This chapter covers various aspects of the preparation for the project, including some of the important concepts I learned about while planning the implementation of various stages, a general overview of the compiler stages, and a description of the tools, tests, and general software development attitude used.

0.1 Requirements Analysis

The requirements of the project were relatively simple: the primary goal was to produce a working compiler for a small subset of Haskell, then add optimisations to meet the success criteria. There were no requirements for interacting with other systems such as a build system. It was desirable to have a comprehensive test suite to ensure correctness, and this was planned in advance.

One issue discovered during requirements analysis was that implementing even a basic subset of Haskell required support for some features which were originally listed as extensions: 'simple' operations such as addition require support for functions, but also for typeclasses and Algebraic Datatypes (ADTs). This caused a reorganisation of the planned schedule during Michaelmas term.

Specifics of the software development processes used are described in Section 0.6, and the following sections on testability and tooling.

0.2 Concepts

There are a number of key concepts that I needed to learn about in order to design and implement various stages in the compiler: these are detailed in the following subsections.

0.2.1 Kinds

Kinds are an essential concept for the type system F_Ω used by Haskell, as described by B. Pierce[**?**]. System F_Ω introduces type constructors, terms in the 'type language' which act as 'functions' between types. `Int` is a traditional type as seen in the polymorphic and simply-typed lambda calculus, but `Maybe Int` is less traditional: `Maybe` is parameterised by a type. Kinds provide a language with which to describe these type constructors, and more importantly check that they are well-formed. After all, `Int Bool` makes no sense as a type, just as the term `1 True` makes no sense as an term.

A Kind is often described as the 'type of a type': we can say that `True :: Bool`, but looking to a type system 'one level up' we can say `Bool :: *` where `*` is the 'type' of a parameterless 'type constructor'. The type constructor `Maybe` has kind `* -> *`, as it takes a single type parameter: applying it to a type of kind `*` yields a type of kind `*`, such as `Maybe Int`, while applying it to a type with a different kind such as `Maybe Maybe` produces an invalid type.

Data and class declarations define type constructors, and the kinds of any parameters can be inferred by their usage. Consider that in the declaration `data Maybe a = Left | Right a`,

we can see that `Maybe :: α -> *` as it has one parameter (`a`, with as-yet unknown kind α), and from the data constructor branch `Right a` we can infer that α = `*`, resulting in `Maybe :: * -> *`. On the other hand, `data A f a = A (f a)` defines a type constructor `A :: (* -> *) -> *` and `data Bool = True | False` simply defines `Bool :: *`.

All typeable terms in Haskell have a type with kind `*`: no values exist for types of other kinds. For example, `(1 :: Int) :: *`, but there are no values of type `Maybe`, only values of `Maybe Int` or `Maybe Bool`, and so on.

The grammar and rules defining kinds sufficient for this compiler are as follows:

$$\text{Kinds } K ::= * \mid K \to K \qquad \frac{\vdash T_1 :: K_1 \to K_2 \quad \vdash T_2 :: K_1}{\vdash T_1\ T_2 :: K_2}$$

The rule is identical to the function application rule from the simply typed lambda calculus, but defined on types instead of terms: this is the type system 'one level up' (more advanced systems for kinds include more rules). Checks for type well-formedness include a check that any type applications result in types of appropriate kinds. In particular, any type inferred for a value must have kind `*`.

0.2.2 Weak Head Normal Form

In a deterministic call-by-value language, `(1+2+3, \x -> x && True)` would evaluate to `(6, \x -> x && True)`. Haskell instead requires that terms are evaluated up to weak head normal form (WHNF), in which terms are evaluated to their 'head', and subexpressions need not have been evaluated. Haskell specifically defines the 'head' to be a literal, a fully or partially applied data constructor, or a partially applied function. Arguments to data constructors/functions are subexpressions, so need not have been evaluated.

