$$\frac{\Gamma \vdash f : A \to A}{\Gamma \vdash fix f : A} \textbf{(Type-Fix)}$$

$$\frac{\Gamma, x : A \vdash t : A}{\Gamma \vdash fix\ (\lambda x.t) \longrightarrow_\beta t[x := (\lambda x.t)] : A} \textbf{(Eval-Fix)}$$

Figure 2.17: `fix` semantics

The semantics of the function `fix` are described in Figure 2.17. This definition is sufficient to define e.g., the recursive computation of a Fibonacci number or a local recursive binding as below.

```
fib = fix (λf. λn. if (isLess n 2) n (add (f (n - 1)) (f (n-2))))

evenOdd
  : (isEven : Nat → Bool) × (isOdd : Nat → Bool) × Top
  = fix (λf. ( if isZero x then true else f.isOdd (pred x)
             , if isZero x then false else f.isEven (pred x)
             , Top
             ))
```

2.4 Remaining constructs

These constructs together form a complete core language capable of forming and evaluating expressions. Already, this would be a usable programming language. However, the *surface language* is still missing: the syntax for defining constants and variables, and interacting with the compiler.

Local definitions The λ-calculus is, to use programming language terminology, a purely functional programming language: without specific extensions, any language construct is an expression. We will use the syntax of Agda, and keep local variable definition as an expression as well, using a `let-in` construct, with the semantics given in Figure 2.18.

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \text{let } x = a \text{ in } b : B} \textbf{(Type-Let)}$$

$$\frac{\Gamma \vdash v : A \qquad \Gamma, x : A \vdash e : B}{\text{let } x = v \text{ in } e \longrightarrow_\zeta e[x := v]} \textbf{(Eval-Let)}$$

Figure 2.18: `let-in` semantics

Global definitions Global definitions are not strictly necessary, as with local definitions and the fixed-point combinator we could emulate them. However, global definitions will be useful later in the process of elaborations, when global top-level definitions will separate blocks that we can type-check separately. We will add three top-level expressions: a declaration that only assigns a name to a type, and a definition with and without type. Definitions without types will have them inferred.

$$\begin{aligned} top &:= id : term \\ &\mid id : term = term \\ &\mid id = term \end{aligned}$$

Holes A construct that serves solely as information to the compiler and will not be used at runtime is a *hole*, which can take the place of a term in an expression and marks the missing term as one to be inferred ("filled in") during elaboration[2]. In fact, the syntax for a global definition without a type will use a hole in place of its type. The semantics of a hole are omitted on purpose as they would also require specifying the type inference algorithm.

$$term := \_$$

Interpreter directives Another type of top-level expressions is a pragma, a direct command to the compiler. We will use these when evaluating the time it takes to normalize or elaborate an expression, or when enabling or disabling the use of "wired-in" types, e.g. to compare the performance impact of using a Church encoding of numbers versus a natural type that uses hardware integers. We will once again use the syntax of Agda:

$$\begin{aligned} top &:= \{-\#\ BUILTIN\ id\ \#-\} \\ &\mid \{-\#\ ELABORATE\ term\ \#-\} \\ &\mid \{-\#\ NORMALIZE\ term\ \#-\} \end{aligned}$$

Polyglot Lastly, the syntax for one language feature that will only be described and implemented in Chapter 5: a way to execute code in a different language in a *"polyglot"* mode, which is a feature that Truffle offers. This is a three-part expression containing the name of language to use, the foreign code, and the type this expression should have:

$$term := [|\ id\ |\ foreign\ |\ term\ |]$$

The syntax and semantics presented here altogether comprise a working programming language. A complete listing of the semantics is included in Appendix B. The syntax, written using the notation of the ANTLR parser generator is in Figure 2. The syntax does not mention constants like *true* or *Nat*, as they will be implemented as global definitions bound in the initial type-checking context and do not need to be recognized during parsing.

With this, the language specification is complete and we can move on to the next part, implementing a type-checker and an interpreter for this language.

[2] Proof assistants also use the concept of a metavariable, often with the syntax $?\alpha$.

```
FILE : STMT (STMTEND STMT)* ;
STMT : "{-#" PRAGMA "#-}"
     | ID ":" EXPR
     | ID (":" EXPR)? "=" EXPR
     ;
EXPR : "let" ID ":" EXPR "=" EXPR "in" EXPR
     | "λ" LAM_BINDER "." EXPR
     | PI_BINDER+ "→" EXPR
     | ATOM ARG*
     ;
LAM_BINDER : ID | "_" | "{" (ID | "_") "}" ;
PI_BINDER : ATOM ARG* | "(" ID+ ":" EXPR ")" | "{" ID+ ":" EXPR "}" ;
ARG : ATOM | "{" ID ("=" TERM)? "}" ;
ATOM : "[" ID "|" FOREIGN "|" TERM "]"
     | EXPR "×" EXPR
     | "(" EXPR ("," EXPR)+ ")"
     | "(" EXPR ")"
     | ID "." ID
     | ID
     | NAT
     | "*"
     | "_"
     ;
 STMTEND : ("\n" | ";")+ ;
 ID : [a-zA-Z] [a-zA-Z0-9] ;
 SKIP : [ \t] | "--" [^\r\n]* | "{-" [^#] .* "-}" ;
// pragma discussed in text
```

Listing 2: Montuno syntax (using simplified ANTLR syntax)

Chapter 3

Language implementation: Montuno

3.1 Introduction

We will first create an interpreter for Montuno as specified in the assignment, and also because evaluation and elaboration algorithms from the literature are quite naturally translated to a functional-style program, which is not really possible in Truffle, the target implementation, where annotated classes are the main building block.

The Truffle implementation, which we will see in the following chapters, has a much higher conceptual overhead as we will need to care about low-level implementation details, e.g. implementing the actual function calls. In this interpreter, though, we will simply use the features of our host language.

We will use Kotlin as our language of choice, as it is a middle ground between plain Java and functional JVM-based languages like Scala or Clojure. While the main target language of Truffle is Java, Kotlin also supports class and object annotations on which the Truffle DSL is based [1]. Functional style, on the other hand, makes our implementation of the algorithms simpler and more concise [2].

The choice of the platform (JVM) and the language (Kotlin) also clarifies the choice of supporting libraries. In general, we are focused on the algorithmic part of the implementation, and not on speed or conciseness, which means that we can simplify our choices by using the most widely used libraries:

- Gradle as the build system,
- JUnit as the testing framework,
- ANTLR as the parser generator,
- JLine as the command-line interface library.

