The compiler is comprised of a number of stages and substages, as shown in Figure 1:

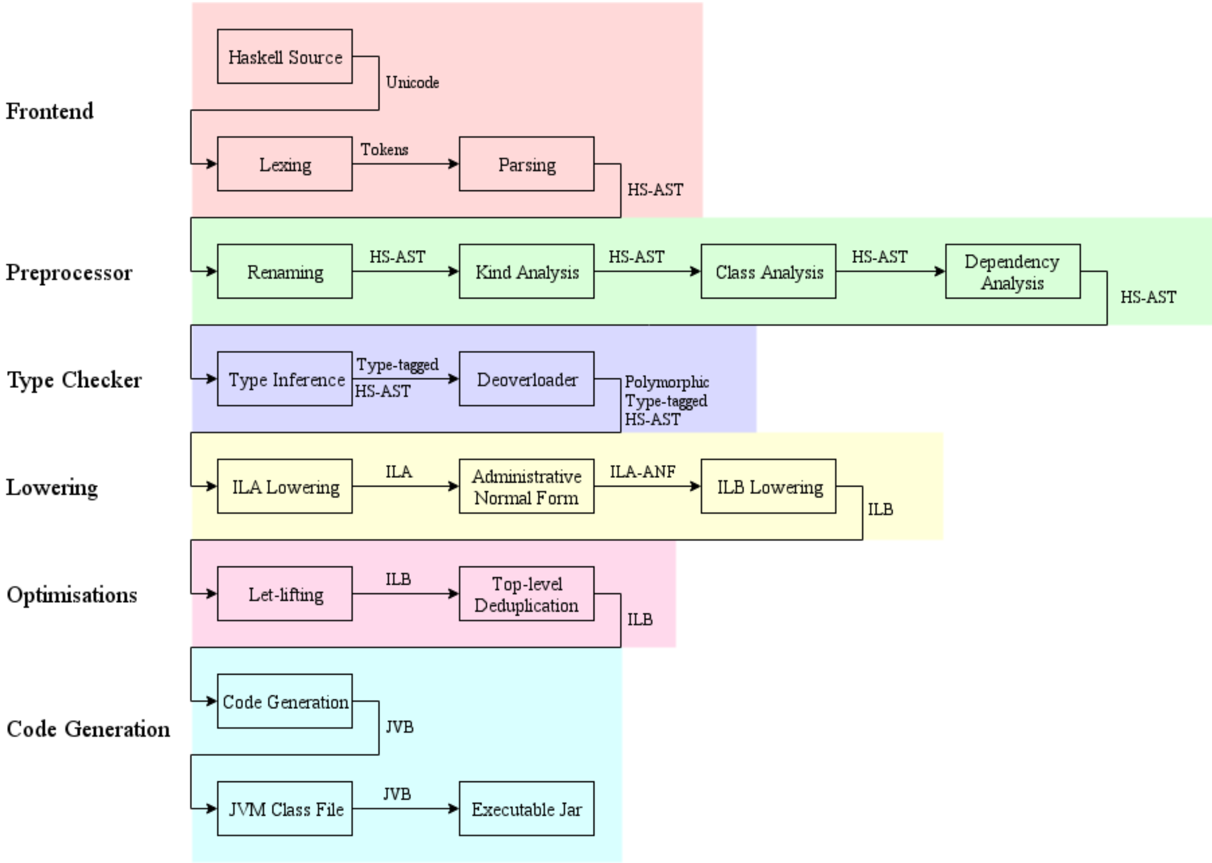

Figure 1

A brief overview of each stage is given here for a 'big picture' view of the compiler, followed by more detailed descriptions below.

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[1]) 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`).
- Kind and Class analysis both simply extract useful information about the declarations in the source so that stages of the type checker are simpler.
- Dependency analysis computes a partial order on the source declarations so that the typechecker can process them in a valid order.

Type Checker

[1]https://github.com/hnefatl/haskell-src

- 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 converts polymorphic overloaded types to polymorphic types similar to those of System F, and alters the AST to implement typeclasses 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 while preserving their semantics.

At time of writing these are done on ILB, might change to ILAANF so should update this accordingly.

If any more optimisations are implemented, update the diagram and here.

Code Generation

ILB is transformed into Java Bytecode (JVB), and a modified version of an existing library (hs-java[2]) 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.

0.1 Implementation Details

0.1.1 Frontend

Lexing and parsing of Haskell source is performed using the `haskell-src`[3] library, which I have modified to provide some additional desirable features:

- Lexing and parsing declarations for built-in constructors like list and tuple definitions (eg. `data [] a = [] | a:[a]`).
- Parsing data declarations without any constructors (eg. `data Int`)[4]. This is a valuable way of introducing built-in types.
- Adding `Hashable` and `Ord` typeclass instances to the syntax AST, so that syntax trees can be stored in associative containers.

The syntax supported is a strict superset of Haskell 1998 and a strict subset of Haskell 2010, but my compiler does not have support for all of the features implied by the scope of the syntax. For example, multi-parameter typeclasses are parsed correctly as a feature of Haskell 2010 but get rejected by the deoverloading stage.

[2] https://github.com/hnefatl/hs-java

[3] https://hackage.haskell.org/package/haskell-src

[4] Declarations of this form are invalid in the original Haskell 1998 syntax, but valid in Haskell 2010: see https://wiki.haskell.org/Empty_type

```haskell
class Convertable a b where
    convert :: a -> b
instance Convertable Bool Int where
    convert True = 1
    convert False = 0
```

Figure 2: An example of a multi-parameter typeclass

0.1.2 Preprocessor

0.1.2.1 Renaming

Haskell allows for multiple variables to share the same name within different scopes, which can increase the complexity of later stages in the pipeline. For example, when typechecking the following code we might conflate the two uses of `x`, and erroneously infer that they have the same type. A similar problem arises with variable shadowing, when the scopes overlap. The problem also applies to any type variables present in the source – the type variable `a` is distinct between the two type signatures:

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

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

Additionally, variables and type variables are in different namespaces: the same token can refer to a variable and a type variable, even within the same scope. The following code is perfectly valid (but loops forever), despite the same name being used for a type variable and a variable:

```haskell
x :: x
x = x
```

To eliminate the potential for subtle bugs stemming from this feature, the renamer pass gives each distinct variable/type variable in the source a unique name (in the above example, the variable `x` might be renamed to `v0` and the type variable renamed to `tv0`, provided those names haven't been already used).

