scoping rules during the code generation process.

4.5.3.1. Foreign imports and exports

The functions that are foreign imports or exported (e.g. functions given by the host environment or functions that will get used by the host environment) are treated specially.

The imported functions receive a wrapper function that evaluates (with `:full_eval`) the arguments of the function passed by a call from another Waskell function. The result of the evaluation is then wrapped into a Waskell value (so a literal or a data structure using `:make_val`). This reduces the overhead the host environment has to do to pass a function to the WebAssembly module. When the imported function is called from the Waskell code, the wrapper function is called instead. The only imported functions that don't get wrapper functions are the ones annotated with `"lib"` (see Section 3.1.10) since these functions are part of the Wasm library and already handle the evaluation of the arguments and result.

The exported functions are wrapped in a function that wraps the arguments of the function (using `:make_val`) and then calls the Waskell function. The result of the function is then evaluated (with `:full_eval`) and returned to the host environment. The

function that is exported is the wrapper function and not the actual function. This is also done to reduce the overhead of the host environment.

The notable exception to this is the exported functions that are annotated with `"unevaluated"` (see Section 3.1.10). These functions are used in the case where we want to export a function that creates a complex data structure. The way the `:full_eval` function works currently is that it recursively replaces the fields of a data structure with the direct values of the fields (integers are left as they are). So when the host environment calls a function that returns a complex data structure, to later pass it to another Waskell function, the `:make_val` doesn't transform it into a valid Waskell value.

This annotation is a temporary solution to this problem. In the future, the `:make_val` function could be modified to recursively wrap the fields of a data structure with- `:make_val` so that the data structure can be passed to another Waskell function without any issues.

4.5.3.2. Function bodies

Each function generates a new thunk from the given arguments, who themselves could be thunks. When a specific thunk is evaluated (with `:full_eval` or `:eval`), the function inside the thunk is called with the arguments in the environment of the thunk. As previously mentioned, the function internally makes a new thunk from the arguments and the function body. Since the implementation of the evaluation functions loops until a non-thunk value is found (in the case of `:full_eval` the fields of a data structure are also evaluated), the function will be called until the result is a non-thunk value.

The function bodies are generated by traversing the expression of the symbol corresponding to the function. First code generator first looks at the top-level expression of the function and then recursively generates the code for the expressions inside the top-level expression. The code generator uses the symbol table to resolve variable references and enforce scoping rules during the code generation process.

The following top-level expressions are handled by the code generator:

- **Lambda abstractions** - The code generator takes the scope of the lambda abstraction and passes it to the function that generates the code for the expression inside the lambda abstraction. Then it takes the result of the code generation function and returns it.
- **Function applications** - The code generator simply calls the function to generate the code for the expression and returns the result.
- **Symbols** (function alias) - The code generator generates calls to the symbols in question, with the arguments of function, directly.
- **All the literals** - The code generator generates the literals and the corresponding WebAssembly instructions. In the case of the string literals, the code generator generates a `List` data structure with the characters of the string

The function that generates the code for the expressions is recursive and generates the code for any expression that is passed to it. It returns an index to a local variable of

the function that contains the result of the expression. So if an expression has sub-expressions, the code generator generates the code for the sub-expressions and stores the result in a local variable. These local variables are then used in the parent expression to generate the final result.

The following expressions are handled by the code generator:

- **Literals** - The code generator generates the literals using a call to the `:make_val` function.
- **Symbols** (possibly a partial application) - The code generator generates checks the arity of the function and generates a thunk or a partial application if the function is not fully applied.
- **Tuple** - The tuples have automatically generated data constructors which are used to generate the data structure.
- **Function application** - has two cases:
  - if the function is a **symbol** - The code generator generates a thunk from the function and the arguments and returns the pointer to the thunk. If the function application is a partial application, the code generator generates a partial application and puts the arguments in the environment of the partial application.
  - if the function is a **argument** of the parent lambda abstraction - In this case we are dealing with a function as value. In the code generator function passed as values are always partial applications. So in the case that the function application partial, the code generator generates a new partial application based on the old one and the new arguments. And of course, if the function application is not partial, it generates a applies the remaining arguments and generates a thunk from the lambda argument.
- **Case expression** - The code generator generates a block for each branch of the case expression. Each branch makes checks on the input expression (evaluating the expression using `:eval` if necessary) and if a check fails, it jumps to the next branch. If none of the checks fail, expression of the branch is returned.