[1] Even though Kotlin seems not to be recommended by Truffle authors, there are several languages implemented in it, which suggests there are no severe problems. "[...] and Kotlin might use abstractions that don't properly partially evaluate." (from https://github.com/oracle/graal/issues/1228)

[2] Kotlin authors claim 40% reduction in the number of lines of code, (from https://kotlinlang.org/docs/faq.html)

Libraries Truffle authors recommend against using many external libraries in the internals of the interpreter, as the techniques the libraries use may not work well with Truffle. Therefore, we will need to design our own supporting data structures based on the fundamental data structures provided directly by Kotlin. Only two external libraries would be too complicated to reimplement, and both of these were chosen because they are among the most widely used in their field:

- a parser generator, ANTLR, to process input into an abstract syntax tree,
- a terminal interface library, JLine, to implement the interactive interface.

For the build and test system, the recommended choices of Gradle and JUnit were used.

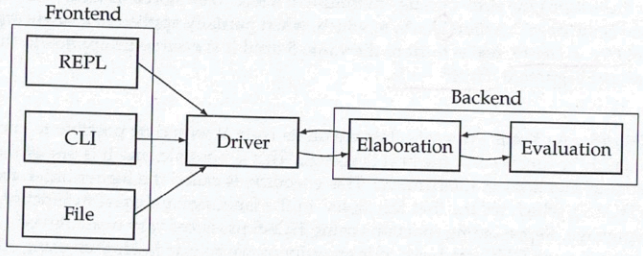

Figure 3.1: Overview of interpreter components

3.1.1 Program flow

A typical interpreter takes in the user's input, processes it, and outputs a result. In this way, we can divide the interpreter into a frontend, a driver, and a backend, to reuse compiler terminology. A frontend handles user interaction, be it from a file or from an interactive environment, a backend implements the language semantics, and a driver connects them, illustrated in Figure 3.1.

Frontend The frontend is intended to be a simple way to execute the interpreter, offering two modes: a batch processing mode that reads from a file, and an interactive terminal environment that receives user input and prints out the result of the command. Proof assistants like Agda offer deeper integration with editors like tactics-based programming or others, similar to the refactoring tools offered in development environments for object-oriented languages, but that is unnecessary for the purposes of this thesis.

Backend The components of the backend, here represented as *elaboration* and *evaluation*, implement the data transformation algorithms that are further illustrated in Figure 3.2. In brief, the *elaboration* process turns user input in the form of partially-typed, well-formed *pre-terms* into fully-annotated well-typed *terms*. *Evaluation* converts between a *term* and a *value*: a term can be compared to program data, it can only be evaluated, whereas a value is the result of such evaluation and can be, e.g., compared for equality.

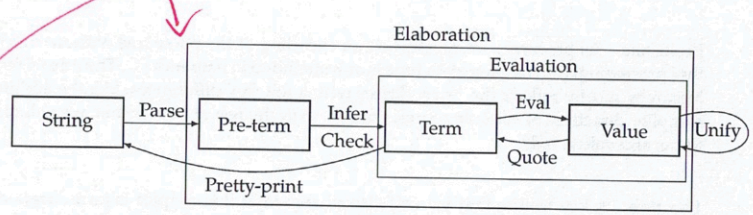

Figure 3.2: Data flow overview

Data flow In Figure 3.2, *Infer* and *Check* correspond to type checking and type inference, two parts of the *bidirectional typing* algorithm that we will use. *Unification* (*Unify*) forms a major part of the process, as that is how we check whether two values are equal. *Eval* corresponds to the previously described βδζι-reduction implemented using the *normalization-by-evaluation* style, whereas *Quote* builds a term back up from an evaluated value. To complete the description, *Parse* and *Pretty-print* convert between the user-readable, string representation of terms and the data structures of their internal representation. For the sake of clarity, the processes are illustrated using their simplified function signatures in Listing 3.1.

```
fun parse(input: String): PreTerm;
fun pprint(term: Term): String;
fun infer(pre: PreTerm): Pair<Term, Val>;
fun check(pre: PreTerm, wanted: Val): Term;
fun eval(term: Term): Val;
fun quote(value: Val): Term;
fun unify(left: Val, right: Val): Unit;
```

Listing 3.1: Simplified signatures of the principal functions

In this chapter, we will first define the data types, especially focusing on closure representation. Then, we will specify and implement two algorithms: *normalization-by-evaluation*, and *bidirectional type elaboration*, and lastly, we finish the interpreter by creating its driver and frontend.

3.2 Data structures

We have specified the syntax of the language in the previous chapter, which we first need to translate to concrete data structures before trying to implement the semantics. Sometimes, the semantics impose additional constraints on the design of the data structures, but in this case, the translation is quite straight-forward.

Properties Terms and values form recursive data structures. We will also need a separate data structure for pre-terms as the result of parsing user input. All of these structures represent only well-formed terms and, in addition, terms and value represent the well-typed subset of well-formed terms. Well-formedness should be ensured by the parsing process, whereas type-checking will take care of the second property.

Pre-terms As pre-terms are mostly just an encoding of the parse tree without much further processing, the complete data type is only included in Appendix C. The `PreTerm` class hierarchy mostly reflects the `Term` classes with a few key differences, like the addition of compiler directives or variable representation, so in the rest of this section, we will discuss terms and values only.

Location A key feature that we will also disregard in this chapter is term location that maps the position of a term in the original source expression, mostly for the purpose of error reporting. As location is tracked in a field that occurs in all pre-terms, terms, and values, it will only be included in the final listing of classes in Appendix C.

$$
\begin{array}{lcllll}
\mathit{term} & := & v & \mid \mathit{constant} & & \\
 & \mid & a\ b & \mid a\ \{b\} & & \\
 & \mid & a \to b & \mid (a : A) \to b & \mid \{a : A\} \to b & \\
 & \mid & a \times b & \mid (l : A) \times b & \mid a.l & \\
 & \mid & \text{let } x = v \text{ in } e & \mid [\mid id \mid foreign \mid type \mid] & & \\
 & \mid & \_ & & & \\
\mathit{value} & := & \mathit{constant} & & & \\
 & \mid & \lambda x : A.b & \mid \Pi x : A.b & & \\
 & \mid & (a_1, \cdots, a_n) & & & \\
 & \mid & \_ & & &
\end{array}
$$

Figure 3.3: Terms and values in Montuno (revisited)

The terms and values that were specified in Chapter 2 are revisited in Figure 3.3, there are two main classes of terms: those that represent computation (functions and function application), and those that represent data (pairs, records, constants).