Unique variable/type variable names are generated by prefixing the current value of an incrementing counter with either `v` for variable names or `tv` for type variable names. The renamer traverses the syntax tree maintaining a mapping from a syntactic variable/type variable name to an associated stack of unique semantic variable names (in Haskell, a `Map VariableName [UniqueVariableName]`):

- When processing the binding site of a new syntactic variable (eg. a let binding, a lambda argument, a pattern match...), a fresh semantic name is generated and pushed onto the stack associated with the syntactic variable.

- Whenever we leave the scope of a syntactic variable, we pop the top semantic name from the associated stack.

- When processing a use site of a syntactic variable, we replace it with the current top of the associated stack.

An analogously constructed mapping is maintained for type variables, but is kept separate from the variable mapping: otherwise the keys can conflict in code such as `x :: x`.

Type constants, such as `Bool` from `data Bool = False | True` and typeclass names like `Num` from `class Num a where ...`, are not renamed: these names are already guaranteed to be unique by the syntax of Haskell, and renaming them means we need to maintain more mappings and carry more state through the compiler as to what they've been renamed to.

0.1.3 Kind/Class Analysis

The typechecker and deoverloader require information about the kinds of any type constructors (the 'type of the type', eg. `Int :: *` and `Maybe :: * -> *`), and the methods provided by different classes. This is tricky to compute during typechecking as those passes traverse the AST in dependency order. Instead, we just perform a traversal of the AST early in the pipeline to aggregate the required information.

0.1.4 Dependency Analysis

When typechecking, the order of processing declarations matters: we can't infer the type of `foo = bar baz` until we've inferred the types of `bar` and `baz`. The dependency analysis stage determines the order in which the typechecker should process declarations.

We compute the sets of free/bound variables/type variables/type constants for each declaration, then construct a dependency graph – each node is a declaration, and there's an edge from A to B if any of the bound variables/type variables/type constants at A are free in B. It is important to distinguish between variables/type variables and type constants, as otherwise name conflicts could occur (as we don't rename type constants). This separation is upheld in the compiler by using different types for each, and is represented in the dependency graph below by colouring variables red and constants blue.

The strongly connected components of the dependency graph correspond to sets of mutually recursive declarations, and the partial order between components gives us the order to typecheck each set. For example, from the dependency graph in Figure 3 we know that: we need to typecheck d_3, d_4, and d_5 together as they're contained within the same strongly-connected component so are mutually recursive; we have to typecheck d_2 last, after both other components.

Typechecking declarations within the same component can proceed in an arbitrary order, we just need to ensure that the all of the type variables for the names bound by the declarations are available while processing each individual declaration.

This process works for languages without ad-hoc overloading, like SML. However, in Haskell there are some complications introduced by typeclasses:

- Typeclass member variables can be declared multiple times within the same scope. For example:

```haskell
class Num a where
    (+) :: a -> a -> a
```

```
data Bool = False | True      d1
x = f True                    d2
f y = g y                     d3
g y = h y                     d4
h y = f y                     d5
```

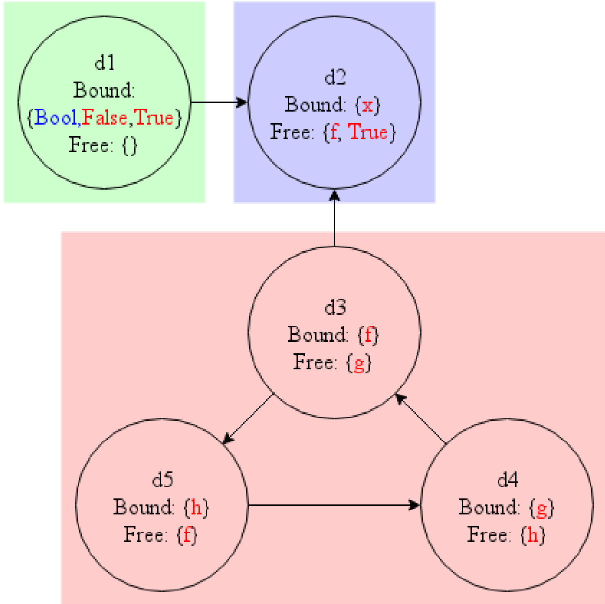

Figure 3: Code labelled with declaration numbers, and the corresponding dependency graph. Variables are in red text, type constants in blue. Strongly connected components are highlighted.

Prettier graph: better colours/some other grouping style, and use a more latex-y font.

```
instance Num Int where
    x + y = ...
instance Num Float where
    x + y = ...
```

Here the multiple declarations of + don't conflict: this is a valid program. However, the following program does have conflicting variables, as `x` is not a typeclass member and is not declared inside an `instance` declaration:

```
x = True
x = False
```

These declaration conflicts can be expressed as a binary symmetric predicate on declarations, as presented in Figure 4, where:

- `Sym` x and `Type` x represent top-level declaration and type-signature declarations for a symbol x, like x `= True` and x `:: Bool`.
- `ClassSym` x c and `ClassType` x c represent `Sym` x and `Type` x inside the declaration for a class c, like `class` c `where { ` x ` = True ; ` x ` :: Bool }`.
- `InstSym` x c t represents a `Sym` x inside the declaration for a class instance c t, like `instance` c t `where { ` x ` = True }`.

Using this table we can see that the multiple declarations for + in the example above are `InstSym (+) Num Int` and `InstSym (+) Num Float` so do not conflict, while the declarations for `x` above are both `Sym x` so do conflict.