In WHNF

```
1  1
2  (+) 1
3  Just True
4  (+) (1 + 2)
5  x = 1:x
```

Not in WHNF

```
1  1 + 2
2  (\x -> x) 1
3  (1 + 2) + 3
```

Evaluation of an expression up to WHNF corresponds to a form of **non-strict evaluation**: partial applications of functions or any data constructor applications do not force their arguments to be evaluated, but when a function is applied to all its arguments, it reduces to the body without necessarily having evaluated its arguments. In particular, the evaluation of a Haskell program is equivalent to its reduction to WHNF.

```
1     (\x y -> (+) x) (1 + 2) 3
2  → (+) (1 + 2)
3  ↛
```

Figure 1: An example of evaluation to WHNF, demonstrating laziness.

This is an important concept for the code generation stage, which must generate code that implements evaluation to WHNF.

0.2.3 Administrative Normal Form

Administrative Normal Form (ANF, presented by Flanagan et al. [**?**]) is a style of writing programs in which all arguments to functions are trivial (a variable, literal, or other irreducible 'value' like a lambda expression). ANF is an alternative to Continuation Passing Style (CPS) as a style of intermediate language that is often seen as being simpler to manipulate.

The expression `f x (1 + 2)` in ANF would be `let y = 1 + 2 in f x y`.

ANF is quite convenient for conceptualising the 'thunk' implementation of lazy evaluation, where expressions are represented as (possibly shared) units of suspended computation: any variable that binds a non-trivial expression acts as the name of the thunk representing that expression, and function arguments now pass around references to expressions.

As part of the lowering process, the intermediate languages are converted into ANF as it makes generating thunk-based code quite intuitive.

0.2.4 Typeclasses

Typeclasses are a language feature used to provide statically typed ad-hoc polymorphism (overloading). The general usage of typeclasses is encapsulated by the following example:

```
1  class Functor f where
2      fmap :: (a -> b) -> f a -> f b
3  instance Functor Maybe where
4      -- fmap :: (a -> b) -> Maybe a -> Maybe b
5      fmap _ Nothing = Nothing
6      fmap f (Just x) = Just (f x)
7  instance Functor [a] where
8      -- fmap :: (a -> b) -> [a] -> [b]
9      fmap _ [] = []
10     fmap f (x:xs) = f x:fmap f xs
```

A typeclass is similar to an 'interface' in object-oriented languages: it defines a set of functions that must be implemented by all instances of the class. In the example above, the `Functor` class defines the `fmap` function and specifies its type, parameterised by a type variable `f`. The instances of the class must provide implementations of this function where `f` is replaced by the type being made an instance.

Functions of a class can be used on any instance of that class, such as in the expression `fmap (*2) [1,2,3]` which evaluates to `[2,4,6]`, or `fmap (fmap (*2)) [Just 1, Nothing]` which evaluates to `[Just 2, Nothing]`.

Typeclasses are one of the most core features of Haskell: comparing values can be done using the functions provided by the `Eq` and `Ord` typeclasses, printing and reading values makes use of the `Show` and `Read` typeclasses, and all numeric types are instances of the `Num` typeclass.

The most commonly used implementation of typeclasses is dictionary passing, which is described in the section on Deoverloading (**??**).

0.2.5 Summary

To briefly summarise the concepts listed above:

- A Kind is the type of a type, usually used to describe type constuctors: `Maybe :: * -> *`.
- Weak head normal form corresponds to non-strict evaluation, by only evaluating to the outermost term.
- Administrative normal form intuitively corresponds to thunks by only allowing trivial arguments to be passed to functions.
- Typeclasses provide statically typed overloading, implemented using dictionary passing.

0.3 Compiler Structure: Big Picture

The general structure of HJC is standard: the specific components within each stage are discussed within the implementation section, but a general overview is useful for context.

Frontend

The frontend consists of standard lexing and parsing from Haskell source code into an Abstract Syntax Tree (AST). A modified version of an existing library (haskell-src[**?**]) is used.

Preprocessing

The renamer renames each variable so that later stages can assume each variable name is unique: this reduces complexity by removing the possibility of variable shadowing (eg. `let x = 1 in let x = 2 in x`).