  Here is the logic of the code generation for the different branches:
  - If the pattern is a **wildcard**, the check always succeeds and the expression of the branch is returned.
  - If the pattern is a **"as" pattern**, the code generator generates a new local variable and assigns the value of the input expression to the local variable. Then the code generator recursively generates the branch code for the optiona rest of the pattern (e.g. "as" pattern can have the form `a @ (Cons _ xs)`).
  - If the pattern is a **constructor pattern** (tuple also included), the code generator evaluates the input expression and checks if the index of the data constructor matches the data constructor of the pattern. If there are fields in the pattern, the code generator recursively generates the code for the patterns of the fields.
  - If the pattern is a **literal pattern**, the code generator evaluates the input expression and checks if the value of the input expression matches the literal value of the pattern.

4.6. Standard library

The standard library of Waskell is implemented in the `lib/prelude.wsk` file. The standard library provides a set of functions and types that are commonly used in functional programming. The design of the standard library is based on the Haskell standard library documentation [1]. To see the full list of functions and types in the standard library, refer to Section 3.2.

In order to include the standard library in all Waskell programs, the code in the `lib/prelude.wsk` file is concatenated with the input source code before being passed to the lexer. This ensures that the standard library functions and types are available to the user without the need for explicit imports. An added benefit of this approach is that the standard library functions and types are optimized by the WebAssembly runtime along with the user-defined functions. It also allows for the standard library to be easily extended and modified by the user.

A disadvantage of this approach is that the standard library functions and types are included in every WebAssembly module generated by the compiler, even if they are not used by the user. This can lead to larger WebAssembly modules and slower execution times due to the increased code size. To mitigate this issue, the `waskellc` compiler includes a `--no-merge` flag that disables the merging of the WebAssembly module with the wasm lib file. This allows the user to manually include the wasm lib file in their WebAssembly runtime and share it across multiple WebAssembly modules.

If in the future the standard library becomes larger and more complex, it may be beneficial to split it into multiple files and modules to improve maintainability and organization. In this case, the introduction of a module system in Waskell would be necessary to allow the user to import specific modules from the standard library. But in the current state of the language, the standard library is kept simple and concise to avoid unnecessary complexity.

4.7. Testing and CI/CD

The Waskell compiler is tested using automated tests on the functions of the standard library. The tests are written in Waskell itself and are located in the `waskellc/examples/prelude_test.wsk` file. The reason for only testing the standard library functions is that the compiler components (lexer, parser, symbol checker, type checker, and code generator) are difficult to test in isolation due to their interdependence and the implementation of such tests would be complex and time-consuming. By testing the standard library functions, the correctness of the compiler components is indirectly verified since the standard library functions exercise the compiler components during compilation.

Inside of the `src/main.rs` file, there is a unit test that compiles the `prelude_test.wsk` file and runs the generated WebAssembly module with the wasmtime runtime. The unit test checks the exit status of the runtime and prints the output of the runtime to the

console. If the runtime exits with a non-zero status, the unit test fails and the output of the runtime is printed to the console for debugging purposes.

The tests in the `prelude_test.wsk` work by using a helper function `test` that takes a function for comparing the expected and actual values, a function to turn a value of that type into a string, the expected value, the actual value, and a string representing the name of the test. The `test` function then compares the expected and actual values using the comparison function and prints if the test passed or failed. In the case of a failure, the expected and actual values are printed to the console for debugging purposes.

Listing 52 shows an example of a test in the `prelude_test.wsk` file.

```
test :: (a -> a -> Bool) -> (a -> String) -> a -> a -> String -> ();
test comparator printer x y name = if'
  (comparator x y)
  (print (name ++ " passed"))
  (error (name ++
      " failed, expected: " ++
      (printer y) ++
      " but got: " ++
      (printer x))
  );

mathTest :: ();
mathTest = test (==) intToString (add3 (3 * 4) (add3 1 2 (-4 `quot` 2)) (10
- 3)) 20 "mathTest";
```

Listing 52: An example of a test in the `prelude_test.wsk` file.

The Waskell compiler uses GitLab CI/CD to automate the testing and deployment of the compiler. The CI/CD pipeline is defined in the `.gitlab-ci.yml` file and consists of two stages: `check` and `build`.

The `check` stage runs a pre-commit hook that:

- Formats the code using `cargo fmt` to ensure consistent code style.
- Checks the code for warnings and errors using `cargo check`.
- Lints the code using `clippy` to enforce best practices and idiomatic Rust code.
- Runs the unit tests using `cargo test` to verify the correctness of the standard library functions.

