Department of Computing
Imperial College London

MEng Individual Project

Haskell-Like S-Expression-Based Language Designed for an IDE

Author:
Michal Srb

Supervisor:
Prof. Susan Eisenbach

June 2015

Abstract

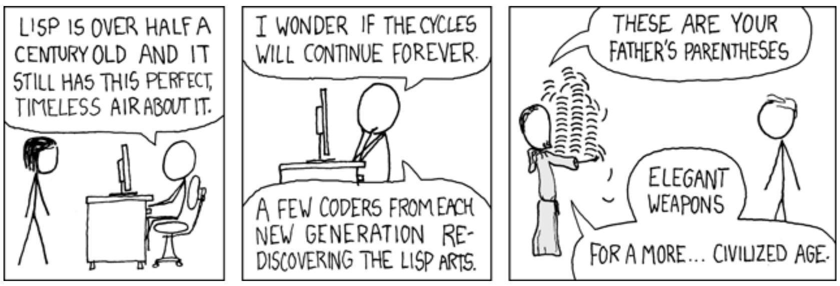
The state of the programmers' toolbox is abysmal. Although substantial effort is put into the development of powerful integrated development environments (IDEs), their features often lack capabilities desired by programmers and target primarily classical object oriented languages. This report documents the results of designing a modern programming language with its IDE in mind. We introduce a new statically typed functional language with strong metaprogramming capabilities, targeting JavaScript, the most popular runtime of today; and its accompanying browser-based IDE. We demonstrate the advantages resulting from designing both the language and its IDE at the same time and evaluate the resulting environment by employing it to solve a variety of nontrivial programming tasks. Our results demonstrate that programmers can greatly benefit from the combined application of modern approaches to programming tools.

I would like to express my sincere gratitude to Susan, Sophia and Tristan for their invaluable feedback on this project, my family, my parents Michal and Jana and my grandmothers Hana and Jaroslava for supporting me throughout my studies, and to all my friends, especially to Harry whom I met at the interview day and seem to not be able to get rid of ever since.

Contents

Chapter 1

Introduction


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

The art of programming has become an ever more important trade in today's society. Yet despite the proliferation of software, both in terms of the quantity of applications and their complexity, the software tools used by professionals to create these programs have gone through a painfully slow evolution over the course of the last four decades. Although programmers are the pioneers of new technology and have been a part of the driving force behind the rapid changes in the way we interact with computers, they are, taken as a community, extremely conservative in the choices they make with regards to the way they give life to their creations.

Recently there have been two major streams of innovation regarding programming tools. Firstly, functional programming has been on a steady rise. There is a substantial amount of research going into the design of new languages and their features[1–3], more powerful type systems[4, 5] and ways of incorporating ideas from this paradigm to the object-oriented world of programming[6]. These are discussed in Sections 2.1 and 2.2.

Secondly, there has been a renaissance of investigation into new ways in which programmers and users in general could interact with the digital medium. This movement has strong ties with the research carried out in 1960's and 1970's by researchers such as Douglas Engelbart[7] and Alan Kay[8] et al. Notably, Bret Victor[9] has produced a number of ideas and these, along with more traditional approaches to development environments are discussed in Section 2.4.

We acknowledge the historical roots of this work and look at some foundational programming language ideas in Section 2.5.

It has been argued by some[10] that a hierarchy among programming languages can be established based on their expressive power and that LISP (in some variation), sits at the top of this ladder, thanks to macros. Since macro systems, even among LISPs, come in various flavors, we explore some of them in Section 2.3. These discussions and inquires provide a background for our own design decisions.

1.1 Objectives

Our primary goal was to design and create the best possible programming environment reflecting the current state of programming-tools research. We believed it important to bring these technologies to a wider audience and hence influence other tool creators to also take advantage of them. We also wanted to demonstrate that these techniques can be used for professional programming, not just for educational purposes. Finally by making the results polished enough we aimed to gather an audience of followers who would help break the conservative mindset of the general programming community. To accomplish this goal, we strove to achieve the following objectives:

1. **Design a new programming language tailored to the programmer's experience.** Most languages are driven by a certain key goal, as discussed in Section 2.1. Very few, though, focus in their design *mainly* on supporting the programmer in his struggle to a successful program completion, and even those which do might have completely missed the point.[1] We formalize our requirements for such a language in Chapter 3. Moreover, almost every popular language of today has been designed agnostic about the tools which programmers might invent to use it, and we boldly took the opposite approach here.
2. **Design a simple yet powerful IDE.** As the popularity of those available shows, IDEs are incredibly helpful to programmers. Yet traditional IDEs can consist of on the order of tens of millions of lines of code[11] and, although designed with customizability and extensibility in mind, can be very difficult to mold to perform the desired tasks or provide a better user interface to existing features. Although writing a plug-in for one of these IDEs could easily be turned into a standalone individual project, we decided to design a simpler, custom-made IDE for our language, using great open-source technology to allow us to accomplish this in a reasonable timeframe. The results are described in Chapter 4.

[1]Ruby touts focus on productivity yet as a dynamicly typed language lacks one of the essential tools which lift some of the burden of program construction and maintanance.

3. **Implement both the language and the IDE.** We could not achieve our goal without providing concrete implementation for our design. It would also have been unrealistic to assume we could design a language and an IDE without solving various implementation issues. As such the processes of design and implementation were intertwined. Although great performance was not our primary goal, we could not neglect the issue, for language design must always balance between the ease of use and compile and runtime performance[12]. Similarly, producing an unusably slow IDE would not allow us to achieve our goal. The technical details of our implementation are summarized in Chapter 5.

1.2 Challenges

These objectives were highly ambitious. The main challenge was the sheer amount of design and implementation decisions we had to make in order to create a usable, fully-featured environment. These were some of the main challenges we faced along the way:

1. **Finding a suitable type system.** Despite the long history of research into type systems, no single silver bullet has been found yet. Specifically, today's type systems in pure functional languages vary between simple, easy to understand but not so powerful to very complex and powerful systems such as dependent types[3]. We had to find a compromise which would not hinder the adoption of our language and at the same time be sufficiently powerful to support the programmer in ways which many mainstream programming languages cannot. Great emphasis was put into making the output of the system understandable, a challenge in itself.

2. **Macros.** One of the unique features of our language is its meta-programming capability. To that end we wanted to combine the macro-directed compilation of LISP-like languages with the convenience of modern Haskell-like languages which support an arbitrary order of definitions, mutually recursive definitions and full-blown type inference. The challenge extends into the design of a suitable programming interface for implementing macros in a pure statically-typed language.

3. **Providing an exhaustive statically typed collections library.** As discussed in Section 3.7, a full-featured collections library is one of the key requirements for a successful adoption of a programming language. Yet the major statically typed functional languages do not come with such a standard library and we had to consider the available libraries and adapt them to the specifics of our language.

4. **Approachable structural editing.** Attempts at popularizing structural editing have failed in the past[13]. We needed to provide the power of structural editing while making

the IDE feel as familiar and easy to use as possible. Our approach is described in Section 4.2.

1.3 Contributions

We have designed and implemented a new programming environment, a synergy between a modern, functional programming language and its novel, interactive IDE. We believe that it demonstrates that a significantly different approach to programming can be realized today. The following are what we consider the key contributions of our work:

1. **Macros.** We implemented a declarative, statically typed language with S-Expression based syntax using a macro-directed implementation which evolved from the compilation strategy of LISPs. This allowed us to provide the facility for user macros, shown in Section 3.4, available to extend the syntax of the language. We have already found many examples of useful macros, both for simplifying syntax and dealing with side-effects.

2. **Implicit functional dependencies.** While functional dependencies are a popular extension of Haskell, they have a negative impact on the complexity of its type system, and have not been included in the base language. We came up with a simplified approach to functional dependencies which is both easier to explain to mortal programmers and, together with the rules our system imposes, does not have a negative impact on type inference completeness or decidability. This mechanism is described in Section 3.5.

3. **Highly polymorphic collections API.** Informing the design of our type system, we implemented an exhaustive collections library polymorphic in the types of collections, which simplifies their use and removes code duplication by increasing the level of abstraction. This API is described in Section 3.7.

4. **Interactive IDE.** In Sections 4.6 and 4.7 we describe the main interface of our IDE and the way in which programmers can freely experiment with programs, including animations and interactive interfaces, and observe their execution. This model works thanks to our language being functional and pure-by-default.

5. **Surfacing type errors through origins of types.** Although the core of the idea did not originate with us, we took great care in implementing a type system which always reports the origin of errors, even when these are coming from explicit type declarations. Our implementation, detailed in Section 4.8.2 can provide a platform for testing this approach towards type error messaging and potentially informing improvements to current Hindley-Milner style type system implementations.

6. **Colorful and powerful syntax.** There have been many implicit influences between the language and the IDE design but the single biggest one is the approach to effective syntax. We have shown in Section 3.3.3 that S-Expression based syntax can be made more readable and less verbose through the use of color and that it can be exploited to provide a more powerful editing experience based on the actual AST of the program, described in Section 4.3.

The project's evaluation is laid out in Section 6.

Chapter 2

State of the Art

In this chapter we discuss the various languages and development environments used today, in relation to our own work. It would be impossible to detail all the languages and tools which had influenced our decision making but for those not mentioned here we make direct references in the rest of the report. It is our aim to focus on those aspects of each technology which has direct relation to our objectives and the challenges we have to tackle, not to provide exhaustive descriptions or a complete enumeration of all available tools.

To structure this chapter, we separate the available technology into these aspects:

1. Languages - syntax and semantics.
2. Type Systems.
3. Macro Systems.
4. Editors and IDEs.

We also look into the foundational work on programming languages and tools at the end of this chapter in Section 2.5.

2.1 Languages

The languages discussed in this section relate to our work in one or more of the following three main aspects. The motivations for adopting these characteristics in our language are discussed in Section 3.

1. **Being functional, strongly and statically typed.** A language is considered functional if it encourages the use of top-level and higher-order functions and immutability rather than dealing with a mutable state[14]. Furthermore, a language is strongly typed if the entities within its programs are assigned a single type that does not change during the execution of the program; it is statically typed when the types of runtime entities corresponding to certain elements of the programs' source can be determined at compile time. Haskell[15] and SML[16] are the most popular examples.

2. **Having an s-expression based syntax.** We use s-expressions, short for symbolic expressions, to name the syntax which is based on the use of parentheses (or other delimiters) and a set of atoms to form the *expression*s of a language, defined inductively as either:

   - *atom*
   - `(`*expression...*`)`

   where the list of *expression*s inside the parentheses is whitespace-separated. We call the languages which employ this syntax for most of its constructs s-expression-based. This syntax originated with LISP and is used by its various dialects.

3. **Targeting the JavaScript runtime.** Our motivation for compiling down to JavaScript (JS) is detailed in Section 3.2.2. We are mainly interested in languages which are designed specifically for the runtime, and are not just mechanically translated (given Emscripten[17], any language which can be compiled to C, C++ or directly to the LLVM bitcode can be run in JavaScript). Such languages are sometimes referred to as *transpiled* to JS. The most popular one, which pioneered this approach, is CoffeeScript[18].

We are interested in both the prime examples of each of these approaches and those languages which combine them. We shall begin with statically typed languages and their variations which compile to JS and then look at mainly dynamic languages and their possible typed variations. A more detailed discussion of type and macro systems is provided in Sections 2.2 and 2.3 respectively.

2.1.1 Haskell

Haskell is a mature, pure, strongly and statically typed functional language originally specified in 1990 with the goal of consolidating the various lazy functional languages of the time and providing a platform for further research[15]. It has significantly evolved since, the two most visible advancements being the use of monads and the do notation for handling input and output (IO) and the addition of functional dependencies to allow for multi-parameter type classes.

Type classes themselves are regarded as the most distinctive characteristic of the language[15, Section 3.3], and are discussed along with functional dependencies in section 2.2.5.

The use of laziness by default, or call-by-need instead of the more traditional call-by-value mechanism, lied at the origin of the Haskell project and is sometimes considered its biggest deficiency. In [19], lazy evaluation is argued for as a means for combining programs, where one program computes a value only if requested for the computation of the successive program. The example given in the paper constructs a sequence of values lazily to find a solution to the problem. It turns out that the need for such composition is real, and is nowadays supported in many traditional imperative languages using the notion of streams or lazy collections. But laziness also comes with a serious drawback, and that is the inability for the programmer to judge the space complexity of their algorithms. It seems to us that the disadvantages of omni-present call-by-need outweigh its advantages.

Haskell is also quite characteristic in its syntax. The language tries to look as much like mathematics as possible, and some find the programs written in it "pretty". Not unlike some later popular imperative languages (Python, Ruby), it uses indentation to denote blocks of definitions, with a fairly complex layouting rule, but in practice this isn't problematic as is the case with CoffeeScript (Section 2.1.8). It also sports custom infix operators with customizable precedence, which can allow for some elegant or unreadable code (depending on the familiarity with given operators). We argue against this approach to the syntax of programming languages in Section 3.3. We find Haskell's biggest contribution in promoting the declaration[1] style of programming, where pattern matching and where clauses are used to denote results first and then detail how these results are obtained from more granular expressions, as opposed to the expression style of using conditionals and let-bindings, as can be seen in Figure 2.1

```
filter p [] = []
filter p (x : xs)
  | p x = x : rem
  | otherwise = rem
    where
      rem = filter p xs
```

```
(defn filter [p list]
  (if (empty? list)
    ()
    (let [x (first list)
         rem (filter p (rest list))]
      (if (p x)
        (cons x rem)
        rem))))
```

FIGURE 2.1: The function filter implemented in declaration style in Haskell and expression style in Clojure.

Although we called Haskell a mature language, we find one particular area in which its built-in facilities lack compared to other popular languages: collections. Some basic collections are included in the standard library, such as `Map` and `Set`, and there are type class declarations for supporting more generic use of collections, but for example hash maps are omitted and it

[1]Not to be confused with the *declarative* vs imperative style, which is more concerned with semantics and is much older than Haskell.

seems that programmers generally rely on a combination of external libraries which may require additional work to function together. This space is also fairly badly documented, compared to other language ecosystems.

2.1.2 Fay, GHCJS

There have recently emerged several Haskell-like languages which are primarily focused on running in-browser, or allowing the use of a single language on both the client and the server (if the same code is then also run on both, such applications are usually called isomorphic[20]). These languages (or technologies) vary in how much they depart from Haskell, or what subset of Haskell features they support. Elm, PureScript, Fay and GHCJS represent different points on this scale. While Elm and PureScript depart from Haskell semantics (as discussed in the next sections), Fay[21] is a proper subset of Haskell, with its own compiler and data types matching those of JavaScript, and GHCJS[22] is a Haskell to Javascript compiler which uses the GHC API and supports many advanced runtime features found in natively-running Haskell. All of these projects have compilers written in Haskell, although there is an effort to implement PureScript in itself. This means that currently none of these compile in-browser. We discuss how these languages deal with foreign-function interface (FFI), that is calling JavaScript code, in Section 3.9.

2.1.3 PureScript

PureScript[2] is a strict (not lazy), Haskell-like language which strives to compile to readable JavaScript without having to provide a large runtime library.

As a *runtime library* we identify the code which needs to be included alongside the programmer's transpiled code. In practice, the size of such library can vary greatly, and sometimes the library can be split and only included optionally, based on whether a certain feature of a language is used. Usually such code is included directly in the resulting JavaScript, as opposed to being imported as a separate module or package. So, for example, PureScript's `Prelude` would not be considered a part of its runtime library.

Apart from the lack of default laziness, PureScript differs from Haskell by having a single type for numeric values, `Number`, and the use of a fine-grained monad system, instead of a single IO monad for dealing with the imperative environment with which the programs interact. PureScript also supports extensible records, while they are not part of standard Haskell, as well as higher-kinded polymorphism. Interestingly, PureScript's compiler does not infer polymorphic types of definitions involving type class constraints, such as the following one:

```
square x = x * x
```

Because the multiplication function `*` requires its operands to have a type which is an instance of the `Num`[2] class, but there is no indication of which instance will be used, the programmer must specify the function's type explicitly, as `square :: forall a. (Num a) => a -> a`. Notice that free type variables need to be explicitly quantified in PureScript.

2.1.4 Elm

The most popular or perhaps most widely known, judging by the number of stars on its source repository on Github[23], and the oldest of the mentioned Haskell derivatives is Elm[1]. Elm is a functional reactive language, aimed specifically at writing interactive applications. The main idea of reactive programming is the use of signals, essentially streams which are observed as values. Mapping a function over a signal yields a corresponding new signal, and successive values can be combined using reduction. Because functional reactive programming does not play well with laziness[1], there are no provisions for laziness in Elm. The support for reactive programming leads to a bigger runtime library in Elm. In terms of its type system, Elm differs mainly by not supporting type classes, which is a highly debated issue within its community, since Haskell relies heavily on type classes. Elm circumvents the issue by special allowances within its compiler which do not scale well beyond built-in types. Elm also misses where clauses, which means that all names defined within a function must be bound before they are used in a let statement, in a stark difference to Haskell code written in a declaration style.

2.1.5 Liskell

Although now abandoned, Liskell[24] is an interesting project to bring LISP syntax to Haskell. Its proposal describes in detail the implementation of a macro system on top of GHC's API and introduces one possible LISP-like syntax to use for creating Haskell programs. We will come back to this proposal in Section 2.3.

2.1.6 ML, SML, OCaml

ML[25] and its most popular variants used today, SML[16] and OCaml[26], are impure, strict, strongly and statically typed functional languages. Because ML is both strict and impure, imperative algorithms can be expressed without the need for special syntax or monads as is required in Haskell. The lack of type classes means that similar functions need to be quantified with corresponding package names, such as `List.map`, and at least in OCaml, which as an efficient natively compiled language distinguishes floating point numbers from integers, even

[2]In PureScript, `Num` is the name of the class which provides arithmetic operations and `Number` is the name of the only numerical type.

such fundamental operations as addition have specific names for different types of arguments: `+` for integer addition and `+.` for floats. The lack of type classes is counter-balanced by ML's more powerful module system. In ML, signatures can be parametrized with type variables, which, among other things, allows the definition of proper abstract data types.

Js_of_ocaml[27] is a compiler from OCaml's bytecode to JavaScript, and as such allows OCaml to run in the browser. The generated JavaScript is unreadable though and as with other languages compiled mechanically, performance suffers. For example, strings in OCaml cannot be directly mapped to JavaScript strings.[28]

2.1.7 Java, C#, Dart, TypeScript, Closure, Flow

Java[29] and C#[30] are the most industrially popular higher-level[3] languages for building large applications. Although they mainly exercise a different paradigm from the languages we consider, they still present important lessons for us. Firstly, they show the importance of type signatures as a way of documenting code, which can often be sparsely documented otherwise. Programmers using these languages are forced to provide type signatures not only for methods of classes, but also for all local definitions, a potentially tedious task. Some of this tediousness can be alleviated with the use of IDEs, discussed in Section 2.4. Another strength of these languages is their clear syntax, borrowed from C and C++. Although verbose, the small number of syntactic forms leads to an easier understanding of unfamiliar code, as opposed to, say, Haskell code extensively using custom operators.

Each language also has a counterpart in the world of languages transpiled to JavaScript. Dart is a language developed by Google, which given its large browser market share can hope to replace JavaScript with a new virtual machine (VM) for their custom language. There is little signal so far that this could happen in the near future, given the size of the JavaScript ecosystem, and Dart is currently mainly used compiled to JavaScript. Dart includes optional typing in the vein of Java, as well as classical object oriented features such as classes.

Similarly, TypeScript is developed by Microsoft. Unlike Dart, it is a proper superset of JavaScript, it adds optional type annotations and object oriented features, and it conforms to the currently proposed ECMAScript 6[31] (ES6) next standard of JavaScript.

Finally, Google Closure[32] and Facebook's Flow[33] are precompilers which preprocess documented or annotated JavaScript code to perform type checking and potentially optimization, such as dead code removal. We mention all of these technologies here to stress the point that many are looking for ways to type check their browser-targeting code, whether it is for programmer's convenience while writing code or for increased documentation and maintainability.

[3]Languages which do not allow direct access to memory locations via pointers.

There is also the desire to make client implementation more familiar to programmers using statically typed languages on the server.

2.1.8 CoffeeScript

CoffeeScript[18] is by far the most popular language transpiled to JavaScript. It is a functional, impure, strict, dynamic language, which encourages functional programming through a lightweight function syntax, similar to Haskell's lambda expressions, while also providing classical object-oriented features such as classes. Its goal is to protect the programmer from "bad parts of JavaScript"[34] while providing a less verbose syntax, as can be seen in Figure 2.2.

```
square = (x) -> x * x
```

```
function square(x) {
  return x * x;
}
```

Figure 2.2: Definition of the square function in CoffeeScript and JavaScript

Particularly attractive is the fact that CoffeeScript has only a single syntax for defining functions. Although its syntax is very elegant, it can cause problems. Consider Figure 2.3.

```
class A
  constructor: ->
    handler = (event) =>
      false
     @initialize handler

    @field = "Should be assigned to object"
```

Figure 2.3: Bug caused by wrong indentation.

The snippet shows a definition of a class `A`. The instances of this class should have a property `field`, set to the given string. Unfortunately, there is a bug not intended by the programmer, the spurious single space in front of the call to `@initialize`, which causes the property to be set on the class object itself. This is caused by CoffeeScript's lenient indentation rules. Although substantial work[4] has gone into improving the language's compiler, many such errors still creep up. As much as CoffeeScript strives to avoid the problems of JavaScript, it adds its own. Another such example is the lack of variable shadowing. Consider Figure 2.4.

```
map = (f, list) -> list.map f
# ...
allKeys = (maps) -> union (keys map for map in maps)
```

Figure 2.4: Accidental shadowing.

[4]See for example my commit https://github.com/jashkenas/coffeescript/pull/3320

`map` is initially defined as a higher order function, and as long as we don't call `allKeys`, it stays that way. But once we use `allKeys`, `map` is reassigned to subsequent values inside the `maps` list. In large files, this can lead to surprising errors even for experienced users of the language, especially if the rogue value is of the same type as the intended value. We should note here that some popular languages, such as Python, lacking lexical scope altogether, have even more surprising semantics.

These examples hopefully show that, along with the lack of static typing, CoffeeScript is certainly not guaranteed to enjoy its position indefinitely. Many languages descendant from CoffeeScript appeared, most notably LiveScript, which aims to support even more functional style, with optional currying and pattern matching.

2.1.9 Clojure

Clojure[35] is a modern LISP dialect, with a primary implementation targeting the Java VM (JVM). It has been designed as a practical language with emphasis on concurrent programming. It is arguably the most popular LISP dialect at present, at least in terms of the growth of its community. Its strengths include good FFI to Java and an exhaustive built-in collections library. The library is supported with special syntax for each basic type of a collection. All collections are immutable and sequences are implemented lazily, which together encourages functional style of programming, even though Clojure has full support for imperative programming as well. Polymorphism is supported via protocols, which are similar to Java's interfaces or Haskell's type classes, but dispatch based only on the type of the first argument to a method.

Clojure also compiles to JavaScript, being envisioned as portable language, and this version of the language is called ClojureScript. It supports a large subset of Clojure with the most obvious deficiency that it lacks runtime compilation capabilities, as Clojure is used for writing and compiling macros.

2.1.10 Racket

Racket[36] is a modern Scheme, and hence LISP, dialect, which originated within the academia and was originally focused on providing a learning environment. In comparison with Clojure, it uses more traditional Scheme-like syntax and has a limited built-in collections library. On the other hand, Racket has the most complex macro system of those used today, as is discussed in Section 2.3. It uses its own runtime[5] and the distribution of its compiler includes a custom development environment called DrRacket, which we discuss in Section 2.4. Both Clojure's and Racket's ecosystem also include libraries which add ad-hoc type checking.

[5]This might be one of the reasons for its low adoption compared to Clojure in the industry.

2.1.11 Shen

Last modern LISP-like language we consider is Shen[37]. It aims to be easily portable by compiling to a simple LISP dialect. It has a rather powerful optional type system based on a calculus with derivation rules and also includes a working Prolog. In terms of semantics it is actually closer to the statically-typed family of functional languages. Functions have fixed arity and partial application is supported. Shen also sports pattern matching as the only way to define functions, with some features which are not supported by Haskell, for example comparing arguments via the use of the same name twice in a pattern. Shen does support variadic calls to macros which expand to fixed arity calls. Although technically most interesting, the language does not seem to enjoy wide popularity and its development seems to be hindered by fundraising efforts.

2.2 Type Systems

Now we discuss in more detail the various static type systems found in languages presented in the previous section. We are mainly interested in these aspects of the systems:

1. **Error messages.** We believe that a type system should be a tool which supports the programmer in creating and maintaining programs. As such, the clarity of error messages given by the system are most important to us. We investigate how the complexity and design of the type system impact how easy it is to see where the error occurred and whether the system can suggest a possible fix.
2. **Expressive power.** We can easily find programs which are very difficult to assign static types, even considering only strongly typed languages. Hence type systems can vary in power, in different areas, such as dealing with higher kinds (types of types), supporting subtyping, supporting computation on the type level and others. We would like to compare and evaluate the different features of the various type systems.
3. **Ease of type annotation.** Another important aspect is the interaction between the programmer and the type system when constructing code. Does the programmer have to specify types by hand and when? How complex are the used type representations? Can types be inferred, are there any exceptions or effects of other features?
4. **Reliability and performance.** Lastly, we would like the type system to be reliable and not have a negative impact on the compilation time. We want to know whether type checking or inference can be done in a modular fashion and whether the implementation supports incremental typing. Again, are there any specific features of the type system that could prevent modularization?

2.2.1 Polymorphism

Today, there are two major classes of type systems used by most of the mentioned languages. The first is a system with Hindley-Milner style type inference. Haskell, ML and their dialects use this system with various extensions. The defining technical aspect of a type system is the way in which it handles polymorphism. These systems support parametric polymorphism. Consider the type of a function `map` in Figure 2.5, which applies the function supplied as its first argument to each element of a given list, returning a list of the results of those applications.

```
map :: (a -> b) -> [a] -> [b]
```

FIGURE 2.5: The type of `map`.

Then `map` is polymorphic, because it accepts arguments of various types, as long as we can substitute concrete types for the *type variables* in its type, in our example `a`, `b`, to obtain the concrete types of those arguments. In most cases (see Section 2.2.4), these type systems can infer any programmer-supplied code, alleviating the need for type annotations altogether, although it is often mentioned that specifying types can still be useful and is regarded as good programming style.

The second class of type systems is used in statically typed languages with classical inheritance. These languages usually have the notion of a *class*, a template which is used to instantiate *object*s, which also represents the type of those *object*s. Polymorphism is achieved by specifying a type which is higher in the type hierarchy of the program than the concrete types of values. For example the method `addAll` in Figure 2.6 accepts as an argument any object which is an instance of a class which inherits from the `Collection` class.

```
boolean addAll(Collection c)
```

FIGURE 2.6: The type of `Collection::addAll`.

We call this polymorphism via inheritance and it is the main kind of polymorphism found in all statically typed object-oriented languages, including Java, C# and their dialects mentioned before. Although attempts have been made in that direction, so far it seems that Hindley-Milner style type inference system cannot deal well with classical inheritance.[38]

Both systems have several extensions which essentially add different kinds of polymorphism. We detail these in the following sections.

2.2.2 Ad-hoc Polymorphism and Type Classes

Apart from parametric polymorphism which is equally supported in ML and Haskell, there is often the need for ad-hoc polymorphism. Consider the case where we would like to obtain a textual representation of a given value. In Java, we could require our arguments to be instances of a subclass of a class which declares a method such as `String toString()`. In Haskell, we could achieve this using two arguments, one which would be our value, of some type `a` and the second which would be a function of type `a -> String`, that is a function which takes our value as an argument and returns its textual representation. This style of programming is usually referred to as *dictionary passing style*, where *dictionary* stands for a collection of functions which can operate on the rest of the arguments.

It turns out that this approach, although feasible, is cumbersome. Consider that we need to pass these functions around everywhere we want to turn our value into its textual representation, possibly through functions which do not care about that representation. In a way, we create undesirable coupling in our program, while also being more verbose than if we used polymorphism via inheritance. Haskell solves this problem with type classes[39]. A type class is essentially a specification of such a *dictionary*, which value is provided for a required type by making it an instance of that type class.[6] Haskell's compiler then uses the information from its type system to pass these dictionaries, possibly several at a time, to our functions automatically. In Figure 2.7, we use the type class method `show`, so the resulting function `wrap` takes an argument of any type that is an instance of the `Show` class which declares the method. Haskell takes care of supplying the correct `show` implementation.

```
wrap value =
  "(" ++ (show value) ++ ")"
```

FIGURE 2.7: Function `wrap` in Haskell, which returns the textual representation of given argument wrapped in parentheses.

Note that both Java and Clojure support their own versions of ad-hoc polymorphism. In Java, we can think of ad-hoc polymorphism as polymorphism "in the other direction". Whereas polymorphism via inheritance *dispatches* methods based on the type of the object containing that method, *method overloading* decides which function to call based on the static type of its arguments. In Java this approach is needed for polymorphism dealing with values of so-called primitive types, which are not objects and hence cannot contain methods.

Clojure includes yet another way of ad-hoc polymorphism, positioned between type classes and method overloading. *Protocols* declare a set of functions which must be implemented for a type to adhere to the given protocol, similarly to type classes in Haskell. These functions then dispatch dynamically, that is based on the actual types at runtime, based on the type of the first argument only, which makes them less general than Haskell's methods and limited in a way compared to Java's overloaded methods.

[6]Notice that although the terms regarding Haskell's type classes coincide with the terminology of object-oriented languages, the two operate on different levels: in Haskell they operate on types, in Java they describe relationships between objects.

It has been shown[40] that ML's modules can be used for simulating ad-hoc polymorphism similarly to Haskell's type classes. Yet in the comparison, it seems that ML's module system does not allow for the same convenience Haskell's compiler provides to the programmer. Consider the example in Figure 2.8, taken from[40].

```
elem x l = any (eq x) l
```

```
fun elem d (x:'a) l =
  let
   structure D as Eq where type t = 'a = d
  in
    any (D.eq x) l end
```

FIGURE 2.8: Use of polymorphic function `eq` in Haskell and its translation to ML.

In Haskell, `eq` is a method of the type class `Eq`, which can be used to compare two values[7]. The important point is that we have to specifically "unify" the type of the argument with the type used inside the `Eq` *structure*, one of ML's module mechanisms used here correspondingly to a Haskell type class. We therefore conclude that although ML's type system provides powerful abstraction tools, it does not give us the same power as Haskell's type classes.

It should be mentioned that some consider this a good property of the ML family of languages. As mentioned in Section 2.1.6, in OCaml there are two separate sets of mathematical operators for dealing with floating point and integral number. Some argue[41] that this makes the language more readable. After all, any of the methods for ad-hoc polymorphism introduce a trade-off between readability and expressive power of the language. Without ad-hoc polymorphism and the possibility of name aliasing, we can quickly identify a definition corresponding to a given name. Java, Haskell and ML do not allow name aliasing of methods or functions. But with the addition of ad-hoc polymorphism, it is more difficult to identify which code corresponds to a given name. In Haskell, any of the `show` implementations defined in various type class instances could be executed at runtime. Similarly in Clojure, any of the protocol implementations could be used, depending on the type of the first argument. In Java, this is not the case, since method overloading is resolved statically, which creates its own problems[42], but its polymorphism via inheritance behaves in the same way.

Although we make our code more ambiguous, and this has severe impact on the complexity of error messages (discussed further below), the popularity of type classes in Haskell and the popularity of polymorphism via inheritance[8] shows that this is a feature worth adopting.

Some even considered extending Haskell to provide for even stronger notion of name overloading, akin to the way object-oriented languages resolve methods.[43] Currently, if two type classes

[7]In the example, in both Haskell and ML, it is partially applied.

[8]Note that we do not distinguish here between extending classes and implementing interfaces in Java, that distinction really exists because of the problem of multiple inheritance but it fundamentally offers the same kind of polymorphism.

define methods with the same name, they must be qualified with the corresponding name of the modules (and hence such classes cannot be defined in the same module).

2.2.3 Parametric Polymorphism

We already discussed parametric polymorphism in Hindley-Milner style typing systems. This kind of polymorphism is very powerful in its generality and *generics* play a similar role in object-oriented languages. We will not cover them in detail, it suffices to say that the inclusion of generics in languages like Java and C# and the loud advocacy for adding them to languages which lack them, notably Go[44], hints at the importance of parametric polymorphism in general. The concept is not directly equivalent between the two classes of type systems though and subtyping means that generics have to deal with the notion of variance[45], which we won't discuss further.

Although it might seem that type inference, which we describe next, is the main culprit of increased type error complexity, it is in fact parametric polymorphism that causes it. In Java or C#, a type error which does not involve generics has a very small scope. Consider the case where a supplied argument to a method has the wrong argument. The expected type of the argument is given by the type of the method, which is always explicitly annotated. The argument can either be a reference to an explicitly typed variable or the result of a method call, which is again explicitly typed. We could also say that there is only a single step between the clashing types.

It is parametric polymorphism that causes types to propagate through the source code, potentially adding steps to type errors. This is why both generics and Hindley-Milner style type systems are often credited with confusing errors. The confusion comes from the fact that types are carried through unification across several method or function applications and compilers usually do not do a great job of surfacing this connection. Because parametric polymorphism is so much more pervasive in languages like ML or Haskell than in Java-like languages, we must acknowledge that we pay a price for the flexibility of their type systems.

2.2.4 Type Inference

An immediately noticeable difference between the two kinds of type systems is that those employing Hindley-Milner type inference do not *require* the programmer to provide type annotations for declarations. This greatly simplifies code creation and modification. It gives the programmer the freedom to decide when a function's type should be explicitly declared and therefore protected against later erroneous changes. Languages which do not support type inference put the burden of updating types after changes to a function on the programmer, and this cannot be fully alleviated by tools because output types depend on annotated input types. Therefore changes to type declarations can only be performed on inputs and outputs separately. The second reason the tools cannot automatically change type declarations is that, in object-oriented setting, the types cannot be easily inferred from method calls.

Type inference does have a slight impact on the complexity of type errors, in cases of groups of mutually dependent declarations. In absence of mutually dependent definitions, type inference is straightforward. Types originate from constants and uses of already typed functions, and every function can be typed in isolation. In cases of recursive and mutually recursive definitions type inference becomes more involved and the inferred types may become less obvious to the programmer.