Data classes Most *data* terms can be represented in a straight-forward way, as they map directly to features of the host language, Kotlin in our case. Kotlin has a standard way of representing primarily data-oriented structures using `data classes`. These are classes whose primary purpose is to hold data, so-called Data Transfer Objects (DTOs), and are the recommended approach in Kotlin[4]. In Listing 3.2 we have the base classes for terms and values, and a few examples of structures that map directly from the syntax to a data object.

```
sealed class Term
sealed class Value

data class TLet(val id: String, val bind: Term, val body: Term) : Term()
data class TSigma(val id: String, val type: Term, val body: Term) : Term()
data class TPair(val left: Term, val right: Term) : Term()

data class VPair(val left: Value, val right: Value) : Value()
```

Listing 3.2: Pair and *let − in* representations

[4] https://kotlinlang.org/docs/idioms.html

Terms that encode computation, whether delayed (λ-abstraction) or not (application) will be more involved. Variables *can* be represented in a straight-forward way, but a string-based representations is not the most optimal way. We will look at these three constructs in turn.

3.2.1 Functions

Closure Languages, in which functions are first-class values, all use the concept of a closure. A closure is, in brief, a function in combination with the environment in it was created. The body of the function can refer to variables other than its immediate arguments, which means the surrounding environment needs to be stored as well. The simplest example is the *const* function: $\lambda x.\lambda y.x$, which, when partially applied to a single argument, e.g., let $five = const5$, needs to store the value 5 until it is eventually applied to the remaining second argument: $five15 \longrightarrow 5$.

HOAS As Kotlin supports closures on its own, it would be possible to encode λ-terms directly as functions in the host language. This is possible, and it is one of the ways of encoding functions in interpreters. This encoding is called the higher-order abstract syntax (HOAS), which means that functions[5] in the language are equal to functions in the host language. Representing functions using HOAS produces very readable code, and in some cases, e.g., on GHC produces code an order of magnitude faster than using other representations [?]. An example of what it looks like is in Listing 3.3.

```
data class Closure<T>(val fun: (T) -> T)

val constFive = Closure<Int> { (n) -> 5 }
```

Listing 3.3: Higher-order abstract syntax encoding of a closure

Explicit closures However, we will need to perform some operations on the AST that need explicit access to environments and the arguments of a function. The alternative to reusing functions of the host language is a *defunctionalized* representation, also called *explicit closure* representation. We will need to use this representation later, when creating the Truffle version: function calls will need to be objects, nodes in the program graph, as we will see in Chapter 4. In this encoding, demonstrated in Listing 3.4, we store the term of the function body together with the state of the environment when the closure was created.

```
data class Closure<T>(val fun: Term, val environment: Map<Name, Term>)

val constFive = Closure<Int>(TLocal("x"), mapOf("x" to 5))
```

Listing 3.4: Defunctionalized function representation

[5] In descriptions of the higher-order abstract syntax, the term *binders* is commonly used instead of function or λ-abstractions, as these constructs *bind* a value to a name.

3.2.2 Variables

Representing variables can be as straight-forward as in Listing 3.4: a variable can be a simple string containing the name of the variable. This is also what our parser produces in the pre-term representation. Also, when describing reduction rules and substitution, we have also referred to variables by their names. That is not the best way of representing variables.

Named Often, when specifying a λ-calculus, the process of substitution $t[x := e]$ is kept vague, as a concern of the meta-theory in which the λ-calculus is encoded. When using variable names (strings), the terms themselves and the code that manipulates them are easily understandable. Function application, however, requires variable renaming (α-conversion), which involves traversing the entire argument term and replacing each variable occurrence with a fresh name that does not yet occur in the function body. However, this is a very slow process, and it is not used in any real implementation of dependent types or λ-calculus.

Nameless An alternative to string-based variable representation is a *nameless* representation, which uses numbers in place of variable names [?]. These numbers are indices that point to the current variable environment, offsets from the top or the top of the environment stack. The numbers are assigned, informally, by *counting the lambdas*, as each λ-abstraction corresponds to one entry in the environment. The environment can be represented as a stack to which a variable is pushed with every function application, and popped when leaving a function. The numbers then point to these entries. These two approaches can be seen side-by-side in Figure 3.4.

	fix	*succ*
Named	$(\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ g$	$\lambda x.x\ (\lambda y.x\ y)$
Indices	$(\lambda(\lambda 1\ (0\ 0))\ (\lambda 1\ (0\ 0)))\ g$	$\lambda 0\ (\lambda 1\ 0)$
Levels	$(\lambda(\lambda 0\ (1\ 1))\ (\lambda 0\ (1\ 1)))\ g$	$\lambda 0\ (\lambda 0\ 1)$

Figure 3.4: Named and nameless variable representations

de Bruijn indices The first way of addressing, de Bruijn indexing, is rather well-known. It is a way of counting from the top of the stack, meaning that the argument of the innermost lambda has the lowest number. It is a "relative" way of counting, relative to the top of the stack, which is beneficial during e.g. δ-reduction, in which a reference to a function is replaced by its definition: using indices, the variable references in the function body do not need to be adjusted after such substitution.

de Bruijn levels The second way is also called the "reversed de Bruijn indexing" [?], as it counts from the start of the stack. This means that the argument of the innermost lambda has the highest number. In the entire term, one variable is only ever addressed by one number, meaning that this is an "absolute" way of addressing, as opposed to the "relative" indices.

Locally nameless There is a third alternative that combines both named and nameless representations, and it has been used in e.g., the Lean proof assistant [?]. De Bruijn indices are used for bound variables and string-based names for free variables. This also avoids any need for bound variable substitution, but free variables still need to be resolved later during the evaluation of a term.

Our choice We will use a representation that has been used in recent type theory implementations [?] [?]: de Bruijn indices in terms, and de Bruijn levels in values. Such a representation avoids any need for substitution as terms are that substituted *into* an existing value do not need to have the "relative" indices adjusted based on the size of the current environment, whereas the "absolute" addressing of levels in values means that values can be directly compared. This combination of representations means that we can doing avoid any substitution at all, as any adjustment of variables is performed during the evaluation from term to value and back.

Implementation Kotlin makes it possible to construct type-safe wrappers over basic data types that are erased at runtime but that support custom operations. Representing indices and levels as `inline classes` means that we can perform add and subtract them using the natural syntax e.g. `ix + 1`, which we will use when manipulating the environment in the next section. The final representation of variables in our interpreter is in Listing 3.5.

```
inline class Ix(val it: Int) {
    operator fun plus(i: Int) = Ix(it + i)
    operator fun minus(i: Int) = Ix(it - i)
    fun toLvl(depth: Lvl) = Lvl(depth.it - it - 1)
}

inline class Lvl(val it: Int) {
    operator fun plus(i: Int) = Lvl(it + i)
    operator fun minus(i: Int) = Lvl(it - i)
    fun toIx(depth: Lvl) = Ix(depth.it - it - 1)
}

data class VLocal(val it: Lvl) : Val()
data class TLocal(val it: Ix) : Val()
```