The `build` stage compiles the Waskell compiler using `cargo build` and releases the compiler as a binary artifact for different platforms (Linux, macOS, Windows). The binary artifacts are then available for download as a release on the GitLab repository (only triggered by a tag push).

4.8. Challenges

The development of the Waskell compiler has been challenging due to the complexity of the WebAssembly runtime and the limitations of the WebAssembly language. The following sections describe some of the challenges encountered during the development of the compiler and the solutions that were implemented to overcome them.

4.8.1. Wasm encoder

The `wasm_encoder` crate relies on the order of the functions in the WebAssembly module to be correct. This means that the order in which the functions are defined in the WebAssembly module is important for the correct generation of the WebAssembly module. The `wasm_encoder` crate does not provide a way to reorder the functions in the WebAssembly module, which makes it difficult to maintain the order of the functions in the module. To work around this limitation, the code generator uses wrappers to keep track of the indices of the functions in the module and ensure that the order of the functions is correct (see Section 4.5.2).

Before the wrappers were implemented, the order in which the functions were compiled was important. Since the symbol table is a `HashMap` the order in which the iterator gave the entries was unpredictable, the compilation order changed all the time. So sometimes the code generation would fail because the function was not defined yet, or even worse a function signature would associated with the wrong function body.

4.8.2. Merging the wasm-lib

Since merging Wasm module is not a quite common task, there are not many tools that can do it. The only tool that was found was the `wasm-merge` tool from the Binaryen project. There is not much documentation on how to use the tool and how it actually merges the modules.

While trying to merge the Wasm library with the Wasm module, it was discovered that the `wasm-merge` overwrites the function table of the Wasm library with the function table of the Wasm module. The first fix was import the function table if the Wasm module.

And since the Wasm module needs the Wasm library to be able to run, the entry point of the compiler was changed to be able to call the `wasm-merge` tool and automatically merge the Wasm library with the Wasm module. This way the user doesn't have to worry about merging the Wasm library with the Wasm module themselves.

4.8.3. The apply function

As previously mentioned, Wasm does not support indirectly function calls without knowing the type of the function. This means that the `:apply` function needs to know

the type of the function that it is calling. The `:apply` function is generated by the code generator and needs to know all the types of the functions that are used in the program. This is done by generating a big switch statement that makes different calls to the functions in the runtime depending on the type of the function.

In earlier versions of the compiler, the `:apply` function was hardcoded in the Wasm library. This was not a good solution since the function would have to be updated every time a new function type was added to the language. But for the time it was enough since there were only few different function types. Instead of receiving the id of a type, it received the number of arguments the function had (back then all function without exception returned an integer).

As soon as the importing and exporting of functions was implemented, some functions could have no return type. This was a problem since the `:apply` function would always return an integer. The solution to pass the type index of the function to the `:apply` function instead of the number of arguments. Now it was also becoming clear that the `:apply` function would have to be generated by the code generator. For each function type of the Wasm module, a new case is automatically generated in the `:apply` function.

4.8.4. Exporting functions for creating recursive data structures

When exporting functions that create recursive data structures, the `:full_eval` function would replace the fields of the data structure with the direct values of the fields. This is because the `:full_eval` function recursively evaluates the fields of the data structure and replaces the fields with the evaluated values. This is a problem when exporting functions that create recursive data structures since the data structure would be transformed into a non-recursive data structure.

To solve this issue, the exported functions that create recursive data structures are annotated with `"unevaluated"`. This annotation is a temporary solution to the problem and allows the exported functions to create recursive data structures without being evaluated by the `:full_eval` function. In the future, the `:make_val` function could be modified to recursively wrap the fields of a data structure with `:make_val` so that the data structure can be passed to another Waskell function without any issues.

4.8.5. Over-applied functions

The type checker does not handle "over-applied" functions correctly. Listing 53 shows an example of what is meant by "over-applied" functions.

```
const :: a -> b -> a;
const x _ = x;

id :: a -> a;
id x = x;

main :: Int;
main = const id 1 2;
```

Listing 53: An example of an "over-applied" function.

As we can see in the example, the `const` function is applied to the `id` function and two integers. This should be cause no problem since `const` returns the first argument and in this case the first argument is the function `id`. The second argument of `const` is the integer `1`, which is ignored. So this means that the `2` is applied to `id` and the result should be `2`. But the type checker does not handle this case correctly and gives an error.