Number of extensions to the Hindley-Milner system, including Haskell's type classes, make type inference more difficult and may require the programmer to supply annotations. The first such extension is polymorphic recursion. Consider the following definition in Haskell:

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

example = Cons 1 (Cons [2, 3, 4] (Cons [[4, 5], [6]] Nil))

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

The function `length` counts the number of *cons*es in the list. Since `x` is unconstrained, it must have the most general type `a`, and therefore `xs` has the type `(List [a])`. This implies that `length` in the call `length xs` has the type `List [a] -> Int`. But this would imply that `x` has type `[a]`, since in a group of mutually recursive definitions all occurrences of a function, including its declaration, must have the same type[46] to preserve the soundness of type inference. Allowing for polymorphic recursion therefore breaks the completeness of type inference - not all valid programs can be fully inferred. This is perhaps why StandardML does not support polymorphic recursion, while OCaml only recently added support for it[47].

We discuss the impact of type classes on type inference in the following section.

2.2.5 Haskell's Type System

Now that we presented the notion of polymorphism in its various forms in the languages discussed, we return back to Haskell's type system and judge its features and extensions according to the criteria presented above. Haskell has possibly the most expressive type system of the major languages mentioned. We have already introduced type classes as its differentiating feature and here we detail their impact and extensions.

2.2.5.1 Ambiguity

Ambiguity arises in Haskell when a function's polymorphic return type is not constrained via its preceding arguments, such function could have type signature such as `a -> b` or simply `a`. There is no way to write such a function in ML, but it is possible thanks to type classes in Haskell. Consider the function `read`, which converts a textual representation of a value to the actual Haskell value:

```
read :: Read a => String -> a
```

Such a function is convenient, because we can simply implement it as a method of the class `Read` for our custom value and such value can now be used anywhere we require reading from a serialized form, such as inputting values through command line arguments. Using such function can be non-trivial though. Consider testing `read` in a REPL:

```
Hugs> read "3.2"
ERROR - Unresolved overloading
*** Type       : Read a => a
*** Expression : read "3.2"
```

Hugs simply states that the overloading, that is the use of the overloaded function `read`, was not resolved, that we haven't given enough information for the compiler to decide which `read` to use. Figure 2.9 includes the result from the Glasgow Haskell Compiler (GHC) for the same call.

GHC's error message is much more verbose, but essentially boils down to the same main cause, providing more information about the error's location and some hints for a possible fix. We would prefer Hugs's message in this trivial case, since the programmer gains little from GHC's verbose output. Crucially, the error message does not provide a recipe for fixing the problem, where one possible solution could look like:

```
read "3.2" :: Double
```

One might argue that having such ambiguously typed functions is a bad idea in the first place, and we could avoid this particular case by specifically naming our functions, such as:

```
readDouble "3.2"
```

```
> read "3.2"
<interactive>:2:1:
    No instance for (Read a0) arising from a use of 'read'
    The type variable 'a0' is ambiguous
    Possible fix: add a type signature that fixes these type variable(s)
    Note: there are several potential instances:
      instance Read () -- Defined in 'GHC.Read'
      instance (Read a, Read b) => Read (a, b) -- Defined in 'GHC.Read'
      instance (Read a, Read b, Read c) => Read (a, b, c)
        -- Defined in 'GHC.Read'
      ...plus 25 others
    In the expression: read "3.2"
    In an equation for 'it': it = read "3.2"
```

FIGURE 2.9: The error message for a function call with an ambiguous type.

Although this works well in this example, such an approach would limit polymorphism. We might naturally define a function which given a list of strings returns a list of values:

```
readMany :: Read a => [String] -> [a]
readMany = map read
```

Without the ambiguous `read`, we would have to redefine such a function for each type or resort to dictionary passing. Ambiguity is therefore another part of the trade-off made by including type classes in Haskell. The issue does not appear often but when it does the language should provide a clear, concise error message. We find that both GHC and Hugs are still lacking in this respect.

2.2.5.2 Defaulting

Perhaps the most common occurrence of ambiguity in Haskell is due to the definition of numeric literals. Surprisingly, numeric literals are themselves functions with ambiguous type:

```
1.5 :: Fractional a => a
  3 ::        Num a => a
```

To avoid running constantly into the type ambiguity error, Haskell implementations employ *defaulting*. Essentially, Haskell specifies an ordering in which instances of the Num class should be selected when an ambiguity arises. Without this rule, even such mundane definitions as:

```
x = 13 + 29
```

would be ambiguous. Although practical, this solution does not scale well beyond the built-in instance of the `Num` class. Surprisingly, although referred to as conservative, this simple mechanism can lead to strange looking errors:

```
 > let x = "0.5"
 > show ((read x) + 2)
"*** Exception: Prelude.read: no parse
```

The runtime error occurs because the literal `2` defaults to an integer, and hence the `read` function tries to translate an integer, although we gave it a representation of a floating point number. Without defaulting, the compiler would report an error due to the ambiguity of the used types and the programmer would have a chance to supply a correct type signature.

Another solution to the ambiguity of numeric literals in the presence of a numerical type class hierarchy is to not make the literals ambiguous (like in OCaml) by default, but this would force the programmer to use specific converting functions to create polymorphic functions, such as a simple adder:

```
addOne :: Num a => a -> a
addOne x = x + (fromInteger 1)
```

Yet another solution would be to have a single numerical type. This was not possible in Haskell due to the desire for both performance and expressiveness.

2.2.5.3 Multi-parameter Type Classes

Type classes turned out to be extremely useful, but very early on the Haskell community investigated the possibility of having type classes with more than one parameter. Consider a scenario in which we would like to operate uniformly on all kinds of sequences: lists, strings, vectors (extensible arrays) and so on. Both Clojure and Java provide such an interface to its collection libraries. In Haskell, we could model such an interface with a multi-parameter type class:

```
class Seq c a where
  cons  :: a -> c -> c
  first :: c -> a
  len   :: c -> Int
```

where the type variable `a` stands for the type of the element of the sequence and the variable `c` stands for the (concrete) type of the sequence itself. Such a declaration allows us to use sequences which are possibly defined only on some concrete types, such as `ByteString`[9]:

```
instance Seq ByteString Char where
  cons  = ... -- :: Char -> ByteString -> Char
  first = ... -- :: Char -> ByteString
  len   = ... -- :: Char -> Int
```

[9]We could use a class of type constructors, but there is no way in Haskell to define `ByteString` as a type constructor taking `Char` as its only possible argument.

Multi-parameter type classes increase the chance of ambiguity, specifically because the type system considers any possible combination of their arguments as valid. Several solutions to this problem have been proposed, with functional dependencies being the most popular. In essence, functional dependencies let the programmer specify relations between the parameters, which limit the number of possible combinations of parameters instances can use, therefore decreasing the ambiguity in the system. Returning back to our example of the `Seq` class, we can declare that the type argument `c` functionally implies the argument `a`:

```
class Seq c a | c ~> a where
  cons  :: a -> c -> c
```

This means that if there is an instance declared with some type for `c`, such as `Seq ByteString Char`, we cannot declare another instance with the same type (instance `Seq ByteString Byte` would be rejected by the compiler). In turn, type inference can infer the type of first argument to `cons` if the type of its second argument is known.

Unfortunately, without restrictions on type classes and functional dependencies type inference becomes incomplete[10]. It has been proven[48] that satisfying the following default properties used in Haskell leads to sound and complete type inference:

- *Basic Conditions.* The basic conditions apply in Haskell regardless of whether functional dependencies are used or not.
    1. Type class constraints (such as `Seq c a`) on instance and class declarations contain only distinct type variables. This disallows constraints such as `Seq String a` or `Seq a a`.
    2. At least one of the types in an instance declaration must not be a type variable. This disallows all-capturing instances such as `Seq a b`.
    3. Instances must not overlap.
- *Functional Dependencies Conditions.* These originate from the original proposal for functional dependencies[49].
    1. Instances are consistent with function dependencies. This disallows clashing instances such as `Seq String Char` and `Seq String Int`.
    2. In instance declarations, all free variables in the types on the right hand side of a functional dependency must be present in the types on the left hand side of a functional dependency. This means that knowing the left hand side types we can infer the right hand side types. This disallows ambiguous instances such as `Seq [a] b`.

[10]Or undecidable. We do not distinguish the two properties further, as they align.

3. Regardless of functional dependencies, the type class constraints on class and instance declarations must not be ambiguous. That is, all free variables in the constraints must occur in the class and instance types.

These conditions, although they might seem complex, are quite intuitive in practice. Unfortunately, they are also quite limiting. We return to them in Section 3.5. It turned out that lifting some of these restrictions actually makes the Haskell type system Turing complete.[50] There are other extensions which offer similar expressive power, namely polymorphic kinds and type families, but they always lead to undecidability of type inference.

To conclude, we return back to the criteria for type systems which we outlined at the beginning of this section. Although we have not discussed specifically in detail the type systems of other statically typed languages besides Haskell, we have hinted at their differences. We are going to shortly summarize the three main families considered:

- The type systems of *Java and C#*, and those inspired by them, have a good support for inheritance-based polymorphism. In the eye of a functional programmer this kind of polymorphism is quite limited, as it constructs a rigid graph between concrete types (the is-a relationship). Their support for ad-hoc polymorphism and parametric polymorphism is on the other hand quite cumbersome. As such we do not consider them to provide the adequate expressive power we would like to see in a type system. Their error reporting is good and effective, as long as the programmer keeps astray of generics. Type annotations are always required and can be a major hurdle when creating and manipulating code. These type systems, partially thanks to the popularity of the respective languages, are extremely reliable and type checking is very fast.
- StandardML and OCaml

2.3 Macro Systems

Often programmers wish to express abstractions which are not representable in the rigid syntax of the programming language they are using, or which require more syntax than the programmer would deem necessary. Consider for example the case where we want to collect the result of one of several expensive computations, in a statement oriented language. Using syntax for basic control flow, we express the program as in Figure 2.10.

Such pattern might be very common, yet in many languages, including Java, this is the most concise way of expressing what we want. Such code is repetitive and easily lends itself to a later change breaking the abstraction. It might be the case that we would prefer code akin to Figure

```
if (c == 'x') {
  result = computationForX(a, b);
} else if (a + b == y) {
  result = anotherComputation();
} else if (check()) {
  result = kComputation(a);
}
```

FIGURE 2.10: Example where rigid syntax leads to repetitive code.

2.11 or similar[11]. The code would execute like a conventional LISP `cond` macro does, returning the value on the RHS of `=>` of the first LHS which evaluates to true.

```
result = cond {
  (c == 'x') => computationForX(a, b);
  (a + b == y) => anotherComputation();
  (check()) => kComputation(a);
}
```

FIGURE 2.11: Using a macro called `cond` to provide a better syntax.

In other cases we would like to replace the original syntax with a better syntax. A similar example to the one presented above would be the `switch` in Java, which unfortunately adopted C's fall-through semantics, causing many programmers a headache with hard to find bugs in their programs. Both cases could be solved with the use of *macros*. Macros are, most generally, functions ran at compile time which produce syntax. They provide the facility for meta-programming, programming at the level of the program itself. We won't concern ourselves with primitive template-like macros systems, such as the one in C, which only allow simple one-step transformations and are usually unsafe, the issue of safety described shortly.

In the systems we are interested in, *macros* are transformations which take as input syntax, in some structured form, that is not plain text, and produce new syntax in the same form. It is the job of the compiler to take the source code provided by the user, usually as text, transform it into the proper form, pass it to the macro and, once all macros have been applied, compiling the final syntax to the target representation, assembly for example. This kind of macro system originated with LISP[51] and we will therefore look at how its modern dialects mentioned before implement macros.

2.3.1 Macro Hygiene

A key observation very early on in the development of macros was the possibility of accidental variable capture. Consider a trivial macro, which expands to an addition of a constant value to given argument. We might define such a macro as in Figure 2.12

[11]The syntax used here might slightly resemble Rust macros, but is indeed entirely fabricated.

```
(define-macro add-seventeen-to [x]
  `(let [y 17]
       (+ ~x y)))
```

FIGURE 2.12: A macro returning a let binding with an addition.

The syntax we use is similar to Clojure's.[12] The macro definition takes one argument, `x`, which stands for some piece of syntax, hopefully evaluating to a number. So before the macro expansion, the code might look like:

```
(add-seventeen-to 1)
```

And after expanding it, we would like to have:

```
(let [y 17]
     (+ 1 y))
```

which would subsequently evaluate to `18`. The back-quote (or back-tick)` in the macro definition signifies that the form following it should not be evaluated in the current context, but instead should be treated as new syntax. Because we want to use the syntax named `x`, not produce a new atom `x`, we precede it with a `~`.

The problem of variable capture appears when we use a name generated within a macro in the surrounding code. Consider what would happen if we first named a value passed to our macro `y`:

```
(let [y 1]
     (add-seventeen-to y))
```

This would expand to:

```
(let [y 1]
     (let [y 17]
          (+ y y)))
```

and inadvertently evaluate to `34`. There are several techniques to avoid this problem, with vast consequences to the way each macro system works. The macro systems which avoid variable capture by default are called *hygienic*. Notably, Clojure and Racket deal with this problem differently.

2.3.2 Source Location Information

A second differentiating aspect of macro systems is the way in which they deal with syntax source location. This is important for correctly reporting errors. As with hygiene, Racket has the most sophisticated approach in this respect.

[12]Clojure uses `defmacro` to define a new macro. Also note that this example would be invalid in Clojure.

We will now describe Clojure's macro system, inspired partly by Common LISP (CML), before delving into the more complex Racket system, which greatly expands upon Scheme's approach.

2.3.3 Macros in Clojure

All LISPs share the homoiconicity property - the source code of their programs is easily expressible using their data structures. For example, the text `(a b c)` represents a list with three values. If we use this text as syntax, its semantics depend on the context in which it is used. Often, such an expression would be treated as an application of an operator, either a function or a macro, `a` to two arguments `b` and `c`. Instead if we prefix the list with a quote (an apostrophe), Clojure will treat it as list. At the REPL:

```
> '(1 2 3)
#=> (1 2 3)
```

If we used such a list as a result of a macro (we have already seen an example of a Clojure-like macro in Figure 2.12), Clojure would try to apply `1` as a function to the rest of the elements in the list. This is the core of homoiconicity, the fact that all of LISP's syntax consists of lists which are easy to produce and manipulate within LISP itself.

Indeed, `quote` applies recursively:

```
> '(1 (22 23) 3)
#=> (1 (22 23) 3)
```

We could use ordinary list operations to construct lists with values which come as arguments to macros, but many LISPs include the convenient back-quote,`, syntax. Back-quote works similarly to quote, but allows the inclusion of tilde prefixed expressions, which are then "evaluated" in the surrounding context. To clarify, consider a macro which should output a list of two elements, arguments to our macro:

```
> (pair (+ 1 2) (* 3 4))
#=> (3 12)
```

The possibly non-obvious solution is presented in the Figure 2.13 (further below). To understand this macro, we can follow the steps to create it. First we would like it to produce a syntax for a list, so we can start with an empty one, ignoring our input for now:

```
(defmacro pair [x y]
  `'())
```

First comes the back-quote, since we want to output new syntax matching the following form, and then comes the normal quote and empty parentheses, which is Clojure's syntax for an empty list[13]. Next we would like to include the syntax represented by `x` and `y` in our lists, so we could write:

```
(defmacro pair [x y]
  `'(~x ~y))
```

But remember that the quote works recursively, so we would not have our arguments evaluated, because they would still be inside the single quoted list. The solution we chose is to use a back-quote instead, and then add another tilde in front of our arguments. That way, the arguments are first evaluated in the context of the macro and then in the context of the outputted list[14].

```
(defmacro pair [x y]
  ``(~~x ~~y))
```

FIGURE 2.13: A macro expanding to a list of its arguments.

Understanding the basics, we can turn to the question of variable capture. There are actually two cases to consider:

1. Binding a name which might shadow (hide) a name at call-site.
2. Using a name which could be redefined at call-site.

In a way, these two cases are duals of each other. Clojure solves the first by using a special macro called `gensym`, which can be used to generate a unique name, which is a working, although possibly partial solution to the problem.[15] The second is solved by resolving names used inside a back-quoted expression to properly namespaced symbols. For example, using the core Clojure function list at the REPL:

```
> `(list 1 2 3)
(clojure.core/list 1 2 3)
```

we can see that the back-quote expanded `list` to its fully namespaced identifier.

Clojure does not seem to have any mechanism to manually handle the source location of syntax forms transformed via its macros. That said, Clojure does track the location of expressions, and if an argument to a macro fails to execute after the macro transformation, its original location is reported. Any errors which happen anywhere outside the expanded subexpressions, within the output of the macro, are simply reported as being located at the beginning of the macro call. That said, Clojure as a language is not famous for good error reporting[52] and in that light it seems logical that the design of its macro system was not driven by the ability to provide good error messages.

[13]Indeed, in Clojure we don't need to quote an empty list and in turn we wouldn't have to back-quote the result either, but we omit this detail here.

[14]There is of course a much simple solution, namely to use the `list` function instead of using a quote, but this wouldn't demonstrate the nested evaluation scheme.

[15]There are few details available on this topic, but in general Clojure maintains the uniqueness during a single runtime of the compiler and this suffices together with the solution to the second case.

2.3.4 Macros in Racket

Racket tackles macros from a completely different angle than we have been discussing so far and substantially extends the ideas found in Scheme. First, let's note that Scheme's[16] syntax takes a different approach to function and macro definition than Clojure. Let us compare:

```
(defn id [x] x)                 (define (id x) x)
(defmacro id [x] x)             (define-syntax (id stx)
                                  (datum->syntax
                                    stx
                                 (cadr (syntax->datum stx))))
```

FIGURE 2.14: The identity function and macro, in Clojure and Racket.

For functions, Scheme's syntax resembles pattern matching, whereas Clojure's can be thought as providing a list of arguments. Unfortunately, the same syntax is adopted for macros in Scheme, although it no longer corresponds, since the argument passed into the macro is the complete form, including the macro identifier.

Let us explain how the Racket example expresses the same macro as the one written in Clojure. Racket distinguishes between *syntax object*s and ordinary values, called *datum*s. This is, perhaps rightfully, criticized by proponents of CML-style macro system, but note that Scheme has a more powerful mechanism for defining macros than the one showed here, which we will discuss in due course. It does indeed partially break the homoiconicity of the language. We can easily convert one type of value to another and once we are dealing with datums, we can use the usual LISP list functions to process them. In the example, we obtain the list representing the complete form of our macro call, use `cadr` to obtain its second element, which is the first argument passed to the macro (called `x` in Clojure) and then convert it back to syntax, using the original form as a "context" for the newly created syntax expression.

From this trivial example we can see that Racket handles syntax metadata, including source location, explicitly.

Before we return to the problem of variable capture, we present the syntactic sugar which makes writing macros in Racket more convenient. In the simplest cases, we can write:

```
(define-syntax-rule (tuple-with-3 x y)
  (let ([z 3])
    (list x y z)))
```

We notice that there are several transformations that have been applied automatically. The result is implicitly converted to syntax (quoted), and arguments passed to our macros used inside of the result are implicitly unquoted, so that their values are used instead of the literal

[16]Scheme has many different versions so we use Racket's syntax in the example, which is equivalent.

names. Therefore `x` and `y` are replaced with whichever value we pass to the macro, whereas `z` is left as is and at runtime replaced with `3`.

Racket deals with the problem of variable capture by distinguishing between the context in which names are bound. Racket does not have the rigid Java namespace used in Clojure so there is no notion of a canonical name. Instead, each layer of compilation or evaluation is treated separately. For example, to use a custom helper within our macro, operating on syntax, we have to define it in a special `begin-for-syntax` block or require it as `for-syntax`. This is achieved by attaching lexical scope information to each identifier[53]. This solves both of the cases of variable capture presented above but adds complexity when constructing macros.

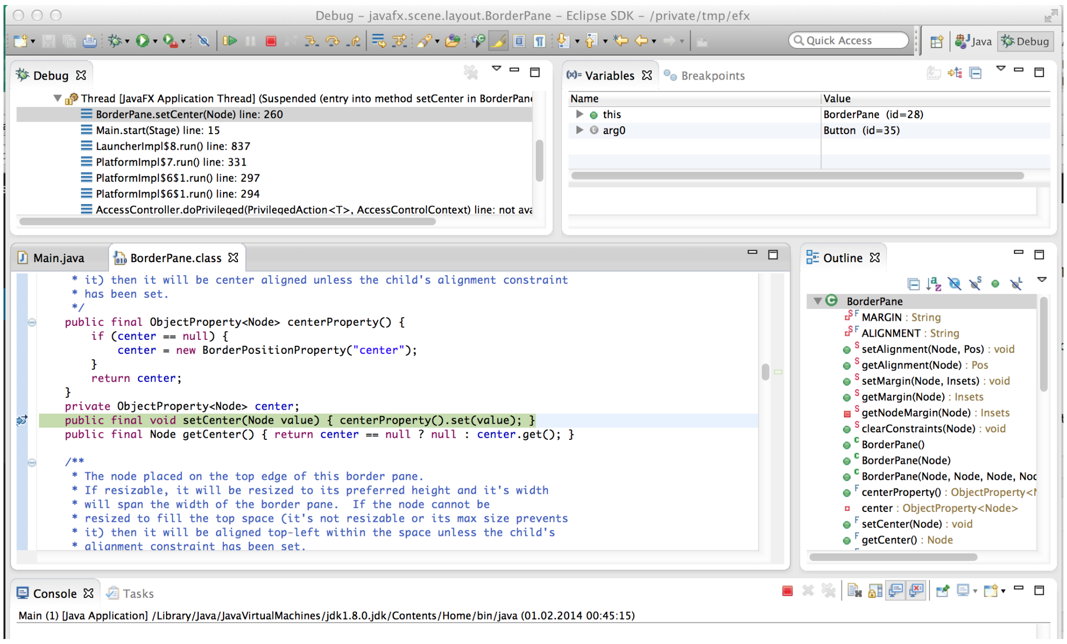
2.4 IDEs

2.4.1 Traditional Big IDEs

IDEs became really mainstream among programmers thanks to the three big Java IDEs (JetBrains IntelliJ IDEA[54], Eclipse[55] and NetBeans[56]) and Microsoft's own Visual Studio[57] for C#. These are now industry standard tools. Indeed, rarely we see programmers using writing in these languages without using one of these environments. They provide many indispensable features: jumping to the definition of a method, smart method call completion and refactoring tools. They also provide numerous not-so-often used features and integrate other parts of the developer's tool chain, such as a debugger or source version control. This sheer amount of functionality makes them some of the most complicated pieces of software in existence. The IDEs are also touted to be "highly extensible", but this extensibility is severely hindered by their complexity and the need for their original inventors to provide suitable APIs which to certain limit determine the extent to which they can be modified. In terms of user interface they are symbols of the era from which they originated - actions might require navigating several stacked menus, followed by filling out information in a popup after which the action is finally performed. Some features might be more interactive, approaching modern text editors, such as renaming a local variable name. In general, the development process supported by these IDEs is iterative rather than interactive. Both refactoring and running code require several steps where the programmer might have to wait on the next step to be performed.

Debuggers are part of all the major IDEs but they are usually fairly isolated from the code, only the last executed line is highlighted while a separate panel is used to display the program state and "watches", expressions which are marked to report their values[17] (see Figure 2.15. The debugger is also isolated from the console view used to print out values, so the programmer

[17]IntelliJ has been the fastest at incorporating features from modern text editors mentioned later in this section, including in-line view of watched values.



```
Thread [JavaFX Application Thread] (Suspended (entry into method setCenter in BorderPan
    BorderPane.setCenter(Node) line: 260
    Main.start(Stage) line: 15
    LauncherImpl$8.run() line: 837
    PlatformImpl$7.run() line: 331
    PlatformImpl$6$1.run() line: 297
    PlatformImpl$6$1.run() line: 294
    AccessController.doPrivileged(PrivilegedAction<T>, AccessControlContext) line: not ava
```

Name	Value
this	BorderPane (id=28)
arg0	Button (id=35)

```
     * it) then it will be center aligned unless the child's alignment constraint
     * has been set.
     */
    public final ObjectProperty<Node> centerProperty() {
        if (center == null) {
            center = new BorderPositionProperty("center");
        }
        return center;
    }
    private ObjectProperty<Node> center;
    public final void setCenter(Node value) { centerProperty().set(value); }
    public final Node getCenter() { return center == null ? null : center.get(); }

    /**
     * The node placed on the top edge of this border pane.
     * If resizable, it will be resized to its preferred height and it's width
     * will span the width of the border pane.  If the node cannot be
     * resized to fill the top space (it's not resizable or its max size prevents
     * it) then it will be aligned top-left within the space unless the child's
     * alignment constraint has been set.
```

```
BorderPane
    MARGIN : String
    ALIGNMENT : String
    setAlignment(Node, Pos) : void
    getAlignment(Node) : Pos
    setMargin(Node, Insets) : void
    getMargin(Node) : Insets
    getNodeMargin(Node) : Insets
    clearConstraints(Node) : void
    BorderPane()
    BorderPane(Node)
    BorderPane(Node, Node, Node, Noc
    centerProperty() : ObjectProperty<
    center : ObjectProperty<Node>
    setCenter(Node) : void
    getCenter() : Node
```

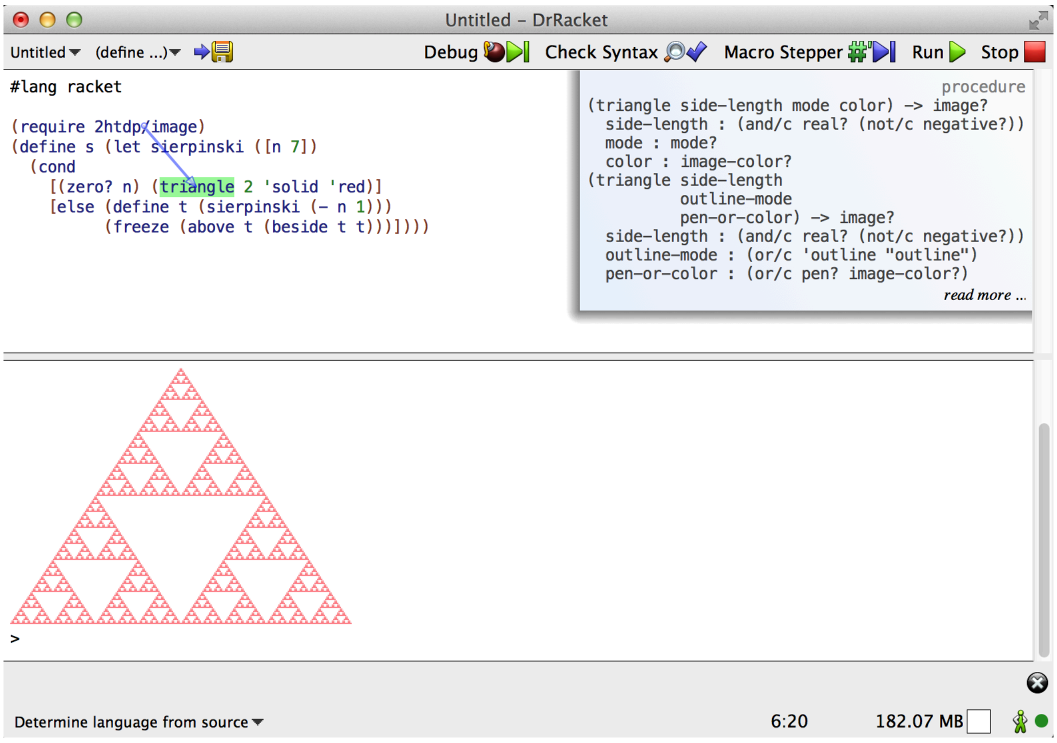
```
Main (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home/bin/java (01.02.2014 00:45:15)
```

FIGURE 2.15: A view of the Eclipse IDE, most popular traditional IDE.

usually has to decide between the two approaches to debugging: stepping through code or printing intermediate values. Today's browsers have actually gotten better at providing useful representation for printed values than IDEs, which only show text using the same language primitives that were designed for the use on a command line.

2.4.2 DrRacket

DrRacket[36] is the editor bundled with the distribution of the Racket language. Its UI feels slightly archaic and certainly is not as feature proof or ergonomic as other IDEs or editors, but it does incorporate several interesting features. Firstly, its REPL (the bottom half of Figure 2.16) allows to display arbitrary values, not just text, which is useful for working with 2D and 3D graphics[58] but also for working with more complicated data structures. Another interesting feature is the highlighting of origin of a reference on hover, which shows an arrow from the location of the definition (either a module import or a global or local definition) to the hovered identifier. Hovering over a module then shows arrows to all the references of its values. We think this feature is valuable, although most likely not surfaced in the best way, especially if both the definition and its reference do not fit on one screen. Another useful LISP specific feature is the built-in macro stepper, which allows the programmer to easily step through the process of expanding a macro. DrRacket provides the best REPL-style programming experience we have seen so far.



```
#lang racket

(require 2htdp/image)
(define s (let sierpinski ([n 7])
  (cond
    [(zero? n) (triangle 2 'solid 'red)]
    [else (define t (sierpinski (- n 1)))
          (freeze (above t (beside t t)))])))
```

```
                                          procedure
(triangle side-length mode color) -> image?
  side-length : (and/c real? (not/c negative?))
  mode : mode?
  color : image-color?
(triangle side-length
          outline-mode
          pen-or-color) -> image?
  side-length : (and/c real? (not/c negative?))
  outline-mode : (or/c 'outline "outline")
  pen-or-color : (or/c pen? image-color?)
                                        read more ...
```

```
>
```

FIGURE 2.16: The user interface of DrRacket, showing the origin-of highlighting.

2.4.3 LightTable

LightTable[59] is a recent competitor in the space of modern text editors, which became popular with the advent of Sublime Text[60] and have been joined by Atom[61]. LightTable is trying to be more than a text editor though. One of the original guiding principles and its main focus is on "shining some light on related bits of code".[62] But it turns out that this is quite difficult and so the released version of LightTable provides more traditional debugging tools - watches, which are conveniently placed next to the line which contains the watched expression as opposed to having a separate table as in traditional IDEs' debuggers. This a rather cosmetic change. Precisely because it is difficult to surface data in other way, our IDE relies on a more traditional approach pioneered by old LISPs - experimenting with expressions and their results. This is why our IDE is more similar to a command line interface than, as LightTable does, showing a single browser page whose content can be edited and autoreloaded on the fly. This approach is reinforced by using a safer, statically typed language, which cements the connections between various parts of code and obviates some of the need for observing intermediate values. But the watching capability is still of course a desirable feature to have in an IDE.

Another interesting guiding principle is "Files are not the best representation of code, just a convenient serialization". Although we principally agree with the sentiment, we must also acknowledge the difficulty of coming up with different code organization that can be easily navigated. This is again why the current version of LightTable focuses on displaying traditional file buffers. In the original LightTable prototype the way to look up a function without knowing its name was to search through the names in a known package. Although this is one useful approach to searching for code, it does not fully replace the combination of visual identification by structure and detail - simply identifying a function based on its content, not just a name, type or documentation; and visual identification by placement, that is remembering where in a static file the function resided (towards the bottom or towards the top). Humans are notoriously bad at keeping multiple objects in their short term memory and so, as projects become larger and more complex, relying on purely on remembering names does not work well in practice.

Apart from these differences LightTable has shared similar philosophy to our IDE, not by the least since their joint influence has been the work done in the several past years by Bret Victor.

2.4.4 Bret Victor and the Future of Programming

In his highly entertaining and educational talk titled The Future of Programming[63] Bret Victor[64] argues that the biggest tragedy for the field of computer science would be if the main programming paradigm of the early 1970s (sequential coding in imperative procedures stored in text files) became the dogma for future generations to the point where programmers stop thinking about new and better ways to program. Whether this is the situation we are in now depends on our point of view. From the industry standards perspective, indeed this model still reigns with a few advancements: procedures are grouped into classes, state is grouped in instances and locking is performed by the runtime. It turns out that we are still severely constrained by the hardware in our computers and that general purpose programming based on ideas as novel as those from 1960s and 70s presented in the talk is still not in sight. On the other hand, there are certainly many people still interested in coming up with better ways to program, some more radical than others. It is our intent to be part of this movement, on the side of less ambitious but more applicable projects.

One of the key topics Bret Victor has been focusing on is making programming better by allowing the programmer to better inspect how the program works and what results the program produces. The relevant articles and videos are: Inventing on Principle[9], The Ladder of Abstraction[65] and Learnable Programming[66]. They all argue for a live editing environment where the changes to code immediately propagate to the results, which are always present and observable. It is of importance to stress that the vast majority of programming today is still done in a way where programmers make a change to source code, then hit a “play button” and then observe the newly generated results, often relying on their memory to compare with the previous results. But the latter two papers also focus on the fact that live programming in itself is not enough. Bret Victor stresses the features he shows are mere examples, which is important because most of them, although they look very appealing, would not work even for slightly more complicated examples. But in general we completely follow the goals set out in the Learnable Programming paper:

1. to support and encourage powerful ways of thinking
2. to enable programmers to see and understand the execution of their programs

The principles outlined are to make the code more readable, to be able to follow the flow of computation, to be able to inspect state, to make creating more reactive and to be able to abstract easily. All these are reflected in the design of our IDE detailed in Section 4.

2.4.5 Swift Playground

Apple has been inspired by the work of Bret Victor when producing their new language called Swift[67]. Swift includes a playground, a plugin to Apple's XCode IDE. The environment includes some of the tools presented in the Learnable Programming paper, such as an automatic graphing of values in an iteration loop and a time debugger. It essentially extends the watches present in LightTable to show any kind of output, such as an image view. It is a great step towards interactive programming, but fails short for two main reasons. It does not support any interaction with the viewed results. So it is completely passive, apart from the few built-in tools. Secondly, and this is where its name comes from, it does not support custom app or framework code. This means that only code pasted into the playground can be used, limiting it to small experiments which must then be copied and pasted into the main program.

2.4.6 Aurora

After "finishing" LightTable, its author has shifted focus to a more general programming model which could be applied not only to programmers. Aurora[68] was a demo of such an approach. The key difference from LighTable is that it focuses on data as opposed to abstract code - indeed, there is no abstract code, as all values are concrete and functions are defined using example values. This was a very interesting experiment which also showed the power of live, interactive editing, but as with many prototypes it remains to be shown whether such approach could scale beyond simple examples.

The second interesting aspect of the prototype was the way the user interacted with it. The interaction relied heavily on mouse clicks, drags and contextual menus, which are generally preferable for less technical audience. The important point is that, unlike LightTable, which in regards to editing remained a very traditional text editor with standard key bindings found in Vim, Emacs or Sublime Text, Aurora was a truly structural editor. We will show later how having a structure-oriented editor can lead to some of the tools described in Learnable Programming.