	`Sym` x_1	`Type` x_1	`ClassSym` x_1 c_1	`ClassType` x_1 c_1	`InstSym` x_1 c_1 t_1
`Sym` x_2	$x_1 = x_2$	`False`	$x_1 = x_2$	$x_1 = x_2$	$x_1 = x_2$
`Type` x_2		$x_1 = x_2$	$x_1 = x_2$	$x_1 = x_2$	$x_1 = x_2$
`ClassSym` x_2 c_2			$x_1 = x_2$	$x_1 = x_2 \wedge c_1 \neq c_2$	$x_1 = x_2 \wedge c_1 \neq c_2$
`ClassType` x_2 c_2				$x_1 = x_2$	$x_1 = x_2 \wedge c_1 \neq c_2$
`InstSym` x_2 c_2 t_2					$x_1 = x_2 \wedge (c_1 \neq c_2 \vee t_1 = t_2)$

Figure 4: The conflict relation: the bottom triangle is omitted as the predicate is symmetric

- Another complication introduced by typeclasses is that variable declarations such as `id = \\x -> x` are usually treated as being the unique binding definition of `id`, and any other uses within the same level of scope must be free rather than binding (otherwise we have conflicting definitions).

  However, we treat binding declarations inside `instance` declarations as actually being free uses rather than binding uses, so that the instance declaration forms a dependence on the class declaration where the variables are bound, ensuring it is typechecked first.

- The dependencies generated by this technique are *syntactic*, not *semantic*: this is a subtle but very important difference. The use of any ad-hoc overloaded variable generates dependencies on the class declaration that introduced the variable, but not the specific instance of the class that provides the definition of the variable used.

```haskell
class Foo a where
    foo :: a -> Bool
instance Foo Bool where
    foo x = x
instance Foo [Bool] where
    foo xs = all foo xs
```

  The declaration of `foo` in `instance Foo [Bool]` semantically depends on the specific overload of `foo` defined in `instance Foo Bool`, and yet no dependency will be generated between the two instances as neither declaration binds `foo` (`foo` is treated as being free within the declarations as described above): they will only generate dependencies to `class Foo a` (and to the declaration of `Bool` and `all`).

  Computing the semantic dependencies is too complicated to be done in this pass, so the problem is left and instead solved during the typechecking stage. A full explanation is given later, but the approach used is to defer typechecking instance declarations until a different declaration requires the information, and then attempt to typecheck the instance declaration then, in a Just-In-Time manner.

  Make sure this is actually explained later

0.1.5 Type Checker

Type inference and checking is the most complex part of the compiler pipeline. The type system implemented is intended to be similar to GHC's dialect of Haskell: approximately System F_ω (the polymorphically typed lambda calculus with type constructors) along with algebraic data types and type classes to provide ad-hoc overloading. The approximation is due to a number of alterations made by the Haskell Report to ensure that type inference is decidable, along with the introduction of terms that aren't covered by the pure type system eg. literals such as `7` and `"foo"`. Finally, due to the complexity of the implementation there will doubtless be bugs affecting correctness in more complex cases not covered by tests.

This is a subset of the type system used by GHC (System F_C), as that compiler provides extensions such as GADTs and type families requiring a more complex type system.

The full grammar of types as defined in the compiler are explained below:

```
data Kind = KindStar
          | KindFun Kind Kind
```

A `Kind` is 'the type of a type', and is used to describe type constructors: what characterises Maybe versus Maybe α?

: while the term `f` might have the type $\alpha \to$ Maybe α, the type has *kind* `*`.

Finish describing kinds and the other type datatypes

Work out how to use — without the red box in the pdf.

```
data TypeVariable = TypeVariable TypeVariableName Kind

data Type = TypeVar TypeVariable
          | TypeCon TypeConstant
          | TypeApp Type Type Kind


data TypePredicate = IsInstance ClassName Type

data Qualified a = Qualified (Set TypePredicate) a
type QualifiedType = Qualified Type

data Quantified a = Quantified (Set TypeVariable) a
type QuantifiedType = Quantified QualifiedType
type QuantifiedSimpleType = Quantified Type
```

Include examples of types demonstrating the features? Examples below might be sufficient. Think an appendix would be best, or should I just include here?

Include simple BNF grammar for the valid types: can modify the one on the HM wiki page

Substituion section explaining the basic notation? Used quite a lot in the next few subsub-sections

0.1.5.1 Type Inference

This whole section needs a lot of attention. The process is complex, maybe need to present the HM rules and their haskell-style variations? Nontrivial

The implementation is inspired by[?], and proceeds similarly to the Hindley-Milner (HM) type inference algorithm presented in[?]. Following the naming convention using in[?], we refer to types such as $\alpha \rightarrow \alpha$ as 'simple types' ('monotypes' in HM), types with constraints such as $\text{Eq}\alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \text{Bool}$ as 'overloaded types', and types involving quantifiers such as $\forall\alpha.\ \text{Monoid}\ \alpha \Rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$ as 'polymorphic types' (similar to 'polytypes' in HM except the quantifiers may wrap an overloaded type or a simple type). It's often helpful to consider simple types as the special case of an overloaded type with an empty context ($\emptyset \Rightarrow t$).

We process each declaration in dependency order, traversing the AST and tagging each expression with a type variable and unifying types together to derive a substitution from type variables to types. Each term in the syntax has an associated rule for combining the type variables representing the types of its sub-terms, which are computed recursively.

One departure from the HM algorithm is that Haskell doesn't require Λ terms in the source language (explicit 'forall' terms that generate $\forall$ quantifiers on type variables): let-bound variables are implicitly polymorphic over all their free type variables, while function parameters are never polymorphic. In practice, this means that in Figure 5, `f ::` $\forall\alpha.\ \alpha \rightarrow a$ whereas `g ::` $\alpha \rightarrow \alpha$. The difference in semantics ensures that type inference remains decidable.

```haskell
let f x = x in const (f True) (f 1) :: Bool -- This is fine
(\\g -> const (g True) (g 1)) (\x -> x)      -- This fails to typecheck
```

Figure 5

Another major difference from HM is the introduction of typeclasses: whenever we encounter an overloaded variable such as +, we split the overloaded type into the constraints $\{\text{Num}\ \alpha\}$ and the simple type $\alpha \rightarrow \alpha \rightarrow \alpha$. Unification can be performed as in HM on the simple types, but we also maintain a global set of the constraints on type variables that we encounter. After unification has completed, we use this set to add the typeclass constraints to the inferred types.

