SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2374

Designing a Novel Programming Language by Analysing the Industry's Current State

Filip Sodić

Zagreb, ožujak 2021.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

I want to thank Prof. Ante Đerek for his mentorship and help in writing the thesis. I also thank Prof. Mile Šikić for attending to my frequent changes of interests, his guidance over the past years, and recommending me to Prof. Đerek. Finally, I want to thank my family and friends for tolerating my absence from various aspects of living during the final few weeks of my work on the thesis.

Želim se zahvaliti profesoru Anti Đereku za mentorstvo i pomoć pri izradi rada. Zahvaljujem se i profesoru Mili Šikiću na razumijevanju za moje učestale promjena interesa, vodstvu tijekom proteklih godina, te za preporuku kod profesora Đereka. Konačno, želim se zahvaliti svojoj obitelji i prijateljima na toleranciji moga izbivanja iz mnogih sfera života tijekom nekoliko tjedana prije konačne predaje rada.

CONTENTS

1. Introduction

In JavaScript, what does the expression `[] + {}` evaluate to? The most appropriate answer is "nothing meaningful."

JavaScript's lack of type safety became the language's bane very soon after its inception. The flexibility of its type system was a reasonable choice at the time, at least considering the language's planned use cases. Brendan Eich designed JavaScript in ten days as a small scripting language for quickly adding dynamic behavior to websites (Cassel, 2018). It was never intended to be used for large-scale software development – its creators anticipated that these kinds of applications would always be written in Java [1]. After twenty-five years, not a single website runs Java anymore. On the other hand, JavaScript, a simple small-scale programming language, is completely dominating the industry – and not just on the web, everywhere – together with its inadequate type system and an abundance of dangerous edge cases.

This thesis aims to design Func, a brand new high-level, general-purpose programming language that harnesses JavaScript's rich semantic capabilities while at the same time preventing its less desirable behaviors. The thesis also aims to design and implement a compiler for the language. Func is transpiled directly into JavaScript, allowing it to cleanly fit inside its already vast ecosystem. The new language intends to be used for data processing and mathematical calculations, a domain left largely undiscovered by JavaScript and most related projects (e.g., TypeScript and Flow). Most of Func's design decisions come from observing the industry's current state and choosing to focus on battle-tested properties and features. To minimize the number of run-time errors without imposing a severe burden on the programmer, Func employs a provably correct type system with full polymorphic type inference. The compiler is written in TypeScript. Its

[1] This is the origin of the name JavaScript. The language's creator, Brendan Eich, said in an interview: *[...] The idea was to make it a complementary scripting language to go with Java, with the compiled language* (Cassel, 2018). JavaScript was meant to be a small scripting language used for frivolous tasks on websites, while Java did all the heavy lifting.

source code is freely available on GitHub under the MIT license[2].

Chapter 2 enumerates projects and languages that inspired Func's design. Chapter 3 explores the language's high-level design goals, offering arguments for every decision made during the design process. Chapter 4 sets the foundation for building Func by introducing its underlying model and describing its core language. Chapter 5 explores Func's type system by referencing the core language introduced in Chapter 4 and offering rules and algorithms for typing it. Chapter 6 describes Func's surface syntax and shows how it relates to the core language. Chapter 7 elaborates on how the compiler transforms the core language into executable JavaScript. Chapter 8 provides an overview of unachived goals and future plans for the language. Finally, the thesis' conclusion is given in Chapter 9.

[2] A demo is available here.

2. Related work

2.1. ML

Robin Milner and his associates designed ML in the early 1970s as a part of the LCF theorem prover (Gordon et al., 1978), the first theoretically based yet practical tool for machine-assisted proof construction (Pierce, 2012). The name ML is an acronym for "Meta Language" and comes from the language's intended application in conducting proofs in PPλ (Polymorphic Predicate λ-calculus), the language of LCF. After realizing ML's potential, the creators lifted it from LCF and continued working on it as a standalone general-purpose language. With time, the language split into two distinct dialects. The present-day variants of these dialects, Standard ML and OCaml, still enjoy general use.

ML was the first language to include polymorphic type inference and a type-safe exception handling mechanism. Its elegant unification-based type inference Algorithm $\mathcal{W}$ has greatly influenced many subsequent works in programming languages (Pierce, 2012). ML's type system, today known as the Hindley-Milner (HM) type system [1] type system, remains the state-of-the-art solution for implementing decidable and sound type systems for languages based on the λ-calculus. Hindley-Milner is the type system used in most functional languages, including Scala, Haskell, and Agda (Pierce, 2012).

Standard ML has been a significant influence for Func. Func's type system is a variant of Hindley-Milner, and its type inference engine is an adaptation of Milner's original Algorithm $\mathcal{W}$. Damas and Milner proved that Algorithm $\mathcal{W}$ was mathematically decidable sound (Milner, 1978) and complete (Damas & Milner, 1982).

[1] J. Roger Hindley independently discovered the type system during his research on combinatory logic (Hindley, 1969). The same type system is also sometimes referred to as ML-style, Damas-Milner, or Damas-Hindley-Milner type system. The text uses the most common name, Hindley-Milner.