2.4.7 Lamdu

Another example of the potential of combining a structural editor with a statically typed language designed for structural editing is Lamdu[69]. Lamdu uses a very specific representation for code using a custom rendering engine where expressions are represented by boxes of various sizes. Arguments to functions in the Lamdu language are always named and there are no tuples, only named records, which don't require type declarations, same as in Elm or PureScript. The resulting representation strives for readability but it remains a question how well it works on large definitions where the font size must decrease to accommodate them.

(screenshot of factors)

But the more interesting aspect of Lamdu is the way code is inserted. Lamdu uses type-based auto-completion, and pressing space both inserts the completion and creates an appropriate hole for the next expression. This enables extremely simplified code creation. For example, to define the factorial function, the key presses required are:

```
fac x=if x=0 1 x*f x-1 // keypresses
fac x = if (x == 0) 1 (x * fac (x - 1)) // resulting definition
```

Of course it is not the number of key presses that's important, but the fact that the IDE helps the programmer create a meaningful program by providing type-correct suggestions and taking away some of the burden of structuring the code. This focus comes from a belief, which we share, that by simplifying the editing experience the programmer can better focus on the actual problem they are trying to solve. We drew inspiration from this aspect of Lamdu, but otherwise don't share the approach of creating a custom representation and custom native language which we believe limits the applicability and hinders the adoption of such technology. Yet Lamdu is a very ambitious project which approaches similar goals to ours from a different perspective.

2.5 Ancient Wisdom

It seems that many of the ideas for a better programming environment have been around for decades without pertaining the mainstream. They can be largely associated with the academia (or similar environments like Xerox PARC) and they have had a rather subtle impact on the industry. Here we find again the struggle between designing useful tools and their performance a key factor. Many of the great ideas of 1960s and 70s were simply too advanced or did not fit well the mainstream hardware of the time. Nevertheless, they have inspired and will be inspiring generations of language and tool designers. The lessons presented here are just some of the ones that had the biggest influence on our design.

2.5.1 Smalltalk

Smalltalk[70] and its modern evolution Pharo[71] are fully interactive object-oriented programming environments. Smalltalk allowed the programmer to shape the environment in which they worked and to experiment by editing running programmers. It pioneered message passing as the fundamental unit of computation in a way which nowadays survives in the actor model and languages like Erlang[72]. But it ultimately fails because it is an isolated, interpreted environment. This means that it is both less performant and difficult to deploy.

2.5.2 APL

APL is a beautifully designed language which tried to provide a better tool of thought[73], focusing on its notation (syntax). The principles behind APL were correct but it tried to model the wrong tool - mathematical notation, instead of coming up with an appropriate tool for the age of computers. The obvious deficiency of the language is that its primitives were named with obscure single characters. Together with the use of higher order operators (akin to higher order functions) and the fact that each name represented two different operations based on its position within an expression (unary and infix binary application), it made programs written in APL incredibly succinct and elegant. This was probably the only advantage: the names were difficult to remember, although they sometimes provided graphical reference to their semantics, hard to input and it made extending the language awkward, because user-defined functions did not look or behaved like the built-in operators. We recognize this urge to simulate mathematics without acknowledging the hardware and software capabilities of computers in Haskell's syntax as well. On the other hand APL is a good role model for a mathematical library: using this library should lead to necessarily as succinct code but it should allow the same form of abstraction and expressiveness.

2.5.3 McCarthy's S-Expressions

Given the syntax of our language we cannot introduce it without referencing the source of its main structure[74]. In it McCarthy realized that:

1. The typical mathematical notation with single letters won't do. 2. Atoms and pairs were all that's required to represent a program.

It's the realization that using lists constructed out of pairs can lead to self-evaluation and simple bootstrapping of the compiler. From a syntactic perspective, S-Expressions are simply the minimal representation of a hierarchical data[75] and in turn allow the programmer to write programs generating programs in a natural way.

2.6 Summary

In this chapter we presented the summary of the related work. The scope of our project is large and so is the spectrum of references we drew inspiration from. Although new languages are appearing constantly, we believe that there is currently no language which matches our design. We have described the issues related to type systems and focused on a couple of aspects of Haskell's type system, since we will need this background for understanding our system in later chapters. Similarly we have described the macro mechanism present in modern-day LISPs. Lastly we have looked at other programming environments, mainly those which try to improve on the traditional "make a change, compile, wait, run the program, wait, make a change..." style of programming and we can conclude that, although improvements are being made, none of the environments provide the sort of experience we describe in Chapter 4.

Chapter 3

Language Design

The design of a programming language is a largely opinionated process, as witnessed by the vast magnitude of varying features among existing languages. This chapter provides the motivation behind design decisions taken with **Shem**[1]. We pay special attention to decisions enabled by targeting our custom IDE, **Golem**[2], described in Chapter 4, which further discusses some of them from the perspective of designing the development environment.

3.1 A Small Extensible Language

In his OOPSLA keynote[76] Guy Steele argues that a language cannot be designed as large, because then a different small language will overtake it, but it also cannot be designed as always small, because programmers would not be able to express themselves without the burden of defining a large vocabulary themselves. Programming languages should therefore be designed to be extensible by their community. Guy Steele meant this on both the language level - programmers should be able to define custom operators and functions - and the community level, in that Java started as a small language but grew with its community.

We must therefore design a small, highly extensible language. This is one of the many reasons we chose S-Expressions for our syntax and decided to support LISP-style macros. It is also why the language is largely inspired by Haskell, which is both at its core and in its standard library a small language, compared to for example Clojure[77]. But we would not call Haskell highly extensible. Its syntax is fixed and difficult to extend via templating and some even resort to copying code[78] instead of using its broken package management system. Yet the problem of packages and their versioning is shared by almost all natively compiled languages. On the other

[1]Shem is the formula which activates the Golem in the famous Prague legend.

[2]According to the legend, Golem is a powerful creature created to help and serve the people of the old Jewish quarter.

hand, JavaScript, which Shem compiles to, is easily extensible, both thanks to the available tools as well as its scripting nature, and Shem builds upon these. The following sections reinforce these choices.

3.2 Runtime

We have decided to target JavaScript, formally ES5[79], as our primary and only runtime. In this section we first explain why we target only a single runtime and then why we chose JavaScript to fulfill this role.

3.2.1 No to Portability

Especially for LISP-like languages it is common that their designers aim for wide portability - the ability of the compiled code to run using various lower lever languages (originally operating systems). Portability is very attractive because it would ideally enable the programmer to learn one language and use it in situations when a certain platform wouldn't support a given language (trying to use C# on a platform without support for .NET) or when some functionality has been implemented only in given lower level language, a particular networking library for example. This approach has several caveats:

- It complicates the design of the language. The design cannot rely on particular runtime features to be available.
- It complicates the compiler implementation. A decision has to be made in regards to the size of the target platform abstraction: the simpler the target implementation required the more complex the language implementation becomes.
- It is difficult to achieve in full. Clojure is a good example of a language which was purposely designed with hosting in mind, yet writing Clojure code which can be compiled for all of its target platforms is non-trivial. This potentially scatters the community, as authors of libraries can opt for a particular target.
- Providing truly equivalent implementations can have impact on performance, for example when native number formats from one target have to be simulated in another one which doesn't support them.

Language design is always fundamentally about the struggle between providing the best possible abstractions and performance. We feel that the desire for portability loses sight of the performance constraint and therefore we designed Shem for a single specific runtime. It helps that the runtime we chose is, in terms of platforms, widely portable itself.

3.2.2 JavaScript

JavaScript is the most ubiquitous runtime of today. It became dominant thanks to the wide popularity of the World Wide Web which evolved from serving static pages to serving dynamic pages and has become increasingly a general application platform. Thanks to the support for direct interfacing with GPUs there are now few types of user-facing applications which could not be implemented in the browser (also referred to as client-side). Indeed there is now a whole operating system which provides only applications based on this technology[80].

This popularity is not superficial, the web comes with a completely different deployment philosophy compared to traditional native platforms. Web applications are available almost instantly and do not require the user to manually install or update them. There are other benefits and some limitations, but this unique aspect will in our opinion drive further development of the platform.

Indeed this increasing popularity was also the primary reason a server-side oriented implementation of the runtime, Node.js, has emerged and become widely popular. Among the arguments for adopting Node.js is that it enables the creation of server-client applications in one language, sharing of common code between the server and the client and, most recently, the pre-execution of client-specific code on the server (for example to prerender so called single-page websites before serving them to the client).

These two use cases of the runtime make JavaScript a perfect platform for general purpose programming. Although the jury is still out on whether Node.js significantly improves or worsens the way server implementations are programmed, it is a fact that no other runtime runs natively in the browser and that Node.js enables the creation of desktop and command-line-interface (CLI) applications for modern hardware[3]. Although JavaScript itself is now evolving, partially thanks to the pressure put on it by transpiled languages, it will stay backwards compatible for the foreseeable future as there are large numbers of old browsers installations still in use worldwide.

Secondly, JavaScript as a language is easily extensible. Libraries can be loaded dynamically, although we do not consider this to be of as much importance as the designers of Java did. More importantly, libraries can be written as single functions or named collections of a functions. Unlike in Java, modules do not have canonical package names and their source code does not have to be placed in a rigid directory structure. There is also no need for PATH executable lookup - it is preferred that dependencies are packaged together and versioned adequately.

[3]Notably, embedded devices and significantly underpowered devices are potentially not a good target platform for Node.js applications as the JavaScript virtual machine used (V8) takes significant time and memory resources to boot up

Server side JavaScript is also credited with one of the best package management systems among programming languages - NPM. The situation in the browser is more complicated, as browsers do not yet support natively loading modules in the same way that Node.js does. There are several ways in which modules can be loaded and we discuss them briefly in Section 3.8. Of course, even more intelligent systems could be created for a statically typed language and Elm is pioneering this approach with its own package management system[81]. The importance of such systems cannot be understated - one of Haskell's biggest problems is its nightmarish dependency resolution and the fact that getting code with dependencies to compile is often a huge ordeal. Clojure, on the other hand, goes as far as versioning its builtin library (the equivalent of Haskell's Prelude) so that a specific version can be easily imported. This is desirable to enable the evolution of the language itself, something Haskell is wrestling with at the moment[82].

Furthermore, by targeting JavaScript, we immediately gain the ability to run our code in the browser, without the performance hit of going through intermediate representations, and this allows us to implement our IDE for the web, making it instantly available to any programmer to try without any associated server costs.

3.2.3 Characteristics

Since we target solely JavaScript we designed the language to take advantage of JavaScript's features and to allow implementation as lightweight as possible on top of JavaScript. Indeed we want to avoid the notion of having our own "runtime" on top of what JavaScript provides. This allows us to focus on more advanced features supporting the programmer, such as our powerful type system and the interaction between the editor and the compiler, while leveraging the implementation of runtime features supported by JavaScript. This is a quality that Shem shares with PureScript over languages like Elm or ClojureScript, which have a significant runtime implementation of their own.

The main JavaScript runtime features we take advantage of are lexical scoping, closures and objects, which are used to implement algebraic data, type class dictionaries and modules. Together with the disadvantages of laziness discussed in Section 2.1.1, this lack of a separate runtime is a reason for our language being strict. There would be no way to implement laziness and adhere to this rule.

3.3 Syntax

Syntax is the user interface (UI) of a programming language. At least a programming language which is meant to be represented in text. Graphical programming languages and graph based

DSLs can employ a more complex representation. We will defer the discussion of why Shem has a textual representation to Section 4.2. Given that Shem is represented in text, what is its syntax? Backus–Naur Form[83] (BNF) rules are often used to describe all the strings which form a syntactically valid program of a given language. Valid in the sense that such statements will not be rejected by the compiler before being checked for semantical errors. Shem is based on LISP's S-Expressions. The full set of rules for complete Shem programs is given in Figure 3.1.

Shem programs are represented by Shem `expressions`. There are 8 rules and whitespace is handled explicitly. For comparison, the official Java specification[86] lists 216 rules[4]. Although the precise count dependents on the granularity of the rules it should be clear from our definition that the "syntax" of Shem is fundamentally simpler than that of a traditional language outside the LISP family, including Haskell.

There are several motivating reasons for using this simple S-Expression style syntax. Firstly, it is simple. It is the simplest representation for trees, and ASTs are intrinsically trees. This removes layers of parsing and rewriting between what the programmer sees and what the compiler operates on. It allows a modern IDE such as ours to display and let the user manipulate code through its structure, as opposed to approximating it from the text. It allows the programmer to easier extend the compiler because the resulting AST is much simpler[24]. It also allows for additional better tooling, such as a version control system which understands the structure of the changing code, unlike today's text line based systems.

But this syntax does not give much information about what valid Shem programs look like. We abandon this definition of syntax and consider a more general formula: syntax is composed of the rules which direct how semantically valid programs, which can be compiled and run, are formed. This is a more useful definition when talking about LISPs or other macro-directed languages, such as Shem.

Shem being macro-directed means that the semantics of a particular syntactical unit are defined by its context - the macro which is used to compile it. A good example is the application of the comment macro[5], invoked as a call to the operator `#`:

```
(# This is a comment. It's contents are syntactically valid.)
```

In this example the comment is called with the arguments `This`, `is`, `a` etc. We can think of macros as procedures that operate on the program syntax and the current context, possibly amending the context and producing compiled JavaScript. The comment macro labels its arguments as comments and does not produce any JavaScript.

[4]Which do not describe how whitespace is handled
[5]Indeed, there is no special "syntax" for comments in Shem

```
expression
    ::=
    atom
    | form
form
    ::=
    "(" terms ")"
    "{" terms "}"
    "[" terms "]"
terms
    ::=
    atom
    | atom (" " | "\n") terms
atom
    ::=
    | character
    | string
    | regular-expression
    | identifier
character
    ::=
    "\" (<JavaScript string literal characters, "'", """,
          latin and unicode escape sequences>
        | "newline"
        | "tab"
        | "formfeed"
        | "backspace"
        | "return")
string
    ::=
    <JavaScript double quoted string literal
      without escaping newlines>
regular-expression
    ::=
    <JavaScript regular expression literal
      where first character is not whitespace>
identifier
    ::=
    <any of a-zA-Z+-*/=@#$%^&*~ ,<>| >+
```

FIGURE 3.1: The EBNF[84] rules for valid Shem expressions, referencing JavaScript strings[85].

The compiler is therefore a collection of macros[6] and the definition of the compiling context. The context carries information about current lexical scopes, types and available macros. This makes the language extremely extensible, as the bulk of what is traditionally checked by the compiler before further processing is delegated to macros. Indeed some might argue that this provides too much freedom, but we believe that the process of defining the canonical form of the language should happen at the community level and the language should be flexible enough to accommodate this design process. After all, bloated languages such as C++ prove that having a large specified syntax does not necessarily lead to homogeneous code.

The issue of the compilation context and how it is used by macros is discussed further in Section 5. The next sections describe the effective syntax of the language, defined by its built-in macros, without discussing their implementation.

3.3.1 Syntax Example

Before we go into the details of the built-in syntax, we present an example of a complete Shem program. It is not essential to understand its semantics, we only want to give the reader a chance to get the feel for how the language looks in real scenarios:

```
Tree (data [a]
  Leaf
  Node [left: (Tree a) value: a right: (Tree a)])

tree-mappable (instance (Mappable Tree)
  map (fn [f x]
    (match x
      Leaf Leaf
      (Node left value right) (Node (map f left) (f value) (map f right)))))

inc-tree (fn [tree]
  (: (Fn (Tree Num) (Tree Num)))
  (map what: (+ 1) over: tree))
```

The example first declares new algebraic data type called Tree, then declares its instance of the Mappable class and finally uses it to define a function which increments the values of a tree of numbers by one.

Shem as a language does not dictate the syntax of a top level view or a file, it leaves this to the tools using it[7]. So another valid Shem program can look like this:

[6]Clojure has a similar concept but calls it "special forms".

[7]Indeed, the way Shem programs are presented in this report is by having the compiler compile them and a trivial script, which converts the annotated AST to valid LaTeX markup. Some examples are standalone while most use the prelude and form a single module, which allows us to use definitions from previous examples.

```
(inc-tree (Node (Node Leaf 1 Leaf) 2 (Node Leaf 3 Leaf)))
```

This program calls the function defined previously on a sample tree. It is a full, valid program, albeit it needs an environment which provides the used functions, macros and types.

We will now give a succinct overview of the built-in syntax.

3.3.2 Design principles

When designing the syntax accepted by macros in Shem our main design principle is *unambiguity*. Contrast this with the syntax of natural languages or mathematical notation - they are both expressive but highly ambiguous. We want to achieve high expressiveness while avoiding ambiguity. Consider the examples of mathematical notation:

$$sin\, x + 1$$
$$\langle 1; 2 \rangle$$
$$3 * 5 \,/\, 2 \,/\, 2 * 3$$

Ambiguity comes from two main sources: the desire for brevity and the lack of standards. Often the meaning of mathematical notation depends on the context in which it is presented. In Section 4.2 we argue that it is the way mathematics were written - by hand, which shaped its syntax. This is why we feel it is a dire mistake to base a syntax of a programming language, which is not written by hand, on mathematical notation. It is simply the wrong tool for the job. It is also important to realize that mathematical notation served two slightly orthogonal purposes. First and foremost, the notation was used to perform calculations. Therefore it was absolutely essential that the notation was as brief as possible. Secondly, mathematical notation provided a tool for thought and conveying ideas to others. It is clear that today code is read more than it is written and hence serves similar purpose. Yet the devil lies in the detail - mathematical papers and proofs have a drastically different format to a large code base - they are relatively short and can expect the reader to study them in full, to remember every definition from introductory chapters, without being reminded of their meaning later. Reading code, on the other hand, is more similar to a search: the programmer needs to find out specific information in the vast amount of code available to him. Descriptive names are preferred to brief shortcuts, homogeneity is valued over brevity. Keeping code short is still important for readability, especially in the sense that superfluous syntax obscures the important parts, but is not the most important aspect for which syntax design should optimize.

3.3.3 Color

The use of S-Expressions as the underlying structure of code does not guarantee simple syntax and readable code. Clearly, if the only rule is that syntax of the language is made out of S-Expressions `(a b c...)` or `[a b c...]`, especially when the latter is just an alias of the former, we can end up with arbitrarily complex syntactic rules referring to the position of terms within the list. Let us compare let bindings and function definition in Racket and Clojure:

```
(let ((x 3) (y 4))              (let [x 3 y 4]
  (+ x y))                        (+ x y))


(define (add x y) (+ x y))      (defn add [x y] (+ x y))
```

We think that Clojure's syntax is slightly better. In the let binding, Racket is unnecessarily verbose, as it is trivial to see which value belongs to which name in Clojure's simplified syntax. When defining a function, Clojure's syntax has a bit more structure and is explicit in that the value being defined is a function, which might not be obvious in Racket. But the key observation should be that reading all of these examples without syntax highlighting is difficult! Indeed, without any sort of highlighting one might argue that the Racket version of let bindings can be more readable than Clojure's for large let bindings.

Colors are essential for understanding Shem's syntax. Traditional languages do not rely on color because their syntax originated in times where coloring on either screen or print was not available. The situations where coloring is not possible have become extremely rare[8] and there is no reason to limit the possibilities in the language design enabled by using colors because of them. Humans have evolved to quickly recognize different colors and it seems too foolish to not take the advantage of this natural ability. Color is less ambiguous than layout rules such as significant indentation and does not form the typical visual clutter of punctuation. We must observe certain limitations though: as with other syntactical features, colors should not be overloaded to signify too many different meanings. We also cannot employ too many colors, since we would loose the ability to easily distinguish between them.

The following is a list of colors used in Golem for the various syntactical features. Every feature will be introduced in more detail later in this section.

[8]There is of course the issue of colorblindness, which can be solved by using other text styles rather than color.

green	new name
red	use of a macro
blue	use of a typed operator or type constructor
yellow	constants, literals, type names
white[9]	reference
orange	special reference
purple	labels

In addition to these we use shades of gray to lower the visual importance of some tokens, namely we highlight parentheses around calls in low contrast to the background. Coloring parentheses is crucial for the legibility of S-Expression-based code, because we do not want them to obstruct the code they surround.

3.3.4 Declaration Style, Where Clauses

We strongly believe that declaration style programs (Figure 2.1), as opposed to expression style[10], relying on let bindings, are easier to read and comprehend, simpler to construct and simpler to maintain. A noticeable difference between the two styles is the reverse order of defining dependencies. We can consider three styles of organizing code within a module:

1. All dependencies are defined before being used. This makes understanding source code easy, as long as the programmer reads, remembers and understands every line of code preceding the area of interest. We believe that for complicated modules and procedures this is inadequate, as the definition of all dependencies obscures the information the reader is looking for.

2. Literate style as envisioned by Donald Knuth. Essentially the idea is to use a text replacement system to present source code in a form similar to a book. This style also presents comments as the main content. Although it influenced some languages to include an *inverted* mode where comments are not delimited but code is (literate Haskell or literate CoffeeScript), the full idea was never adopted. The main problem with focusing on comments is that they are not necessary for the successful completion of a program, therefore maintaining an excessive documentation becomes a burden.

3. Dependencies are defined after being used. This corresponds to the declaration style and helps focus on top-down programming. We can think of the source code as a directed graph (with occasional loops) of dependencies with a root in the API definition or main procedure of a program. The process of reading the code can then be thought of as trying

[10]Not to confuse with expression languages. Shem is an expression language in the sense that there are no statements which cannot be treated as values, although some macros, like class and instances, require to be defined at the top level.

to comprehend this graph and we believe the best way to achive this is to walk the graph from the root, exploring those branches which interest the reader. Hence naturally the code corresponding to nodes closer to the root should come before its dependencies.

Whichever style is used, the more important aspect of the declaration style is that it takes the responsibility of ordering bindings according to the required order of execution within pure code away from the programmer and, on the other hand, gives them freedom to order the definitions based on importance and readability. In the case that definitions are cyclically dependent, the programmer might have to put in additional effort to provide this ordering. For example, in Clojure, the programmer has to use the `declare` macro to be able to use names of values before they are defined, which might be the case with mutually dependent functions.

Lazy languages, such as Haskell, have the additional benefit that bindings which would result in an error can be included among a declaration list as long as the values they define are not required by the current execution path. This issue extends to the notion of referential transparency - we should be able to give any subexpression a name and put its definition into the adjacent definition list. Strict languages, such as PureScript, do not have this property. Consider the example in Figure 3.2.

```
factorial n = case n of
  0 -> 1
  _ -> n * (factorial (n - 1))
```

```
factorial n = case n of
  0 -> 1
  _ -> n * recursiveCall
  where
    recursiveCall = factorial (n - 1)
```

FIGURE 3.2: Naming a subexpression causes an infinite loop in strict languages.

Following PureScript's strict semantics, pushing the subexpression into the definition list causes an infinite loop, as the *where* clause gets executed first. The solution taken in Shem is lifting the definition of subexpressions to the appropriate conditional branch. Therefore the second form of the example runs fine in Shem. This is a great improvement to functional declarative programming in a strict language and a feature we believe is quite unique.

As practical example of the syntactic clarity gained by referential transparency and definition lists, consider the following example, which is an implementation of the modulo operation for very large numbers, in Haskell:

```
modPow x y n
  | y == 0 = 1
  | even y = (half * half) `mod` n
  | otherwise = (mod x n * minusOne) `mod` n
  where
```

```
    half     = modPow x (y `div` 2) n
    minusOne = modPow x (y - 1) n
```

Another need for let bindings arises when pattern matching introduces new names to the scope and we would like to use these inside named subterms. A similar mechanism to the one introduced above can be used to allow the use of these names in the function's definition list[11]. Without this feature, the only way to avoid a let binding is to use a helper function.

```
inc-some x = case x of
  Just value -> incremented
  where
    incremented = 1 + value

inc-some x = case x of
  Just value -> inc value
  where
    inc value = 1 + value
```

Haskell supports this kind of referencing not only from the main expression of a function but also for any pattern matching, again thanks to laziness.

3.3.5 Labels

Many LISPs have the facility of keywords. These can be used by macros in various ways, for example to give names to values in a dictionary, or to provide optional named arguments to functions. Shem does not have keywords, but it does use labels. Labels are identifiers ending in a ':' and how they are used depends on the controlling macro. For calls, labels are used to name arguments, which can be then passed in arbitrary order:

```
(map over: {1 2 3} what: (fn [x] (+ x 1)))
```

Labels are optional, unlike in Objective-C[87], because there are many situations where they would only be an unnecessary burden:

1. Arguments have the same type but the function is commutative:

   ```
   (+ x: 1 y: 1)
   ```

[11]Not currently implemented

2. Arguments have disparate types, especially when local names are the same as the parameter names:

   ```
   what (^ 2)
   over {1 2 3}
   squared (map what: what over: over)
   ```

The main advantage of having labels is in reinforcing code safety when the type system cannot guarantee it, that is when arguments are of the same type and it is easy to involuntarily swap them, causing semantic errors. In Haskell this situation is usually avoided by using `newtypes`, essentially cheap wrapper objects which give the arguments different types. This is a good solution but it might add significant boilerplate if the value needs to be converted often between the wrapper and the original type.

Another advantage is that currying becomes even more powerful. Programmers sometimes use the `flip` function in Haskell to avoid the need to use a lambda or declare a new function by currying a different argument of a function. Consider[12]:

```
map (flip at "ABCDE") ['A', 'C', 'E']
```

```
(map (at in: “ABCDE”) {\A \C \E})
```

In Shem, we can name the argument instead of flipping the order of the arguments, which, unlike flip, scales well beyond two argument functions:

```
(zip (put in: {a: 1 b: 2 c: 3}) {“a” “c”} {4 5})
```

In this example, put takes three arguments, key, value and a dictionary and returns an updated dictionary. Using labels is much simpler and shorter than using flip or a new lambda:

```
zipWith (\at what ->
  put at what (fromList [("a", 1), ("b", 2), ("c", 3)])) ["a", "c"] [4, 5]
```

Labeled arguments can be specified in any order. The remaining arguments are applied in the order they are supplied as the remaining parameters to the function. If there are fewer arguments than parameters, the function is applied partially. This translation is done at compile time and has no additional impact on performance.

This example demonstrates another reason Shem includes labels: they are useful for creating dictionaries with keys as strings, which is arguably an extremely useful data structure to have (Clojure also has built-in syntax for them).

[12] at is similar to Haskell's lookup.

3.3.6 Describing values

The following table lists the default values represented by atoms and forms, shown by example:

`1`
`1.5`
`-3`
`Infinity`
`-Infinity`
	Numerical values, `-` is used for negation

`\A` `\newline`	Characters

`“Hello world”`
`“Multiline`
`  string”`
	Strings

`/\w+/`	Regular expressions
`x`	References
`True` `Green`	Constants
`global.Math.PI`	Name-spaced references
`(+ 1 2)`	Calls
`{3 4 5}`	Arrays
`{a: 1 b: 2}`	Maps with string keys
`[1 2]`	Tuples

Notably, Shem inherits from Haskell the distinction between data constructors using capitalized names and other references using lower-case names. This namespacing nicely corresponds to the practice of capitalizing constants in JavaScript and other C-like languages.

Also note that strings use proper quotation marks for presentation, although they are typed as the usual straight quotes.

3.3.7 Defining names

In Haskell, there are multiple ways new names are defined. There are function declarations and value assignments (semantically different), using the equal sign, and also other syntax for declaring types and classes. In LISPs, such as Clojure, names are usually added to scope via specific macros, such as `define`, `defn`, `defmacro`, depending on the kind of value being assigned. As we already discussed, an important difference is that Haskell allows definitions in arbitrary order, while in Clojure the macros execute imperatively on the context.

In Shem the order of definitions does not matter and all naming is done via pairing of the name with its definition. Consider:

```
x 3
y z
z 42
```

In the example, the value `3` is given the name `x`, `42` is given the name `z` and the value represented by `z`, which we know is 42, is given the name `y`. It is important to see this example in color, as the coloring makes it obvious which part of the pair is the name and which is the value.

The same syntax is used with all other built-ins:

```
f (fn [x] x)

D (data A B C)

C (class [a]
  m (fn [x] (: (Fn a a))))

d-c (instance (C D)
  m f)
```

Macros can of course define other ways of defining names within them, so for example the `fn` macro defines names for its parameters. The same pattern of pairs of names and values, which is what we refer to as a definition list, is repeated within the `class`, `instance` and `fn` macros, detailed next.

3.3.7.1 Pattern Matching

Like in Haskell, Shem supports pattern matching on collections and algebraic data, introduced below. Patterns can be used in definition lists and in the `match` macro (Section 3.3.9.4).

```
[x y] pair
pair [1 2]
```

`x` is assigned the first value in the tuple `pair` and correspondingly `y` is assigned the second value.

There is syntax for matching built-in arrays, although we would like it to match any sequences in the future:

```
{start ..middle last-but last} list
list {1 2 3}
```

`middle` will be an empty array. Similarly literal maps can be matched as well as other built-in collections using their constructor macros, described in Section 3.7.

3.3.8 Style Guidelines

Even though we tried to minimize the number of unnessary choices programmers have to make, there are cases which are better left to them. This is where the official language style guide comes into play, to guide programmers in cases which cannot be enforced by the compiler. Some examples of rules are:

- Pure functions returning a `Bool` should have a name ending in `?`.
- Procedures with side effects should have a name ending in `!`.
- Built-in macros should be neither aliased nor shadowed, especially if the shadowing definition is a macro.

3.3.9 Built-in Macros

3.3.9.1 Functions

Functions in Shem are defined using the `fn` macro:

```
increment-by-slate (fn [x]
  (: (Fn Num Num))
  (# Increments x by a small slate.)
  (+ slate x)
  slate (* 2 half-slate)
  half-slate 0.5)
```

The macro takes a tuple of parameter names, which become the named arguments to the function, an optional explicit type, an optional documentation comment and the result of the function followed by a possibly empty definition list. The names defined within the definition list are local to the function, as in Haskell's `where` clause. Polymorphic recursion and mutual recursion, which is demonstrated in Figure 3.3, are supported.

There are no function patterns in Shem. That means that every function needs to specify the names of its paramaters. This is intentional - we find that Haskell's function patterns lead to

```
fibonacci (fn [month] (adults month))

adults (fn [month]
  (match month
    1 0
    n (+ (adults previous-month) (babies previous-month)))
  previous-month (- 1 month))

babies (fn [month]
  (match month
    1 1
    n (adults (- 1 month))))
```

Figure 3.3: Computing an element of the Fibonacci series using mutual recursion derived from its original definition[88].

badly documented code. We also made sure that in Shem's syntax documentation comments can be read by the enclosing macros and written into the environment so that our IDE can display them during auto-completion. Documentation in Shem is therefore first-class, not ad-hoc, and we hope this will lead to more understable code bases than are usually those written in Haskell.

3.3.9.2 Types

Explicit type declaration can be used to specify the type of a definition which is then checked against the inferred type. For functions, the syntax uses the : operator, followed by the type of the function, followed by any constraints:

```
wrap (fn [x]
  (: (Fn a String String String) (Show a)))
```

The `Fn` type expands to fixed-arity function type constructor application, for the example above `(Fn a (Fn String (Fn String String)))`. This type reflects the curried nature of all functions in Shem. Tuple types have the same syntax as tuples, while lists and arrays have no special syntax, unlike in Haskell. Type class constraints are expressions where the operator is a type class name.

3.3.9.3 Algebraic Data Types

Algebraic data is used to define new constants, similarly to enums in object-oriented languages, and composite values. The `data` macro is used to declare a new algebraic data type:

```
Color (data Red Blue Green)
```

In this example, `Color` is the new type of algebraic data representing colors. This data has three possible values, `Red`, `Blue` and `Green`. This is the simplest variation of algebraic data, where we specify each possible constant value. We can also create data which has values composed at runtime, by specifying the names and types[13] of each encapsulated value:

```
Person (data Person [name: String age: Num])
```

Here, `Person` is a type of algebraic data for which we can create values by using the `Person` constructor function, which the macro generates. Like any other function, we can use labels or bare arguments in the correct order. The record macro is a shortcut for such algebraic data types where the name of the type matches its only constructor.

```
Person (record name: String age: Num)
tom (Person “Tom” 24)
alex (Person age: 22 name: “Alex”)
```

There is an obvious difference between Haskell and Shem in the way algebraic data types are declared: in Haskell, the arguments to data constructors are unnamed. Haskell offers the *record* syntax to deal with constructors with many arguments. The syntax allows the programmer to tag the arguments of a constructor with names, which are then used as function names, where the functions take the composite value and return the corresponding embedded value. The main issue with Haskell's approach is that the record field names can be used only once within a module. Shem does not have this problem because the generated *getters* are prefixed with the constructor name.

```
harry's-age (Person-age harry)

harry (Person “Harry” 21)
```

Haskell also provides syntax for matching and updating records. In Shem, matching using labels follows the use of labels for calls, and is therefore a natural extension of matching without labels. In Haskell, the two syntaxes are quite different:

```
Person {name = harrysName} = harry        (Person name: harrys-name) harry
Person _ harrysAge = harry                (Person _ harrys-age) harry
```

Updating values[14] within records can be done with a generated macro, which accepts any number of labeled arguments to the corresponding constructor and a value to base the new value on. The following assigns `(Person “Harry” 75)` to `older-harry`.

[13]Type class constraints are not allowed in types of algebraic data constructors.[89]
[14]Not implemented yet.

```
older-harry (Person.with age: 75 harry)
```

The advantage of having a macro as opposed to Haskell's special syntax is that it can be also used as a function, simply by not passing in the algebraic data, in a similar manner to how calls are curried.

```
all-older (map (Person.with age: 75) {tom alex harry})
```

As in Haskell, the two variations of algebraic data, also referred to as sum and product types[90], can be combined. Shem of course supports recursive algebraic data types:

```
Lines (data
  End
  Line [line: String rest: Lines])

two-lines (Line “Hi,” (Line “John.” End))
```

To define polymorphic algebraic data, the first argument to the `data` macro must be a tuple of type variables, which can be used withing the declaration of the data type. As in Haskell, this list needs to be explicitly declared so that there is a known ordering of the argument types when we refer to the polymorphic algebraic data type. In the following example, `small-character-tree` has the type `(Tree Char)`:

```
Tree (data [a]
  Leaf
  Node [left: (Tree a) value: a right: (Tree a)])

small-character-tree (Node Leaf \M Leaf)
```

Such values, when we do not know the constructor that has been used to construct them, are deconstructed using the `match` macro.