The final important difference from HM is in the instantiation of polymorphic types: the [`Inst`] rule in HM allows for terms with polymorphic types to be reused at call sites with different argument types, but it needs a slight modification to handle type variable constraints introduced by overloaded types:

$$\frac{\Gamma \vdash e : \tau' \quad \tau' \sqsubseteq \tau}{\Gamma \vdash e : \tau}\,[\text{Inst}]$$

The original rule from HM

$$\frac{\Gamma \overset{\Delta}{\vdash} e : C \Rightarrow \tau' \quad \tau' \sqsubseteq \tau \quad \theta = \text{mgu}(\tau, \tau')}{\Gamma \overset{\Delta \cup C\theta}{\vdash} e : C\theta \Rightarrow \tau}\,[\text{Inst}]$$

A modified rule handling type classes. Δ is the set of type variable constraints, built up throughout uses of the rules.

Bother explaining $\sqsubseteq$ or just say it means 'x is more general than y'? Not exactly going into detail here anyway

Yeah this is kinda obscure...

Here is an example of how the type inference algorithm proceeds on a simple Haskell program:

Want to distinguish as a different section from the above somehow

$$
\begin{aligned}
S_{\text{add1}} &= \text{mgu}(\alpha, \beta \to \eta) \circ \text{mgu}(\delta, \epsilon \to \eta) \circ \text{mgu}(\gamma \to \gamma \to \gamma, \beta \to \delta) \\
&= [\gamma \to \gamma/\alpha, \gamma \to \gamma/\delta, \gamma/\epsilon, \gamma/\eta]
\end{aligned}
$$

```
add1 x = x + 1
y = add1 2
```

As the definition of `y` depends on the definition of `add1` we process `add1` first, converting the infix operator into a prefix operator, and producing a type-variable-tagged AST:

```
add1^α x^β = (((+)^(γ→γ→γ) x^β)^δ 1^ε)#\(^\eta\)|
```

We know the type of + is $\forall \omega.\ \text{Num}\ \omega \Rightarrow \omega \to \omega \to \omega$ from the definition of + within the typeclass `Num`, which would have been processed prior to this example. We instantiate this to fit this specific call site (remove the quantifier and replace the quantified type variables with fresh type variables) $\text{Num}\ \gamma \Rightarrow \gamma \to \gamma \to \gamma$, then split the overloaded type into the type variable constraints ($\{\text{Num}\ \gamma\}$) and the simple type ($\gamma \to \gamma \to \gamma$), and use the simple type for unification and add the constraints to a global set of type variable constraints.

Unifying the

The type of `add1` is therefore $\alpha\ S_{\text{add1}} = \gamma \to \gamma$, and as this declaration is let-bound we insert quantifiers to make it polymorphic: `add1 ::` $\forall \gamma.\ \gamma \to \gamma$.

Processing `y` gives:

```
y^ι = (add1^(θ→θ) 2^κ)#\(^\lambda\)|
```

$$
\begin{aligned}
S_{\text{y}} &= \text{mgu}(\iota, \lambda) \circ \text{mgu}(\theta \to \theta, \kappa \to \lambda) \\
&= [\theta/\kappa, \theta/\iota, \theta/\lambda]
\end{aligned}
$$

As `y` is let-bound again, we conclude `y :: |\(\forall \theta. \; \)|`.

0.1.5.2 Deoverloader

The deoverloading stage performs a translation which eliminates typeclasses, resulting in an AST tagged with types that no longer have type contexts. The implementation of typeclasses chosen was dictionary passing.

To convert `add1 x = (+) x 1` with type $\forall \alpha.\ \text{Num}\ \alpha \Rightarrow \alpha \to \alpha$ to a non-overloaded function, we can add an extra argument that carries the 'implementation' of the $\text{Num}\ \alpha$ constraint, which we then pass down to the + function: `add1' dNum x = (+) dNum x 1`.

Alternative approaches than dictionary passing?

The approach used here is to perform a source-to-source transformation on the AST that replaces typeclass/instance declarations with datatype/value/function declarations.

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
```

```haskell
instance Eq Bool where
    (==) = ...
    (/=) = ...
```

Typeclasses are replaced by datatypes equivalent to tuples with an element for each function defined by the class, and instances are replaced by values of the respective class' datatype, filling in the elements using the implementation provided by the instance declaration. Extractor functions are added which pull specific elements out of the datatype to get at the actual implementation of the function.

```haskell
-- Implementation of the typeclass
data Eq a = Eq (a -> a -> Bool) (a -> a -> Bool)

-- The functions defined by the typeclass extract the implementation functions
(==), (/=) :: Eq a -> a -> a -> Bool
(==) (Eq eq _) = eq
(/=) (Eq _ neq) = neq

-- The implementation of the typeclass instance
dEqBool :: Eq Bool
dEqBool = Eq dEqBoolEq dEqBoolNeq