The reason for this is that the function type is internally uncurried to simplify the type checking and the code generation. Every partial application is manually checked by comparing the number of arguments that have been applied to the number of arguments the function needs. This means that in this case the `const` function is seen as a function that takes 3 arguments and not as a function that takes 2 arguments and returns a function that takes 1 argument.

This issue has not yet been resolved and is a limitation of the current implementation of the compiler. The reason for why it has not been resolved is that it would require a significant redesign of the type checker and the code generator to handle this case correctly. The type checker would need to be able to handle functions that return other functions and the code generator would need to generate the correct WebAssembly instructions for these functions.

4.8.6. Issue with scanr and pattern matching

The `scanr` function does not work correctly in the current implementation of the compiler. The `scanr` function is used to apply a function to each element of a list from right to left and accumulate the results. The issue with the `scanr` function it the values of the list are not evaluating correctly. In the test the function is supposed to return a list of integers, which it does, but the integers are just the integer value corresponding to a memory address.

This signifies that there is an edge case where the partial evaluation of the list is not working correctly. The issue is most likely in the code generation of the `scanr` function.

The code generation of the `scanr` function is quite complex since it involves generating a recursive function that applies the function to each element of the list and accumulates the results. The issue could be in the recursive function that generates the list of results or in the code that generates the thunk for the recursive function.

Section 5
Conclusion

All the objectives of the thesis were achieved. The Waskell compiler is capable of compiling a subset of Haskell to WebAssembly and running it in a WebAssembly runtime. The compiler is able to handle the core features of Haskell such as parametric polymorphism, pattern matching, and lazy evaluation. The compiler is also able to generate WebAssembly code that can be run in a WebAssembly runtime and produce the expected output. Embedding the WebAssembly runtime in different programming languages was also successful.

The only thing that had to be skipped was the benchmarking of the compiler. This was due to the time constraints of the thesis and an underestimation of the writing of the thesis. The benchmarking would have been done by comparing the performance of the Waskell compiler with the GHC compiler on a set of benchmarks. The benchmarks would have tested the performance of the compiler on different types of programs and measured the execution time and memory usage of the generated WebAssembly code.

5.1. Future work

The Waskell compiler is a work in progress and there are several areas that could be improved and extended in the future. The following sections outline some of the potential future work that could be done to enhance the Waskell compiler.

5.1.1. Garbage collection

Currently, the Waskell compiler allocates memory for the heap and stack but does not deallocate memory when it is no longer needed. This can lead to memory leaks and inefficient memory usage. Implementing a garbage collector in the WebAssembly runtime would allow the compiler to automatically manage memory and free up memory that is no longer in use. A garbage collector would improve the performance and reliability of the compiler by preventing memory leaks and reducing memory usage.

There are different ways to approach garbage collection in WebAssembly. The first way would be to rely on the garbage collection proposal for WebAssembly that is currently in development (see Section 5.1.1). The proposal aims to add garbage collection support to WebAssembly and provide a standard interface for garbage collectors to interact with WebAssembly modules. By using the garbage collection proposal, the Waskell compiler could integrate with existing garbage collectors and benefit from automatic memory management.

This first approach would be the most efficient and reliable way to implement garbage collection in WebAssembly. However, the garbage collection proposal is still in development and is not widely supported by WebAssembly runtimes. As an alternative, the Waskell compiler could implement a custom garbage collector in Rust that manages memory in the WebAssembly runtime. The custom garbage collector would be responsible for tracking memory allocations and deallocations and freeing up memory that is no longer in use. While this approach would be less efficient and reliable than using the garbage collection proposal, it would provide a way to implement garbage collection in WebAssembly without relying on experimental features.

Implementing custom garbage collection in the WebAssembly runtime would be a challenging task that requires a deep understanding of memory management and garbage collection algorithms. A possible approach would be to implement a simple mark-and-sweep garbage collector that traverses the heap and identifies unreachable objects. For this there would be a need to make a table that tracks the references to the heap objects and every time a reference is created or removed, the table would be updated. The garbage collector would then free up memory for the unreachable objects and compact the heap to reduce fragmentation.