3.3.9.4 Match Macro

In Shem, pattern matching with several cases is always done with `match` macro, similar to Haskell's case. Although this is a more verbose approach to pattern matching than Haskell's function patterns, it does not require repetition of function names and improves code readability by requiring the programmer to always specify names for function parameters. Apart from algebraic data we can also match on numbers, booleans and collections. Indeed, the `match` macro leaves the matching to the pattern, and therefore new macros can be used as patterns.

```
tree-value (fn [tree]
  (match tree
    Leaf None
    (Node left: l value: v right: r) (Some v)))
```

The function above converts a `Tree` to an optional value, depending on whether the tree is just a leaf or has at least one node.

```
factorial (fn [n]
  (match n
    0 1
    1 1
    _ (* n (factorial (- 1 n)))))
```

Underscore `_`, which is otherwise invalid identifier character, can be used to match any value without assigning it a name. We present the syntax for matching on collections in Section 3.7.

3.3.9.5 Type Classes

Type classes are declared with the `class` macro:

```
Show (class [a]
  show (fn [x]
    (: (Fn a String))
    (# A textual representation of x .)))
```

The macro takes a tuple of type variables, optional list of super class constraints and a definition list with methods, which are explicit typed function declaration without their bodies.

An example of super class constraints is the `Ord` class extending the `Eq` class:

```
Eq (class [a]
  = (fn [x y] (: (Fn a a Bool))
    (# Whether x is equivalent to y .)))

Ord (class [a]
  {(Eq a)}
  <= (fn [than what] (: (Fn a a Bool))
    (# Whether what is less or equal to than .)))
```

Instances are correspondingly defined using the `instance` macro. The macro takes the instance type, which is type class constraint describing the instance, optional list of additional constraints and a definition list of methods. The main difference from Haskell's instance definitions is that each definition in Shem is given a name[15]:

```
show-triplet (instance (Show [a b c])
  {(Show a) (Show b) (Show c)}
  show (fn [triplet]
    (format "[%s %s %s]" (show first) (show second) (show third))
    [first second third] triplet))
```

In the example, the name of the instance definition is `show-triplet`. This will allow for explicit importing or hiding of instances, which solves Haskell's problem of undesired instance clashes[16].

3.4 User Macros

The simplified construction of macros is one of the primary reasons Shem's syntax is S-Expression based. We believe that macros are essential for the high expressive power of our language[10]. We already discussed systems which are built on top of a language, such as Template Haskell, and conclude that these systems are not an adequate substitute for a macro-based language, for several reasons[24]:

1. The ASTs of languages such as Haskell or Java are very complex. Template Haskell uses 48 data constructors to represent Haskell's syntax.
2. The languages also do not posses the homoiconicity property: the translation from syntax that the programmer sees to the data structures representing it is not straightforward.
3. The macro systems require special syntax which allows them to distinguish macro application from ordinary code or, more importantly, code to be processed from actual compile-time data.

In Shem, macros fulfill two purposes. They are used to generate JavaScript and to generate new syntax. The two corresponding macros are `macro` and `syntax`:

[15]PureScript's instances do as well.
[16]We wanted to use modules instead of explicit signatures[91].

```
plus (macro [x y]
  (: (Fn Num Num Num))
  (Js.binary "+" x y))

+ (syntax [..args]
  (curried-plus args)
  curried-plus (fn [args]
    (match args
      {x} (` plus ,x)
      {x y} (` plus ,x ,y)
      {x ..xs} (` plus ,x (, curried-plus xs)))))

summed (+ 1 2 3 4)
```

The `plus` macro expands at the call site to a JavaScript addition. Any macro which is explicitly typed can be curried like a function. The `+` macro expands depending on the number of arguments given, to corresponding, possibly curried or nested, application of the `plus` macro. The resulting JavaScript code is simple:

```
var plus = function (x, y){
  return (x + y);
};
var summed = (1 + (2 + (3 + 4)));
```

First the `plus` macro is declared as a function. This allows us to pass it as a function to higher order functions. After that the `summed` value is defined exactly as expected, after expanding the call to `+` and then expanding the resulting call to `plus`.

The syntax quote (`` ` ``) and unquote (`,`) macros are used to produce (or match) syntax. Shem does not currently support reader macros[92], so these macros are used as ordinary operators. However, they both provide a shortcut - by passing more arguments to them, they automatically create a call containing the arguments. Therefore `` (` a b c) `` and `` (` (a b c)) `` are equivalent. Additionally, `` ` `` automatically applies `,` to identifiers starting with a comma. These identifiers, if they are lists, can be also splatted.

It turns out that macros are much more useful when dealing with imperative and side-effecting code. A great example of macros from Shem's prelude is the `do` macro, providing Haskell-like do notation[93] sugaring atop monadic bindings[17]:

[17]The actual macro in the Shem's prelude allows for pattern matching in the assignment.

```
do (syntax [..actions]
  (match actions
    {x} x
    {x ..xs} (match x
      (` set ,to ,what) (` chain ,what (fn [,to] (do ,..xs)))
      _ (` follow ,x (do ,..xs)))))
```

Here `chain` and `follow` correspond to Haskell's `>>=` and `>>` operators.

Our approach to variable capture is a combination between Racket's and Clojure's approaches. Macros operate on `Expressions`, which is a built in data type. There are built-ins which can check whether an expression is a form and convert it to a list of its terms. The ` macro tags references which are in current scope with their source scope index and mangles names of those which are not. The author of the macro can force new identifiers to keep their name by prefixing them with `!`. The mangled names contain underscores which are invalid for normal identifiers and the current module name to avoid clashes with other macro generated identifiers.

3.5 Type System

We chose the Haskell type system to base Shem's type system on because we believe that it is the most powerful type system with sound type inference and we want the type system to be a tool which helps the programmer, not a nuisance which requires them to repeat obvious type annotations. We believe that the increased complexity of type errors can be alleviated through better error messaging and the continuous feedback loop of our IDE.

Shem's type system is an extension of the Haskell98 type system[93–95]. We extend this system with multi-parameter type classes and implicit functional dependencies, which provide the required flexibility for expressing our collections API. We will first describe the motivation for these extensions and how they work in practice. We will then argue that these extensions preserve the soundness and completeness of type inference.

3.5.1 Multi-parameter Type Classes

The collections API requires type classes which abstract type constructors. We would like to have methods that can work both on lists and sets, for example, as the `add` method of `Collection` class does in Java. Haskell98 already provides a mechanism for dealing with type constructors in type classes: constructor classes. Constructor classes have as an argument the type constructor itself. Consider:

```
class Collection c where
  elem :: e -> c e -> Bool
  insert :: e -> c e -> c e
```

Here `c` stands for a collection type constructor and `e` for the type of its elements. One possible instance of such class could be:

```
instance Collection List where
  -- elem :: e -> List e -> Bool
  elem = ...
  -- insert :: e -> List e -> List e
  insert = ...
```

There are two reasons constructor classes will not provide enough flexibility. Firstly, because `e` doesn't appear as a class parameter we cannot impose any additional constraints on it in the instances (instances cannot change the signatures of methods). This means we cannot implement Sets and similar (which constrain the types of their elements). Multi-parameter type classes remedy this:

```
class Collection c e where
  elem :: e -> c e -> Bool
  insert :: e -> c e -> c e
```

The only difference is that now `e` became an argument of the class. Instances can now declare constraints, for example:

```
instance Ord a => Collection TreeSet a where ...
```

The second reason is that constructor classes only accept type constructors with one argument. This means we can use neither concrete types such as `BitSet` or `String`, nor type constructors with multiple arguments, such as `HashMap k v`. But multi-parameter type classes alone don't solve this problem. We additionally need functional dependencies. In the above example, the `c e` in the method type signatures signals that `c` stands for a type constructor, and therefore `a` is constrained (via the method types) to be its argument. Functional dependencies allow us to express similar relationship and more. Our running example becomes:

```
class Collection ce e | ce ~> e where
  elem :: e -> ce -> Bool
  insert :: e -> ce -> ce
```

`ce` now stands for a complete type of some *collection* with some *elements*. This allows us to declare instances for type constants and type constructors with more arguments.

```
instance Collection String Char where ...
instance Collection (Map k a) a where ...
```

Note that without functional dependencies the types resulting from use of the methods in our example would be imprecise:

```
addToString x = insert x "Hello"
```

would have the type `(Collection String a) => a -> String`, although passing in anything but a `Char` would result in an `instance not found` error. With functional dependencies the type is simply `Char -> String`. Even worse, methods such as `empty :: ce` would have a completely ambiguous type `(Collection ce e) => ce`; the `e` in the constraint does not occur inside of the type.

3.5.2 Flexible Instances and Contexts

Multi-parameter classes with functional dependencies, as implemented by the Haskell extensions `MultiParamTypeClasses` and `FunctionalDependencies`, still do not provide the expressive power which we require for our collections API. In Haskell, there is a restriction imposed on the signatures of instances which requires all types of the instance to be of the form $T\,a_1...a_n$, where a_i are distinct type variables;[96] note that there can be zero type variables, in which case T is a type constant (such as `Char`). This restriction prevents us from having instances such as:

```
instance Collection (List a) a where...
```

because the second type `a` does not use a type constructor.[18]. In Haskell this instance signature is allowed with the use of the `FlexibleInstances` compiler extension. This extension allows for the use of arbitrary types, therefore the following would be allowed:

```
class C a where
  m :: a -> Bool

instance C (Maybe Char) where ..
```

[18]Without multi-parameter classes it makes no sense for an instance type to be just a type variable, since it would capture all possible instance types of that class, unless overlapping instances are allowed[97].

But lifting this restriction has an impact on class constraints as well; for example, we could write a valid function `f x = m (Just x)` with a most general type:

```
(C (Maybe a)) => a -> Bool
```

Notice the class constraint, which can be neither reduced to a normalized form $C\,v_1...v_n$ nor rejected (especially since there is an instance which partially satisfies it, the `C (Maybe Char)` instance). In Haskell we can enable another compiler extension[19], `FlexibleContexts`, which lifts the restrictions on the normalization of class constraints during type generalization. This has an impact on the runtime performance - because we don't reduce contexts, we might have to pass in compounded dictionaries to functions with constraints. The lack of enforced context reduction (context reduction is now only forced by explicit types in the program), can in turn lead to failures in identifying type errors.[98] Most importantly, we lose the property that programs which require certain instances are rejected if those instances are not present at the time of compiling those programs.

This alone is enough for us to reject this path to the desired flexibility in our type system. To find a way in which we could use functional dependencies but still have a fairly simple type inference mechanism, we considered imposing a different set of restrictions on the format of classes, instances and class constraints.

3.5.3 Implicit Functional Dependency

The solution we chose has two main components:

1. All multi-parameter type classes have a single, implicit, functional dependency. For classes $TC\,a_1...a_n$ the implied functional dependency is $a_1 \rightarrow a_2...a_n$.
2. We retain the original Haskell98 restriction on the first argument in the instance type. That is, for instance type $TC\,t_1...t_n$, t_1 must have the form $T\,a_1...a_n$, where a_i are distinct. Additionally, any free variables in $t_2...t_n$ must be present in t_1.

There are several advantages to our solution:

1. We get some benefits of functional dependencies without the programmer having to understand the concept of functional dependencies. Although in this report we talk about functional dependencies, the resulting rules are simple enough to be explained in their own terms.

[19]Haskell requires this extension to be enabled only since version 7.10, before that it was willing to infer constraints not in a normal form.

2. We retain original context reduction which reduces all constraints to a normal form. This means that we do not pass unnecessary dictionaries to functions and that the boundary between valid and invalid programs is clear.

3. The increase in likelihood of type ambiguity introduced by multi-parameter type classes is avoided. Even with functional dependencies available, the programmer is still free to make highly ambiguous type classes such as `Coerce a b` where both argument types need to be known for successful type inference. Our system prohibits them.

To show that this system preserves sound and complete type inference, we return to the conditions defined in Section 2.2.5.3. Although we do not satisfy the condition imposed by Haskell on instance types, we satisfy both the Basic Conditions and the Functional Dependencies Conditions, and therefore preserve sound and complete type inference[48].

3.6 Prelude and Mathematics

Although not part of the core language, a language prelude, or built-in standard library, has a large impact on how programs written in the language actually look. Apart from collections, described in the next section, we discuss here our choices for functions included in Shem's Prelude, specifically how we deal with logical and mathematical operations.

When compared to Haskell, Shem's prelude differs in several ways:

- Basic algebraic operators: `+, -, *, /, mod, div` are not part of a type class hierarchy, along with numerical literals.
- Operators which are infix in Haskell and other languages take the right hand operand first. This means that `(- 1 5)` gives 4, not -4.
- Negation operator is `~` and negative numerical literals are `-1`, `-5` etc, but `-` cannot be prefixed to other literals or forms.

The main guiding principle in designing the prelude has been consistency and avoiding type ambiguity. Additionally, we had to take into consideration the performance impact of type classes. In designing the basic algebra to use a single type, `Num`, we have traded type safety and some part of expressiveness for programmer convenience, type unambiguity and performance. Additionally, this greatly simplifies the implementation of our type system, since we avoid the need for defaulting[15], which is far from trivial to implement[46], and avoids the problems described in Section 2.2.5.2. We end up with an algebraic system which will be familiar to

ECMAScript[20] programmers. The main practical potential for errors comes from the lack of integers, which are used often, for example for indexing, and programmers will have to make sure the numbers they pass in are actually integers. From our experience programming in ECMAScript languages, this error rarely occurs.

On the other hand, ambiguity and surprising results of defaulting in Haskell's approach to algebra arise more often than one would like. Yet there is good argument for having a full-blown category theory based hierarchy for algebraic operations, and by not creating one for the built-in library we leave the freedom to design one to the users of our language. Additionally, for dealing with common numbers and vectors, a simplified hierarchy can be devised, substituting `plus, minus, times, over` for corresponding operators and possibly using a `from-num` function to convert numerical literals to the desired class of values.

The reason for swapping the order of operands of infix operators is that in this way we achieve greater consistency with other functions in the prelude, which all obey the rule that the object (usually a collection) of a functions comes as the last parameter, simplifying currying. This means that curried calls like `(- 1)` and `(/ 2)` have the intuitive semantics, "subtract one" and "divide by 2". This change might be confusing to programmers used to LISP, but we believe that knowing the rules of our language, `(- 1 5)` is easier to understand than LISP's `(- 1 2 3 4)`[21]. The same rules apply to comparison operators, therefore `(> 2)` check whether the next argument is "bigger than 2" and `(> 2 3)` is true.

Comparison and equality are implemented using type classes, as in Haskell, since apart from a small performance hit they have no negatives but allow for the implementation of collections based on these constraints for arbitrary element types. Logical operators don't use symbols, therefore `and` and `and-map` are equivalent to Haskell's `&&` and `and` functions respectively. The rest of the prelude comprises standard basic functions, such as `even?`, `const`, `curry` etc. and our collections library.

3.7 Collections

Designing a good collections library is no easy task, but we believe such a library is fundamental for a language to become a useful programming tool. Without well designed collections programmers are constantly forced to re-implement known data structures and algorithms, with a great possibility for functional and performance errors. We believe that a good collections API is one of the good parts of Java and one of the main reasons the language has become so popular.

[20] JavaScript, ActionScript etc.
[21] Equals −8.

It was essential for us that our API was polymorphic in the types of collections, for two reasons:

1. Sharing method names across collection types leads to a simpler API, which is easier to learn and use, and leads to simpler code which is easier to maintain, because collections implementing the same type classes can be interchanged.

2. Functions can be written in terms of abstract type classes instead of concrete types, allowing greater code reuse.

There are two possible ways of using a non-polymorphic collections library in a functional language: either we have to prefix clashing names with module names or make sure that names in the library do not clash. The latter way is extremely cumbersome and not employed in practice, as the names would soon mimic the module prefixing. How would we call the `map` function on arrays, linked lists, hash sets, tree sets etc.? Therefore the former method is used. Consider this hypothetical[22] example in Elm[23]:

```
Set.foldl Array.append Array.empty
  (Set.map (\x -> Array.fromList [x ^ 2, 1]) (Set.fromList [-2, 1, 2, 3]))
```

The program takes a list of numbers, squares each and produces an array of the squares followed by a one. Elm would show the result as `Array.fromList [4,1,1,1,9,1]`. We believe the same code is much more readable using highly polymorphic functions, in Shem:

```
(concat-map (fn [x] {(^ 2 x) 1}) (Set -2 1 2 3))
```

Producing the equivalent result `{4 1 1 1 9 1}`. There are two steps to the translation from Elm to Shem. Firstly, functions such as `append` (`join` in Shem), `fold`, `empty` and `map` are methods of corresponding type classes and can be used on any data structure implementing them, and, secondly, this polymorphism allows us to provide a generic `concat-map` function, which concatenates any foldable collection of any joinable collections (details follow).

Some argue against making such general API because they expose inefficient operations on the underlying data structures. We do not believe that constraining the API of a data structure to only the operations which it implements efficiently leads to a good design. First of all, many operations which are inefficient for large number of elements work fine for small numbers and programmers often have to deal with collections of single-digit sizes. Secondly, there are situations where a non-efficient operation performed once is more efficient than the conversion

[22]The example would be a valid Elm program if Elm supported sets of arrays, which it does not.
[23]The same example in Haskell would use the module Vector and its function (++) instead of append.

to a more suitable data structure; for example, searching linearly through a list is faster than converting the list to a set (generally loglinear[99]) and then performing a constant lookup. We therefore leave the complexity analysis and choice of appropriate data structure to the programmer, while providing them with the most flexible API possible.

3.7.1 Collections Class Hierarchy

We will now discuss the collections class hierarchy currently implemented in Shem's prelude. It is clear to us that this hierarchy is not perfect and that further improvements and additions will have to be made, especially taking full account of performance. Nevertheless the resulting API is already a big improvement, in our opinion, over the state of collections in PureScript, Elm or Haskell.

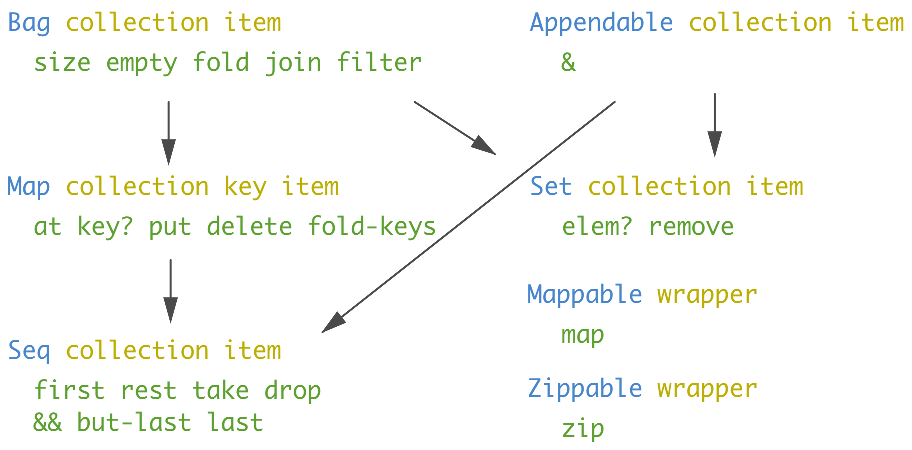


FIGURE 3.4: The type classes forming Shem's collections API. Arrows point from super class constraints towards class types.

The class hierarchy is shown in Figure 3.4. The two top type classes are `Bag` and `Appendable`. The reason they are separate is that we wanted sequences to be maps, but we cannot add values to maps without their keys. This is actually the central issue of our design - whether to include maps in the hierarchy along with sequences or to keep them separate, as in Java's collections(Figure 3.5). We gain a lot by using this design - all the methods which work over generic maps work over all sequences (Lists, Arrays, Strings) with keys being numbers. `Map` therefore extends `Bag` and `Seq` in turn extends both `Map` and `Appendable`. `Set` is similar to `Seq` in that it extends both `Bag` and `Appendable`. By having a simple interface for the `Set` class we can allow all collections to implement it, provided they have a means of comparing their elements (usually via an `Eq` constraint).

There are also separate type classes for mapping and zipping, `Mappable` and `Zippable` (zip in Shem is equivalent to Haskell's zipWith). They are not multi-parameter, because their methods are fundamentally different to those of the classes above: they change the type of the element. Such method types cannot be expressed with multi-parameter type classes, because they only

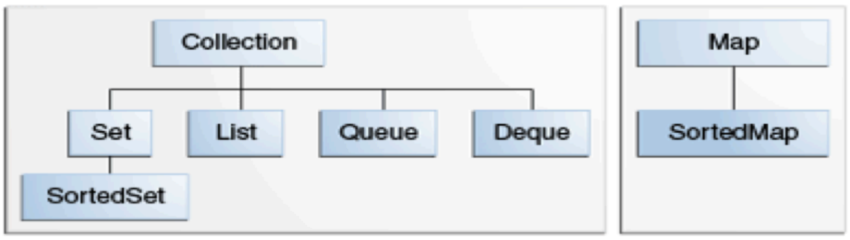


FIGURE 3.5: Core collection interfaces from the Java Development Kit[100].

have access to the concrete type, say `ce`, which represent both the collection type and the type of its elements. Additionally, since `Mappable` corresponds to Haskell's `Functor` we extend it further with other classes independent of the collections (more on them in Section 3.9).

3.7.2 Collections Data Types

Shem's built-in collections use the Immutable.js library[101], which provides a basic set of immutable, persistent[35] data structures. All method implementations are strict at the moment, although laziness could be used to improve performance in the same way it is used in Clojure, which inspired the use of persistent data structures. The collections which are available using the prelude are:

- `Array`[24], which provides efficient lookup and addition/removal on both ends. Ex: `{1 2 3}`.
- `List`[25], which is a singly-linked list with usual properties. Ex: `(List 1 2 3)`.
- `Set`, persistent HashSet with efficient addition/removal. Ex: `(Set 1 2 3)`.
- `Map`, persistent HashMap with efficient addition/removal on keys. Ex: `(Map \A 1 \B 2)` or `{one: 1 two: 2}`.
- `String`, JavaScript's strings, which are natively immutable.

Each can be created with a corresponding macro, and arrays and maps with string keys have the aforementioned special syntax. These forms can be used for pattern matching as well. For example, we will spell out all the instances of the `Array` data type:

- `(Seq (Array item) item)`
- `(Set (Array item) item)` given `(Eq item)`
- `(Map (Array item) Num item)`

[24]List in Immutable.js v3.
[25]Stack in Immutable.js v3.

- `(Appendable (Array item) item)`
- `(Bag (Array item) item)`
- `(Mappable Array)`
- `(Zippable Array)`

The same instances are declared for Lists and with `item` being `Char` for `String`. The type constructor `Set` implements the `Set` type class and its super classes. `Map` implements the type class `Map` with generic `key`, `item` arguments and also implements `Set` and `Appendable`, in the following ways:

- `(Set (Map key item) item)` given `(Eq item)`
- `(Appendable (Map key item) [key item])`

We would normally insert items into a Map using the `put` method (f.e. `(put “a” 1 (Map))`) but it can be useful for higher order functions to use the more polymorphic method `&` (f.e. `(& [“a” 1] (Map))`). There are situations in which it is useful to treat maps as collections of pairs instead of collections of their values, in which case can we can wrap the map in a “view”, similarly to the approach taken in Haskell’s Data.Foldable package[102].

This concludes the description of the built-in collections in Shem. We present more examples for comparison between Shem and other languages in Chapter 6.

3.8 Modules and Namespacing

Shem has a simple module system inspired mainly by Node.js. Modules can be required[26] and functions, macros, types, and instances can be imported under original names or renamed[27]. Shem does not support importing all definitions automatically, which is Haskell’s default behavior, because this can cause unexpected name clashes when the list of definitions within a module changes. On the other hand, once a module is required, its functions and macros can be accessed via a namespaced reference (prefixed with the module name). There is currently no syntax for exporting, but we considered explicit hiding of names, either via a name affix or wrapping macro.

The reason functional languages usually allow the import of all definitions of a module, as opposed to statically typed OO languages, is that in the former there are many more functions

[26]Prelude is currently required automatically and so are all its definitions.
[27]Renaming has not been implemented.

than classes in the latter, which must be accessible directly - there are no objects holding references to them. Therefore requiring the explicit import of every function can be a big burden on the programmer - that is, if they must type the imports themselves. Shem, on the other hand, relies on its IDE, which handles these imports automatically for the programmer, similarly to traditional IDEs in OOP languages.

For example, we could require a module HTML and import the function `animate` from its namespace, while accessing a different function via the module name directly:

```
[animate] (req HTML)

args HTML.cli-arguments!
```

This makes code more understandable[103], because the origin of every name is explicit, provided we know the contents of the whole module. Here again it is better to rely on appropriate tools, like our IDE, which easily surfaces the origin of every name.

3.8.1 Module Implementation

The module system in Shem fulfills two purposes: it allows for separate compilation of the modules and also for importing code across projects. For the second purpose it must compile modules to isolated JavaScript objects that can be safely combined together. In Node.js the runtime takes cares of isolating and loading modules, provided the correct `require` and `exports` declarations are used. In the browser there are several different ways of joining modules: static ones like Browserify[104], which concatenate all of the JavaScript sources into one file or dynamic ones like RequireJS[105], which load modules dynamically as required[28]. In any case, since Shem does not dictate the format of files, it also does not dictate the format of modules, and the compiled JavaScript can be generated in several forms, including a universal module wrapper which supports both Node.js and the browser tools.

3.9 JavaScript Interop

We discussed in Section 3.2.2 the reasons for designing a lightweight implementation on top of JavaScript. Similarly to Clojure, we also want to provide a simple interoperability between Shem and JavaScript code (sometimes called JavaScript FFI), so that Shem programmers can easily take advantage of the existing ecosystem of JavaScript projects. We provide a way to import JavaScript modules, call methods on objects and access JavaScript's built-ins:

[28]This is a gross oversimplification of both systems, but it is often how they are identified by the community.

- The current global object (whether it is the browser's `window` or Node's `global`) can be referenced with the `(global)` macro call.
- JavaScripts built-ins are accessed through namespaced references starting with `global`: `global.parseInt`, `global.Math.PI`.
- Method calls use operators prefixed with `.`, first argument is the JavaScript object containing the method and subsequent arguments are passed to the method: `(.method x y z)` calls `method` on `x` passing it arguments `y` and `z`.
- Fields are accessed using operator with `.-` prefix, f.e. `(.-field x)`.[29]

Any of these methods produce values of the built-in `Js` type. These values can be passed around, have methods called on them and fields accessed, but they cannot be used as arguments to typed functions. It is therefore good practice to explicitly type them as soon as possible:

```
DomElement (data)

integer-value (fn [element]
  (: (Fn DomElement Num))
  (parse-int (:: String (.-value element))))
```

The function `integer-value` returns the DOM value of a DOM element such as `input` converted from a string representation to an integer. The fact that we access the field `value` on the parameter `element` suggest it is itself a `Js` value, which was tagged with a dummy type `DomElement`, (again, using the `::` macro).

It is even better to use a function which checks that the `Js` value passed to it corresponds to the required type, although the macro `::` does not perform this check to not impair performance and because there is no way to infer this check for user defined types. It would be possible to implement a macro which checks built-in types and algebraic data types, though.

Compare our approach with the following systems from other Haskell-like languages compiling to JavaScript:

- **PureScript** requires[30] that JavaScript code be provided in a separate file named as the one which tries to call it, with matching names. For example:

  ```
  -- Test.purs
  foreign import intAdd :: Int -> Int -> Int
  ```

[29]We cannot have a single accessor pattern like Clojure, because unlike in Java every method of a JavaScript object can also be treated as a field with a functional value.

[30]From version 7[106]

```
// Test.js
// module Test
exports.intAdd = function (x) {
  return function (y) {
    /* jshint bitwise: false */
    return x + y;
  };
};
```

- **Elm** does not allow simply calling a JavaScript function, because the whole computation is controlled by its runtime. Values can be sent from and to the runtime and functions written in Elm can be called from JavaScript.
- **Fay** has actually the simplest FFI to JavaScript of the three languages, which can be used at both the definition or expression level:

  ```
  unixTime :: Fay Int
  unixTime = ffi "new Date().getTime()"

  add3 :: Int -> Int -> Int -> Int
  add3 x y z = x + (ffi "%1 + %2" :: Int -> Int -> Int)
  ```

  The obvious disadvantage of this approach compared to ours is that any string can be passed to `ffi`.

None of these languages allow `Js` values to be passed around. We believe that Shem's FFI is a simpler and more flexible mechanism than any of these.

3.10 Imperative Computation

Shem, being a strict language, has an easier way to supporting imperative computation than lazy languages like Haskell. Mainly for simplicity and because it is a well understood approach, we included Haskell style monad-based IO in Shem's prelude. Although not strictly required for the implementation, we added thunk-like zero-arity functions to Shem. A simplified IO can then be implemented using an algebraic data type wrapping a zero-arity function, which gets evaluated once the action is run, as shown in Figure 3.6. We have already shown the `do` macro implementing the usual syntactic sugar in Section 3.4. The type classes hierarchy implementing monads in Shem consists of these classes:

- `Mappable` with method `map`, corresponding to Functor in Haskell.
- `Liftable` with methods `lift` and `apply`, corresponding to Applicative with `pure` and `<*>`.
- `Chainable` with method `chain`, corresponding to Monad with and `>>=` from Haskell's prelude.

The latter classes extending the former. There are several improvements over Haskell's implementation: Shem does not inherit the historical error of Monad subsuming the Applicative interface[31][82], it provides more intuitive names that do not originate from category theory and does not use hard-to-remember and read infix operators[103]. The implementation of the `chain` method is then straightforward:

```
Io (data [a] Io [content: (Fn a)])

run-io (fn [action] ((Io-content action)))

chain (fn [action through]
  (: (Fn (Io a) (Fn a (Io b)) (Io b)))
  (Io (fn [] (run-io (through (run-io action))))))
```

FIGURE 3.6: The implementation of IO in Shem's Prelude using zero-arity functions.

The `run-io` function first unwraps the `Io` data type and then evaluates the enclosed zero-arity function. The `chain` function, which would be the implementation of the `chain` method for the instance of `Chainable` of `Io`, returns a new action which when evaluated first evaluates the passed-in `action`, then passes its result to the supplied `through` function and evaluates its result, which must be an action itself. This enforces the order of evaluation of the functions stored inside actions.

The added benefit of having the monadic mechanism in place is that we can implement different kinds of IO, for example one to deal with asynchronous computations[107].

3.11 Summary

In this chapter we introduced Shem, a new statically typed strict functional language. Its main design goals are expressiveness and unambiguity. It is a language tailored for the use within its custom designed IDE. It compiles to JavaScript, itself a high level, portable and extensible platform. Its type system and basic library are inspired by standard Haskell, but they are a significant improvement in terms of both simplicity in the way they deal with numbers and expressive power thanks to highly polymorphic collections, making Shem a much more usable language from the get-go.

[31]In Haskell, Monad extends Functor directly and does not extend Applicative.

Chapter 4

IDE Design

Many of the design decisions taken with Shem have been influenced by us relying on its IDE, Golem, to provide the corresponding features. Yet there is a more general influence of the IDE. Without a good IDE aspects of the language such as an S-Expression-based syntax and static typing would have too many drawbacks for us to base an ideal programming tool on them. Although we could have hoped that a suitable IDE was developed later for such a language its design could not draw lessons from the concurrent design of the IDE and we would not be able to fully demonstrate and evaluate the strengths of these aspects.

We drew a lot of inspiration from the work of Bret Victor (Section 2.4.4) and perhaps the single most important principle for the design of our IDE is providing an interactive experience. What we mean by interactive will become clearer as we discuss the various aspects of our IDE, but for now it will suffice to say that we do not consider a REPL-style interface interactive. Neither live reloading - recompiling and reloading programs on save - does itself provide the kind of programming experience we are aiming at.

4.1 Supporting the Programmer

IDEs are some of the largest pieces of software and it is not feasible for us to implement every little feature of every available IDE. Indeed, we aimed for a simple IDE that would showcase our fundamentally different approach to programming. Yet the IDE should provide enough features that it can be used for real work and the rest of the features, such as version control, performance profiling and package management, should be available via existing external tools. The crucial aspects of the environment for us are therefore how source code is changed and navigated, the testing and interaction with running programs, good error reporting and debugging facilities. These are all areas where we innovated beyond the industry standard, or at least beyond tools available for functional programming.

What is more, we aimed at making this experience instantly available. We implemented both the compiler and the IDE in CoffeeScript without any reliance on server side technology. This is important both to support interactive programming and because it allows us to distribute our IDE in the form of a static web page. This in-browser version is fully capable of compiling and running Shem programs which do not require the desktop environment. Today's client-side web applications show us that the limits of programming for the browser are stretching further every day. APIs which are not available in the browser can be easily simulated. We do not expect programmers to work on real projects in the browser, but having this environment readily available is of great value for building a community, as we do not require the programmers to go through any installation process to try it out. A desktop version of the IDE is available which supports editing real files and running code against Node.js APIs.

It is important to mention that an ideal IDE should be highly user-extensible but we primarily focused on designing a great out-of-the-box experience. Certainly more work would have to be done in Golem's implementation to provide the kind of customization we would like to have, but design-wise this area is fairly well researched[59].

4.2 Input

We begin discussing the design of our IDE from the absolute basics: how should user code be represented? Some[68, 108–110] have argued that we should employ a richer representation for code. There are applications where this approach has been successfully used, such as graphical post-processing. Flowhub[111] is an early example of taking that approach to general programming. We believe that these graphical approaches are not appropriate for professional programming for two reasons. The most important one being the *input* hardware that is currently used by programmers. Keyboard is the fastest input mechanism we have to this day. Faster than a mouse, faster (and certainly more precise) than a touchscreen. Easily faster than speech. Until we have a good mind reading equipment, keyboards will stay the fastest input devices and with them, text will stay the natural medium of communication, including communication between the programmer and the computer.

There is a possibility of using text for input but choosing different visual representation for output. There are certainly ways in which richer representation could be made easily editable by keyboard[1]. Yet this approach abandons the plethora of tools based on text which already exist, which means that they would have to be all reimplemented: storage, copying, versioning. Microsoft Word[112] is the prime example of this approach and everything that can go wrong with it. Smalltalk is in a way also an example of this, as it creates an isolated ecosystem within its editing environment. Most importantly for us, there is an opaque layer between the actual

[1]Aurora, although heavily relying on mouse input, would be a good base to start.

source and its representation and we believe this is not a feasible solution for programmers who often have to dig deep into concrete implementations.

One of the biggest problems with all rich graphical representations is that they are necessarily richer and hence more verbose then their code equivalent. Therefore they usually work well on small examples but do not allow the programmer to quickly skim large amounts of code, which is often crucial.

A simpler approach would be to treat textual source code as markup and allow a view on top of it. This would arguably be even worse, as programmers would have to deal with two different representations of their programs. Shem strives to present code in line with the WISIWIG user experience (UX) philosophy: “What you see is what you get”. There should be no barrier between the input and the output, as this would almost certainly slow the programmer down. Additionally, this allows Golem to be built on top of familiar text-editing concepts. After all, there are already two representation layers of the programmer’s code, Shem code and the compiled JavaScript, and we should not add another one.