Listing 3.5: Variable representation

3.2.3 Class structure

Variables and λ-abstractions were the two non-trivial parts of the mapping between our syntax and Kotlin values. With these two pieces, we can fill out the remaining parts of the class hierarchy. The full class listing is in Appendix C, here only a direct comparison of the data structures is shown on the *const* function in Figure 3.5, and the most important differences between them are in Figure 3.6.

```
PLam("x", Expl,     TLam("x", Expl,     VLam("x", Expl,
  PLam("y", Expl,     TLam("y", Expl,     VCl([valX], VLam("y", Expl,
    PVar("x")))         TLocal(1)))         VCl([valX, valY], VLocal(0)))))
```

Figure 3.5: Direct comparison of `PreTerm`, `Term`, and `Value` objects

	Variables	Functions	Properties
`PreTerm`	String names	`PreTerm` AST	well-formed
`Term`	de Bruijn index	`Term` AST	well-typed
`Value`	de Bruijn level	Closure (`Term` AST + `Values` in context)	normal form

Figure 3.6: Important distinctions between `PreTerm`, `Term`, and `Value` objects

3.3 Normalization

3.3.1 Approach

Normalization-by-evaluation Normalization is a series of $\beta\delta\zeta\iota$-reductions, as defined in Chapter 2. While there are systems that implement normalization as an exact series of reduction rules, it is an inefficient approach that is not common in the internals of state-of-the-art proof assistants. An alternative way of bringing terms to normal form is the so-called *normalization-by-evaluation* (NbE) [?]. The main principle of this technique is interpretation from the syntactic domain of terms into a computational, semantic domain of values and back. In brief, we look at terms as an executable program that can be *evaluated*, the result of such evaluation is then a normal form of the original term. NbE is total and provably confluent [?] for any abstract machine or computational domain.

Neutral values If we consider only closed terms that reduce to a single constant, we could simply define an evaluation algorithm over the terms defined in the previous chapter. However, normalization-by-evaluation is an algorithm to bring any term into a full normal form, which means evaluating terms inside function bodies and constructors. NbE introduces the concept of "stuck" values that cannot be reduced further. In particular, free variables in a term cannot be reduced, and any terms applied to a stuck variable cannot be further reduced and are "stuck" as well. These stuck values are called *neutral values*, as they are inert with regards to the evaluation algorithm.

Semantic domain Proof assistants use abstract machines like Zinc or STG; any way to evaluate a term into a final value is viable. This is also the reason to use Truffle, as we can translate a term into an executable program graph, which Truffle will later optimize as necessary. In this first interpreter, however, the computational domain will be a simple tree-traversal algorithm.

The set of neutral values in Montuno is rather small (Figure 3.7): an unknown variable, function application with a neutral *head* and arbitrary terms in the *spine*, and a projection eliminator.

$$neutral \;:=\; var \;\mid\; neutral\ a_1 \ldots a_n \;\mid\; neutral.l_n$$

Figure 3.7: Neutral values

Specification The NbE algorithm is fully formally specifiable using four operations: the above-mentioned evaluation and quoting, reflection of a neutral value (*NeVal*) into a value, and reification of a value into a normal value (*NfVal*) that includes its type, schematically shown in Figure 3.8. In this thesis, though, will only describe the relevant parts of the specification in words, and say that NbE (as we will implement it) is a pair of functions $nf = quote(eval(term))$,

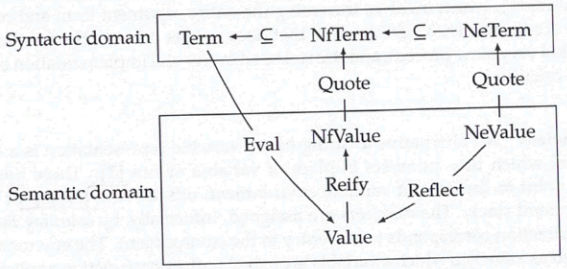

Figure 3.8: Syntactic and semantic domains in NbE [?]

3.3.2 Normalization strategies

Normalization-by-evaluation is, however, at its core inefficient for our purposes [?]. The primary reason to normalize terms in the interpreter is for type-checking and inference and that, in particular, needs normalized terms to check whether two terms are equivalent. NbE is an algorithm to get a full normal form of a term, whereas to compare values for equality, we only need the weak head-normal form. To illustrate: to compare whether a λ-term and a pair are equal, we do not need to compare two fully-evaluated values, but only to find out whether the term is a pair of a λ-term, which is given by the outermost constructor, the *head*.

In Chapter 2 we saw an overview of normal forms of λ-calculus. To briefly recapitulate, a normal form is a fully evaluated term with all sub-terms also fully evaluated. A weak head-normal form is a form where only the outermost construction is fully evaluated, be it a λ-abstraction or application of a variable to a spine of arguments.

Reduction strategy Normal forms are associated with a reduction strategy, a set of small-step reduction rules that specify the order in which sub-expressions are reduced. Each strategy brings an expression to their corresponding normal form. Common ones are *applicative order* in which we first reduce sub-expressions left-to-right, and then apply functions to them; and *normal order* in which we first apply the leftmost function, and only then reduce its arguments. In Figure 3.9 there are two reduction strategies that we will emulate.

$$x \xrightarrow{name} x \qquad x \xrightarrow{norm} x$$

$$\frac{}{(\lambda x.e) \xrightarrow{name} (\lambda x.e)} \qquad \frac{e \xrightarrow{norm} e'}{(\lambda x.e) \xrightarrow{norm} (\lambda x.e')}$$

$$\frac{e_1 \xrightarrow{name} (\lambda x.e) \qquad e[x := e_2] \xrightarrow{name} e'}{(e_1\ e_2) \xrightarrow{name} e'} \qquad \frac{e_1 \xrightarrow{name} (\lambda x.e) \qquad e[x := e_2] \xrightarrow{norm} e'}{(e_1\ e_2) \xrightarrow{norm} e'}$$

$$\frac{e_1 \xrightarrow{name} e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \xrightarrow{name} (e_1'\ e_2)} \qquad \frac{e_1 \xrightarrow{name} e_1' \not\equiv \lambda x.e \qquad e_1' \xrightarrow{norm} e_1'' \qquad e_2 \xrightarrow{norm} e_2'}{(e_1\ e_2) \xrightarrow{norm} (e_1''\ e_2)}$$

(a) Call-by-name to weak head normal form

(b) Normal order to normal form

Figure 3.9: Reduction strategies for λ-calculus [?]

In general programming language theory, a concept closely related to reduction strategies is an evaluation strategy. These also specify when an expression is evaluated into a value, but in our case, they apply to our host language Kotlin.

Call-by-value Call-by-value, otherwise called eager evaluation, corresponds to applicative order reduction strategy [?]. Specifically, when executing a statement, its sub-expressions are evaluated inside-out and immediately reduced to a value. This leads to predictable program performance (the program will execute in the order that the programmer wrote it, evaluating all expressions in order), but this may lead to unnecessary computations performed: given an expression `const 5 (ackermann 4 2)`, the value of `ackermann 4 2` will be computed but immediately discarded, in effect wasting processor time.