5.1.2. Fix remaining issues

There are several issues and limitations in the current implementation of the Waskell compiler that need to be addressed. These issues are explained in more detail in the challenges section (Section 4.8). Some of the issues include:

- The `scanr` function does not work correctly which could indicate some untested edge cases in the code generator.
- The type checking of parametric polymorphism is not completely tested and could have some edge cases that are not handled correctly.
- Generic functions that return another function do not work correctly in the type checker because the function type is internally uncurried.

5.1.3. Layout rules

The layout rules of the language could be improved to allow for more flexibility and expressiveness in the code. Currently, the layout rules are quite strict and require explicit indentation to define blocks of code. By relaxing the layout rules, the compiler could allow for more flexible code formatting and make the language more user-friendly. For example, the compiler could allow for optional indentation and use braces to define blocks of code. This would make the language more familiar to programmers coming from other languages and reduce the learning curve for new users.

The reason for why they are not currently implemented is that the layout rules are a core feature of the language and changing them would require a significant redesign of the parser and lexer. The layout rules are also a key part of the language's design and contribute to its readability and elegance. However, by relaxing the layout rules, the

compiler could provide more flexibility and expressiveness in the code and make the language more user-friendly.

5.1.4. Refactoring

The codebase of the Waskell compiler could be refactored to improve readability, maintainability, and performance. The codebase is currently quite large and complex, with many interdependent components that interact with each other. By refactoring the codebase, the compiler could be split into smaller modules and components that are easier to understand and maintain. This would make it easier to add new features, fix bugs, and optimize the compiler.

5.1.5. More optimizations

The Waskell compiler could be optimized to generate more efficient WebAssembly code and improve the performance of the generated code. There are several optimizations that could be implemented in the compiler, such as:

- Constant folding: The compiler could evaluate constant expressions at compile time and replace them with their result. This would reduce the amount of computation needed at runtime and improve the performance of the generated code.
- Dead code elimination: The compiler could remove code that is never executed from the generated WebAssembly module. This would reduce the size of the WebAssembly module and improve the performance of the runtime.
- Tail call optimization: The compiler could optimize tail-recursive functions to avoid stack overflow errors and improve the performance of recursive functions.
- Inline expansion: The compiler could inline small functions into their callers to reduce function call overhead and improve the performance of the generated code.
- Call graph reduction: The GHC compiler uses a call graph reduction technique to evaluate functions in a lazy language. This technique could be implemented in the Waskell compiler to improve the performance of lazy evaluation and ensure that no unnecessary computations are performed.

5.1.6. More features

Right now the language is quite limited in terms of features and could be extended to support more advanced programming concepts. The main feature that would improve the language is the addition of type classes. Type classes are a powerful feature of Haskell that allows for ad-hoc polymorphism and type constraints. By adding type classes to the language, the compiler could support more advanced type checking and code generation techniques and enable more expressive and concise code.

With type classes, the compiler could support Monads, Functors, Applicatives, and other advanced programming concepts that are commonly used in functional programming. Type classes would also allow for more advanced type inference and type checking

techniques that could improve the performance and reliability of the compiler. Since IO is done with the IO monad in Haskell, the addition of type classes would also allow to implement pure IO in Waskell.

5.2. Personal opinion

In this thesis, I have presented the design and implementation of a compiler for a functional programming language that targets WebAssembly. The compiler is implemented in Rust and consists of several components that work together to parse, type-check, and compile functional code to WebAssembly. The functional language is a subset of Haskell and includes features such as pattern matching, polymorphism, and lazy evaluation. The compiler translates the functional language constructs into WebAssembly instructions and generates a WebAssembly module that can be executed by a WebAssembly runtime.

I have demonstrated the implementation of the compiler components, such as the lexer, parser, symbol checker, type checker, and code generator. I have also shown the design of the standard library, which includes functions for working with booleans, numbers, lists, tuples, and ratios. The standard library is based on the Haskell standard library and provides a set of functions and types that are commonly used in functional programming.

This thesis has allowed me to explore and learn about the design and implementation of compilers, programming languages, and WebAssembly. I have gained experience in working with Rust, parsing, type checking, code generation, and WebAssembly. I have also learned about functional programming concepts, such as pattern matching, polymorphism, and lazy evaluation. It was overall a very fascinating and educational experience.

If I were to start this project again, with the knowledge I have now, I would do the following thing differently.

I would start by designing a minimal standard library that I could implement and test early on. This would allow me to test the compiler components with real code and ensure that the standard library functions are working correctly. After implementing the minimal standard library, I would gradually add more functions and types to cover a wider range of functionality. This approach is different from the one I took in this project, where I only did a minimal code example to test the compiler components. When I started implementing the standard library, I encountered several edge cases and issues that could have been avoided with a more incremental approach.