4.3 Modification of Source Code

It is often argued that the majority of the time programmers are not modifying code: they are trying to understand existing code, thinking of a solution, sketching solutions on paper, or being forced to sit through “boring” meetings. It is certainly the case that there is much more to programming than writing code, but we feel that this argument does not justify negligence of how programmers interact with the code. Even if 10 or 20 percent of the time was spent writing and modifying the code (and it is certainly more for certain periods of the development cycle) it would be worthwhile to minimize it. More importantly, writing code is tedious and programmers have to waste either time or mental capacity on figuring out how to perform it. We want our IDE to provide the best possible editing experience. This is tightly connected with the input ergonomics. We must be able to perform all editing actions easily from the keyboard while leveraging the pointing device[2] for targeted actions. Any interface which does not support *and* take advantage of both keyboard and mouse is in our opinion not ergonomically optimal.

Golem’s editing model, although based on text, is fundamentally more powerful because it allows the programmer to manipulate the code at the level of its AST. This ties in with our decision to use S-Expression based syntax for our language. The difference is subtle, yet hugely empowering. Imagine we want to swap the order of elements in a list or arguments passed to a function, in a language with C-like syntax:

[2]Mouse/trackpad.

```
x = [a, b];
f(a, b);
```

There are several ways to achieve this. We can select the inside of the parentheses and simply retype the arguments. This is not a good solution if the arguments are longer names or even more complex expressions. We can cut the first argument, delete the comma and the space, move cursor after the second argument, type a comma and a space, paste the first argument. A lot of error-prone editing for such a simple change. In a modern text editor, such as Sublime Text, which supports multiple cursors, we can select the first argument, then select the second argument using a new cursor and invoke the built-in transpose command, which swaps the two selections. This is a much better solution then the previous ones but still has some drawbacks: the second selection cannot be done with keyboard and might be difficult if the argument is a complex expression.

We can argue that this operation is not very common, although this might be the result of how cumbersome it is to perform it. But let us imagine further that we want to perform this operation many times, on different arguments, while we observe the results of the change. In such a case we might come up with an IDE function which performs this operation. But such a function will be, at best, specific to comma separated lists. It will not work on a list of expression separated by semicolon. In the end to implement a general function to perform this operation we must consider every syntactical structure of our language but even then the function will not be as general as we would like it to be because the syntax of the language is not homogeneous.

In Golem, we simply shift the expression using a combination of a modifier key and an arrow key. There are no delimiters that have to be adjusted because Shem's syntax is made of S-Expression only. This operation is general: it works with a larger selection, it works on multiple selections (cursors) simultaneously, it works across forms[3]. This example shows the advantage of moving the editing from the text level to the AST level. More powerful editing commands can then be defined when we take into consideration the semantical value of the AST nodes.

4.3.1 Design Principles for Editing

There have been many attempts at bringing true structural editing[13] to the programming toolbox but they have all mostly failed, as witnessed by the fact that they are virtually non-existent today[113]. The reasons this mode of editing has not become popular is because it constraints the programmer too much and disrupts the natural flow[114] of working with source code. Although editing modes which try to simulate structural editing, like Vim's Paredit[115], exist, they only try to provide a few structure-like editing commands. This shows us that

[3]See the AST reference in Section 3.3.

the most important consideration we have to make when designing a more structured editing experience is to mirror as close as possible the text editing programmers are used to. This reinforces our decision to have a purely textual representation of our code.

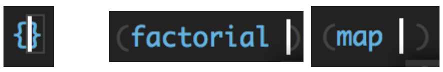
Although structural editing on the AST level has never become too popular, there is a precedent for providing a more powerful editing experience which differs significantly from standard text editing: Vim[116]. Vim along with Emacs[117] form the “holy” duo of traditional text editors despite the fact that Vim’s approach is quite unorthodox. Instead of working directly with the text of a buffer[4] in Vim the user issues commands which are executed on the buffer. Usually this system is described as having several “modes”, one for commands and one for inserting text, although this is slightly imprecise. It would be better to say that for some commands Vim allows the user to see the result or part of it directly as they are typing the command. We believe this is not a great approach to source code editing because it is not interactive, it is more alike to processing (and certainly Vim can be used as a very powerful text processing tool). But it does show that programmers are willing to put up with different text editing approaches and modal interaction patterns. Golem’s editing and selection model is in a way modal but we believe we avoid the sharp divide between modes experienced in Vim.

4.3.2 Representing Structure and Partial Programs

Although S-Expressions seem to have straightforward textual representation we spent considerable amount of time designing the details of how the AST nodes are manipulated and how new nodes can be inserted. Traditionally LISPs ignore the amount and kind of whitespace which separates expressions. This is not the case with Shem and the reason is that we needed to be able to tell partial from complete programs. Complete programs use only a single space or newline to separate terms within a form. Newlines may be followed by indentation, but the programmer does not have control over how big the indentation is or which characters are used for it. This frees the programmer from dealing with space-vs-tab and size-of-indentation wars. This way we strike balance between expressibility and automation, giving the programmer the freedom to decide when a line should break, which we believe cannot be decided mechanically.

Restricting whitespace this way allows us to represent partial programs simply as programs containing extraneous whitespace. We use the concept of “holes” present in other programming environments to discuss places within the AST where new nodes can be inserted. Holes are present both in complete and partial programs. From a textual perspective holes are cursor positions surrounded by delimiters or whitespace. Figure 4.1 shows three different holes, where the first one is in a complete program while the other two are part of partial programs.

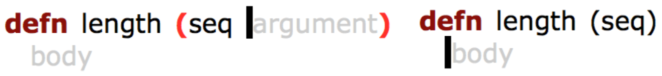
[4]File.



```
{}    (factorial )(map )
```

Figure 4.1: A hole between delimiters, a space and a delimiter, and two spaces.

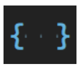
This might seem like an implementation detail but it actually is an extremely important decision. Usually[68, 69, 118, 119] some sort of added placeholder is used to represent holes, as shown in Figure 4.2.



```
defn length (seq argument)    defn length (seq)
  body                          body
```

Figure 4.2: A use of a placeholder for inserting a new argument[119]. After navigating away the placeholder disappears.

The problem with this solution is that such holes are not normally visible and easily selectable, they only work while the programmer is inserting new code. Such approach would break the strictly textual representation we require and would add friction to changing partial programs. Our approach on the other hand corresponds to traditional editing, the programmer simply places a cursor at the position where they want to insert text, we just limit which positions are valid. Holes represented this way are so natural that we highlight them only slightly[5]:



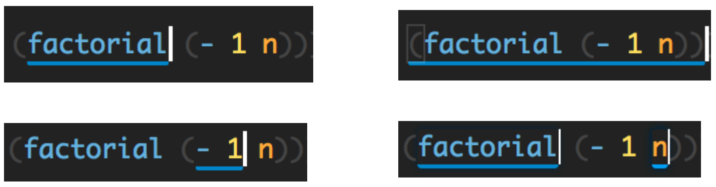
```
{ · }
```

We will now describe how selection, inserting and editing commands work in Golem.

4.3.3 Mouse Selection

As can be expected from an editor which is designed to manipulate the AST directly selection in Golem is by expression by default. Clicking on an atom, including its beginning, will select the whole atom. The same applies to forms, a big complicated expression can be selected simply by clicking on one of its two delimiters[6]. Having some expression selected, the user can select a range by holding `Shift` and clicking on another expression - all expressions between the two will be selected, possibly expanding the selection to parent forms so that all selected expressions are on the same level in the AST. Holding `Command`[7] and clicking on an expression triggers the multi-selection mode (multiple cursors). All basic and most advanced commands work on multi-selections, mostly by executing the same operation on each selection, although sometimes reconciliation is needed and some editing commands use the multiple selections in a specific way. Figure 4.3 shows how these selections look in Golem.

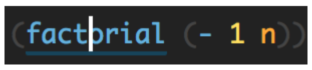
[5]There are three small dots between the parentheses corresponding to each hole position.
[6]As per Section 3.3 these are parentheses, brackets and braces.
[7]We will use the OS X shortcuts in the report, different sets of bindings are used on other operating systems.



```
(factorial (- 1 n))
```
```
(factorial (- 1 n))
```
```
(factorial (- 1 n))
```
```
(factorial (- 1 n))
```

FIGURE 4.3: Selecting atom, form, multiple expressions and multi-selection.

Clicking on a hole places the cursor on that position. Because clicking is by default used for selection, we use long press for placing cursor anywhere within an atom:



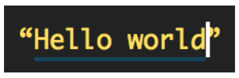
```
(factorial (- 1 n))
```

4.3.4 Keyboard Selection

The obvious approach to selection is traversal through the AST. The naive approach of walking up and down the levels and left and right within the levels of the tree does not work well in practice, because the "movement" does not map directly to the 2D textual representation of the tree. This why in Golem, the default movement using arrow keys happens on the list of atoms and holes. Left and right arrows are used to travel between previous and next atoms/holes, the leaves of the AST. The up and down arrows travel between atoms/holes on previous and next lines and similarly to a cursor in a modern text editor maintain horizontal position between lines. `Command` with arrows perform the naive tree traversal. Since there are two possible ways to travel down the tree - to the first or the last term within a form, we chose the first child to be the default and additionally pressing `Shift` moves the selection to the last child. This is a common pattern in our controls, where adding `Shift` key to the combination performs the sibling version of an action.

`Shift` with left/right arrow expand the selection to sibling and parent expressions, that is when there are no more sibling terms the parent expression is selected. This is another important design decision - actions should naturally extend so that the user does not feel constrained by the AST structure.

`Command-Down` combination when an atom is selected places the cursor at its end ready for adding more characters. Since Golem preserves the validity of the text representation of the AST, if the atom is itself delimited, that is if it is a string or a regular expression literal, Golem places the cursor automatically within the delimiters:



```
"Hello world"
```

4.3.5 Inserting Source Code

In line with our design principles, inserting works simply by typing or pasting text. Selections are overwritten, they always form a valid place to insert into the AST since they are in general a range of sibling expressions. Selecting a hole brings up auto-completion, which we discuss separately in Section 4.4. Inserting text within a delimited atom ensures that delimiters are properly escaped and that keys bound to actions just print their characters. Pressing `Space` creates a new sibling hole to insert the next expression, whether the user is currently editing an atom or selecting an expression, while `Shift-Space` creates a hole before the current selection. `Enter` and `Shift-Enter` work correspondingly.

Pressing `(`, `[` or `{` opens a new correspondingly delimited form while typing any closing delimiter selects the parent expression. Atom delimiters work similarly. These bindings mean that typing a Shem program character by character requires actually *exactly the same keys* as it would in a normal text editor! Golem therefore preserves the natural flow of inserting text while providing more advanced commands for manipulating existing code.

4.3.6 Editing Commands

Golem provides many useful commands for transforming code. We will list the most interesting ones. The more advanced ones take advantage of the information provided by the compiler which annotates the AST.

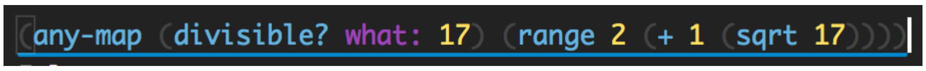
- `Command-(` *Wrap in a call* wraps the selection in parenthesized form, with a hole at its start for the new operator.
- `Ctrl-Space` *Flatten* flattens the selection into a single line, removing any new lines.
- `Ctrl-P` *Replace parent* replaces the parent of the current selection with the selection.
- `Alt-Left/Right` *Move* is the command described at the beginning of Section 4.3, shifting the selection between the leaves of the tree.
- `Alt-Up/Down` *Move line* similarly to text editors but preserves the AST. Useful for shifting definitions in definition lists.

The following commands are Shem-specific. Although some of them are present as refactoring tools in traditional IDEs by leveraging multi-selections Golem allows the programmer to perform them more fluidly.

- `Command-)` *Select definition* as extension of `)` selects the whole enclosing definition within a definition list.

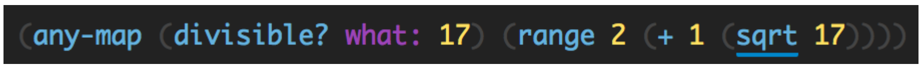
- `Ctrl-F` *Wrap in a lambda* replaces selection with a new function with it as its body.
- `Ctrl-M` *Wrap in a match* replaces selection with a call to the match macro such that the selection is being matched on.
- `Ctrl-Shift-M` *Wrap as the result of a match* replaces selection with a call to the match macro such that the selection is the result of the first branch.
- `Ctrl-A` *Add a parameter* removes selection and adds another cursor to the parameter list of the current function.
- `Ctrl-U` *Push to upper scope* moves the enclosing definition pair to the parent definition list, if there is one.
- `Ctrl-D` *Define* removes selection and adds a new cursor followed by the selection to the current definition list, effectively allowing the programmer to name the selected expression.
- `Ctrl-I` *Inline* either replaces a selected reference with its definition or removes the selected definition name and replaces all its references with its definition.

Many of these commands are important for the kind of interactive experience we want to provide, as well as allowing the programmer to easily abstract code, which was one of the principles outlined in Section 2.4.4. We think the following three commands are a particularly compelling example of the power of structural editing. Imagine we have constructed the following expression which computes whether 17 is not a prime number:



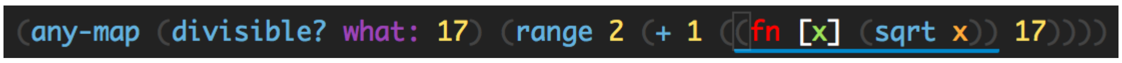
```
(any-map (divisible? what: 17) (range 2 (+ 1 (sqrt 17))))
```

The result is correct (how the programmer knows is described in section 4.6) so we would like to transform this expression into a function which takes an arbitrary number. We start of by selecting the operator which is being called with the hard-coded value:



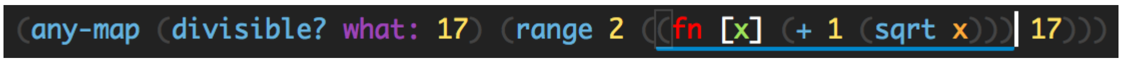
```
(any-map (divisible? what: 17) (range 2 (+ 1 (sqrt 17))))
```

Then we use `Ctrl-L` (abstract with a *lambda*) command which turns the enclosing call into a lambda application:



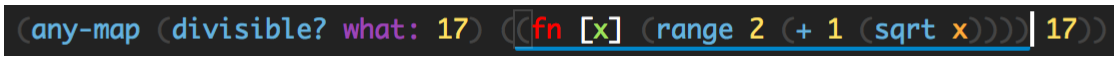
```
(any-map (divisible? what: 17) (range 2 (+ 1 ((fn [x] (sqrt x)) 17))))
```

The lambda is selected and we can use `Ctrl-O` to push it to the *outer* expression, replacing the parent call with the lambda application and wrapping the lambda's body with the call.

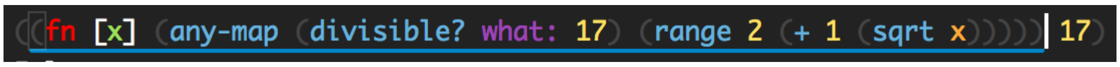


```
(any-map (divisible? what: 17) (range 2 ((fn [x] (+ 1 (sqrt x))) 17)))
```

Using this command twice more gives us almost the function we wanted:

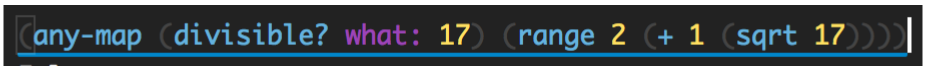


```
(any-map (divisible? what: 17) ((fn [x] (range 2 (+ 1 (sqrt x)))) 17))
```



```
((fn [x] (any-map (divisible? what: 17) (range 2 (+ 1 (sqrt x))))) 17)
```

We can now give the function a name using the define command and replace the 17 in its body with the name of the parameter. Or we can hit `Ctrl-I` which also inlines lambda applications and get back the original expression:

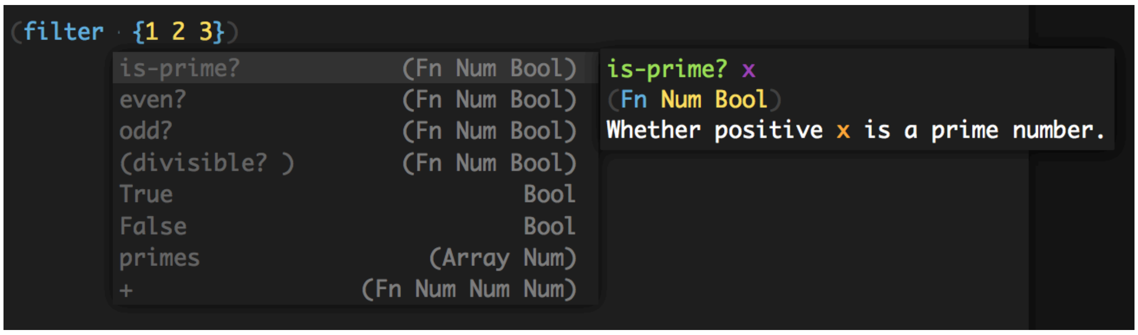


```
(any-map (divisible? what: 17) (range 2 (+ 1 (sqrt 17))))
```

The power of all these commands is difficult to demonstrate in a written report but after getting used to them we already found ourselves missing them in other editors.

4.4 Type-based Auto-Completion

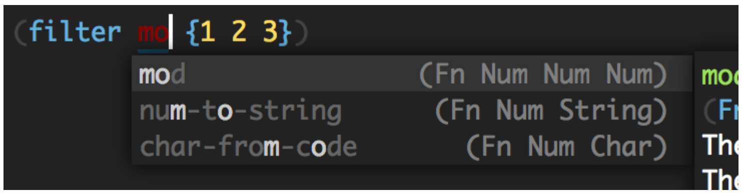
Traditional IDEs sometimes include a "smart auto-completion" mode, where the user can manually bring up a list of options where the top few are based on typing. Golem's auto-completion is automatically based on types and is much more advanced than these modes. First of all, the compiler accepts partial programs with holes, typing them as far as possible and providing this typing information via the AST to Golem. When the user creates or navigates to a hole, the auto-completion list automatically opens with suggestions sorted by definition proximity and how closely their types match.



```
(filter  {1 2 3})
  is-prime?          (Fn Num Bool)   is-prime? x
  even?              (Fn Num Bool)   (Fn Num Bool)
  odd?               (Fn Num Bool)   Whether positive x is a prime number.
  (divisible? )      (Fn Num Bool)
  True                        Bool
  False                       Bool
  primes               (Array Num)
  +              (Fn Num Num Num)
```

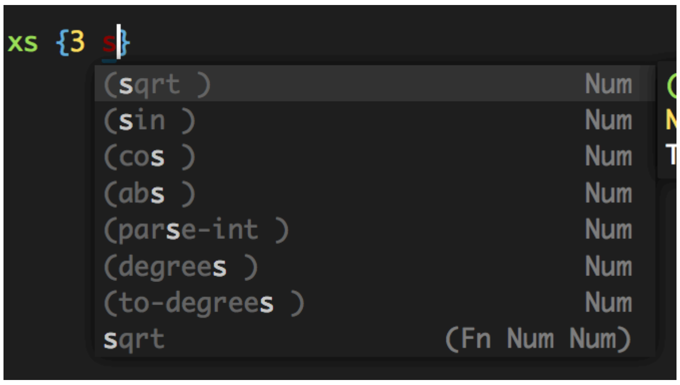
FIGURE 4.4: The auto-completion suggests is-prime? first because it is defined in the same module.

In the example of Figure 4.4 there are three functions which match the inferred type and the one defined in the same module is suggested as the first. Hitting the `Tab` key inserts this name. The programmer is of course free to start typing a name of a function which might not match the currently inferred type and the completion list will present them with all the matching existing definitions:



```
(filter mo {1 2 3})
        mod              (Fn Num Num Num)   mo
        num-to-string    (Fn Num String)    (F
        char-from-code   (Fn Num Char)      Th
                                            Th
```

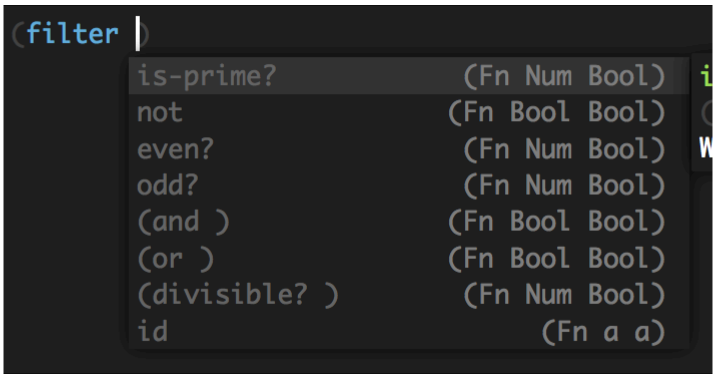
But Golem's auto-completion is even “smarter”. It suggests calls and curried calls which also precisely match the type of the hole:



```
xs {3 s}
      (sqrt )                 Num   (
      (sin )                  Num   N
      (cos )                  Num   T
      (abs )                  Num
      (parse-int )            Num
      (degrees )              Num
      (to-degrees )           Num
      sqrt          (Fn Num Num)
```

In this example we have a list of numbers, so the the hole must be have the `Num` type. The programmer typed `s` and is presented with a list of calls which have the correct result type. `Tab` inserts the call and places the cursor into the new hole for the missing argument of the inserted call. We already saw en example of a completion which is a curried call in Figure 4.4, where the function `divisible?` has type `(Fn Num Num Bool)`.

In the previous examples the type of the hole was not polymorphic - it only contained concrete types. But Golem's autocompletion works well on polymorphic types as well. In the following example the type of the filtered collection and its elements is unknown, Golem therefore suggests functions which would work if the elements were `Bool`s, such as `id`, which is itself polymorphic, as well as functions accepting first argument of some type and producing `Bool`, such as `even?`:



```
(filter |)
        is-prime?            (Fn Num Bool)
        not                  (Fn Bool Bool)
        even?                (Fn Num Bool)
        odd?                 (Fn Num Bool)
        (and )               (Fn Bool Bool)
        (or )                (Fn Bool Bool)
        (divisible? )        (Fn Num Bool)
        id                       (Fn a a)
```

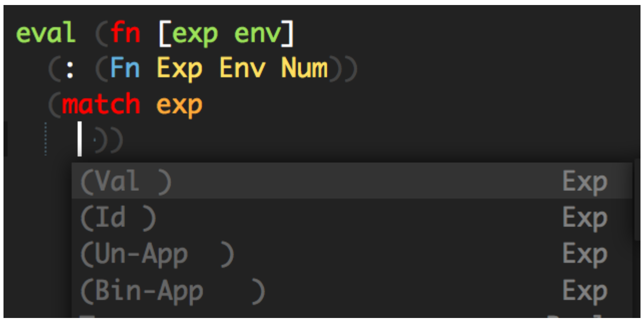
The definitions considered come from all the loaded modules in the project. Inserting a name which has not been imported yet adds it to the imports.

4.4.1 Auto-Completing Types and Patterns

Golem's auto-completion works on patterns and types as well. Consider these algebraic data definitions:

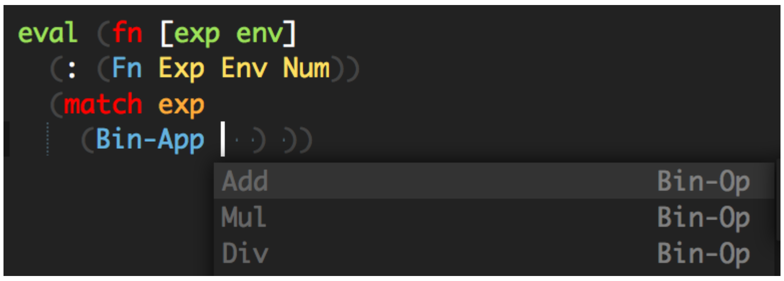
```
Un-Op (data Neg Sin Cos Log)
Bin-Op (data Add Mul Div)
Exp (data
  Val [value: Num]
  Id [name: String]
  Un-App [op: Un-Op exp: Exp]
  Bin-App [op: Bin-Op left: Exp right: Exp])
```

We would now like to write an `eval` function which evaluates `Exp`ressions using pattern matching. Since we have given the function an explicit type, Golem knows that `exp` argument has the type `Exp` and suggests the corresponding patterns, which are calls to constructors or constants:



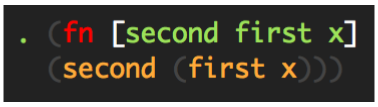
```
eval (fn [exp env]
  (: (Fn Exp Env Num))
  (match exp
    |))
     (Val )                          Exp
     (Id )                           Exp
     (Un-App  )                      Exp
     (Bin-App   )                    Exp
```

After hitting `Tab` we immediately get a completion list for the next hole, which is now the operator of the binary application:

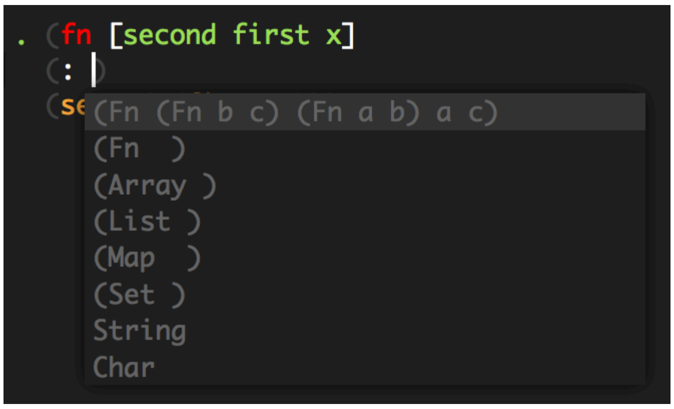


```
eval (fn [exp env]
  (: (Fn Exp Env Num))
  (match exp
    (Bin-App | ) ))
              Add                    Bin-Op
              Mul                    Bin-Op
              Div                    Bin-Op
```

Golem also auto-completes types. An inferred type of a function can be inserted via auto-completion:



```
. (fn [second first x]
  (second (first x)))
```



```
. (fn [second first x]
  (: |)
  (se(Fn (Fn b c) (Fn a b) a c)
     (Fn  )
     (Array )
     (List )
     (Map  )
     (Set )
     String
     Char
```

As with normal values, calls to type constructors are suggested when a concrete type is required. Completion on types is also useful when the function definition has not been filled in yet, the same rules about sorting completions based on proximity of definition apply to types.

Auto-completion is one of the areas where the fact that we designed and developed our language and IDE together immensely helped to achieve the level of experience we wanted. We can have an arbitrary number of typed holes and let the programmer figure out how to fill them out. GHC has recently added typed holes as well, although the programmer needs to specify they want to get a warning by inserting `_` but in theory a similar interface could be constructed on top of this feature.

4.5 Codebase Navigation

As we have mentioned programmers spend a large amount of time trying to understand code created by either someone else or themselves in the past[8]. During all parts of the development cycle it is important that the programmer can easily surface information about any part of the code. We have already shown part of this surfacing in the last section. During auto-completion Golem displays properly the documentation for suggested functions. The same information can be shown simply by hovering over any reference.

We can jump between references of a name using the `Tab` (`Shift-Tab`) key and perform multi-selection using `Ctrl-S` (`Ctrl-Shift-S`). Golem has the advantage over modern text editors that it understands from the AST which atoms correspond to which definitions, so that we can multi-select references to a concrete definition without selecting occurrences of the same name coming from a different scope. `Ctrl-R`, for *rename*, selects all references and the name in the pattern of a definition. This again shows how simple syntax and powerful text editing features can be used instead of needing many specialized commands. Golem does not support the display of multiple panels/buffers, but definitions from other modules can be printed to the output area, detailed in the next section.

Another useful tool for understanding code from other modules is the inlining command, which also expands macros (supplementing the role of Racket's macro stepper). Consider the following threading macro and its application:

```
-> (syntax [..args]
  (match args
    {x} x
    {x f ..fs} (` -> (,f ,x) ,..fs)))

summed-evens (-> {1 2 3 5 4}
  (map (+ 1))
  (filter even?)
  sum)
```

[8]Often there is no difference between the two.

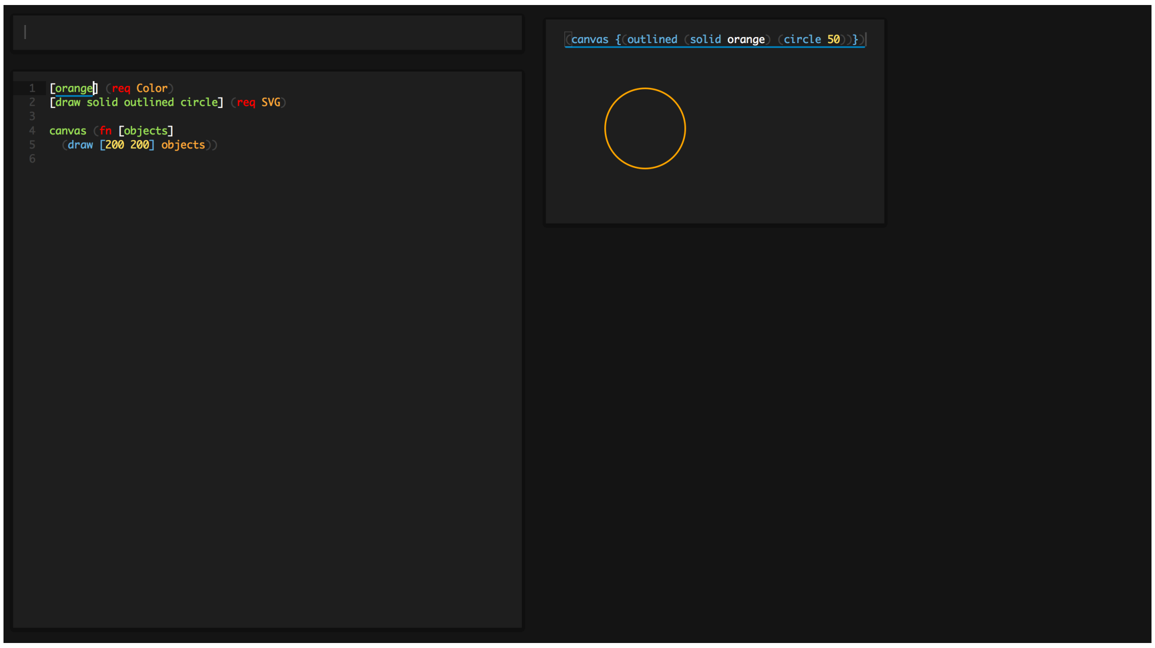
We can now inline the macro `->` as if it was a function, but instead of inserting a function Golem expands the macro:

```
summed+evens (-> ((map (+ 1)) {1 2 3 5 4})
  (filter even?)
  sum)
```

Inlining further we can see how the recursive definition of the macro works.

4.6 Interaction and Testing

In the previous sections we discussed how Golem makes editing and navigating code more fluid. In this section we describe the core of Golem's experience - working with running programs. The Figure 4.5 shows Golem's UI.



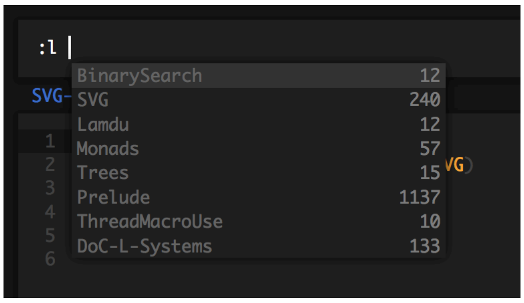
```
[orange] (req Color)
[draw solid outlined circle] (req SVG)

canvas (fn [objects]
  (draw [200 200] objects))

```

```
(canvas {(outlined (solid orange) (circle 50))})
```

FIGURE 4.5: The main interface of our IDE, with execution line, module editor and output area.

The interface is intentionally clean and consists of three areas. The majority of the left side is occupied by the module editor, where the programmer edits Shem source code of the currently open module. The right side is the output area, which displays executed programs, such as the program drawing a circle in the figure above, logged expressions, as well as output of the editor commands. The top left is occupied by the execution line. This input box is used for executing editor commands, such as switching modules or printing compiled JavaScript, and adding new programs to the output area.

Editor commands on the execution line are prefixed with `:`[9] and can support auto-completion. For example the `load` command, with the `l` shortcut, suggests modules based on the order in which they were last accessed:



```
:l
BinarySearch      12
SVG              240
Lamdu             12
Monads            57
Trees             15
Prelude         1137
ThreadMacroUse    10
DoC-L-Systems    133
```

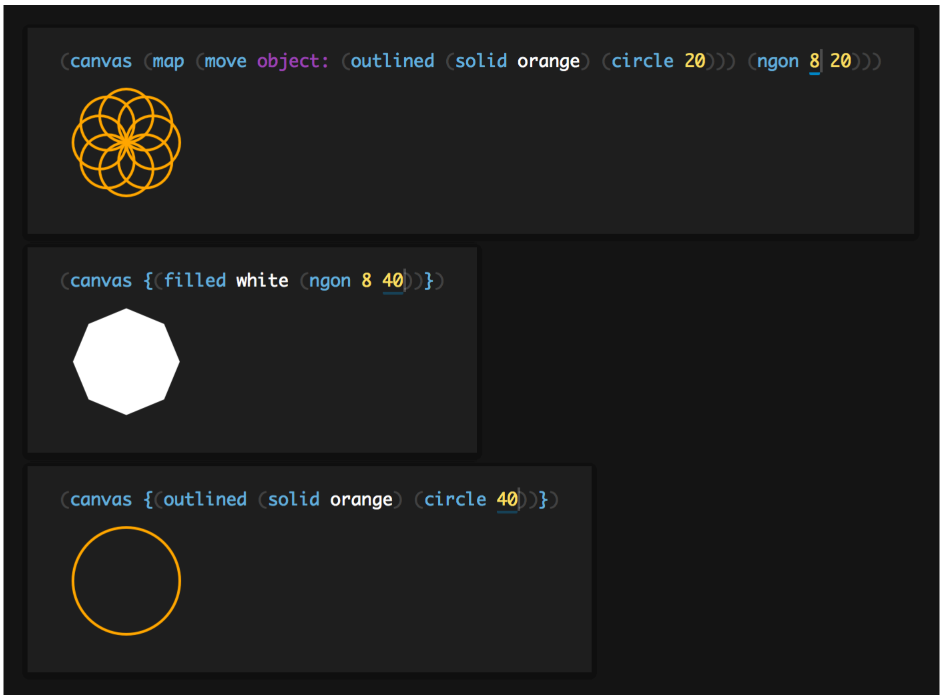
All other input is interpreted as a Shem program. When the programmer finishes inserting the code they hit `Enter` to evaluate the program and add it to the output area. The execution line preserves history similarly to traditional command line interfaces. After insertion the program stays editable within its output box and any change automatically causes the program to reevaluate. There can be many programs in the output area at the same time. Changing the code in the current module reevaluates all programs in the output area which belong to this module. The programs are attached to the module which was active when they were inserted, so it is possible to edit one module and observe changes to programs which are defined in the scope of a different module requiring this edited module. Golem takes care of tracking dependencies between modules and recompiling them as necessary, similarly to *online type checking* performed by Flow[33] and Hack[120].