-- The function implementations defined in the instance
dEqBoolEq, dEqBoolNeq :: Bool -> Bool -> Bool
dEqBoolEq = ...
dEqBoolNeq = ...
```

To deoverload declarations using overloaded values, we essentially convert declarations of functions with types like Num $\alpha \Rightarrow \alpha \rightarrow \alpha$ to Num $\alpha \rightarrow \alpha \rightarrow \alpha$: we replace typeclass constraints with formal arguments to carry the implementation dictionaries. Similarly when using an expression of overloaded type, we add an actual argument to the call site for each typeclass constraint to account for the extra arguments we added to the definition.

For example, `foo x = x == 1` deoverloads to `add1 dEqa x = (==) dEqa x 1`: we add an extra formal argument to the function declaration, and insert it in the call site to `+`.

In order to know what variables to insert at call sites, the names of in-scope variables holding values of typeclass implementations are stored while traversing the AST. Initially the mapping contains the ground instances provided by top-level instance declarations (eg. $\{$Num Int $\mapsto$ `dNumInt`, Eq Int $\mapsto$ `dEqInt`, ...$\}$), and we update it to include any in-scope extra arguments we add to functions while deoverloading them. When we encounter an overloaded variable, we insert arguments for each type constraint by finding matching constraints within our mapping.

One special case in the process is the handling of literals. Whilst arbitrary expressions with overloaded types always require a dictionary to be passed (consider that the innocent-looking `x` in `let x = 1 + 2 in x` involves an application of `+` which requires a dictionary to provide the implementation), literals do not require a dictionary as they can never perform any computation.

0.1.6 Lowering and Intermediate Languages

There are two intermediate languages within the compiler, imaginatively named Intermediate Languages A and B (ILA and ILB respectively). There is also a minor language named ILA-Administrative Normal Form (ILA-ANF), which is simply a subset of ILA that helps restrict the terms to those in Administrative Normal Form (ANF).

For each: why needed, BNF grammar, strengths. Probably don't go into details of translation?

0.1.6.1 Intermediate Language A

ILA is a subset of GHC's Core intermediate language, removing terms which are used for advanced language features like GADTs, as they are not supported by this compiler. Haskell 98 has hundreds of node types in its AST[5], whereas ILA has far fewer: this makes it far easier to transform. Below is the full definition of ILA:

Give BNF instead of Haskell ADTs?

```haskell
data Expr = Var VariableName Type
          | Con VariableName Type
          | Lit Literal Type
          | App Expr Expr
          | Lam VariableName Type Expr
          | Let VariableName Type Expr Expr
          | Case Expr [VariableName] [Alt Expr]
          | Type Type

data Literal = LiteralInt Integer
             | LiteralChar Char

data Alt a = Alt AltConstructor a

data AltConstructor = DataCon VariableName [VariableName]
                    | Default

data Binding a = NonRec VariableName a
               | Rec (Map VariableName a)
```

A Haskell program is lowered by this pass into a list of `Binding Expr`: a list of recursive or non-recursive bindings of expressions to variables.

Use paragraph to split this into better titled chunks

One notable feature of ILA is that it carries type information: leaf nodes such as `Var` are tagged with a type. GHC's Core IL is fully explicitly typed under a variant of System F,

[5] `https://hackage.haskell.org/package/haskell-src/docs/Language-Haskell-Syntax.html`

Find version-independent link

which allows for 'core linting' passes in-between transformations to ensure they maintain type-correctness. The type annotations on ILA are not sufficient for such complete typechecking, but do allow for some sanity checks and are necessary for lower stages of the compiler such as code generation.

ILA is still quite high-level, so many of the language constructs have similar semantics to their Haskell counterparts. The main benefit in this lowering pass is to collapse redundant Haskell syntax into a smaller grammar.

Most of these constructors have obvious usages, but some are more subtle: `Con` represents a data constructor such as `True` or `Just`. `App` is application of expressions, which covers both function applications and data constructor applications (eg. `App (Var "f" (Bool → Bool)) (Con "True" Bool)` and `App (Con "Just" (Int → Maybe Int)) (Var "x" Int)`).

Work out why spacing is weird here

`Lam` x t e represents a lambda term like $\lambda x : t.\ e$, and `Let` x t e_1 e_2 represents a term like let $x : t = e_1$ in e_2.

Lam and Let are most easily explained as lambda terms, but App is most easily demonstrated with some code, and the switch between explanation styles is a bit jarring.

`Case` e vs as represents a multi-way switch on the value of an expression e (the 'head' or 'scrutinee'), matching against a number of possible matches ('alts') from the list as, where the evaluated value of e is bound to each of the variables in vs. The additional binding variables can be useful when the scrutinee expression is reused within some number of the alts.

`Type` isn't really used afaik, it's a leftover from when I was trying to do System F style types. Should see if I can remove it.

The alts in a `Case` expression, of the form `Alt` c b, match the value of evaluating the scrutinee against the data constructor c, then evaluates the b from whichever alt matched. `AltConstructor` represents the potential matches: either a data constructor with a number of variables to be bound, or a 'match-anything' default value.

Many syntax features in Haskell are just syntactic sugar, and are simple to desugar (list literals like `[1, 2]` are desugared to `1:2:[]`). Others are slightly more involved, such as converting `if x then y else z` expressions into `case x of { True -> y ; False -> z }` (`Bool`is just defined as an ADT in Haskell, there's no special language support for it).

Other language features are non-trivial to lower, such as the rich syntax Haskell uses for pattern matching. An example pattern match could be `Just x@(y:z) = Just [1, 2]`, binding `x = [1, 2]`, `y = 1`, and `z = [2]`. Multiple pattern matches can also be related, as in function definitions:

```haskell
f (x, Just y) = x + y
f (x, Nothing) = x
```

Additionally, pattern matches can occur in a number of places: pattern-binding declarations such as `let (x, y) = z in ...`, functions definitions like the example above, lambda expressions, and `case` expressions (`case Just 1 of { Nothing -> ... ; Just x -> ... }`). The heterogeneity of use sites demands a flexible approach to translating pattern matches that can be reused for each instance.

My initial implementation worked correctly for single-pattern uses, such as the let example above, but didn't support multiple parallel patterns as used in case expressions and function definitions. The current implementation is now based off the approach given in Chapter 5 of [?], which is a more general version of my initial algorithm.

Feel like this needs a little bit more. Probably give a very brief overview, explain that everything's converted into case statements in the end, etc.

Finally, note that in Haskell a pattern will eventually match against a data constructor, a literal, or anything (with the wildcard pattern `_`). However, in the grammar for ILA's `AltConstructor`, there's no constructor corresponding to literals. This is due to case expressions generally only making sense for data constructors, where there are a finite number of constructors to check a value against for a given datatype. On the other hand, literals normally have a cumbersomely large (or infinite) number of 'constructors' (one can imagine the `Int` type, which is bounded, as being defined as `data Int = ... | -1 | 0 | 1 | ...`, but `Integer` cannot be defined in this way as it is unbounded). As a result, literals are 'pattern matched' by using equality checks from the `Eq` typeclass: the expression `case x of { 0 -> y ; 1 -> z ; _ -> w }` is essentially translated to `if x == 0 then y else if x == 1 then z else w`, which is then lowered into case expressions match on `True` and `False` as described above.

0.1.6.2 Intermediate Language A - Administrative Normal Form

Administrative Normal Form (ANF) is a style of writing programs in which all arguments to functions must be 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 but can perform transformations in a single pass that would take multiple passes on a CPS program[?].

This compiler uses ANF as it lends itself well to conceptualising lazy evaluation: as each complex expression is referred to through a variable, if the expression is evaluated by one computation then all other references to the variable transparently reference the resulting value, rather than a duplicate of the computation.

ILA-ANF is a subset of ILA which uses a more restricted grammar to enforce more invariants on the language and guide the AST into ANF. The full definition of ILA-ANF is given below, and reuses the definitions of `Binding` and `Alt` from ILA.

In the case of ILA-ANF, 'trivial' terms are taken to be variables, data constructors, and literals. Note that this excludes lambda terms, which is somewhat unusual. Instead, lambda terms must immediately be bound to a variable: this restriction is enforced by the `AnfRhs` term in the grammar below.

Why lambda terms not trivial? General design-decision explanation. Makes thunk model explicit + easier to generate code?