I intend to continue working on this project and improving the compiler and standard library. I plan to fix the remaining issues, refactor the codebase, and add more features to the language and standard library. I also plan to write more tests, improve the error messages, and optimize the generated WebAssembly code. I find this project very interesting and challenging, and I look forward to continuing to work on it in my free time.

5.3. Acknowledgements

I would like to thank my supervisors, Dr. Jacques Supcik and Dr. Serge Ayer, for their guidance, support, and feedback throughout this project. They helped me navigate the challenges of this project despite its large scope and complexity.

I would also like to thank Dr. Baptiste Wicht and Mr. Valentin Bourqui for their help and advice throughout this project as external experts. Their expertise in project management, and software development was invaluable to me.

Finally, I would like to thank my family and friends for their encouragement and understanding during this time. Their support has been invaluable to me, and I am grateful for their patience and encouragement.

Declaration of honor

In this project, we used generative AI tools, namely GitHub Copilot for coding and Claude AI for paraphrasing. Copilot was employed as an advanced autocomplete feature, but it did not generate a significant portion of the project. I, the undersigned Noah Godel, solemnly declare that the submitted work is the result of personal effort. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and author citations have been clearly acknowledged.

Glossary

Claude AI: Claude AI is a LLM based AI chat bot by Anthropic. 99

GitHub Copilot: GitHub Copilot is an AI pair programmer that uses AI to make context related code suggestions. 99

***HEIA-FR* – Haute école d'ingénierie et d'architecture de Fribourg**: The Haute école d'ingénierie et d'architecture de Fribourg (HEIA-FR) is a technical school of applied sciences located in Fribourg, Switzerland. The name in english is: School of Engineering and Architecture of Fribourg. 1

Bibliography

[1] The Haskell community. 2024. Haskell Prelude documentation. Retrieved from https://hackage.haskell.org/package/base-4.7.0.1/docs/Prelude.html

[2] Brian O'Sullivan. 2012. Lecture notes on the GHC compiler. Retrieved from https://www.scs.stanford.edu/11au-cs240h/notes/ghc.html

[3] Bytecode Alliance. 2024. WebAssembly Component Model proposal. Retrieved from https://component-model.bytecodealliance.org/

[4] Mikael Brockman. 2022. Wisp compiler github repository. Retrieved from https://github.com/mbrock/wisp

[5] Niklaus Wirth. 1996. *Compiler Construction*. Addison-Wesley.

[6] Noah Godel. 2024. *Functional language compiler to WebAssembly - Specification*.

[7] Simon Marlow. 2010. Haskell language specification. Retrieved from https://www.haskell.org/onlinereport/haskell2010/

[8] Tweag I/O. 2022. Asterius compiler github repository. Retrieved from https://github.com/tweag/asterius

[9] United Nations. 2024. United Nations Sustainable Development Goals. Retrieved from https://sdgs.un.org/goals

[10] WasmEdge. 2024. List of supported Wasm proposals for the Wasmedge runtime. Retrieved from https://wasmedge.org/docs/start/wasmedge/extensions/proposals/#standard-webassembly-features

[11] WasmEdge. 2024. List of SDKs for the Wasmedge runtime. Retrieved from https://wasmedge.org/docs/embed/overview

[12] Wasmer. 2024. List of SDKs for the Wasmer runtime. Retrieved from https://github.com/wasmerio/wasmer?tab=readme-ov-file#wasmer-sdk

[13] Wasmtime. 2024. List of SDKs for the Wasmtime runtime. Retrieved from https://github.com/wasmerio/wasmer?tab=readme-ov-file#wasmer-sdk

[14] WebAssembly Community Group. WebAssembly Reference Types proposal. Retrieved from https://github.com/WebAssembly/reference-types

[15] WebAssembly Community Group. WebAssembly Function References proposal. Retrieved from https://github.com/WebAssembly/function-references

[16] WebAssembly Community Group. 2024. WebAssembly Garbage Collection proposal. Retrieved from https://github.com/WebAssembly/gc

[17] WebAssembly Community Group. 2024. WebAssembly Tail Call proposal. Retrieved from https://github.com/WebAssembly/tail-call

[18] WebAssembly Community Group. 2024. WebAssembly feature extensions. Retrieved from https://webassembly.org/features/

Table of figures