This approach combines live-reloading with smart REPLs in a highly interactive way. The programmer can select, either using the keyboard or by clicking on them, any of the output boxes in the output area and move or remove them. This creates a sort of working area which the user has control over, but unlike in Smalltalk there are no windows or popups which need to be dragged and closed, just a stack of boxes. Because the programs run directly in the current browser `window` they can persist state between changes although this is left to the programs.

It is not an accident that Shem is pure-by-default. Pure programming fits nicely into the model where the programmer experiments directly with expressions, since programs are easily decomposable. It is both useful to see multiple results of the same function on different inputs and the results of different functions to construct more complex functions, as demonstrated in Figure 4.6, which extends the last example. The programmer combines the circle shape with the shape of the polygon by creating a circle at each corner of the polygon.

Graphical examples demonstrate well the capabilities of the output area and the power of auto-evaluation, but there is nothing inherently tying Golem to graphics, these capabilities come

[9]The Space key can be used to type this initial colon.



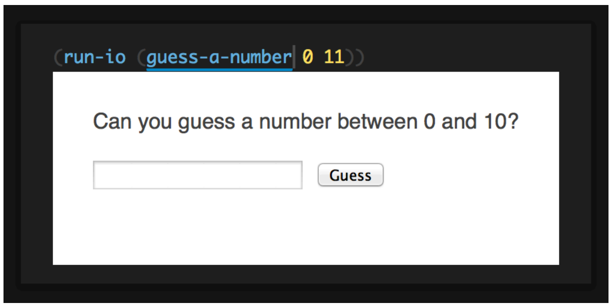
```
(canvas (map (move object: (outlined (solid orange) (circle 20))) (ngon 8 20)))
```

```
(canvas {(filled white (ngon 8 40))})
```

```
(canvas {(outlined (solid orange) (circle 40))})
```

Figure 4.6: Using visual output to create more complex programs out of simpler components.

simply from its design and can be utilized in constructing any kind of a program. The graphics API used in the example is implemented purely in Shem and works by producing SVG which Golem renders, as any other HTML string output. This means that programs can also create user interfaces within the output area, as shown in Figure 4.7.



```
(run-io (guess-a-number 0 11))
```

Can you guess a number between 0 and 10?

Figure 4.7: An interface built within Golem.

This is an example of a simple guessing game, running inside Shem. The player inserts a number, clicks on the button, the UI changes to a "thinking" state for half a second and then displays the result. The game is built purely in Shem and uses the standard browser APIs and callback style asynchronous computation.

Some commands are specifically adapted to work between an output box and the module editor. For example the `Ctrl-D` define command allows the programmer to define a selected expression from within an output box at the top level of the module. The same command used on an undefined reference defines it and if it was used as an operator it defines a function. `Ctrl-Shift-D`, typed define, will also insert an explicit type for the function. Analogously `Ctrl-T` can be used to insert a type of the selected expression at the current selection with the module editor. These commands are useful for type directed programming.

4.7 Debugging - Observing Code

Observing code at work was one of the essential requirements imposed by Bret Victor on interactive programming tools. LighTable took this idea and improved on the traditional watch mechanism by showing results of subexpressions on the same line these expressions end, instead of in a separate panel, as discussed in Section 2.4. Golem takes a different approach which allows for multiple states of a value to be displayed at the same time. Golem uses its output area to display logged expressions, which the programmer tagged using the `log` macro[10].

Consider the following definition of binary search in Figure 4.8. The code contains a non-obvious bug. We correctly find a number present in the list but for some numbers and lists the program loops and causes a stack overflow instead of returning `None`. This is a good example of a situation where decomposing the expression to smaller parts does not work well, because the function `search-in` is recursive. We suspect that the error might have something to do with the bounds we search, so we want to log the value of `min`, `max` and `half` every time the `search-in` function is called.

```
search (fn [n list]
  (search-in n 0 (size list) list)
  search-in (fn [n min max list]
    (if (>= max min)
      None
      (match (compare n middle)
        LT (search-in n min half list)
        GT (search-in n half max list)
        EQ (Some n)))
    middle (!! (at half list))
    half (log min max (+ min (div 2 (- min max))))))
```

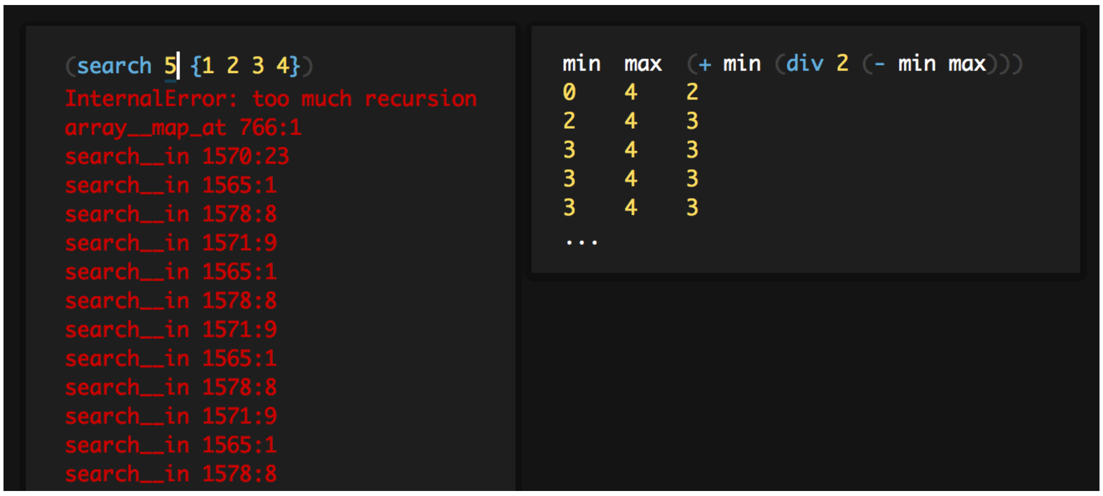
FIGURE 4.8: Binary search algorithm containing an error.

The `log` macro accepts an optional string literal as a label for the logging output box, then any number of arguments to print where the last one is the value which is returned. When we are done we can get rid of the log simply by replacing it with its last argument[11]. Golem's output is shown in Figure 4.9. The left box shows the result, which is the stack overflow error with printed stack trace coming from JavaScript. The second box is the logged output in the form of a table. The first row shows the printed expressions (the last expression is the definition of

[10]Our original design was to only highlight the logged expression to avoid having to change the source code, but we have not had the time to implement this. The advantage of the current trace[121]-like solution is that it is easily preserved across sessions.

[11]Via the common Ctrl-P command.

`half`), the next rows are values as they are being logged. Golem notices that values repeat and checks for a stack overflow before printing more rows.



```
(search 5 {1 2 3 4})
InternalError: too much recursion
array__map_at 766:1
search__in 1570:23
search__in 1565:1
search__in 1578:8
search__in 1571:9
search__in 1565:1
search__in 1578:8
search__in 1571:9
search__in 1565:1
search__in 1578:8
search__in 1571:9
search__in 1565:1
search__in 1578:8
```

```
min  max  (+ min (div 2 (- min max)))
0    4    2
2    4    3
3    4    3
3    4    3
3    4    3
...
```

FIGURE 4.9: Debugging an infinite loop in the binary search algorithm using logging.

We can see that once we get to the values `min` is 3 and `max` is 4 the bounds stop changing! We would expect that after the row `2 4 3`, where the last number in the list is checked, the next one should be `4 4 4` and the program should exit. Indeed, we forgot to start the search in the right half of the list from the next number after the number we just checked, the fix is easy, we add 1 to `half` in the `GT` branch. This way of printing values is especially powerful when we use multiple logs within the same program, which would be all mangled together if we were just printing to a stream such as standard output (stdout).

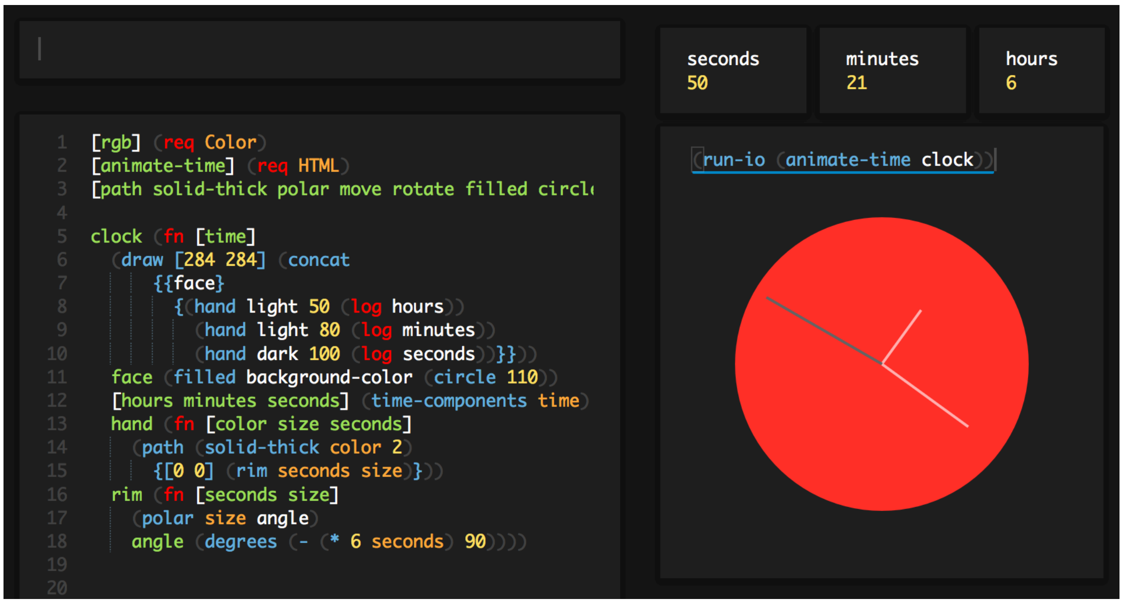
For debugging long running code, such as animations, it does not make sense to keep printing values. Golem therefore checks whether the stack of the current log is the same as the previous one, and if it is not, which is the case for any asynchronously issued code or code on a attached as listener to an event, the log is essentially restarted. An example of using this feature is shown in Figure 4.10.

Apart from this built-in debugging Golem also supports debugging using the browser tools. The `debug` macro inserts the JavaScript `debugger` directive into the compiled code and the programmer can then debug the compiled JavaScript. This is also useful when the program is run directly in a browser page and not inside Golem's environment. Similarly profiling is left to the existing JavaScript browser[12] tools.

4.8 Error reporting

There are three kinds of errors, or situations which make the program partial.

[12]In Node.js the Chrome developer tools can be used for profiling.



```
[rgb] (req Color)
[animate-time] (req HTML)
[path solid-thick polar move rotate filled circl

clock (fn [time]
  (draw [284 284] (concat
      {{face}
        {(hand light 50 (log hours))
          (hand light 80 (log minutes))
          (hand dark 100 (log seconds))}}))
  face (filled background-color (circle 110))
  [hours minutes seconds] (time-components time)
  hand (fn [color size seconds]
    (path (solid-thick color 2)
      {[0 0] (rim seconds size)}))
  rim (fn [seconds size]
    (polar size angle)
    angle (degrees (- (* 6 seconds) 90))))
```

```
seconds
50
```

```
minutes
21
```

```
hours
6
```

```
(run-io (animate-time clock))
```

FIGURE 4.10: The logged numbers in the output boxes in the top right update as the animation progresses.

1. The programmer can leave holes in code to be filled in later.
2. The code can be "malformed", which means that invalid arguments have been passed to built-in or user macro.
3. The program can contain type errors.

Shem always tries to compile the given source code as much as possible, it will only stop locally if one of the above errors occurs, and will still compile the rest of the program to JavaScript. This is important because otherwise almost any change within the IDE would cause the program to be momentarily invalid and we want programs which do not rely on this change to remain runnable. Golem treats these classes of errors differently. If a program in an output box contains any kind of error, it is not evaluated. Programs in output boxes are short and compile immediately, so we wanted to avoid running programs which would most likely lead to runtime errors. Contents of a module on the other hand can be both malformed or contain holes and the programmer can still run programs against them.

In either case, type errors are always displayed and programs containing type errors are never run. Unlike holes and malformed expressions, which are displayed inline, type errors are shown with their full description to guide the programmer and therefore Golem can show only one such error at a time. There is another reason Golem shows only a single type error. Type errors often cause further type errors, and we do not want to bury the programmer in error messages as is the case in GHC. Therefore it would make no sense to run programs containing type errors and the programmer is forced to fix type errors as soon as they occur. This works well in practice, because the feedback loop from the compiler is almost instantaneous and type errors are much easier to fix when we know which change (the last one) to the code caused them. The compiler always annotates the AST with type information, so Golem can display types of expressions even if there is a type error, which is not possible with the current interface to GHC.

4.8.1 Syntax Errors

One of the reasons we opted for S-Expression based syntax is that it avoids a large number of syntax errors.

Consider the following PureScript program:

```
class Sized a where
  size :: a -> Number

instance sizedNumber :: Sized Number where
  size n = n

instance sizedArray :: (Sized a) => Sized [a]
  size [] = 0       -- line 10
  size (x : xs) = size x + size xs

main = do
  print $ size 100
  print $ size [1, 2, 3, 4, 5]
```

Which generates the following error:

```
"" (line 10, column 11):
unexpected Equals
expecting no indentation or end of input
```

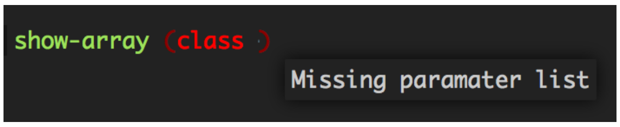
This error points to the `=` sign on line 10. It provides practically no information as to the true cause of the problem. This is because Haskell's syntax is very complex and making a small error can lead to a program which produces an AST which is completely different to what the programmer would expect from reading the program text. In this example, we forgot the `where` keyword after the `sizedArray` instance declaration head. A worse problem is that a "wrong syntax" from the programmer's perspective can produce a type error instead of a syntax error. Consider the following example:

```
fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)

main = print $ take 20 fibonacci
```

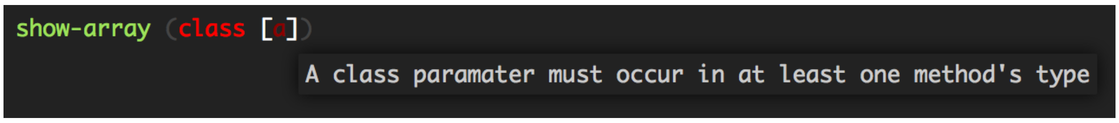
Now imagine we accidentally leave out or delete the second `:` operator. This is the "comprehensive" error message produced by GHC:

```
test.hs:1:13:
    No instance for (Num a2) arising from the literal '0'
    The type variable 'a2' is ambiguous
    Possible cause: the monomorphism restriction applied to the following:
      fibonacci :: [a2] (bound at test.hs:1:1)
    Probable fix: give these definition(s) an explicit type signature
                  or use -XNoMonomorphismRestriction
    Note: there are several potential instances:
      instance Num Double -- Defined in 'GHC.Float'
      instance Num Float -- Defined in 'GHC.Float'
      instance Integral a => Num (GHC.Real.Ratio a)
        -- Defined in 'GHC.Real'
      ...plus three others
    In the first argument of '(:)', namely '0'
    In the expression: 0 : 1 zipWith (+) fibonacci (tail fibonacci)
    In an equation for 'fibonacci':
        fibonacci = 0 : 1 zipWith (+) fibonacci (tail fibonacci)

test.hs:1:17:
    No instance for (Num
                       (((a0 -> b0 -> c0) -> [a0] -> [b0] -> [c0])
                        -> (a1 -> a1 -> a1) -> [a2] -> [a2] -> [a2]))
      arising from the literal '1'
    Possible fix:
      add an instance declaration for
      (Num
         (((a0 -> b0 -> c0) -> [a0] -> [b0] -> [c0])
          -> (a1 -> a1 -> a1) -> [a2] -> [a2] -> [a2]))
    In the expression: 1
    In the second argument of '(:)', namely
      '1 zipWith (+) fibonacci (tail fibonacci)'
    In the expression: 0 : 1 zipWith (+) fibonacci (tail fibonacci)

test.hs:1:27:
    No instance for (Num a1) arising from a use of '+'
    The type variable 'a1' is ambiguous
    Possible fix: add a type signature that fixes these type variable(s)
    Note: there are several potential instances:
      instance Num Double -- Defined in 'GHC.Float'
      instance Num Float -- Defined in 'GHC.Float'
      instance Integral a => Num (GHC.Real.Ratio a)
        -- Defined in 'GHC.Real'
      ...plus three others
    In the second argument of '1', namely '(+)'
    In the second argument of '(:)', namely
      '1 zipWith (+) fibonacci (tail fibonacci)'
    In the expression: 0 : 1 zipWith (+) fibonacci (tail fibonacci)

test.hs:3:8:
    No instance for (Show a2) arising from a use of 'print'
    The type variable 'a2' is ambiguous
    Possible cause: the monomorphism restriction applied to the following:
      fibonacci :: [a2] (bound at test.hs:1:1)
    Probable fix: give these definition(s) an explicit type signature
                  or use -XNoMonomorphismRestriction
    Note: there are several potential instances:
      instance Show Double -- Defined in 'GHC.Float'
      instance Show Float -- Defined in 'GHC.Float'
      instance (Integral a, Show a) => Show (GHC.Real.Ratio a)
        -- Defined in 'GHC.Real'
      ...plus 23 others
    In the expression: print
    In the expression: print $ take 20 fibonacci
    In an equation for 'main': main = print $ take 20 fibonacci
```

This would never occur in Golem, because the AST is apparent. If we deleted an operator, there would be a hole, and if we deleted the hole the argument would be called as a function, producing an error which would directly point to it. There are situations when the supplied arguments don't match the operator, possibly a macro. Golem highlights such cases and an informative message is provided when the user hovers the malformed expression:



```
show-array (class )
                    Missing paramater list
```



```
show-array (class [a])
                    A class paramater must occur in at least one method's type
```

These messages are defined in the built-in macros.

4.8.2 Type Errors

There has been substantial research carried out concerning the issues of finding, constructing and displaying type errors in Hindley-Milner style type systems, [122] presents a good summary. Helium[123] implements a constraint-based type checker to improve on existing error messages but the current implementation contains bugs and lacks code generation[124, 125], which makes us doubtful of its approach, both from theoretical and practical position. We have therefore used typical Hindley-Milner type unification and focused on the content of error messages and their display within Golem.

There are several different type errors which can occur during type checking:

1. Failure to unify two concrete types. This is the most common type error. Applications of different type constructors can never be unified, such as a `String` with a `Num` or a `(Fn a a)` with `(Array a)`.
2. The "occurs check" fails. This happens when we try to unify a type variable with a type which contains that type variable, effectively trying to construct an infinite type, such as when we try to unify `Fn a b` with `b`.
3. Kinds of unified types do not match. Type variables always carry their kind information, so that unified types are always valid. For example, we cannot unify variable `a` from the type `(Fn a b)` with the type constructor `Array`.
4. Inferred type is ambiguous. This means that the inferred type contains a type class constraint which includes some unconstrained type variable (variable not implied by a the implicit functional dependency or in an argument to a type constructor) which is not contained in the constrained type, such as `(Fn Num Num)` with a type class constraint `(Eq a)`.
5. Inferred type is ambiguous due to the "dreaded monorphism restriction"[13]. This error occurs when the inferred type of a definition which is not a function includes type class constraints. This restriction might be lifted in the future, but for now Shem enforces definitions to be either new functions or not constrained.
6. The declared type is too general. This means that the type which was explicitly declared for an expression matches the inferred type, but is more polymorphic. This can be the case when a function has declared type `(Fn a b)` but the compiler inferred the type `(Fn a Num)`.
7. The declared type is missing a constraint. The compiler inferred a constraint which has not been declared in the given explicit type.

[13]Its definition can be found in Section 4.5.5 of [126].

8. No instance found. There is no instance matching an inferred or declared constraint.

The general approach we take to display these errors is as follows. We try to provide a single-line concise and clear error message, which talks about the actual problem, rather than its circumstances. We then highlight the *origins* of types included in the message in the source code. The origin of a type is either its explicit declaration or the reference which introduced it. We adapted this approach from MIT's pH compiler[127]. We will now give several examples of the errors introduced above and compare them to GHC errors and discuss the pros and cons of each approach. We make comparisons with GHC as it is the de facto standard in functional programming tools and we can therefore demonstrate the added value of our approach to functional programming.

4.8.2.1 Unification Errors

We will start with the simplest type error, the failure to unify two types, and show how our error reporting provides a better insight than GHC[14] or Hugs.[15] Consider the following Haskell code

```
x = (uncurry (id mystic)) (flip (,) True 4)
mystic :: Int -> Int -> Int
mystic = (+)
```

We have obscured the call to +, which we typed for integers to avoid an instance error. GHC reports:

```
@1:37-1:41 Couldn't match expected type Int
 with actual type Bool

In the second argument of ‘flip’, namely ‘True’
In the second argument of ‘uncurry’, namely ‘(flip (,) True 4)’
In the expression: (uncurry (id mystic)) (flip (,) True 4)
```

At this point, the programmer might have the following questions:

1. How come the actual type is `Bool`? Type of what? GHC guides the programmer well to the literal value `True` which indeed has the type.

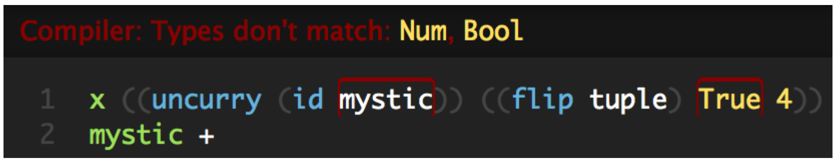
[14]Version 7.8.3.

[15]We leave the comparison with Elm and PureScript to Chapter 6, but in general they provide even less information than Haskell's compilers.

2. What are the other lines of the error message trying to tell us? How come the error is in the second argument to `uncurry` when `uncurry` only has one argument?

3. *Why is the expected type Int?* How do I fix the problem? Should I change `True` to something else?

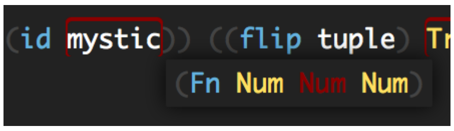
An experienced Haskell programmer will know that for 2. the lines suggest where the conflict lies. `(flip (,) True 4)` type checks fine but suddenly when calling `uncurry` the expected type is different. The reason for this is left for the programmer to figure out. Haskell mentions second argument to `uncurry` because it has unwrapped its partial application. We can now compare this experience to how this situation plays out in Golem. Consider Figure 4.11.



```
Compiler: Types don't match: Num, Bool

x ((uncurry (id mystic)) ((flip tuple) True 4))
mystic +
```

FIGURE 4.11: Unification error displayed in Golem.

Shem's error message is simpler and more importantly, does not assume that one of the offending types is expected. This is because although Haskell's "expected" type corresponds to how the type checker performs unification this description might very well be the exact opposite of what the programmer perceives. Perhaps in our example, it is `mystic` which has the wrong type, not the `True` argument. Golem therefore helps the programmer answer 1. by simply showing where `Bool` originates. The types within the error message are sorted according to the position of the origins, so we know that the second type corresponds to the second highlight within the IDE. Golem also answers 3. by highlighting not only the origin of the "actual" type but also the origin of the "expected" type. In this case, it is the `mystic` function. This is much better than GHC's message which only pointed out the whole call to `uncurry`. But Golem can give the programmer more information. If it is not obvious how `Num` originates from `mystic`, the programmer can hover the highlighted expression to display its type, as shown in Figure 4.12.



```
(id mystic)) ((flip tuple) Tr
            (Fn Num Num Num)
```

FIGURE 4.12: Displaying the type of an origin of a conflicting type with the conflicting type highlighted in red.

The programmer can also see the type of any other expression and we found this leads to a quite pleasant debugging experience.

Surprisingly, Hugs[16] shows the other side of the type error than GHC:

[16]Hugs is no longer in development and we tested on its last available version, from September 2006.

```
ERROR :1 - Type error in application
*** Expression     : uncurry (id mystic) (flip (,) True 4)
*** Term           : id mystic
*** Type           : Int -> Int -> Int
*** Does not match : a -> Bool -> Int
```

This fails to provide all the necessary information similarly to GHC and even more so because it is not shown why the expression (in this particular example trivial) `id mystic` gets the type `Int -> Int -> Int`. It is also left to the programmer to figure out why the two types do not match, which can be difficult with more involved types. It might also be confusing that the derivation of the "expected" type `a -> Bool -> Int` is based on both sides of the type error and is prematurely stopped when the error is encountered. A more logical type from the programmer's perspective would be `Int -> Bool -> a`, implied by the the second argument[17] to uncurry. Yet, this error message might be more welcomed by the programmer if it is indeed the operator which has an unexpected type and not the argument pointed out by GHC.

The issue of "expected" vs "actual" is even better demonstrated by the following example of type checking a list:

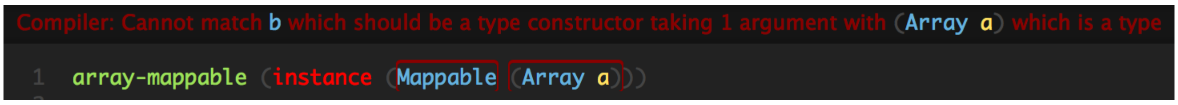
```
x = [y, False, True]
y = 'C'


@1:9-1:14 Couldn't match expected type Char
 with actual type Bool

In the expression: False
In the expression: [y, False, True]

@1:16-1:20 Couldn't match expected type Char
 with actual type Bool

In the expression: True
In the expression: [y, False, True]
```

For the programmer, it is perhaps `y` which does not have the expected type `Bool`. This example also reinforces our decision to only show one type error at a time.

GHC has a special message when kinds do not match in an instance declaration, while Shem does not mention kinds but talks about type constructors instead:

```
instance Functor [a]

@1:18-1:21 The first argument of Functor should have kind '* -> *',
  but [a] has kind *

In the instance declaration for 'Functor [a]'
```

[17]Hugs unwraps the partial call as well, which means that the contents of the error message do not correspond to the source code.



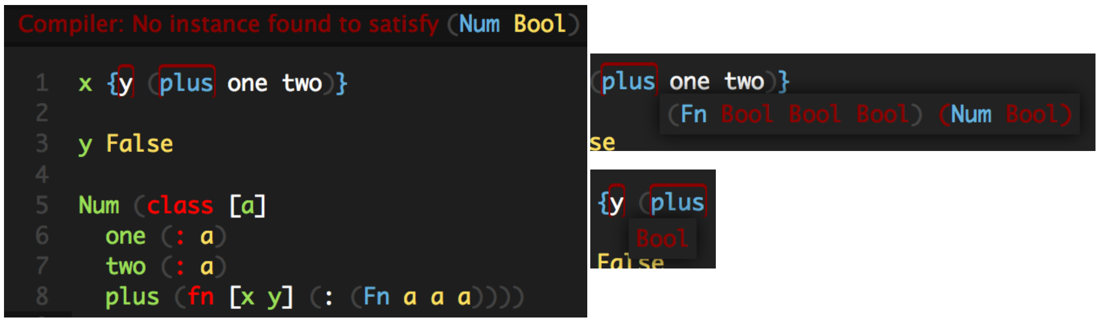
```
Compiler: Cannot match b which should be a type constructor taking 1 argument with (Array a) which is a type
array-mappable (instance (Mappable (Array a)))
```

4.8.2.2 Type Class Errors

These are errors which occur when type class constraints are involved in types and therefore do not occur in variants of ML, unlike unification errors. The most common one is the instance lookup failure, where the inferred type requires an instance which does not exist. In Haskell this error occurs quite frequently due to the numerical literals having an ambiguous type (in the sense that their return type is constrained). Consider the trivial example:

```
list = [y, 1 + 2]
y = False

@1:14-1:15 No instance for (Num Bool) arising from a use of +

In the expression: 1 + 2
In the expression: [y, 1 + 2]
In an equation for ‘list’: list = [y, 1 + 2]
```

Similarly to unification errors, GHC tells only half the story. It shows where the constraint for the instance originates, but not why the instance type, `Bool`, is required. Golem highlights both, in Figure 4.13.



```
Compiler: No instance found to satisfy (Num Bool)
x {y (plus one two)}

y False

Num (class [a]
  one (: a)
  two (: a)
  plus (fn [x y] (: (Fn a a a))))
```

```
(plus one two)}
    (Fn Bool Bool Bool) (Num Bool)
se
```

```
{y (plus
    Bool
False
```

FIGURE 4.13: Highlighting both the origin of a class constraint and its argument during a failed instance lookup.

When context reduction is forced on values defined via pattern binding (not function binding, see the definition of monorphism restriction[126]) the origin of the constraint can be in a completely different expression making it very difficult to identify the origin of the required instance type:

```
list = [x, y, z, 1 + 2]
x = 3
y = False
w = 42
z = w

@4:5-4:7 No instance for (Num Bool) arising from the literal 42

In the expression: 42
In an equation for ‘w’: w = 42
```

This error message seems misleading because GHC reports the origin of the constraint which is the last in the file and does not report the origin of the instance type. This means that if the programmer swapped the definition of `x` and `z` the reported origin would be the literal `3` in its definition. Shem improves the situation by reporting the first encountered origin, which is more likely to be within the expression which causes the error and by always showing the origin of the instance type.

The second common error is the presence of an ambiguous type. Programs must not contain ambiguous types because their runtime semantics would be ambiguous (see Section refCompilingTypeClasses). Consider the following example:

```
import Control.Applicative
k x = fst (x, empty)
```

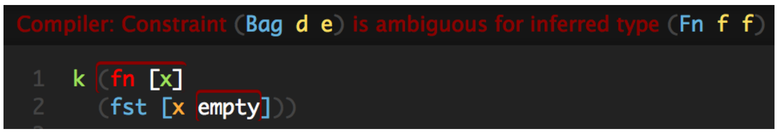
`empty` has the type `Alternative a => a`. Because the result of the function does not depend on `empty`, the compiler cannot infer which instance of `Alternative` it should use. GHC correctly points out the origin of the ambiguous type:

```
@2:15-2:20
No instance for (Alternative f0) arising from a use of empty

The type variable f0 is ambiguous
Note: there are several potential instances:
  instance Control.Arrow.ArrowPlus a =>
           Alternative (Control.Arrow.ArrowMonad a)
    -- Defined in Control.Applicative

  instance Alternative Maybe -- Defined in Control.Applicative

  instance Alternative Text.ParserCombinators.ReadP.ReadP
    -- Defined in Control.Applicative

  ...plus five others

In the expression: empty
In the first argument of ‘fst’, namely ‘(x, empty)’
In the expression: fst (x, empty)
```

The message is verbose because GHC is trying to explain that there are multiple matching instances, although in this case, without enabling the FlexibleContexts extension discussed in Section 3.5.2, it does not matter. The type would be ambiguous even if there was only one instance available. It might not be clear to the programmer why the type is ambiguous, when in the following case the compiler does not complain:

```
import Control.Applicative
k x = (x, empty)
```

Shem does better by pointing out which type is ambiguous as a result of the rogue type class constraint:



```
Compiler: Constraint (Bag d e) is ambiguous for inferred type (Fn f f)

k (fn [x]
  (fst [x empty]))
```

4.9 Summary

In this chapter we introduced Golem, a modern browser-based IDE for Shem. Its main design goal is to enable highly interactive programming. It can be used directly and immediately in the browser without requiring a server. It provides powerful, type-aware editing capabilities for Shem's S-Expression based code. It allows the programmer to experiment with and combine multiple programs which are always kept synchronized with the source code. It helps the programmer resolve errors through iterative logging and insightful type error display.

Chapter 5

Implementation

Although we have conceived this project as a platform for researching the idea of designing a programming language and its IDE hand in hand we have created for ourselves a significant technical challenge, as the scope of features to implement was large and many of them have never been implemented. This chapter therefore gives a brief overview of the implementation of both our compiler and the IDE. Both are implemented in CoffeeScript[18] in slightly over 10000 lines of code[1], evenly split between the two. While the compiler has no external dependencies, the IDE uses several open source JavaScript libraries, most importantly the Ace[129] embeddable code editor. We will first discuss the challenges with implementing a macro-directed statically typed language and then explain the architecture of our IDE and how it relies on the compiler.

5.1 Not a LISP

We have talked about Shem as a macro-directed language, but this does not mean that Shem is a LISP. The core idea of LISP is imperative. Macros, called special forms in Clojure, can change the current environment, whether it is namespaced or simply lexically scoped. When a function is defined in a LISP, it is completely compiled and attached to the environment, from which it can be invoked. This implies several restrictions:

1. No automatic circular dependencies or definitions in arbitrary order. If we want to define a function after it is used we must declare it first, which prepares the environment and other functions can then reference it.
2. The compilation performs a single pass. The macros are run and perform their actions, without being translated to some intermediate translation.

[1]Not counting comments and blanks. Lines of codes are a terrible metric for any purpose, but to get a sense of scale, note that CoffeeScript is one of the most expressive and terse languages[128].

It is certainly possible to implement a statically typed ML-derivative which uses this compilation strategy, provided that mutually dependent groups of definitions are explicitly typed[2]. Yet this cannot be done for a Haskell-like language because its defining feature, type classes, require two compilation passes. This is because context reduction is performed only at the level of definitions or programs and only after context reduction (also called generalization) the compiler can translate the uses of constrained (overloaded) functions and methods. We discuss how this is done in Shem in Section 5.4.1.

We did not want to abandon type inference for groups of mutually dependent definitions or force the programmer to keep their definitions in topologically sorted order, especially since this would not make any sense for where clauses (definition lists within functions). Yet we wanted to retain the ability to define macros which can be used in the same way as functions are used and use functions within their definition.