```haskell
data AnfTrivial = Var VariableName Type
                | Con VariableName Type
                | Lit Literal Type
                | Type Type

data AnfApplication = App AnfApplication AnfTrivial
                    | TrivApp AnfTrivial
```

```haskell

data AnfComplex = Let VariableName Type AnfRhs AnfComplex
                | Case AnfComplex Type [VariableName] [Alt AnfComplex]
                | CompApp AnfApplication
                | Trivial AnfTrivial

data AnfRhs = Lam VariableName Type AnfRhs
            | Complex AnfComplex
```

An ILA program is lowered from a list of `Binding Expr` to a list of `Binding AnfRhs` by this pass. The translation is quite simple compared to the other lowering passes – most of the terms are similar to those in ILA (including carrying type information), with notable exceptions being the introduction of `AnfApplication`, which restricts application arguments to purely trivial terms, and `AnfRhs`, to enforce that lambda terms can only be bound to variables.

0.1.6.3 Intermediate Language B

ILB is the final intermediate language of this compiler and is inspired by GHC's STG (Spineless Tagless G-Machine) IL. ILB maintains the ANF style from ILA-ANF. It has a number of extremely useful features for code generation: the only term that performs any evaluation of an expression is the `ExpCase` $e\ t\ vs\ as$ term (which evaluates e then branches to one of the as), and the only term which performs any memory allocation is the `ExpLit` $v\ r\ e$ term, which allocates memory on the heap to represent a datatype/literal/uneveluated expression then evaluates e.

Additionally, this language makes lazy evaluation 'explicit', in the sense that expressions to be evaluated are always encapsulated within an `RhsClosure` (thanks to ANF style which names each subexpression) that can be implemented as a not-yet-evaluated thunk.

```haskell
data Arg = ArgLit Literal
         | ArgVar VariableName

data Exp = ExpLit Literal
         | ExpVar VariableName
         | ExpApp VariableName [Arg]
         | ExpConApp VariableName [Arg]
         | ExpCase Exp Type [VariableName] [Alt Exp]
         | ExpLet VariableName Rhs Exp

data Rhs = RhsClosure [VariableName] Exp
```

ILB is similar in grammar to ILA-ANF, and the translation pass is relatively simple. There are some key differences between the languages, that reflect the changes from a relatively high-level IL down to a lower-level one:

Didn't use bullet points for the previous IL explanations: change them?

- There are now two terms for applications, one for functions (`ExpApp`) and one for data constructors (`ExpConApp`). The distinction is necessary for code generation, when a function application results in a jump to new executable code while a constructor application creates a new heap object.

ExpConApp also requires all its arguments to be present: it cannot be a partial application. Haskell treats datatype constructors as functions, so the following is a valid program:

```
data Pair a b = Pair a b
x = Pair 1
y = x 2
```

At the implementation level however, functions and data constructors are necessarily very different, so distinguishing them within this IL makes code generation easier.

Include how they're distinguished? Or too much detail?

- Right-hand-side terms in ILA (AnfRhs) were either lambda expressions or a let-binding/case expression/...– in ILB, the only right-hand-side term is a RhsClosure. A closure with no arguments is essentially a thunk, a term that exists purely to delay computation of an expression, while a closure with arguments is the familiar lambda term.

  ILB's RhsClosure takes a list of arguments, whereas ILA-ANF's lambda terms only take a single argument (multiple-argument functions are just nested single-argument lambdas). This is another translation aimed at making code generation easier. Single-argument lambdas allow for simpler logic when handling partial application in higher-level languages, but is inefficient in implementation. ILB is the ideal IL to perform this switch from the high-level convenient-to-modify grammar to a lower-level efficient representation.

- ILB only allows variables in many of the places where ILA-ANF allowed variables, literals, or 0-arity data constructors (like True). This is another step towards making laziness explicit, by keeping expressions simple so that only one step of the evaluation needs to happen at a time.

0.1.7 Code Generation

Left optimisations until after codegen: cover the main pipeline first then the optional stages later?

Code generation is, from the surface, quite a mechanically simple process. ILB is a small language, so there aren't many terms to lower into bytecode. Implementing the semantics of these terms in Java Bytecode is complex, however.

The `hs-java` library was used to provide a Haskell representation of bytecode that could then be serialised to a Java `.class` file, but a number of modifications were made to the library by me to add support for Java 8 features required by the compiler, as well as a number of smaller improvements: the forked project can be found at `https://github.com/hnefatl/hs-java`.

A number of Java classes have been written to provide the 'primitives' used by generated bytecode: including the implementation of Haskell's primitive datatypes like Int and Char, as well as the base class for all ADTs definable within the language (`BoxedData`, described later). The compiler is aware of these 'builtin' classes and uses a set of 'hooks' when generating code to provide Java implementations of Haskell functions. This is covered in more detail later.

0.1.7.1 Weak Head Normal Form

A Haskell expression is in weak head normal form (WHNF) if it is either a partially applied function (including lambda terms), a fully/partially applied data constructor, or a literal. Any arguments need not have been evaluated.

Evaluation of an expression up to WHNF corresponds to a form of non-strict evaluation: partial applications of functions or any data constructor applications don't 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 evaluation to WHNF.

The following Haskell expressions are either valid or invalid WHNF terms, as indicated:

```haskell
1                -- In WHNF
(+) 1            -- In WHNF
1 + 2            -- Not in WHNF
3                -- In WHNF
Just             -- In WHNF
Just True        -- In WHNF
(\\x -> x) 1     -- Not in WHNF
(+) (1 + 2)      -- In WHNF
(1 + 2) + 3      -- Not in WHNF
```

0.1.7.2 Heap Objects

Literals, datatype values and closures are all represented at runtime by values on the heap, as they are all first-class values in Haskell, and will be referred to as 'objects': this intentionally overloads the terminology used by Java for an instance of a class, as the two concepts are essentially interchangeable here as Java objects are heap-allocated.

Thunks are represented simply as closures without arguments: all of the closure logic described below is the same between thunks and functions.

All objects on the heap inherit from a common abstract base class, `HeapObject`:

```java
public abstract class HeapObject implements Cloneable {
    public abstract HeapObject enter();

    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

The abstract `enter` method evaluates the object to WHNF and returns a reference to the result, and the `clone` method simply returns a shallow copy of the object. This method is critically for implementing function applications, described later.

Literals

Literals are builtin types that can't be defined as an Haskell ADT, such as `Int`. Any such type is a subclass of the `Data` class, which is itself a subclass of the `HeapObject` class that a rather boring implementation of the abstract `enter` method. Any literal is already in WHNF, so evaluation to WHNF is trivial:

```java
public abstract class Data extends HeapObject {
    @Override
    public HeapObject enter() {
        return this;
    }
}
```

Here is an example literal implementation for `Integer`, Haskell's arbitrary precision integral value type. It is implemented using Java's `BigInteger` class to perform all the computation. The copious uses of underscores is explained in the JVM Sanitisation section below.

```java
import java.math.BigInteger;

public class _Integer extends Data {
    public BigInteger value;
    public static _Integer _make_Integer(BigInteger x) {
        _Integer i = new _Integer();
        i.value = x;
        return i;
    }
    public static _Integer _make_Integer(String x) {
        return _make_Integer(new BigInteger(x));
    }

    public static _Integer add(_Integer x, _Integer y) {
        return _make_Integer(x.value.add(y.value));
    }
    public static _Integer sub(_Integer x, _Integer y) { ... }
    public static _Integer mult(_Integer x, _Integer y) { ... }
    public static _Integer div(_Integer x, _Integer y) { ... }
    public static _Integer negate(_Integer x) { ... }

    public static boolean eq(_Integer x, _Integer y) { ... }

    public static String show(_Integer x) { ... }
}
```

The `_make_Integer(String)` function allows a Java `_Integer` object to be constructed from a Java string representation, which is used by the compiler to construct `Integer` values:

Brief two-line bytecode demonstrating this

The `add`, `sub`, etc. methods are Java implementations of the functions required by Haskell's `Num`, `Eq` and `Show` typeclass instances for `Integer`. The section on Hooks covers this aspect of code generation in more detail.

Datatypes

An Haskell ADT can be represented simply by a class generated by the compiler which inherits from the `BoxedData` builtin abstract class:

```
public abstract class BoxedData extends Data {
    public int branch;
    public HeapObject[] data;
}
```

The `branch` field is used to identify which constructor of the type has been used, and the `data` field contains any arguments given to the constructor. An example generated class[6] for the datatype `data Maybe a = Nothing | Just a` might be:

```
public class _Maybe extends BoxedData {
    public _make_Nothing() {
        _Maybe x = new _Maybe();
        x.branch = 0;
        x.data = new HeapObject[] {};
        return x;
    }
    public _make_Just(HeapObject val) {
        _Maybe x = new _Maybe();
        x.branch = 1;
        x.data = new HeapObject[] { val };
        return x;
    }
}
```

Note that as `BoxedData` inherits from `Data`, the `enter` method has the same simple implementation – as any data value is already in WHNF.

Closures

Closures are the most complicated objects stored on the heap. There are three main lifecycle stages of a closure:

- Creation: construction of a new closure representing a function of a given arity, without any arguments having been applied yet but possibly including values of free variables in scope of the closure.

[6]The compiler doesn't generate a class described in Java source as shown, it just generates the bytecode for the class directly.

- Argument application: this may be a partial application or a total application, or even an over-application: consider `id (+1) 5`, which evaluates to `6`. `id` has arity 1, but is applied to 2 arguments here.
- Evaluation: after a total application, reducing the function to its body (as specified by WHNF reduction).

These behaviours are provided by the `Function` builtin class:

Formatting + it's quite a lot of code, but I think necessary to understand the details below.

```
import java.util.ArrayList;
import java.util.function.BiFunction;

public class Function extends HeapObject {
    private BiFunction<HeapObject[], HeapObject[], HeapObject> inner;
    private HeapObject[] freeVariables;
    private ArrayList<HeapObject> arguments;
    private int arity = 0;

    public Function(BiFunction<HeapObject[], HeapObject[], HeapObject> inner,
                    int arity, HeapObject[] freeVariables) {
        this.inner = inner;
        this.arity = arity;
        this.freeVariables = freeVariables;
        arguments = new ArrayList<>();
    }

    @Override
    public HeapObject enter() {
        if (arguments.size() < arity) { // Partial application
            return this;
        }
        else if (arguments.size() > arity) { // Over-applied
            try {
                Function result = (Function)inner
                    .apply(
                        arguments.subList(0, arity).toArray(new HeapObject[0]),
                        freeVariables)
                    .enter()
                    .clone();
                for (HeapObject arg : arguments.subList(arity, arguments.size()))
                    result.addArgument(arg);
                return result;
            }
            catch (CloneNotSupportedException e) {
                throw new RuntimeException(e);
            }
```

```java
        }
        else { // Perfect application
            return inner.apply(
                arguments.toArray(new HeapObject[0]), freeVariables
            ).enter();
        }
    }

    public void addArgument(HeapObject arg) {
        arguments.add(arg);
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        Function f = (Function)super.clone();
        f.inner = inner;
        f.arity = arity;
        f.freeVariables = freeVariables.clone();
        f.arguments = new ArrayList<>(arguments);
        return f;
    }
}
```

A function f (either defined locally or at the top-level) in Haskell of arity n_a and using n_{fv} free variables is translated into two Java functions:

- `_`f`Impl`, which takes two arrays of `HeapObject`s as arguments, one holding the arguments for the Haskell function (of length n_a) and one holding the free variables used by the Haskell function (of length n_{fv}), and returns a `HeapObject` representing the result of applying the function.