Dependency analysis computes a directed graph representing syntactic dependencies between the source declarations so that the typechecker can process them in dependency order.

Type Checker

The type inference stage infers polymorphic overloaded types for each symbol, checks them against any user-provided type signatures, and alters the AST so that each expression is tagged with its type.

Deoverloading removes typeclasses from the type-level by implementing them as datatypes using dictionary-passing.

Lowering

The lowering stage transforms the Haskell source AST into Intermediate Language A (ILA), then rearranges that tree into Administrative Normal Form (ILA-ANF), before finally transforming it into Intermediate Language B (ILB).

Optimisations

Optimisations transform the intermediate languages into more efficient forms (with respect to runtime performance or generated code size) while preserving their semantics.

Code Generation

ILB is transformed into Java Bytecode (JVB), and a modified version of an existing library[**?**] is used to convert a logical representation of the bytecode into a set of class files, which are then packaged into an executable Jar file.

0.4 Testability

Given the number of stages and the scale of the project, tests are important to ensure that each component has the intended behaviour.

Haskell is a good language for writing testable code in: pure functions are usually easier to unit-test than impure functions as their behaviour is only affected by the parameters, independent of any global mutable state. The pipeline of HJC is entirely pure, with impure code only for reading the source file and writing the compiled files. This made testing each stage reliable and strictly independent of the adjacent pipeline sections.

Regression tests were implemented for all major bugs discovered, and ensure that the compiler stages do not reintroduce incorrect behaviour.

Finally, end-to-end tests ensure that HJC successfully processes a given Haskell source file and that the executable produced computes the correct result, treating the compiler as a black box. This extremely coarse testing method was very effective for discovering the existence of bugs, which could then be tracked down using standard debugging techniques and isolated using the finer-grained unit and regression tests.

0.5 Tools

I chose to write HJC in Haskell as it has a number of desirable features for large projects: purity ensures that components of the project cannot interact in unexpected ways, and the static type system guarantees that modifications are checked for a shallow (type-level) degree of correctness across the entire system.

The natural choice of compiler for Haskell is the industry-leading Glasgow Haskell Compiler (GHC), and the Stack build system is also relatively uncontested for Haskell build tooling, ensuring reproducible builds through a strict dependency versioning system.

Documentation has been written using Haddock, a tool that generates documentation from the code and comments: this documentation is rebuilt on every successful build and provides an easily-navigable description of commented modules and functions.

Git was used for version control, allowing me to develop features on distinct branches, use bisection to find the commits which introduced bugs, and keep a remote repository of code on Github as a backup.

Continuous integration was performed using Travis CI, which ensures that tests are run on every pushed commit and that builds are reproducible: the project can be built and run on different machines.

The benchmarking framework was written in Python 3, and plots generated using `matplotlib`. The Java Microbenchmark Harness (JMH) was used for gathering runtime statistics about the performance of output programs.

0.6 Software Development Model

I mainly used the waterfall development model, building each stage in the pipeline sequentially and testing it both in isolation and in sequence with the previous stages. The only stage which broke this model was type inference, which required multiple refining iterations to properly implement.

This approach was effective for most stages as there was a well-defined set of unchanging requirements. When the approach failed to work it was due to an incomplete set of requirements, so an iterative approach was more suitable to introduce support for the new requirements.

0.7 Starting Point

HJC uses a number of open-source packages from the de-facto Haskell standard library, such as `containers`, `text`, `mtl`, ... The full list is available in the `packages.yaml` configuration file in the root of the code repository.

The `haskell-src` lexing/parsing library for Haskell 98 source code was used, although forked and modified minorly (`187++`, `68--`). The bytecode assembly library `hs-java` was forked and significantly modified and extended to meet the requirements of this project (`1,772++`, `1,431--`).

No new languages had to be learnt for this project: I was already familiar with Haskell, Python 3, and Java. I had not worked with `haskell-src`, `hs-java`, `matplotlib`, or `JMH` before.