5.2 Macro Directed Compilation

The grand technical challenge of this project was the reimplementation of the Haskell 98 type system, with multi-parameter type classes and implicit functional dependencies, and the type class compilation, for a language which does not have a known syntax. Consider the normal steps taken in compiling a Haskell program by GHC[130]:

1. *The parser* converts text to a Haskell AST which completely matches all available Haskell syntax.
2. *Renaming* tags identifiers with unique numbers.
3. *Type checking* is performed on this original syntax to produce good error messages. The type checker itself consists of several phases, which deal with translating classes and instances, performing type inference, context reduction and method translation.
4. The original AST is *desugared* into a small Core language.
5. Repeated simplification and several passes towards producing native code.

In particular, the type checker performs a dependency analysis to construct bindings groups. Each binding group is then checked separately, as described in [46]. But Shem cannot perform any static analysis over the source code because at the time of the analysis it does not know which calls are function calls and which are macro calls that must be executed. The compiler therefore cannot assert a priori that a token represents an identifier referencing a certain definition. Hence

[2]Which would be a major nuisance for the programmers of such a language though.

Shem's compiler combines the single pass approach of LISPs with the multi-pass approach of traditional compilers using a monadic context and a system of *deferring*.

The compilation of a bare top level expression, such as the one used in a Golem's output box, proceeds as follows:

1. The source code is tokenized and "astized" into a bare AST. The AST is made of JavaScript arrays for forms and objects for tokens. Because JavaScript arrays are also objects, forms can be tagged with the same information as tokens are.
2. The "top level expression" macro is called with a new compilation context and the parsed AST. "Macro" is perhaps not the best name, since there is no syntax to invoke this function but it follows the general signature of all the built-in macros in Shem's compiler, accepting context and an AST node. This is why we call Shem a macro-directed language, because it is not made of separate passes but essentially a one big recursive decent parser operating on the AST, where every step of the decent is a call to a macro. The top level macro sets up the context for a bare definition, later forcing context reduction and invokes the generic expression compiling macro.
3. The "expression compile" macro is used whenever an expression should be compiled based on its own syntax. This macro in turn invokes the macros for compiling a hole, an atom, a tuple, an array, a map with string keys or a call to an operator.
4. The "call compile" macro in turn dispatches based on the operator and calls macros for compiling function calls or calls to macros. If the operator is not an atom, it will first be compiled using the above expression compiling macro.

Once a macro which is bound to a name, such as `instance` or `fn` is called it has full control over how its node in the AST will be processed but it cannot access any node from the parent tree (there are no parent links on the AST used in the compiler). The way macros interact across nodes is through the compilation context. We call the context monadic because only its updated values are observed by the macros. It was our hope that by not using references we could easily translate the compiler to a pure language supporting monads, such as Shem. Perhaps more importantly code written this way is simpler because there is no hidden local state, only a single updating object. The macros also enrich the AST, but because of the way the tree is passed around these labelings could also be modeled using a monad. We had to break this principle in one area of the compiler though, and that is the type substitution, which uses references for performance reasons, detailed in Section 5.6.

The compilation context holds information about the current scopes, the current definition pattern and counters for variables and type variables. The scopes then contain information about declared functions, macros, types, classes etc. The context is also used for marking a definition as *deferred*.

Each macro produces an intermediate representation (IR). This is a mix of JavaScript AST nodes and a couple of special IR nodes which are used for compiling type classes. Once a set of definitions is translated to IR and type checked we initiate the second pass which translates the IR nodes to JavaScript AST, finally a third pass translates the purely JavaScript AST to plain text JavaScript. In the case of a bare top level expression the set of definitions consists of the one expression.

5.3 Deferring

Deferring is a mechanism through which we achieve the arbitrary order of definitions and mutually dependent definitions. In essence it allows a macro to mark the current node as deferred. This is done mainly by the macro responsible for compiling references - if the reference does not correspond to any binding in scope the context is marked as deferred. The macros which are up in the chain of the recursive decent can then check whether the current definition is deferred and act accordingly. Since a deferred node does not know its type the calling macros must make sure they do not attempt to perform type inference. The deferred definitions in a definition list are then collected and once the list was fully traversed are tried again. It would be obviously extremely inefficient if we relied solely on this mechanism and it would not work for mutually recursive functions. This is why definitions of functions are predeclared when first compiled and calls to them are not deferred further, even if we do not know their type. The references of these functions, which cannot be typed, are given a fresh type variable as a type and are marked in the context for type resolution. After all the deferred definitions have been tried we perform Tarjan's strongly connected components algorithm on the remaining untyped definitions and type them using the strategy described in [46], where the components are either implicitly typed binding groups or explicitly typed bindings.

This strategy also has the advantage that the produced list of JavaScript assignments is already in the correct order, where the order matters due to JavaScript being strict. This description paints only a faint picture of the actual implementation, where definition lists can be embedded within definitions, deferring can be pushed to the parent definition and type resolution must for the same reason work across scopes. An example of these mechanisms is given in the next section.

5.4 Type Inference

Our gratitude goes to Mark Jones for his paper [46] which has been an indispensable resource in implementing Shem's type system. Unlike authors of most papers on type inference, Jones describes the process in which binding groups are type checked, which as he points out is one of the most complicated parts of the system, but is usually completely ignored by type systems theory. A simplification which we found useful was to make the return type of any type inference method, which in our case is simply the type marked in the AST, to be a constrained[3] type, whereas in [46] some methods return a type and a list of constraints, some return only a type etc. These are the types which get shown during type errors to the user.

To give an example of how our macro-directed compilation deals with type inference and binding groups, consider the following definitions:

```
f (fn [x]
  (g x)
  g (fn [y]
    (h y)))

h (fn [x]
  (h x))
```

In terms of type checking and type inference[4], g depends on h, h is recursive and f depends on g. In Haskell, the type inference proceeds as follows:

1. The definitions are split into two binding groups: $\{h\}$ and $\{f\}$, where the latter depends on the former.
2. The $\{h\}$ implicit binding group is type checked: h is assigned a new type variable and its definition is type checked, unifying both the type inferred for the reference of h in its body and the type of the whole definition with this type variable. h now has a known type.[5]
3. Now the $\{f\}$ group is type checked. Most importantly, h has an assigned type, so inferring the type of g is straightforward, and so is inferring the type for f.

In Shem, we do not have the information about dependencies. We attempt to compile in the first pass. The process is as follow:

[3]We use different terminology from the paper, where constrained types are called qualified, finalized types are quantified and , constraints are predicates. The paper as a whole is a great example of the mathematical style of writing code, where single letter names make the code extremely difficult to understand.

[4]The fact that a call to any of the functions would loop is irrelevant for us now.

[5]a -> b, although this is irrelevant.

1. f gets compiled. g gets compiled in the process, the reference to h is encountered and the definition of g is deferred. Because there is no definition for h in the definition list containing g, the deferral is pushed to the parent definition, which is f. Now f is *deferred for* h.

2. h gets compiled. Its declaration happens first, so the call to itself is not deferred, but because the type of h is unknown, the typing of h is deferred, noting the dependency on h.

3. f now gets recompiled. g gets compiled, h is correctly declared but we do not know its type, so the typing of g is deferred. When we try to resolve the deferred typings within f we find out that g depends on a definition which is not in the current scope. We therefore defer the typing of f. Importantly, unlike with deferring definitions, where we could forget about g, when we defer typing we must properly trace all untyped definitions. Therefore g's type will depend on h and f's type will depend on g.

4. We now resolve the deferred typings in the top scope, obtaining the groups $\{h\}$, $\{g\}$, $\{f\}$. Here we follow Haskell's algorithm. Crucially, we remembered which scope g belonged to, so we can assign its type in the correct scope. Shem's context keeps track of all scopes, even those which were exited already. This is important both for the IDE, so that we always know the type of every definition, and also for the type checking, because the scopes carry the information of deferred type class constraints.

Unlike deferred definitions and typings, deferred constraints also appear in Haskell. They are simply the constraints which relate to type variables coming from a parent scope. These constraints will form dictionaries, which must be added to the correct, parent definition. Consider the following trivial example and its translation to JavaScript:

```
ff (fn [x]
  (gg x)
  gg (fn [y]
    (= y x)))
```

```
var f = function f(_Eq_1, x) {
    var g = function g(y) {
        return _Eq_1['='](y, x);
    };
    return g(x);
};
```

The constraint `(Eq a)` is inferred from the use of the `=` operator. When inferring the type for g we must recognize that this constraint should not be "retained" and defer it to the parent definition, as described in [46]. Our deferred type checking resolution procedure ensures that these constraints are propagated even in the cases, such as the one in the previous example, where g was typed among the definitions of its parent scope.

5.4.1 Compiling Type Class Applications

As we have mentioned type classes pose additional constraint on the implementation, because unlike normal functions, which can be compiled once their signature has been found, references to methods and overloaded functions, functions with class constraints in their type, cannot be translated until context reduction. We follow the implementation of type classes described by Jones[131] but we use our IR instead of the placeholders used in the paper. Type class dictionaries are objects constructed using the class JavaScript pattern, possibly on the fly if they themselves are constrained. We have not implemented any optimization such as preconstructing composed type class dictionaries.

5.4.2 Context Reduction with Implicit Functional Dependencies

In [46] entailment is given by a constraint being equal to one of the normalized constraints (or normalized constraints attached to classes). This will in turn reduce constraint sets such as `(Eq a)`, `(Eq a)` to `(Eq a)`. But with implicit functional dependencies the normalized constraints are not truly normalized (our definition is that the first parameter is a type variable), which leads to constraint sets such as `(Bag b i)`, `(Bag b j)`. Therefore entailment involves instead of a mere equality check a unification step on the additional parameters of the constraint (those which are implied by the implicit functional dependency, in the example `i` and `j`), of which result must be applied to the rest of the type checking context. If the two constraints cannot be unified a type error occurred. Because functional dependencies can introduce new substitutions, we re-normalize all normalized constraints, which might have become reduced.

5.5 Definitions and Pattern Matching

In Shem values are mostly defined as pairs of a pattern and a value. This is achieved using the compilation context. The built-in macro responsible for compiling a definition list extracts pairs from its arguments and then for each definition pair attaches information of the pattern to the context. It is actually the value in the pair (its right hand side) which is responsible for compiling the whole pair. This allows macros such as `class` to use the current pattern, which must be a name, to name the newly declared class, which is stored in the context. Most of the built-in macros, which compile real runtime values, call a helper function which compiles the pattern, marking in the context that the current compilation environment is a pattern. This is the only special environment which is currently defined. It is essentially just a value on the context, which is the compiled right hand side of the definition. Macros such as the one responsible for compiling atoms will then use this value when compiling a pattern. Consider the following example:

```
[i j] (tuplize 3)
```

`tuplize` takes a single value and returns a pair containing that value twice. The call compiling macro will, once it compiled the call to `tuplize`, call the assignment helper, which in turn initiates the compilation of the `[i j]` pattern. The macro responsible for compiling tuples will be called, will check that it is being called inside of a pattern, and will in turn call the general expression compiling macro with an updated assigned value. For each `i`, `j`, this value will be an intermediate representation of the access of indices 0 and 1, respectively, on the current assignment value, which is the compiled call to `tuplize`.[6] Yet a different macro, such as `req` can simply use the pattern, its structure and syntax contents, without actually compiling a tuple pattern matching.

5.6 Compiler Performance

After profiling our compiler on large programs we have observed that the majority of time is spent in type checking system. As we have followed the naive implementation in [46], which completely ignores time complexity of the algorithms and data structures it uses, the original immutable implementation of unification spent a large portion of time combining substitutions. We have first tried to regain performance by optimizing the used data structures, using integer type variables and dense arrays tagged with initial index instead of strings and JavaScript objects as hashtables. We improved the performance of the whole compiler by 30 to 60% depending on the JavaScript VM used using these optimizations, but the naive substitution algorithm was still too slow to provide acceptable performance for medium sized programs (the compilation took 7 seconds for 1000 line program).

We therefore had to abandon the naive unification and instead use a more efficient substitution mechanism described in [132]. This paper was useful because it compared the algorithm we adapted from Jones with the substitution by sharing. We have observed corresponding improvement, about 7 to 8 times faster compilation, to the one reported in the paper. We would like to point out though, that there is an error in the algorithm as presented in the paper. The unification algorithm, as presented in the paper, binds unbound type variables to the second unified type, regardless of whether the type is a type variable or not. This can cause the construction of an infinite substitution chain. Imagine the following unification steps:

1. We unify type variable `a` with some type `T`.
2. We unify type variable `b` with variable `a`. Following the algorithm presented in the paper, this makes the reference in `b` point to `a` which points to `T`.

[6]Because tuples are represented by JavaScript arrays at runtime.

3. Now we attempt to unify `a` with `b`. The two are not the same type variable and so `a` is bound to `b`. This does not fail the occurs check because the occurs check only considers free variable within `T`. Because we are dealing with references, binding `a` to `b` creates a loop, since `b` is bound to `a`.

The corrected algorithm adds a case to check whether the second type is a bound type variable and in this case unifies the first type with the type bound to that variable.

Moving from combining substitutions to using references superficially simplifies the code, in that we do not have to update the context by extending the current global substitution with the one produced by the last unification, but makes the code rely on side-effects which are much harder to debug and control. This aspect manifested itself especially in the context reduction of constraints with implicit functional dependencies, where we had to make sure that the substitution is performed only when the whole constraint matched, as is usual in imperative code, by defensively cloning the constraints. It would be great if we could use a more pure approach, but [132] did not convince us that this was possible, given their pure approach had sufficient performance only when the occurs check was ignored.

5.7 Golem's Architecture

Golem is implemented using React[133] for its UI and Ace[129] for the editor components.

We have reasoned about why we wanted to have a purely browser-based environment in Section 4.1, through its availability and to achieve high interactivity. But it cannot be understated that we would not be able to implement our IDE in the given time frame had we chosen some other technology, a natively compiled language with a UI framework or perhaps tried to extend existing open source IDE platforms such as Eclipse or IntelliJ. It is not unusual that making a plugin, which does not drastically modify the behavior of the platform and only adds a feature on top of it, constitutes a whole individual project. This is understandable because firstly, programming in a language such as Java, being a verbose and less expressive language, takes longer than using a highly expressive language such as CoffeeScript[128], and secondly, these platforms have a huge API beyond which a modification to their behavior is virtually impossible. On the other hand with Ace we could make any changes we needed, usually in a fairly compatible way given it is designed to be highly extensible, and in the worst case we could make a change or a fix which got pulled back into the Ace project. In the course of the project we have successfully submitted 4 Pull Requests and filed several bugs which were fixed. This would not be possible with the much bigger platforms due to the high traffic in their bug trackers, number of open issues and their release cycle.

Given that the compiler runs directly in the browser, we must ensure that the compilation does not interfere with editing. We therefore run the compiler and the module manager within a Web Worker[134], which ensures that the compilation happens on a separate thread. We use the worker wrapper provided by Ace which gets updated with the changes to the currently edited document and automatically compiles it using Shem's compiler. The module manager takes care of compiling any required modules and recompiling modules which had their dependencies updated. The compiler produces an annotated AST, which contains information about labels, types and scopes, used for debugging and type-based auto-completion. It also produces any type errors. The information from the AST is copied onto the AST which is kept in sync by the editor mode. Ace provides modes the capability to override how the editor behaves when the mode is active. The Shem mode overrides almost all of the text editing functions of Ace to work on both the AST and the text within the editor. These "editing commands" are described declaratively in terms of changes to the AST and a single `mutate` method performs the actual mutation on the AST as well as the text within the editor. Ace asks the mode for tokens on each line, which are converted from the labeled tokens of the AST, to perform syntax highlighting. Syntax highlighting is therefore performed by a couple of the main macros within the compiler, such as the atom compiling macro which labels strings as constants, references as references etc.

The module manager produces complete JavaScript which is then used within output boxes to run the programs. All output boxes use the same worker which belongs to the main module editor.

In the browser version Local Storage[135] is used to preserve modules and the state of the compiler. This means that reloading the browser window does not cause loss of any data, as the modules are automatically saved. In the desktop version Electron[136] is used to wrap the browser-based implementation and bundle it with a version of Chromium and Node.js that allows us to use real files, both for saving modules and from the programs.

5.8 Summary

In this chapter we sketched the implementation of our compiler and the IDE. The compiler performs recursive descent over the language AST, passing around a context object, enriches the AST and produces an intermediate representation which is then translated to JavaScript. It uses deferring to recompile definitions with missing dependencies and to infer types of nested, mutually dependent definitions. The IDE is built on Ace and React and runs the compiler on a separate thread to avoid impairing the performance of the user interface.

Chapter 6

Evaluation

The goal of this project was to design and implement the best possible programming environment. The ideal evaluation would therefore entail surveying a large number of programmers, with the same level of experience with our environment and some other environment, and quantitatively comparing the results. Unfortunately there do not exist such programmers yet. We must therefore use a more qualitative approach. We will first try to argue about various aspects of our environment judged by general criteria and later take a more critical view and make a direct comparison to applicable alternatives.

6.1 Evaluating Shem as a General Purpose Language

There are many criteria for evaluating programming languages.[137] In this section we try to provide a qualitative survey of the language, as it is used within its IDE, starting with criteria imposed by the users of the language:

- **Rapid development**. Together with its IDE Shem has a great potential for rapid development because we are no longer programming in the dark - results of our programs are always available and we can easily observe running code. Static typing prevents runtime errors which saves us time otherwise spent debugging. On the other hand, Shem being pure-by-default, imperative algorithms are not as easy to express, at least for now.

- **Easy maintenance**. Static typing provides a great aid to maintaining a functioning code base. Along with tests it ensures that changes made to the code in one place do not break other code. There are no built-in facilities for testing at the moment, although testing can be performed using any of the available frameworks for JavaScript.

- **Reliability and safety**. Shem code itself is very safe. The prelude avoids partial functions, so for example the equivalent of `head`, `first`, returns an optional (`Maybe`) value. Interfacing with JavaScript obviously opens up the program to runtime errors and such code must be thoroughly tested.

- **Portability**. Shem is portable as much as JavaScript is, but it does not itself target different language runtimes. We are quite comfortable with this decision, especially since C++ can be used to write Node.js extensions for truly performance intensive applications.

- **Performance**. We put significant amount of effort into making the compiler fast to provide an interactive experience. Current performance of the compilation is shown in Figure 6.1[1]. We do not perform any caching currently within the compiler so there could be opportunities for further improving its running time. In terms of the language itself, high performance was not our top priority in designing it and we definitely favored simplicity over efficiency[2]. See Section 6.2 below for more detail.

- **Learnability**. As we will discuss further the learning curve for our environment is quite steep. We never intended our environment to serve as a learning environment, as we believe that some of its underlying principles, such as structured editing and static typing, are more suited for experienced programmers. Yet this is definitely an area where Shem lacks at the moment and more work needs to be done to soothe the path of new users of the language.

- **Reusability**. High polymorphism leads to higher reusability and together with the extensibility of our language we believe Shem will lead to less repetitive code.

Apart from these "user requirements" a good language should also achieve the following qualities:

- **Readability**. We have strove for readability of our language. Indeed it was one of the main reasons for abandoning Haskell's syntax which we find too ambiguous. But readability also depends largely on conventions adopted by the users of the language. Together, we believe that expressions such as `(fn [action folded] (apply (map & action) folded))` are more readable than `cons_f x ys = (:) <$> f x <*> ys`[3]. It is also our IDE which makes code more readable by making it more tangible: the programmer can select any expression to highlight it among other expressions. This is one of the criteria which is hard to compare without a long term experience, but we believe that simplicity and unambiguity lead to readability.

[1]We report the means from 20 test runs of each module. The relative standard deviation was high, often over 30%. There might be many reasons, such as JIT optimization withing the VM and memory cache hits/misses. The reported numbers were collected on a 2.5GHZ Intel Core i5 CPU in Firefox.

[2]Compare with Scala[138].

[3]This is part of the definition of Traversable instance for Lists in GHC's Prelude.

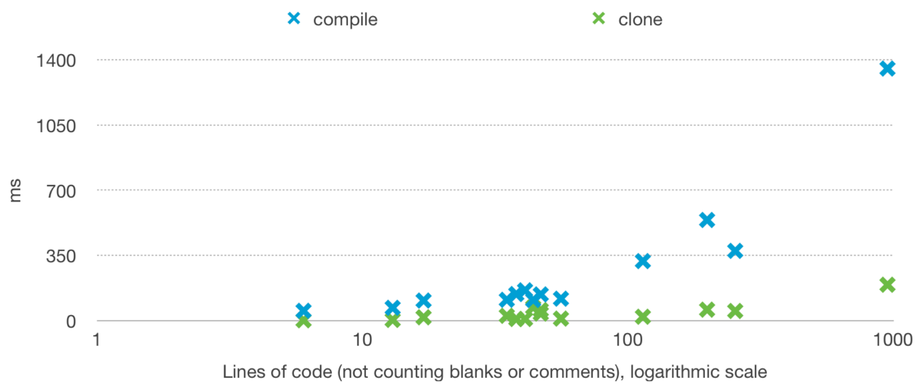


Figure 6.1: How long programs of certain sizes take to compile. The 1000 line program is the current Prelude. Such monolithic modules are not usual, for example our full solutions to the first year Haskell exercise have all from 50 to 100 lines of code. The green labeled points represent the additional time required to clone the result of the compilation from the worker to the main thread.

- **Writeability**. Shem is already a concise language, although in some places we preferred simplicity and consistency over brevity, for example by excluding function patterns or pattern matching within function parameter lists. Yet we believe that the high extensibility of our language will allow it to evolve towards what programmers find most useful. Consider the current `fn` macro, which makes writing simple lambdas one character longer than in Haskell. We believe that not having two separate syntaxes for functions is a good property to have, but if we later find out that writing lambdas this way is too onerous, a shorter syntax is just a macro away[4]:

  ```
  l (syntax [..args]
    (match args
      {..xs res} (` fn [,..xs] ,res)))

  test (zip (l x y (x y)) {+ - * /} (range 1 5))
  ```

  To provide a measurement, we compare the number of key strokes required to input the factorial function using Lamdu, Shem, Haskell and other languages in Figure 6.2. We use this particular example because it is the only one available where we can compare with Lamdu (recall Section 2.4.7). Shem dominates this comparison even more with more complex examples, but we do not necessarily consider this a valuable metric, as programmers rarely type programs from start to finish. The graph does manifest the difference in verbosity between Shem, Haskell and CoffeeScript on one side and JavaScript and Java on the other side, and it is not far fetched to believe that less verbose languages are easier to write.

[4]Lambdas are not reusable, so we prefer to define and name functions which can be reused. The lambda in the example has an equivalent in Shem's prelude called apply-1.

- **Simplicity**. There are perhaps simpler languages, but we hopefully struck a nice balance with our approach to typing, FFI and macros. We expect the language to get more complex as it grows, even though we would like to preserve this initial simplicity.
- **Consistency**. Consistency and unambiguity go hand in hand. "No surprises" are often mentioned to embody this philosophy. It is difficult to achieve, as every new challenge in the design of the language presents an opportunity to trade consistency for a quick solution to the problem. We have paid attention to consistency but there are certainly area for improvement, such as the editing controls and the prelude.
- **Abstraction, Expressiveness**. We believe that our language, with its user macros and highly polymorphic collections, is highly expressive. We do lose some expressiveness compared to lazy languages, but we gain in evaluation and performance predictability and the simplicity of its compilation. Imperative concepts can also be more difficult to express. This prompts us to consider easier ways of embedding imperative computations directly within the language, potentially sacrificing purity.

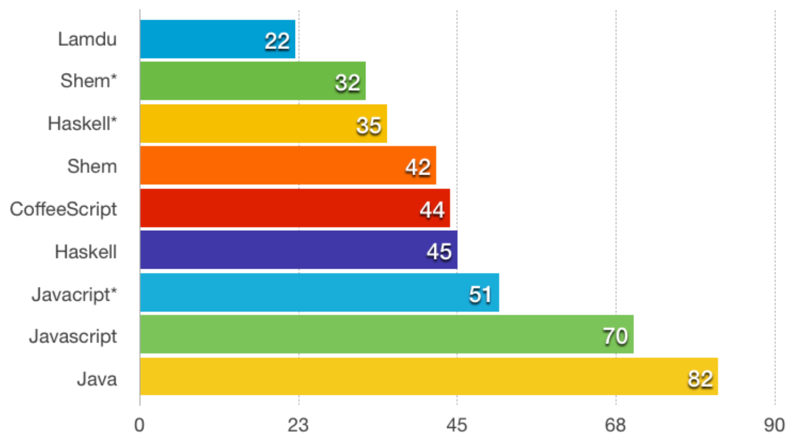


Figure 6.2: Comparing the number of key strokes required to write the factorial function, from example given in the Lamdu tutorial[139]. The * labeled rows correspond to programs written using snippets or commands, while the rest are typed character by character (except for Lamdu). The keystrokes and programs are given in Appendix A.1.2.

6.2 Performance

Comparing languages, unlike frameworks or libraries, performance-wise is rather difficult. It is complicated to construct a suite of programs which would truthfully represent each language and usually the libraries and algorithms used dominate any performance differences[140]. There are differences between classes of programming languages, lower-level languages such as C are generally faster than higher-level languages such as Java or Python. In this sense Shem's performance is limited to that of JavaScript. In general Shem runs as fast as PureScript or

Elm, with differences depending on the used functions and collections. Although we have made sure that no decision in the design of the language would severely impact the performance, achieving the best possible performance was not our goal. Macros obviously allow for some optimizations, although their impact would have to be studied further, as JavaScript VMs often inline function calls anyway.

6.3 User Studies

Although we cannot perform a comparative survey of programming environments we carried out several user studies during the later stages of the project development. Because we cannot teach a new programming language and the controls of a new IDE in the time of a short user session we used a script which prescribed what the programmer should do. We specifically focused on:

- Difficulty of getting used to structural editing.
- Whether the language constructs present made sense to the programmer.
- Readability of the resulting code, especially code which the programmer did not write themselves.

We have engaged users with very mixed backgrounds, from different years of the Computing degree at Imperial College London, of different programming aptitudes. In general, beginners[5] were more willing to follow the instructions and therefore encountered less surprises with structural editing. Very experienced programmers[6] tried to go ahead and use familiar shortcuts from text editors, which did not map to corresponding actions in our editor.

The following feedback was often repeated across sessions:

- Majority of the programmers found the language readable.
- Type completion is “really nice”. Especially beginners relied on auto-completion whenever it was available.
- Movement across atoms and holes works well. It is intuitive and easy to learn.
- Some often used commands, such as *define*, *replace parent* and *wrap in a call* are useful and easy to internalize.
- Testing on OS X, having both `Command-` and `Ctrl-` based shortcuts is confusing.

[5]Students from 1st year which have finished courses on Haskell, Java and C and had no prior experience with programming before starting the degree.

[6]Students of later years, with industrial experience.

- Where clauses without the where keyword are not as obvious as in Haskell.
- Types should be always visible, possibly using a bottom tray displaying the type of the current selection.

General reaction to the environment depended mainly on the personal preference of each programmer. Those without LISP experience found the language odd, but often liked the IDE and would want to use it for they favorite language. Programmers who knew both Haskell and LISP appreciated the language, but some would much prefer to use a normal text editor to program the language. Those who did not remember Haskell too well found it difficult to understand the basic concepts behind the language.

The lesson we took from these user studies is that there is still a room for improvement in how familiar the editing experience is. One observation we made was that users often tried to use the `Right` arrow key to move the cursor behind a delimiter, instead of typing a closing delimiter. This kind of action pattern is often very ingrained and if we do not change the current model then at least we should provide a customization option to enable this behavior. We have also made several changes based on the gathered user feedback, such as using the `Enter` key for filling in auto-completion or using `-` for negative number literals[7]. These have some negative impact on consistency but the benefits of making the environment more familiar and less strange outweigh them. User studies are time consuming but we wish we conducted more tests, focusing on different parts of the system, such as its type error messaging.

One particular problem we noticed ourselves when using the IDE to prepare the some of the examples for this report is that converting examples from other languages is not currently simple. We cannot just insert them, we have to either copy them viewing the source side by side or first convert them to Shem in a text editor. There is a simple solution to this problem discussed in Section 7.1.8.

6.4 Implementation Evaluation

The scope of the project had its negative impact on its implementation. The main problem is the relatively low test coverage. There are currently 83 language tests (in 1300 lines of Shem), which are run by compiling Shem programs, executing them and comparing their results. We could probably write another 50 tests just to test known behavior and added functionality, plus a lot more for testing error behavior, for partial and malformed programs and programs with type errors. The IDE lacks tests completely, as it is more difficult to test user interfaces, but

[7]The original design used the   symbol, in line with the negation operator.

it could certainly be done. Increasing test coverage should be a priority for further developing the project.

For the IDE specifically, we need to spent more time testing the controls to make sure they work well across platforms and user settings. We have mentioned that the IDE does not currently provide facility for customization, although this would be fairly simple to implement, it is certainly something missing from the current state of the project.

6.5 Comparison with Current Alternatives

Here is a summary of how Shem and Golem compare as a programming environment to PureScript, Elm and Haskell today:

Advantages

- *No set up.* The full environment is available in the browser and the desktop version is a single executable. PureScript and Elm offer pre-built executables which provide a REPL; text editors must be set up separately and there is no true IDE support. Haskell requires multiple steps to install the compiler, the platform and text editor support. There are more IDE-like features available for some text editors.
- *Proper collections library.* Shem has a collections library polymorphic in collection types and has built-in lists, arrays, maps and sets. PureScript has separate packages which implement some collections in PureScript itself and its built-in lists are JavaScript arrays. Elm has built-in sets, maps lists and arrays, but they each have a separate interface and since Elm does not have type classes, collections of collections have almost no support apart from lists of lists.
- *Interactive programming.* There are no tools currently available for any of the three languages which enable the kind of interactive programming experience that Golem provides.
- *Simple JavaScript interop.* Calling JavaScript from Shem is really easy, Shem's functions can be passed directly to JavaScript code expecting functions. This is not possible in PureScript because functions are statically curried and nearly impossible in Elm. This makes taking advantage of the large NPM ecosystem in Shem much simpler.
- *Less errors and better error messages.* In section 4.8 we compared our syntax and type error messaging to Haskell's popular compilers and mentioned that Elm and PureScript provide the programmer with even less information.

Disadvantages

- *Lack of pure libraries.* A disadvantage of a new language is the obvious lack of libraries. Although good interop alleviates this problem somewhat, because Shem is statically typed JavaScript, dynamically typed libraries must be additionally typed for truly easy use and where applicable converted to pure style. It would be useful to simplify this task by borrowing some libraries from PureScript, but there is no easy way to share types between the two type systems, since types are extracted during compilation and the two type systems miss features of each other. PureScript has quite a few libraries and a very large category theory based type class hierarchy but being a new language itself a lot of these are poorly documented and still evolving. Elm has a large built-in library for implementing user interfaces, which is Elm's main targeted use case, implemented partly in JavaScript for efficiency. There is also a number of external libraries by members of the community. In numbers, Bower[141] used by PureScript for package distribution, has 210 PureScript packages listed[8] while Elm's own package management system lists 187 packages. For comparison Hackage[142] lists 8268 packages. Both PureScript and Elm are still evolving their core concepts, while Haskell, given its experimental nature, is still in flux in terms of libraries and language extensions.
- *Reliability of the type system.* We have not proven our type system error-free and judging from the past months of implementation we suspect that new bugs might appear. On the other hand our type system sophistication is only rivaled by Haskell, as Elm completely ignores type classes and PureScript does not infer constrained types, making programming in Elm cumbersome and in PureScript not as effortless as programming in Shem.
- *Learning curve.* In order to push the boundary of current programming environments we had made some unorthodox and novel choices. Although we find our structural editor quite powerful and easy to use, it certainly takes time to master the new editing model, although given its simplicity and similarity to text editing this is probably less severe than in the case of Emacs or Vim. To programmers who have never encountered either Haskell or LISP our language will feel unfamiliar either from semantics or syntax perspective. Haskell, Elm and PureScript certainly have a steep learning curve as well for programmers without functional background and we have made this curve even steeper.
- *Lack of learning resources.* We have not had the time to create a proper tutorial or reference, which are a necessity for building a community around the language.

6.6 Summary

In this chapter we tried to evaluate Shem and Golem, qualitatively, with the help of user studies, and comparatively, with respect to other strict Haskell-like languages targeting JavaScript and

[8]At the time of writing.

Haskell itself. Overall, programming in Shem and Golem is currently hindered by the lack of existing library support. We expect the experience to improve significantly as we build up this ecosystem. We would obviously not recommend to use our system for production-level applications and systems today but it is already a great tool for exploratory programming and, in our opinion, a good starting point for a better future of programming.

Chapter 7

Conclusion

Our main goal was to design and implement a programming environment tailored to the programmer's experience. To achieve this we designed a new language from the ground up, taking the best aspects of different powerful, high level languages, which would fit into our vision of a truly interactive IDE. The language targets the most popular runtime of today and the IDE is purely browser based, available instantly for anyone to try out.

We have met the success conditions of the original project proposal. Shem is a general purpose programming language and with its simple JavaScript FFI can be used today to write interactive applications, servers, crawlers, 2D, 3D and GUI applications. Golem supports all of Shem's features. Because it is aware of the AST, it can perform more complex operations than modern text editors and, thanks to the full support of multi-selections, can easily perform complex operations not available in traditional IDEs.

And yet we seem to be far away from achieving the objectives we set out for ourselves at this beginning of this report. The project's level of polish is not yet high enough for us to advertise it to the world's programming community. There are still rough edges concerning the editing experience and more user testing needs to be done to identify the decisions where we went too far in designing an ideal but unfamiliar experience. The compiler needs to be properly tested to ensure that more type-system related bugs are not lurking among the thousands of lines of its (dynamically typed) code.

Yet from the user studies we have conducted we are optimistic about finding a niche group of supporters who will appreciate the simplicity and elegance of our language and the interactivity and power of our IDE. After all

> *The language must start small, and the language must grow as the set of users grows.*[76]

7.1 Extensions and Future Work

Given the large scope of this project it is not surprising that there is a big number of possible extensions and future work. In this section we discuss the most interesting ones.

7.1.1 Source Control

Traditional IDEs integrate traditional source control systems such as Git[143] or Mercurial[144]. These tools work on any text file on a line by line basis. This is usually inappropriate for code[110]. Given our IDE operates directly on the AST, it would be possible to implement smarter source control. More importantly, the IDE has an actual idea of the meaning of the changes and could automatically generate commits and merge commits. Ideally, we would like a system where the undo/redo stack is indistinguishable from the source control. The IDE can generate changes such as “renamed x to y” and merge them based on time, the amount/complexity of the change and its scope, to create larger commits such as “changed the implementation of f to use g”, “removed method m” etc. It should still be possible for the programmer to tag a range of changes with a custom message.