Call-by-need Call-by-need, also lazy evaluation, is the opposite paradigm. An expression will be evaluated only when its result is first accessed, not when it is created or defined. Using call-by-need, the previous example will terminate immediately as the calculation `ackermann 4 2` will be deferred and then discarded. However, it also has some drawbacks, as the performance characteristics of programs may be less predictable or harder to debug.

Call-by-value is the prevailing paradigm, used in all commonly used languages with the exception of Haskell. It is sometimes necessary to defer the evaluation of an expression, however, and in such cases lazy evaluation is emulated using closures or zero-argument functions: e.g., in Kotlin a variable can be initialized using the syntax `val x by lazy { ackermann(4, 2) }`, and the value will only be evaluated if it is ever needed.

Call-by-push-value There is also an alternative paradigm, called call-by-push-value, which subsumes both call-by-need and call-by-value as they can be directly translated to CBPV–in the context of λ-calculus specifically. It defines additional operators *delay* and *force* to accomplish this, one to create a *thunk* that contains a deferred computation, one to evaluate

the thunk. Also notable is that it distinguishes between values and computations: values can be passed around, but computations can only be executed, or deferred.

Emulation We can emulate normalization strategies by implementing the full normalization-by-evaluation algorithm, and varying the evaluation strategy. Kotlin is by default a call-by-value language, though, and evaluation strategy is an intrinsic property of a language so, in our case, this means that we need to insert `lazy` annotations in the correct places, so that no values are evaluated other than those that are actually used. In the case of the later Truffle implementation, we will need to implement explicit *delay* and *force* operations of call-by-push-value, which is why we introduced all three paradigms in one place.

3.3.3 Implementation

The basic outline of the implementation is based on Christiansen's [?]. In essence, it implements the obvious evaluation algorithm: evaluating a function captures the current environment in a closure, evaluating a variable looks up its value in the environment, and function application inserts the argument into the environment and evaluates the body of the function.

Environments The brief algorithm description used a concept we have not yet translated into Kotlin: the environment, or evaluation context. When presenting the λ→-calculus, we have seen the typing context Γ, to which we add a value context.

$$\Gamma \ ::= \ \bullet \ \mid \ \Gamma, x : t$$

The environment, following the above definition, is a stack: defining a variable pushes a pair of a name and a type to the top, which is then popped off when the variable goes out of scope. An entry is pushed and popped whenever we enter and leave a function context, and the entire environment needs to be captured in its current state whenever we create a closure. When implementing closures in Truffle, we will also need to take care about which variables are actually used in a function. That way, we can capture only those that need to be captured and not the entire environment.

Linked list The natural translation of the environment definition is a linked list. It would also be the most efficient implementation in a functional language like Haskell, as appending to an immutable list is very cheap there. In Kotlin, however, we need to take care about not allocating too many objects and will need to consider mutable implementations as well.

Mutable/immutable In Kotlin and other JVM-based languages, an `ArrayDeque` is a fast data structure, a mutable implementation of the stack data structure. In general, array-backed data structures are faster than recursive ones on the JVM, which we will use in the Truffle implementation. In this first interpreter, however, we can use the easier-to-use immutable linked list implementation. It is shown in Listing 3.6, a linked list specialized for values; an equivalent structure is also implemented for types.

```
data class VEnv(val value: Val, val next: VEnv?)

fun VEnv?.len(): Int = if (this == null) 0 else 1 + next.len()
operator fun VEnv?.plus(v: Val): VEnv = VEnv(v, this)
operator fun VEnv?.get(n: Ix): Val
   = if (n.it == 0) this!!.value else this!!.next[n - 1]
```

Listing 3.6: Environment data structure as an immutable linked list

Environment operations We need three operations from an environment data structure: insert (bind) a value, look up a bound value by its level or index, and unbind a variable that leaves the scope. In Listing 3.6, we see two of them: the operator `plus`, used as `env + value`, binds a value, and operator `get`, used as `env[ix]`, looks a value up. Unbinding a value is implicit, because this is an immutable linked list: the reference to the list used in the outer scope is not changed by any operations in the inner scope. These operations are demonstrated in Listing 3.7, on the `eval` operations of a variable and a *let – in* binding.

There we also see the basic structure of the evaluation algorithm. Careful placement of `lazy` has been omitted, as it splits the algorithm into two: parts that need to be evaluated lazily and those that do not, but the basic structure should be apparent. The snippet uses the Kotlin `when-is` construct, which checks the class of the argument, in this case we check if `this` is a `TLocal`, `TLet`, etc.

```
fun eval(ctx: Context, term: Term, env: VEnv): Val = when (term) {
  is TLocal ->
    env[term.ix] ?: VNeutral(HLocal(Lvl(ctx.lvl - term.ix - 1), spineNil))
  is TLet -> eval(ctx, term.body, env + eval(ctx, term.defn, env))
  is TLam -> VLam(term.name, VCl(env, term.body))
  is TApp -> when (fn := eval(ctx, term.lhs, env)) {
    is VLam -> eval(ctx, fn.cl.term, fn.cl.env + eval(ctx, term.rhs, env))
    is VNeutral -> VNeutral(fn.head, fn.spine + term.right)
  }
  // ...
}
```

Listing 3.7: Demonstration of the `eval` algorithm

Eval In Listing 3.7, a variable is looked up in the environment, and considered a neutral value if the index is bigger than the size of the current environment. In `TLet` we see how an environment is extended with a local value. A λ-abstraction is converted into a closure. Function application, if the left-hand side is a `VLam`, evaluates the body of this closure, and if the left-hand side is a neutral expression, then the result is also neutral value and its spine is extended with another argument. Other language constructs are handled in a similar way,

Quote In Listing 3.8, we see the second part of the algorithm. In the domain of values, we do not have plain variable terms, or *let – in* bindings, but unevaluated functions and "stuck" neutral terms. A λ-abstraction, in order to be in normal form, needs to have its body also in normal form, therefore we insert a neutral variable into the environment in place of the argument, and eval/quote the body. A neutral term, on the other hand, has

at its head a neutral variable. This variable is converted into a term-level variable, and the spine reconstructed as a tree of nested `TApp` applications.

```
fun quote(ctx: Context, v: Val): Term = when (v) {
  is VNeutral -> {
    x = TLocal(Ix(ctx.depth - v.head - 1))
    for (vSpine in v.spine.reversed()) {
        x = TApp(x, quote(ctx, vSpine))
    }
    x
  }
  is VLam -> TLam(v.name,
      quote(ctx, eval(ctx, v.cl.body, v.cl.env + VNeutral(HLocal(ctx.lvl)))))
  // ...
}
```

Listing 3.8: Demonstration of the `quote` algorithm

These two operations work together, to fully quote a value, we need to also lazily `eval` its sub-terms. The main innovation of the normalization-by-evaluation approach is the introduction of neutral terms, which have the role of a placeholder value in place of a value that has not yet been supplied. As a result, the expression $quote(eval(term, emptyEnv))$ produces a lazily evaluated normal form of a term in a weak head-normal form, with its sub-terms being evaluated whenever accessed. Printing out such a term would print out the fully normalized normal form.