- `_make_`f, which takes n_{fv} arguments representing the free variables of the Haskell function, and returns a Java `Function` object representing the closure, where the `inner` field points to the `_`f`Impl` function.

`Function`'s `freeVariables` field has type `HeapObject[]` as we know at initialisation time exactly how many free variables the function has, and it doesn't change. The `arguments` field is an `ArrayList<HeapObject>` so that we can handle partial applications and over-applications by only adding arguments when they're applied.

Haskell function applications are lowered into bytecode that:

1. Fetches the function, either by calling the appropriate `_make_` function with the free variables, or just loading a local variable if the function has already been partially applied and stored or passed as a function argument.

2. **Clones** the `Function` object. This step is subtle but vital, as each argument applied to the function mutates the `Function` object by storing additional arguments.

   If we're using a local closure like `let add1 = (+) 1 in add1 2 * add1 3` then `add1` will be a local `Function` object with `inner` pointing to the implementation of `(+)` and one applied argument (a `Data` instance representing 1). Both `add1 2` and `add1 3` will mutate the object to add the argument being applied (see the next step for details), which leads to the `Function` object after `add1 3` having 3 stored arguments.

Cloning the function essentially maintains the same references to arguments and free variables, but creates new (non-shared) containers to hold them, avoiding the above issue.

This is a shallow clone – if we used a deep clone, recursively cloning the arguments and free variables, then we'd lose the performance benefit of graph reduction where we can use an already computed value instead of recomputing it ourselves, and increase memory usage.

3. Invokes `addArgument` on the cloned object for each argument in the application, storing them later use.

4. Invokes `enter` on the function object. This will reduce the object to WHNF, which has three cases:

   - The function is partially applied, so hasn't yet received all of the necessary arguments to be evaluated. Such a function is already in WHNF, so we can just return it.
   - The function has exactly the right number of arguments, so WHNF demands we reduce it. This is implemented by calling the `inner` function that performs the actual implementation of the Haskell function with the free variables and arguments we've stored, then ensuring the result has been evaluated to WHNF by calling `enter`, then returning it.
   - The function is over-applied. This case looks complicated, it's two simple steps. We pretend we have an application of exactly the right number of arguments as in the above case, then instead of returning the result we cast it to a `Function` object and perform a normal function application with all the leftover arguments.

All of the functions defined in a Haskell program are compiled into their pairs of Java functions within a single class, the 'main' class. Datatypes are compiled into their own classes which are then referenced by the main class. This approach to function compilation differs from the approaches taken by Scala and Kotlin (other languages targeting the JVM), which compile lambda expressions into anonymous classes.

In Haskell, the vast majority of expressions are function applications by the time the source has reached ILB. To provide lazy semantics, each expression has to be evaluatable without forcing other expressions, so each function implementation is quite small. This results in a lot of functions being generated. Using anonymous classes to implement Haskell functions would result in hundreds or thousands of small Java classes, whereas using Java functions results in far fewer classes and more functions inside a single class.

This is meant to be a reflective design discussion: say something about it being interesting to compare tradeoffs in performance and size? Also cost of class loading?

0.1.7.3 JVM Sanitisation

Haskell[7], Java[8], and JVB[9] all allow different sets of strings as valid identifiers: for example, in Java and JVB `Temp` is a valid variable name, but in Haskell it's not (identifiers with uppercase Unicode start characters are reserved for constructor names like `True`). `+` is a valid identifier in Haskell and JVB, but not in Java.

Additionally name conflicts can occur between builtin classes used by the compiler (eg. `Function` and `Data`) and constructor names in the Haskell source (eg. `data Function = Function`).

JVM Sanitisation is a name conversion process used in the code generator to prevent conflicts and invalid variable names when everything's been lowered into JVB:

- All names that have come from Haskell source are prefixed with an underscore, and any builtin classes are forbidden from starting with an underscore. This prevents name clashes.

- Any non-alphanumeric (Unicode) characters in a Haskell source identifier are replaced by their Unicode codepoint in hexadecimal, flanked on either side by $ symbols. This is more restrictive than necessary, as JVB allows most unicode characters, but is a safe and simple defence against conflicts. Using $ symbols to mark the start and end of a sanitised character ensures that identifiers are uniquely decodable and prevents two distinct identifiers from clashing when sanitised (without delimiters, the valid Haskell identifiers π and `CF80` are sanitised into the same identifier: `_CF80`. With the delimiters, π is sanitised into `_$CF80$`).

0.1.7.4 Notable Instructions

JVB uses variable-length instructions, and also instructions where the length varies between usages (not the same length each time). The `nop` (no-op) instruction is a single byte, whereas the `goto` instruction is 3 bytes (the opcode followed by a two-byte operand forming the address to jump to).

The `lookupswitch` instruction is a low-level implementation of a `switch` statement in Java: it compares an `int` on the top of the stack with a set of values, jumping to an address paired with each value or to a default address if no values match. The interesting part of this instruction is that the length varies between usages:

Opcode	Padding $0 \leq p \leq 3$	Default offset	Num. cases $0 \leq n \leq 2^{31}$	Match_1	Offset_1	… $n-1$
1 byte	p bytes	4 bytes	4 bytes	4 bytes	4 bytes	$8(n-1)$ bytes

[7] https://www.haskell.org/onlinereport/lexemes.html
[8] https://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.8
[9] https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.2.2