The same information can be used to provide a much better `blame` system, since the authorship of code would be known on the AST level and would not get mangled by programmers making slight changes or renames. Such a system does not have to implement the whole vertical of a source control manager (SCM), it could serialize these changes into a format understood by Git/Mercurial and use the existing infrastructure of these tools.

7.1.2 Versioning

It would be great to be able to require specific versions of certain functions or modules. This is currently done using external tools like NPM, Golem does not keep track of semantic versioning. There are some good ideas in this space, such as hashing functions to provide unique identifiers[145] and versioning based on types used in Elm’s package manager[81].

7.1.3 Code Hosting

Given that our IDE runs directly in the browser, we could integrate with a hosting code service, such as GitHub. GitHub now supports CORS[146] requests so this could be done without accessing a server. NPM does not support CORS at the moment, so it is not possible to load libraries in the browser version of the IDE. This would enable anyone to edit any Shem project in the IDE with no setup costs and make changes via pull requests. Github provides online editing facilities but these are obviously very limited compared to what Golem provides.

7.1.4 Asynchronous and Imperative Computation

Given we based our language largely on Haskell, imperative and asynchronous computations are not as straightforward as pure computations. We currently use monads to deal with these (see Appendix A.2.3), which do have the advantages of type safety but can be a bit unwieldy when we have to wrap every call with potential side effects in an action. We could have chosen to implicitly wrap calls to JavaScript in an action, but this would not allow us to distinguish between blocking and asynchronous calls and there is no way to determine which kind of control flow is used statically from the code.

To make the current monadic approach efficient we would have to implement the inlining of monad chaining, as is done in PureScript, which translates monadic IO chains into normal imperative assignments and calls in JavaScript. This works well for chains within a single function but does not optimize the case of running many actions across functions - on the other hand, that's where the abstraction of an action is a useful concept.

In our original proposal we have touted the possibility of designing a simpler approach to contained imperative computations than using monads. Although some of this can be achieved today with macros, we have not yet provided the facility of running loops, for example, which is important for getting the best performance out of the JavaScript VM.

The asynchronous problem is currently hotly debated in the JavaScript community, with different people preferring differing solutions: generators, streams, functional reactive programming (used in Elm) and communicating sequential processes (popularized by Go). It would be interesting to see which approach fits Shem best and whether our implicit functional dependencies could be used in the solution.

7.1.5 UI Programming

In Appendix A.2.1 we showed a very crude approach to building a UI, using HTML strings. A better library is obviously needed. React[133] is currently among the most popular frameworks for building web UI.[1] The exciting prospect is that Shem's semantics match React's model of pure rendering and because Shem uses immutable data structures, React can optimize its checking for changes in its update loop. Thirdly, because Shem has macros the syntax for constructing DOM trees can be made much nicer than the equivalent Haskell or JavaScript

[1]After all, Golem is built using React.

code.[2] It would be very interesting to see if these aspects actually align, as supporting a popular framework has helped some languages[3] to gain in popularity.

7.1.6 IDE Integration

Although we chose to build our IDE on top of Ace we could have taken a different approach and integrated our editing and experimentation experience with an existing browser-based IDE, such as LightTable[59] or Atom[61]. The reason we did not chose this path is because we were not sure whether we could deliver the editing experience we wanted in these more controlled environments (neither uses Ace, LightTable uses CodeMirror[147], an alternative to Ace, and Atom has its own implementation of the text editor). The advantage of integration would be that the other features of the IDE, handling files and integrating external tools, would be already taken care of. We might run into similar issues as the ones we mentioned in Section 5.7, as the bigger the main project the more difficult it is to integrate beyond the aspects envisioned by its creators.

7.1.7 Truly Lazy Compiler

Currently, the smallest unit of compilation is a module. It would be interesting to explore the idea of making the compiler truly lazy, in the sense that instead of keeping track of dependencies between modules it would keep track of dependencies between definitions and only recompile parts of modules as necessary. This could lead to further performance improvements. Such an approach is not impossible to imagine, as at the end of the compilation we have all the necessary information. The question then would be what impact can a change have on this state.

7.1.8 Plain Text Mode

Based on the issues we discussed in Section 6.3, it is sometimes useful to edit the source code using strictly text, such as when converting from a different language to Shem. This would be very simple to implement using our current architecture. We could simply switch to a plain text mode for the duration of the conversion. The IDE would have to check that the source is valid or reformat it before switching back to the Shem mode.

[2]Indeed, JavaScript would be so bad at expressing HTML that the team behind React came up with an XML syntax, which gets compiled to JavaScript.

[3]Yes, Ruby.

Appendix A

Sample Programs

A.1 Basic

These are simple pure programs. Haskell is presented for comparison.

A.1.1 Algebra

The following is not an efficient solution. A `find-first` function or a lazy collection should be used. This is a solution to the third Euler Project problem[148]:

```
euler3 (fn []
  (largest-prime-factor 600851475143))

largest-prime-factor (fn [x]
  (larger-prime-factor 1 x)
  larger-prime-factor (fn [than x]
    (match (first (filter (divisible? what: x) (range 2 (+ 1 (sqrt x)))))
      None x
      (Some factor) (larger-prime-factor factor (/ factor x)))))
```

A.1.2 Factorial

Factorial without pattern matching:

```
fac (fn [x]
  (if (= 0 x)
    1
    (* x (fac (- 1 x)))))
```

```
fac x =
  if x == 0 then 1
  else x * fac (x - 1)
```

The key strokes counted in Figure 6.2. Special characters are used to represent non-alphabetical keys and key combinations:

```
fac x=if x=0 1 x*f x-1 # Lambdu | 22
fac fx®(if =†0 x) 1 *†x f† -†1 x # Shem using Ctrl-F and autocompletion | 32
fac x = if†x == 0†1†x * fac (x - 1) # Haskell with if snippet | 35
fac (fn [x] (if (= 0 x) 1 (* x (fac (- 1 x # Shem 42
fac = (x) ->®if x is 0®1®†else x * fac x - 1 # CoffeeScript | 44
fac x = if x == 0 then 1 else x * fac (x - 1) # Haskell | 45
fun†fac†x†ife†x == 1†return 1†return x * fac(x - 1)
                      # Javacript with good fun and ife snippets | 51
function fac(x) {if (x === 0) {return 1} else {return x * fac(x - 1)}}
                      # Javascript | 70
class Fac {int fac(int x){if (x === 0) {return 1;}
                            else {return x * fac(x - 1);}}} # Java | 82
```

A.1.3 Binary Search

```
search2 (fn [n list]
  (search-in 0 (size list))
  search-in (fn [min max]
    (if (>= max min)
      None
      (match (compare n middle)
        LT (search-in min half)
        GT (search-in (+ 1 half) max)
        EQ (Some half)))
    middle (!! (at half list))
    half (+ min (div 2 (- min max)))))
```

```
import Data.Array
search n list
  = uncurry searchIn (bounds list)
  where
    searchIn min max
      | max <= min = Nothing
      | otherwise =
        case compare n middle of
          LT -> searchIn min half
          GT -> searchIn (half + 1) max
          EQ -> Just middle
      where
        half = (min + max) `div` 2
        middle = list ! half
```

A.1.4 L-Systems

Adapted from first year exercises. Notice that maps are used instead of looking up in lists in Haskell. Comments and types had to be stripped for presentation:

```
Rules (type (Map Char String))
System (record
  angle: Num
  base: String
  rules: Rules)

tree (fn [angle] (System
    angle
    “M”
    (Map
      \M “N[-M][+M][NM]”
      \N “NM”
      \[ “[”
      \] “]”
      \+ “+”
      \- “-”)))

l-system (fn [system n]
  (trace
    (expand-one mapper
      (expand
        (System-rules system)
        (System-base system)
        n))
    (System-angle system)
    [1 0.8 0.6]))

lookup-char (fn [char in]
  (!! (at char in)))

expand-one (fn [rules base]
  (concat (map-into
      (lookup-char in: rules) {} base)))

expand (fn [rules base n]
  (reapply (expand-one rules) base n))

move (fn [command state rotation]
  (match command
    \F [[(+ x (cos a)) (+ y (sin a))] angle]
    \L [pos (+ rotation angle)]
    \R [pos (- rotation angle)])
  [x y] pos
  a (degrees angle)
  [pos angle] state)

trace (fn [commands rotation color]
  lines
  [end empty lines] (fold step initial commands)
  step (fn [command current]
    (match command
      \[ [state (& state stack) lines]
      \] [(!! (first stack)) (rest stack) lines]
      dir (do-move dir))
    [state stack lines] current
    do-move (fn [command]
      [next stack (& [from to color] lines)]
      [to _] next
      [from _] state
      next (move command state rotation)))
  initial [[[0 0] 270] (List) (List)])

mapper (Map
  \M “F”
  \N “F”
  \+ “R”
  \- “L”
  \[ “[”
  \] “]”)
```

```haskell
tree angle = ( angle, "M",
  [ ( 'M', "N[-M][+M][NM]"),
    ( 'N', "NN"),
    ( '[', "[" ),
    ( ']', "]" ),
    ( '+', "+" ),
    ( '-', "-" ) ] )

lSystem s count
  = expandOne mapper (expand (rules s) (base s) count)

angle system = let ( x, _, _ ) = system in x

base system  = let ( _, x, _ ) = system in x

rules system = let ( _, _, x ) = system in x

lookupChar ch rs = head [s | ( i, s ) <- rs, i == ch]

expandOne rs cmd = concat (map (`lookupChar` rs) cmd)

expand rs cmd n = iterate (expandOne rs) cmd !! n

move r ( ( x, y ), ang ) rot
  | r == 'F' = ( ( x + cosD ang, y + sinD ang ), ang )
  | otherwise = ( ( x, y ), turn ang rot)
  where
    turn = if r == 'L' then (+) else (-)

cosD = cos . dTR
sinD = sin . dTR

dTR x = pi * x / 180

trace pattern rot color = gen pattern [initPos]
  where
    gen [] stack = []
    gen (ch : chs) stack@(start : rest)
      | ch == 'F' = line : advance (to : rest)
      | ch == '[' =        advance (start : stack)
      | ch == ']' =        advance rest
      | otherwise =        advance (to : rest)
      where
        advance = gen chs
        line = ( fst start, fst to, color, 1 )
        to = move ch start rot
    gen (ch : chs) stack@(start : rest)
      | ch == 'F' = line : advance
      | otherwise = advance
      where
        line = ( getVert start, getVert to, color, 1 )
        to = move ch start rot
        advance = gen chs next
        next
          | ch == '[' = (start : stack)
          | ch == ']' = rest
          | otherwise = (to : rest)

initPos = ( ( 0, 0), 90 )

getVert = fst

mapper  = [ ( 'M', "F" ),
            ( 'N', "F" ),
            ( 'X', "" ),
            ( 'Y', "" ),
            ( 'A', "" ),
            ( '[', "[" ),
            ( ']', "]" ),
            ( '+', "L" ),
            ( '-', "R" ) ]
```

A.2 Advanced

A.2.1 A Guessing Game

Source for the UI shown in Figure 4.7. Uses plain strings for producing HTML and callback-style event programming. First the crude DOM library, using barebones DOM API[1]. Also shows that we can peform impure operations without monads:

```
time-out! (fn [duration callback]
  (: (Fn Num (Io Void) (Io Void)))
  (io (.setTimeout (global) (fn [] (run-io callback)) duration)))

on-id (fn [id event-name callback]
  (: (Fn String String (Fn Js (Io a)) (Io Void)))
  (io (.addEventListener global.document event-name (fn [event]
        (if (== target-id id)
          (exec-io (callback event))
          Void)
        target-id (:: String (.-id (.-target event)))))))

element-by-id (fn [name]
  (: (Fn String (? DomElement)))
  (from-nullable (.getElementById global.document name)))

set-html! (fn [html element]
  (: (Fn String DomElement (Io Void)))
  (io (set! (.-innerHTML element) html)))
```

Now we can put these together. Clearly, this demonstrates that we can use the native JavaScript APIs, but also that we should use a better abstraction layer:

[1] Which is horrible.

```
guess-a-number (fn [min max]
  (do
    (set r (random-int min max))
    (on-id “button” “click” (fn [event]
        (do
          (set-result! “...”)
          (time-out! 500 (set-result! (answer r guess))))
        set-result! (fn [result]
          (set-html! result (!! (element-by-id “result”))))
        guess (integer-value (!! (element-by-id “guess”)))))
    (lift (ui min max))))

answer (fn [goal guess]
  (match (compare guess goal)
    LT “That's too little.”
    GT “That's too much.”
    EQ “That's correct!”))

ui (fn [min max]
  (concat
    {“<div style='
      padding: 20px;
      background: white;
      color: #444;
      font: 16px Helvetica; '>”
      (format “<div style='padding: 10px'>
        Can you guess a number between %n and %n?
        </div>” min (- 1 max))
      “<input
        id='guess'
        style='margin: 10px 10px 20px'></input>”
      “<button id='button'>Guess</button><br />”
      “<span id='result' style='margin: 10px'></span>”
      “</div>”}))
```

A.2.2 A Server

Using the Express.js[149] library, replies with `“Hello world!”` to requests on port 8080:

```
server (do
  (set app (io (express)))
  (io (.get app “/” (fn [req res]
        (.send res “Hello world!”))))
  (io (.listen app 8080 (fn []
        (log “Success”))))
  (lift “starting...”))

express (:: (Fn Js) (require “express”))
```

A.2.3 Asynchronous Computations

Defining an Async monad:

```
Async (data [a] Async [content: (Fn (Fn a Void) Void)])

start-async (fn [async]
  (# Fires of an async action. Similar to run-io.)
  ((Async-content async) (fn [x] Void)))

chain-async (fn [wrapped through]
  (# Similar to chain-io, but using callbacks to chain asynchronous
    computations.)
  (Async (fn [x]
      ((Async-content wrapped) (fn [y]
          ((Async-content (through y)) x))))))

async-mappable (instance (Mappable Async)
  map (fn [what over]
    (chain-async over
      (fn [x] (Async (fn [cb] (cb (what x))))))))

async-liftable (instance (Liftable Async)
  lift (fn [x] (Async (fn [cb] (cb x))))
  apply (fn [what to]
    (chain-async what (fn [unwrapped-what]
        (chain-async to (fn [unwrapped-to]
            (Async (fn [cb] (cb (unwrapped-what unwrapped-to))))))))))

async-chainable (instance (Chainable Async)
  chain chain-async)

sync (fn [io]
  (# Takes an io action and turns it into an Async action.
    This would be an instance of MonadIO in Haskell.)
  (Async (fn [cb]
      (cb (run-io io)))))

async (syntax [..expressions]
  (` (io (start-async (do ,..expressions)))))
```

Now we can define some asynchronous actions[2] and use them with the `do` macro:

[2]"require" requires a Node module.

```
greet... (do
  (say... “Hello,”)
  (wait... second)
  (say... “world!”))

second 1000

wait... (fn [ms]
  (# Block for ms .)
  (Async (fn [cb]
      (:: Void (.setTimeout (global) cb ms)))))

say... (fn [what]
  (# Pronounce what using the OS X say command.)
  (Async (fn [cb]
      (:: Void (.exec child-process (format “say %s” what) cb)))))

child-process (require “child_process”)
```

A universal Traversable instance for all appendable bags (see Haskell's `mapM`) allows us to chain collections of Async actions (or any other Liftable instances). Compare with equivivalent JavaScript, without using libraries for dealing with asynchronocity:

```
greetings... (fold-do (repeat 10 greet...))

fold-do (fn [actions]
  (: (Fn (b (a c)) (a (b c)))
    (Bag (b (a c)) (a c))
    (Bag (b c) c)
    (Appendable (b c) c)
    (Liftable a))
  (fold-right add (lift empty) actions)
  add (fn [action folded]
    (apply (map & action) folded)))
```

```
var child_process = require('child_process');

var greetings = function(cb) {
  var count = 10;
  var callNext = function(){
    if (count > 0) {
      count--;
      greet(callNext);
    }
  });
}
```

```
var greet = function(cb) {
  say("Hello", function() {
    wait(1000, function() {
      say(", world!", cb);
    })
  });
}

var say = function(what, cb) {
  child_process.exec("say" + what, function(error, result) {
    cb();
  });
}

var wait = function(ms, cb) {
  setTimeout(function(){
    cb();
  }, ms);
}
```

Bibliography

[1] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Notices*, volume 48, pages 411–422. ACM, 2013.

[2] Phil Freeman. Purescript by example, 2014. URL `https://leanpub.com/purescript`.

[3] Edwin Brady. Programming in Idris: a tutorial. Technical report, Technical report, University of St Andrews, 2013.

[4] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of functional programming*, 17(01):1–82, 2007.

[5] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *ACM Sigplan Notices*, 43(9):51–62, 2008.

[6] Oracle. What's New in JDK 8, 2015. URL `http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html`. [Accessed Jan 29th, 2015].

[7] Douglas C Engelbart. The mother of all demos. 1968. URL `https://www.youtube.com/watch?v=yJDv-zdhzMY`.

[8] Alan C Kay. The early history of smalltalk. In *History of programming languages—II*, pages 511–598. ACM, 1996.

[9] Bret Victor. Inventing on principle, 2012. URL `http://worrydream.com/#!/InventingOnPrinciple`.

[10] Paul Graham. Beating the averages, 2004. URL `http://www.paulgraham.com/avg.html`.

[11] DZone. Eclipse indigo release train now available: 46 million lines of code across 62 projects, 2011. URL `http://eclipse.dzone.com/articles/eclipse-indigo-released`. [Accessed Jan 29th, 2015].

[12] Michael Lee Scott. *Programming language pragmatics.* Morgan Kaufmann, 2000.

[13] Andreas Gomolka and Bernhard Humm. Structure editors: Old hat or future vision? In *Evaluation of Novel Approaches to Software Engineering*, pages 82–97. Springer, 2013.

[14] Ben Lippmeier. *Type inference and optimisation for an impure world.* Australian National University, 2009.

[15] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[16] Andrew W Appel and David B MacQueen. Standard ML of new jersey. In *Programming Language Implementation and Logic Programming*, pages 1–13. Springer, 1991.

[17] Alon Zakai. Emscripten: an LLVM-to-Javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011. URL `http://emscripten.org/`.

[18] Jeremy Ashkenas et al. Coffeescript, 2012. URL `http://coffeescript.org/`.

[19] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[20] Airbnb. Isomorphic Javascript: The future of web apps, 2013. URL `http://nerds.airbnb.com/isomorphic-javascript-future-web-apps/`. [Accessed Jan 29th, 2015].

[21] faylang/fay · github. `https://github.com/faylang/fay`. (Visited on 12/06/2015).

[22] ghcjs/ghcjs · github. `https://github.com/ghcjs/ghcjs`, . (Visited on 12/06/2015).

[23] elm-lang/elm-compiler · github. `https://github.com/elm-lang/elm-compiler`, . (Visited on 12/06/2015).

[24] Clemens Fruhwirth. Haskell semantics with lisp syntax. 2007. URL `http://clemens.endorphin.org/ILC07-Liskell-draft.pdf`.

[25] Ml (programming language) - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/ML_%28programming_language%29`. (Visited on 12/06/2015).

[26] Ocaml – ocaml. `http://ocaml.org/`. (Visited on 12/06/2015).

[27] Ocsigen. Js_of_ocaml. `http://ocsigen.org/js_of_ocaml/`. (Visited on 12/06/2015).

[28] Performances of js_of_ocaml compiled programs. URL `http://ocsigen.org/js_of_ocaml/2.5/manual/performances`. [Accessed Jan 29th, 2015].

[29] Oracle. java.com: Java + you. `https://www.java.com/en/`, . (Visited on 12/06/2015).

[30] C sharp (programming language) - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/C_Sharp_%28programming_language%29`. (Visited on 12/06/2015).

[31] harmony:specification_drafts [es wiki]. `http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts`. (Visited on 12/06/2015).

[32] Closure compiler | google developers. `https://developers.google.com/closure/compiler/`. (Visited on 12/06/2015).

[33] Facebook. Flow | flow | a static type checker for javascript. `http://flowtype.org/`, . (Visited on 12/06/2015).

[34] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts.* " O'Reilly Media, Inc.", 2008.

[35] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.

[36] The racket language. `http://www.racket-lang.org/`. (Visited on 12/06/2015).

[37] Mark Tarver. Shen. URL `http://www.shenlanguage.org/`.

[38] Martin Odersky. (comment) universal type inference is a bad thing. URL `http://www.codecommit.com/blog/scala/universal-type-inference-is-a-bad-thing#comment-3438`.

[39] John Peterson and Mark Jones. Implementing type classes. In *ACM SIGPLAN Notices*, volume 28, pages 227–236. ACM, 1993.

[40] Stefan Wehr and Manuel MT Chakravarty. ML modules and haskell type classes: A constructive comparison. In *Programming Languages and Systems*, pages 188–204. Springer, 2008.

[41] Yaron Minsky. Effective ML. URL `http://vimeo.com/14313378`.

[42] Differences with Java, multi-methods. URL `http://groovy-lang.org/differences.html#_multi_methods`. [Accessed Jan 29th, 2015].

[43] Simon Peyton Jones. Type directed name resolution, 2010. URL `https://ghc.haskell.org/trac/haskell-prime/wiki/TypeDirectedNameResolution`.

[44] Bouke van der Bijl. Idiomatic generics in go. URL `http://bouk.co/blog/idiomatic-generics-in-go/`. [Accessed Jan 29th, 2015].

[45] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for c^{\ sharp} generics. In *ECOOP 2006–Object-Oriented Programming*, pages 279–303. Springer, 2006.

[46] Mark P Jones. Typing haskell in haskell. In *Haskell workshop*, volume 7, 1999.

[47] nlinger. Ensuring that a function is polymorphic in ocaml 3.12 :: Jane street tech blogs. `https://blogs.janestreet.com/ensuring-that-a-function-is-polymorphic-in-ocaml-3-12/`. (Visited on 06/12/2015).

[48] Martin Sulzmann, Gregory J Duck, Simon Peyton-Jones, and Peter J Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 17(01):83–129, 2007.

[49] Mark P Jones. Type classes with functional dependencies. In *Programming Languages and Systems*, pages 230–244. Springer, 2000.

[50] Robert Dockins. [haskell] the ghc typechecker is turing-complete. `https://mail.haskell.org/pipermail/haskell/2006-August/018355.html`. (Visited on 12/06/2015).

[51] Macro (computer science), syntactic macros, . URL `http://en.wikipedia.org/wiki/Macro_%28computer_science%29#Syntactic_macros`. [Accessed Jan 29th, 2015].

[52] Brian Goslinga, Eugene Butler, and Elena Machkasova. Improving error messages in the clojure programming language.

[53] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that work together. *Journal of Functional Programming*, 22(02):181–216, 2012.

[54] JetBrains. Intellij idea — the most intelligent java ide. https://www.jetbrains.com/idea/specials/idea/idea-tools.jsp?utm_expid=85132606-37.5-enLWJnTxyshuS9opeVLA.2&utm_referrer=https%3A%2F%2Fwww.google.com%2F. (Visited on 12/06/2015).

[55] Eclipse - the eclipse foundation open source community website. https://eclipse.org/. (Visited on 12/06/2015).

[56] Oracle. Netbeans. https://netbeans.org/, . (Visited on 12/06/2015).

[57] Microsoft. Visual studio - microsoft developer tools. https://www.visualstudio.com/. (Visited on 12/06/2015).

[58] Neil Toronto. (fourth racketcon): Neil toronto — purely functional 3d in typed racket - youtube. https://www.youtube.com/watch?v=t3xdv4UP9-U. (Visited on 12/06/2015).

[59] Light table. http://lighttable.com/. (Visited on 12/06/2015).

[60] Sublime text: The text editor you'll fall in love with. http://www.sublimetext.com/. (Visited on 12/06/2015).

[61] Inc. Github. Atom. https://atom.io/, . (Visited on 08/06/2015).

[62] Chris Granger. Light table - a new ide concept. http://www.chris-granger.com/2012/04/12/light-table---a-new-ide-concept/, . (Visited on 12/06/2015).

[63] Bret Victor. The future of programming on vimeo. https://vimeo.com/71278954, . (Visited on 12/06/2015).

[64] Bret Victor. Bret victor, beast of burden. http://worrydream.com/, . (Visited on 12/06/2015).

[65] Bret Victor. Up and down the ladder of abstraction. http://worrydream.com/LadderOfAbstraction/, . (Visited on 12/06/2015).

[66] bret Victor. Learnable programming. http://worrydream.com/LearnableProgramming/, . (Visited on 12/06/2015).

[67] Inc. Apple. Swift - overview - apple developer. https://developer.apple.com/swift/. (Visited on 12/06/2015).

[68] Chris Granger. Aurora (strangeloop 2013 demo) - youtube. https://www.youtube.com/watch?v=L6iUm_Cqx2s, . (Visited on 06/12/2015).

[69] Eyal Lotem. Lamdu. http://peaker.github.io/lamdu/, . (Visited on 06/12/2015).

[70] Smalltalk - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Smalltalk. (Visited on 12/06/2015).

[71] Pharo - welcome to pharo! http://pharo.org/. (Visited on 12/06/2015).

[72] Erlang programming language. http://www.erlang.org/. (Visited on 12/06/2015).

[73] Kenneth E Iverson. Notation as a tool of thought. *ACM SIGAPL APL Quote Quad*, 35 (1-2):2–31, 2007.

[74] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[75] Ron. Rondam ramblings: Why lisp? http://blog.rongarret.info/2015/05/why-lisp.html. (Visited on 12/06/2015).

[76] Guy Steele. Growing a language, by guy steele - youtube. https://www.youtube.com/watch?v=_ahvzDzKdB0. (Visited on 06/12/2015).

[77] eduardoejp. **simplicity** my ideal of simplicity is more like haskell's than clojure's. c... | hacker news. https://news.ycombinator.com/item?id=9489927. (Visited on 06/12/2015).

[78] Joel Burget. The third age of computing is dependently typed. http://joelburget.com/third-age-dependently-typed/, . (Visited on 06/12/2015).

[79] Ecmascript language specification - ecma-262 edition 5.1. http://ecma-international.org/ecma-262/5.1/. (Visited on 12/06/2015).

[80] Google. chromeos - google search. https://www.google.com/search?q=chromeos&oq=chromeos&aqs=chrome..69i57j0l5.157j0j4&sourceid=chrome&es_sm=91&ie=UTF-8. (Visited on 12/06/2015).

[81] Elm package manager. http://elm-lang.org/blog/announce/PackageManager.elm, . (Visited on 08/06/2015).

[82] #9586 (implement traversable/foldable-burning-bridges proposal) – ghc. https://ghc.haskell.org/trac/ghc/ticket/9586. (Visited on 12/06/2015).

[83] Backus–naur form - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form. (Visited on 06/12/2015).

[84] Extended backus–naur form - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form. (Visited on 06/12/2015).

[85] Mozzila. Grammar and types - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#String_literals, . (Visited on 12/06/2015).

[86] Oracle. Chapter 19. syntax. http://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html, . (Visited on 06/12/2015).

[87] Objective-c - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Objective-C. (Visited on 12/06/2015).

[88] Manuel Rubio-Sánchez and B Pajak. Fibonacci numbers using mutual recursion. *Koli Calling*, pages 174–177, 2005.

[89] C. A. McCann. haskell - typeclass constraints on data declarations - stack overflow. http://stackoverflow.com/questions/12770278/typeclass-constraints-on-data-declarations. (Visited on 06/12/2015).

[90] Algebraic data type - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Algebraic_data_type, . (Visited on 06/12/2015).

[91] Wolfram Kahl and Jan Scheffczyk. Named instances for haskell type classes. In *Proceedings of the 2001 Haskell Workshop, number UU-CS-2001-23 in Tech. Rep*, pages 71–99. Citeseer, 2001.

[92] Rich Hickey. Clojure - reader. `http://clojure.org/reader`. (Visited on 12/06/2015).

[93] S Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, et al. Haskell 98: A non-strict, purely functional language, 1999.

[94] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[95] Mark P Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, 5(01):1–35, 1995.

[96] Flexibleinstances – haskell prime. `https://ghc.haskell.org/trac/haskell-prime/wiki/FlexibleInstances`, . (Visited on 12/06/2015).

[97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, pages 1–16, 1997.

[98] Flexiblecontexts – haskell prime. `https://ghc.haskell.org/trac/haskell-prime/wiki/FlexibleContexts`, . (Visited on 12/06/2015).

[99] Big o notation - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Big_O_notation`. (Visited on 12/06/2015).

[100] Oracle. Lesson: Interfaces (the java™ tutorials > collections). `https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html`, . (Visited on 12/06/2015).

[101] Facebook. Immutable.js. `https://facebook.github.io/immutable-js/`, . (Visited on 12/06/2015).

[102] Data.foldable. `https://hackage.haskell.org/package/base-4.7.0.2/docs/Data-Foldable.html`, . (Visited on 12/06/2015).

[103] Evan Czaplicki. Elm designguidelines. `http://package.elm-lang.org/help/design-guidelines`. (Visited on 12/06/2015).

[104] James Halliday. Browserify. `http://browserify.org/`. (Visited on 12/06/2015).

[105] James Burke. Requirejs. `http://requirejs.org/`. (Visited on 12/06/2015).

[106] Phil Freeman. Release v0.7.0-rc.1 · purescript/purescript · github. `https://github.com/purescript/purescript/releases/tag/v0.7.0-rc.1`. (Visited on 12/06/2015).

[107] Andrey Popp. Fighting node callback hell with purescript. `https://andreypopp.com/posts/2014-07-21-fighting-node-callbacks-with-purescript.html`. (Visited on 12/06/2015).

[108] Jamie Brandon. Incidental complexity. `http://incidentalcomplexity.com/2015/02/24/january--february/`. (Visited on 12/06/2015).

[109] Josh Marinacci. Typographic programming wrapup. `http://joshondesign.com/2014/10/06/typoplwrapup`. (Visited on 12/06/2015).

[110] Joel Burget. Programmer tooling beyond plain text. `http://joelburget.com/plaintext/`, . (Visited on 06/12/2015).

[111] The Grid. Flowhub v0.8.4. `http://app.flowhub.io/`. (Visited on 12/06/2015).

[112] Microsoft word - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Microsoft_Word`. (Visited on 12/06/2015).

[113] Source code editor - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Source_code_editor`, . (Visited on 12/06/2015).

[114] seanmcdirmid. The main problem with structured editing is that it requires programmers to writ... | hacker news. `https://news.ycombinator.com/item?id=6964727`. (Visited on 12/06/2015).

[115] Emacswiki: Par edit. `http://emacswiki.org/emacs/ParEdit`. (Visited on 12/06/2015).

[116] welcome home : vim online. `http://www.vim.org/`. (Visited on 12/06/2015).

[117] Gnu emacs - gnu project - free software foundation (fsf). `http://www.gnu.org/software/emacs/`. (Visited on 12/06/2015).

[118] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[119] Unknown. sediment.io. `http://sediment.io/`. (Visited on 12/06/2015).

[120] Hack. `http://hacklang.org/`. (Visited on 12/06/2015).

[121] Haskell/debugging - wikibooks, open books for an open world. `http://en.wikibooks.org/wiki/Haskell/Debugging`, . (Visited on 12/06/2015).

[122] Bastiaan J Heeren. *Top quality type error messages*. Utrecht University, 2005.

[123] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62–71. ACM, 2003.

[124] Jurriaan Hage. Knownbugs < helium < uucs. `http://foswiki.cs.uu.nl/foswiki/Helium/KnownBugs`, . (Visited on 12/06/2015).

[125] Jurriaan Hage. Hiw 2014: The past, present and future of the programmer-friendly helium compiler - youtube. `https://www.youtube.com/watch?v=YioWTDi_Jyc`, . (Visited on 12/06/2015).

[126] 4 declarations and bindings. `https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-930004.5.5`, . (Visited on 12/06/2015).

[127] Lennart Augustsson. Hiw 2014: Lennart augustsson: Better type-error messages - youtube. `https://www.youtube.com/watch?v=rdVqQUOvxSU`. (Visited on 12/06/2015).

[128] Berkholz. Programming languages ranked by expressiveness – donnie berkholz's story of data. `http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/`, 2013. (Visited on 06/07/2015).

[129] Mozzila Cloud9 IDE Inc. Ace - the high performance code editor for the web. `http://ace.c9.io/#nav=about`. (Visited on 07/06/2015).

[130] Commentary/compiler/hscmain – ghc. `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscMain`, . (Visited on 12/06/2015).

[131] John Peterson and Mark Jones. Implementing type classes. In *ACM SIGPLAN Notices*, volume 28, pages 227–236. ACM, 1993.

[132] Atze Dijkstra, Arie Middelkoop, and S Doaitse Swierstra. Efficient functional unification and substitution, 2008.

[133] Inc. Facebook. A javascript library for building user interfaces | react. `https://facebook.github.io/react/`, . (Visited on 08/06/2015).

[134] Mozzila. Web workers api - web api interfaces | mdn. `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API`, . (Visited on 12/06/2015).

[135] Window.localstorage - web api interfaces | mdn. `https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage`. (Visited on 12/06/2015).

[136] Inc. Github. Electron. `http://electron.atom.io/`, . (Visited on 08/06/2015).

[137] University of Washington. Cse 341 – evaluating programming languages. `http://courses.cs.washington.edu/courses/cse341/02sp/concepts/evaluating-languages.html`. (Visited on 12/06/2015).

[138] EPFL. The scala programming language. `http://www.scala-lang.org/`. (Visited on 10/06/2015).

[139] Eyal Lotem. lamdu/tutorial at master · peaker/lamdu · github. `https://github.com/Peaker/lamdu/blob/master/TUTORIAL`, . (Visited on 12/06/2015).

[140] Comparison of programming languages - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Comparison_of_programming_languages#Benchmarks`, . (Visited on 12/06/2015).

[141] Bower. `http://bower.io/`. (Visited on 12/06/2015).

[142] Introduction | hackage. `https://hackage.haskell.org/`. (Visited on 12/06/2015).

[143] Git. `https://git-scm.com/`. (Visited on 12/06/2015).

[144] Selenic. Mercurial scm. `https://mercurial.selenic.com/`. (Visited on 12/06/2015).

[145] Chiusano. Unison: May 7th 2015 update and roadmap. `http://unisonweb.org/2015-05-07/update.html`. (Visited on 08/06/2015).

[146] Cross-origin resource sharing - wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Cross-origin_resource_sharing`. (Visited on 12/06/2015).

[147] Codemirror. `http://codemirror.net/`. (Visited on 08/06/2015).

[148] Problem 3 - project euler. `https://projecteuler.net/problem=3`. (Visited on 12/06/2015).

[149] StrongLoop. Express - node.js web application framework. `http://expressjs.com/`. (Visited on 12/06/2015).