Primitive operations Built-in language constructs like *Nat* or *false* that have not been shown in the snippet are mostly inserted into the initial context as values that can be looked up by their name. In general, though, constructs with separate syntax, e.g. Σ-types, consist of three parts:

- their type is bound in the initial context;
- the term constructor is added to the set of terms and values, and added in `eval()`;
- the eliminator is added as a term and as a spine constructor, i.e., an operation to be applied whenever the neutral value is provided.

The full listing is provided in the supplementary source code, as it is too long to be included in text.

3.4 Elaboration

3.4.1 Approach

The second part of the internals of the compiler is type elaboration. Elaboration is the transformation of a partially-specified, well-formed program submitted by a user into a fully-specified, well-typed internal representation [?]. In particular, we will use elaboration to infer types of untyped Curry-style λ-terms, and to infer implicit function arguments that were not provided by the user, demonstrated in Figure 3.10.

function signature:	$id : \{A\} \to A \to A$
provided expression:	$id\ id\ 5$
elaborated expression:	$(id\ \{Nat \to Nat\}\ id)\ \{Nat\}\ 5$

Figure 3.10: Demonstration of type elaboration

Bidirectional typing Programmers familiar with statically-typed languages like Java are familiar with type checking, in which all types are provided by the user, and therefore are inputs to the type judgment $\Gamma \vdash e : t$. Omitting parts of the type specification means that the type system not only needs to check the types for correctness, but also infer (synthesize) types: the type t in $\Gamma \vdash e : t$ is produced as an output. In some systems, it is possible to omit all type annotations and rely only on the type constraints of built-in functions and literals. Bidirectional systems that combine both input and output modes of type judgment are now a standard approach [?], often used in combination with constraint solving.

Judgments The type system is composed of two additional type judgments we haven't seen yet that describe the two directions of computation in the type system:

- $\Gamma \vdash e \Rightarrow t$ is "given the context Γ and term e, infer (synthesize) its type t", and
- $\Gamma \vdash e \Leftarrow t$ is "given the context Γ, term e and type t, check that t is a valid type for t".

The entire typing system described in Chapter 2 can be rewritten using these type judgments. The main principle is that language syntax is divided into two sets of constructs: those that constrain the type of a term and can be checked against an inferred term, and those that do not constrain the type and need to infer it entirely.

$$\frac{a : t \in \Gamma}{\Gamma \vdash a \Rightarrow t}\ \textbf{(Var)} \qquad \frac{c \text{ is a constant of type } t}{\Gamma \vdash c \Rightarrow t}\ \textbf{(Const)}$$

$$\frac{\Gamma, x : t \vdash e \Leftarrow u}{\Gamma \vdash \lambda x.e \Leftarrow t \to u}\ \textbf{(Abs)} \qquad \frac{\Gamma \vdash f \Rightarrow t \to u \qquad \Gamma \vdash a \Rightarrow t}{\Gamma \vdash f\ a \Rightarrow u}\ \textbf{(App)}$$

$$\frac{\Gamma \vdash a \Rightarrow t \qquad \Gamma \vdash a = b}{\Gamma \vdash a \Leftarrow b}\ \textbf{(ChangeDir)} \qquad \frac{\Gamma \vdash a \Leftarrow t}{\Gamma \vdash (a : t) \Rightarrow t}\ \textbf{(Ann)}$$

Figure 3.11: Bidirectional typing rules for the $\lambda\!\to$-calculus

Bidirectional $\lambda\!\to$ typing In Figure 3.11, this principle is demonstrated on the simply-typed λ-calculus with only variables, λ-abstractions and function application. The first four rules correspond to rules that we have introduced in Chapter 2, with the exception of the constant rule that we have not used there. The two new rules are **(ChangeDir)** and **(Ann)**: **(ChangeDir)** says that if we know that a term has an already inferred type, then we can satisfy any rule that requires that the term checks against a type equivalent to this one. **(Ann)** says that to synthesize the type of an annotated term $a : t$, the term first needs to check against that type.

Rules **(Var)** and **(Const)** produce an assumption, if a term is already in the context or a constant, then we can synthesize its type. In rule **(App)**, if we have a function with an

inferred type then we check the type of its argument, and if it holds then we can synthesize the type of the application $f\ a$. To check the type of a function in rule **(Abs)**, we first need to check whether the body of a function checks against the type on the right-hand side of the arrow.

While slightly complicated to explain, this description produces a provably sound and complete type-checking system [?] that, as a side effect, synthesizes any types that have not been supplied by the user. Extending this system with other language constructs is not complex: the rules used in Montuno for local and global definitions are in Figure 3.12.

$$\frac{\Gamma \vdash t \Leftarrow \star \qquad \Gamma \vdash a \Leftarrow t \qquad \Gamma, x : t \vdash b \Rightarrow u}{\Gamma \vdash \text{let } x : t = a \text{ in } b \Rightarrow u}\ \textbf{(Let-In)}$$

$$\frac{\Gamma \vdash t \Leftarrow \star \qquad \Gamma \vdash a \Leftarrow t}{\Gamma \vdash x : t = a \Rightarrow t}\ \textbf{(Defn)}$$

Figure 3.12: Bidirectional typing rules for *let* − *in* and top-level definitions

Meta-context One concern was not mentioned in the previous description: when inferring a type, we may not know all its component types: in rule **(Abs)**, the type of the function we check may only be constrained by the way it is called. Implicit function arguments $\{A\ B\} \to A \to B \to A$ also only become specific when the function is actually called. The solution to this problem is a *meta-context* that contains *meta-variables*.

These stand for yet undetermined terms [?], either as placeholders to be filled in by the user in interactive proof assistants (written with a question mark, e.g. as $?\alpha$), or terms that can be inferred from other typing constraints using unification. These meta-variables can be either inserted directly by the user in the form of a hole "_", or implicitly, when inferring the type of a λ-abstraction or an implicit function argument [?].

There are several ways of implementing this context depending on the scope of meta-variables, or whether it should be ordered or the order of meta-variables does not matter. A simple-to-implement but sufficiently useful for our purposes is a globally-scoped meta-context divided into blocks placed between top-level definitions.

```
id : {A} → A → A = λx.x
?α = Nat
?β = ?α → ?α
five = (id ?β id) ?α 5
```

Listing 3.9: Meta-context for the expression `id id 5`

The meta-context implemented in Montuno is demonstrated in Listing 3.9. When processing a file, we process top-level expressions sequentially. The definition of the *id* function is processed, and in the course of processing *five*, we encounter two implicit arguments, which are inserted on the top-level as the meta-variables $?\alpha$ and $?\beta$.

3.4.2 Unification

Returning to the rule (**ChangeDir**) in Figure 3.12, a critical piece of the algorithm is the equality of two types that this rule uses. To check a term against a type $\Gamma \vdash a \Leftarrow t$, we first infer a type for the term $\Gamma \vdash a \Rightarrow u$, and then test its equivalence to the wanted type $t = u$.

Conversion checking The usual notion of equivalence in λ-calculus is *α-equivalence of β-normal forms*, that we discussed in Chapter 2, which corresponds to structural equality of the two terms. *Conversion checking* is the algorithm that determines if two terms are convertible using a set of conversion rules.

Unification As we also use meta-variables in the type elaboration process, these variables need to be solved in some way. This process of conversion checking together with solving meta-variables is called *unification* [?] and is a well-studied problem in the field of type theory.

Pattern unification In general, solving meta-variables is undecidable [?]. Given the constraint $?\alpha\ 5 = 5$, we can produce two solutions: $?\alpha = \lambda x.x$ and $?\alpha = \lambda x.5$. There are several possible approaches and heuristics: first-order unification solves for base types and cannot produce functions as a result; higher-order unification can produce functions but is undecidable; *pattern unification* is a middle ground and can produce functions as solutions, with some restrictions.

Renaming In this thesis, I have chosen to reuse the algorithm from [?] which, in brief, assumes that a meta-variable is a function whose arguments are all local variables in scope at the moment of its creation. Then, when unifying the meta-variable with another (non-variable) term, it builds up a list of variables the term uses, and stores such a solution as a *renaming* that maps the arguments to a meta-variable to the variables with which it was unified. As the algorithm is rather involved but tangential to the goals of this thesis, I will omit a detailed description and instead point an interested reader at the original source [?].

3.4.3 Implementation

As with the implementation of normalization-by-evaluation, we will look at the most illustrative parts of the implementation. This time, the comparison can be made directly side-by-side, between the bidirectional typing algorithm and its implementation.

What was not mentioned explicitly is that the type elaboration algorithm has `PreTerms` as its input, and produces `Terms` in the case of type checking, and pairs of `Terms` and `Values` (the corresponding types) in the case of type inference. Unification, not demonstrated here, is implemented as parallel structural recursion over two `Value` objects.

In Figure 3.13, we see the previously described rule that connects the checking and synthesis parts of the algorithm and uses unification. Unification solves meta-variables as a side-effect, here it is only in the role of a guard as it does not produce a value. The code exactly

follows the typing rule, the pre-term is inferred, resulting in a pair of a well-typed term and its type, the type is unified with the wanted type and, if successful, the produced term is the return value.

$$\frac{\Gamma \vdash a \Rightarrow t \qquad \Gamma \vdash a = b}{\Gamma \vdash a \Leftarrow b}\ (\textbf{ChangeDir})$$

```
fun LocalContext.check(pre: PreTerm, wanted: Value): Term = when (pre) {
  // ...
  else -> {
    val (t, actual) = infer(pre.term)
    unify(actual, wanted)
    t
  }
}
```

Figure 3.13: Side-by-side comparison of the **ChangeDir** rule

Figure 3.14 shows the exact correspondence between the rule and its implementation, one read left-to-right, the other top-to-bottom. Checking of the type and value are straightforward, translation of $\Gamma, x : t \vdash b \Rightarrow u$ binds a local variable in the environment, so that the body of the *let − in* expression can be inferred, and the result is a term containing the inferred body and type, wrapped in a `TLet`.

$$\frac{\Gamma \vdash t \Leftarrow \star \qquad \Gamma \vdash a \Leftarrow t \qquad \Gamma, x : t \vdash b \Rightarrow u}{\Gamma \vdash \text{let } x : t = a \text{ in } b \Rightarrow u}\ (\textbf{Let-In})$$

```
fun LocalContext.infer(pre: PreTerm): Pair<Term, Value> = when (pre)
  is RLet -> {
    val t = check(pre.type, VStar)
    val a = check(pre.defn, t)
    val (b, u) = localDefine(pre.name, a, t).infer(pre.body)
    Pair(TLet(pre.name, t, a, b), u)
  }
  // ...
}
```

Figure 3.14: Side-by-side comparison of the **Let-in** rule

Lastly, the rule for a term-level λ-abstraction is demonstrated in Figure 3.15. The type produced on the last line of the snippet is a `VPi` unlike the rule, as the rule was written for the λ→-calculus; it is semantically equivalent, however. This rule demonstrates the creation of a new meta-variable as without a placeholder, we are not able to infer the type of the body of the function. This meta-variable might or might not be solved in the course of inferring the body: either way, both the term and the type only contain a reference to a globally-scoped meta-variable and not the solution.

3.5 Driver

This concludes the complex part of the interpreter, what follows are rather routine concerns. Next part of the implementation is the driver that wraps the backend, and handles

$$\frac{\Gamma, x : t \vdash e \Leftarrow u}{\Gamma \vdash \lambda x.e \Leftarrow t \to u}\ (\textbf{Abs})$$

```
fun LocalContext.infer(pre: PreTerm): Pair<Term, Value> = when (pre)
  is RLam -> {
    val a = newMeta()
    val (b, t) = localBind(pre.name, a).infer(pre.body)
    Pair(TLam(pre.name, b), VPi(pre.name, a, VCl(env, t.quote())))
  }
  // ...
}
```

Figure 3.15: Side-by-side comparison of the **Abs** rule

its interaction with the surrounding world. In particular, the parser, pretty-printer, and state management.

Parser Lexical and syntactic analysis is not the focus of this work, so simply I chose the most prevalent parsing library in Java-based languages, which seems to be ANTLR[6]. It comes with a large library of languages and protocols from which to take inspiration[7], so creating the parser was a rather simple matter. ANTLR provides two recommended ways of consuming the result of parsing using classical object-oriented design patterns: a listener and a visitor. I used neither as they were needlessly verbose or limiting[8].

Instead of these, a custom recursive-descent AST transformation was used that is demonstrated in Listing 3.10. This directly transforms the `ParseContext` objects created by ANTLR into our `PreTerm` data type.

```
fun TermContext.toAst(): PreTerm = when (this) {
  is LetContext -> RLet(id.toAst(), type.toAst(), defn.toAst(), body.toAst())
  is LamContext -> rands.foldRight(body.toAst()) { l, r -> RLam(l.toAst(), r) }
  is PiContext -> spine.foldRight(body.toAst()) { l, r -> l.toAst()(r) }
  is AppContext -> operands.fold(oprator.toAst()) { l, r -> r.toAst()(l) }
  else -> throw UnsupportedOperationException(javaClass.canonicalName)
}
```

Listing 3.10: Parser to `PreTerm` transformation as a depth-first traversal

The data type itself is shown in Listing 3.11. As with terms and values, it is a recursive data structure, presented here in a slightly simplified manner compared to the actual implementation, as it omits the part that tracks the position of a term in the original source. The grammar that is used as the source for the parser generator ANTLR was already presented once in the conclusion of Chapter 2, so the full listing is only included in Appendix B.

Pretty-printer A so-called pretty-printer is a transformation from an internal representation of a data structure to a user-readable string representation. The implementation of

[6] https://www.antlr.org/

[7] https://github.com/antlr/grammars-v4/

[8] In particular, ANTLR-provided visitors require that all return values share a common super-class. Listeners don't allow return values and would require explicit parse tree manipulation.


```
sealed class PreTerm
typealias Pre = PreTerm
typealias N = String

sealed class TopLevel
data class RDecl(val n: N, val type: Pre) : TopLevel()
data class RDefn(val n: N, val type: Pre?, val term: Pre) : TopLevel()
data class RTerm(val cmd: Command, val term: Pre) : TopLevel()

object RU : Pre()
object RHole : Pre()
data class RVar(val n: N) : Pre()
data class RNat(val n: Int) : Pre()
data class RApp(val lhs: Pre, val rhs: Pre) : Pre()
data class RLam(val n: N, val body: Pre) : Pre()
data class RPi(val n: N, val type: Pre, val body: Pre) : Pre()
data class RLet(val n: N, val type: Pre, val defn: Pre, val body: Pre) : Pre()
data class RForeign(val lang: N, val eval: N, val type: Pre) : Pre()
```

Listing 3.11: Data type `PreTerm`

such a transformation is mostly straight-forward, complicated only by the need to correctly handle operator precedence and therefore parentheses.

This part is implemented using the Kotlin library `kotlin-pretty`, which is itself inspired by the Haskell library `prettyprinter` which, among other things, handles correct block indentation and ANSI text coloring: that functionality is also used in error reporting in the terminal interface.

An excerpt from this part of the implementation is included in Listing 3.12, which demonstrates the precedence enumeration `Prec`, the optionally parenthesizing operation `par`, and other constructions of the `kotlin-pretty` library.

```
enum class Prec { Atom, App, Pi, Let }
fun Term.pretty(ns: NameEnv?, p: Prec = Prec.Atom): Doc<Nothing> = when (this) {
  is TVar -> ns[ix].text()
  is TApp -> par(p, Prec.App,
    arg.pretty(ns, Prec.App) spaced body.pretty(ns, Prec.Atom))
  is TLet -> {
    val d = listOf(
      ":".text() spaced ty.pretty(ns, Prec.Let),
      "=".text() spaced bind.pretty(ns, Prec.Let),
    ).vCat().align()
    val r = listOf(
      "let $n".text() spaced d,
      "in".text() spaced body.pretty(ns + n, Prec.Let)
    ).vCat().align()
    par(p, Prec.Let, r)
  } // ...
}
```

Listing 3.12: Pretty-printer written using `kotlin-pretty`

State management Last component of the driver code is global interpreter state, which consists mainly of a table of global names, which is required for handling incremental interpretation or suggestions (tab-completion) in the interactive environment. It also tracks the position of the currently evaluated term in the original source file for error reporting.

Overall, the driver receives user input in the form of a string, parses it, expression by expression supplies it to the backend, receiving back a global name, or an evaluated value, which it pretty-prints and returns back to the user-facing frontend code.

3.6 Frontend

We will consider only two forms of user interaction: batch processing of a file via a command-line interface, and a terminal environment for interactive use. Later, with the Truffle interpreter, we can also add an option to compile a source file into an executable using Truffle's capability to produce *Native Images*.

```
> cat demo.mt
id : {A} -> A -> A = \x. x
{-# TYPE id #-}
{-# ELABORATE id 5 #-}
{-# NORMALIZE id 5 #-}

> montuno demo.mt
{A} -> A -> A
id {Nat} 5
5
> montuno --type id
{A} -> A -> A
```

Listing 3.13: Example usage of the CLI interface

CLI We will reuse the entry point of Truffle languages, a `Launcher` class, so that integration of the Truffle interpreter is easier later, and so that we are able to create single executable that is able to use both interpreters.

`Launcher` handles pre-processing command-line arguments for us, a feature for which we would otherwise use an external library like `JCommander`. In the Truffle interpreter, we will also use the *execution context* it prepares using various JVM options but for now, we will only use `Launcher` for argument processing.

Two modes of execution are implemented, one mode that processes a single expression provided on the command line and `--normalizes` it, `--elaborates` it, or find its `--type`. The second mode is sequential batch processing mode that reads source code either from a file or from standard input, and processes all statements and commands in it sequentially.

As we need to interact with the user we encounter another problem, that of error reporting. It has been mentioned in passing several times, and in this implementation of the interpreter, it is handled only partially. To report an error well, we need its cause and location. Did the user forget to close a parenthesis, or is there a type error and what can they do

to fix it? Syntactic errors are reported well in this interpreter, but elaboration errors only sometimes.

Error tracking pervades the entire interpreter, position records are stored in all data structures, location of the current expression is tracked in all evaluation and elaboration contexts, and requires careful placement of update commands and throwing and catching of exceptions. As error handling is implemented only passably and is not the focus of this thesis, it is only mentioned briefly here.

In Listing 3.13, a demonstration of the command-line interface is provided: normalization of an expression, batch processing of a file, and finally, starting up of the REPL.

REPL Read-Eval-Print Loop is the standard way of implementing interactive terminal interfaces to programming languages. The interpreter receives a string input, processes it, and writes out the result. There are other concerns, e.g., implementing name completion, different REPL-specific commands or, in our case, switching the backend of the REPL at runtime.

From my research, JLine is the library of choice for interactive command-line applications in Java, so that is what I used. Its usage is simple, and implementing a basic interface takes only 10s of lines. The commands reflect the capabilities of the command-line interface: (re)loading a file, printing out an expression in normalized or fully elaborated forms, and printing out the type of an expression. These are demonstrated in a simple way in Listing 3.14.

```
> montuno
Mt> :load demo.mt
Mt> <TAB><TAB>
Nat Bool zero succ true false if natElim id const
Mt> :normalize id 5
5
Mt> :elaborate id 5
id {Nat} 5
Mt> :type id
{A} -> A -> A
Mt> :quit
```

Listing 3.14: REPL session example