2.2. Haskell

Haskell is a high-level purely functional programming language with non-strict semantics featuring ML-style type inference and several notable extensions of the Hindley-Milner type system.

A committee of functional programmers developed Haskell over a period of twelve years. The committee ceased to exist in 1999, entrusting the project with Simon Peyton Jones and the open-source community, who continue their work on the language to this day (Hudak et al., 2007). Haskell was conceived in 1987. to replace many different non-strict, functional programming languages present at the time, all similar in expressive power and semantic properties. The coauthors felt that the lack of a common language was hampering widespread use of this particular class of programming languages, so they set out to create one (Hudak et al., 2007). Judging by Haskell's success, they were right. Today, the language is perceived by many as the de facto standard for learning about functional programming and its applications.

Haskell's main contributions to the field are monadic side-effects (Wadler, 1993) and ad-hoc polymorphism achieved with type classes (Wadler & Blott, 1989). The language also helped popularize many features and idioms, many even finding their use outside of the functional programming space. Examples include sections, type signatures, pattern matching, currying, and function application by juxtaposition (Hudak et al., 2007).

Haskell's influence in Func can be seen in its semantics (e.g., curried functions, ad-hoc polymorphism based on type classes), its syntax (most notably the composition operator explored in 6.2.3) and its type notation (i.e., the syntax for describing type schemes described in 5.3.2). Since I did not have access to an SML compiler, Haskell also played an important practical role in implementing Func's compiler, as I would often verify the validity of Func's semantics by comparing its behavior to Haskell.

2.3. JavaScript and Its Ecosystem

Brendan Eich developed JavaScript in only ten days in 1995. because his employer, Netscape, was in a rush to beat Microsoft in a race for market dominance (Crockford, 2008). They wanted to be the first ones to support interactive websites through a simple and familiar programming language, so they told Eich to *put Scheme in the browser and make it look like Java* (Cassel, 2018). In other words, JavaScript was supposed to be a small language for performing trivial tasks on webpages, approachable to people who knew nothing about software development. In Eich's own words, *It's just this sort*

of silly little brother language, right? The sidekick to Java (Cassel, 2018). Of course, this was not quite how things turned out.

Today, JavaScript is the world's most popular programming language. It is used everywhere, from its original domain of animating web pages to embedded programming. Jeff Atwood, the co-founder of Stack overflow, famously said in 2007. that *any application that can be written in JavaScript, will eventually be written in JavaScript* (Atwood, 2019) and, sure enough, his prophecy is only becoming more accurate as time passe. The fact that the language managed to become so popular despite being plagued by problems both large and small tells a great deal about its power[2].

According to Douglas Crockford, JavaScript was the first language to introduce function-based lexical scoping into the mainstream programming space (Crockford, 2008). It remains the only widely-used programming language that features prototypical inheritance. In a sense, JavaScript has been a great ambassador for concepts found in academic programming languages (e.g., Haskell), such as first-class functions and closures.

JavaScript's great expressive power and versatility made it an ideal compiler target, leading to many compile-to-js languages aiming to fix its shortcomings while remaining in the same ecosystem. Func also fits in this category. As Func's foundation and compiler target, JavaScript has been a significant influence on its design (both in terms of syntax and semantics, made clear in 3.2) and, of course, on its run-time behavior. After all, Func's run-time *is* JavaScript.

JavaScript's problems (more precisely, its lack of type safety and an overwhelming presence of dangerous features and edge cases) were the single most important motivator for creating Func.

2.3.1. TypeScript

Microsoft started developing TypeScript in 2010. to address JavaScript's problem with type safety and a complete lack of compile-time type verification (IDG, 2012). The language is most accurately described as "a statically checked superset of JavaScript." Despite being a relatively new language, TypeScript has seen great success and has all but overtaken JavaScript in its own ecosystem. The engineers at Microsoft made the wise choice of not making a new language but instead extending an existing one. This decision, together with TypeScript's use in AngularJS and Angular2, significantly accelerated its mass adoption.

[2] And yes, JavaScript being the only option for web programming did not hurt either.

However, that same decision of not being too revolutionary acts as TypeScript's largest limitation in ways that will become clear in Chapter 3. Func aims to avoid such limitations by approaching the problem from a more radical and less marketable perspective – building a new language from scratch.

TypeScript has also influenced Func's design in numerous ways. However, TypeScript's most notable contribution to this project remains its crucial role in implementing Func's compiler.

2.3.2. Flow

Flow is an open-source static type checker for JavaScript backed by Facebook (Flow, 2020a). It once represented the main competition to TypeScript, but TypeScript has since cemented its role as the statically checked improvement to JavaScript.

Despite the low market share, Flow remains interesting because it attempts to adapt the Hindley-Milner type system to a very complex multi-paradigm programming language without forcing any limitations upon it. Still, the project is restrained the same way TypeScript is – radical progress is hampered by Flow's insistence on compatibility with JavaScript (a problem Func aims to eliminate).

2.3.3. PureScript

PureScript is a strongly-typed purely functional programming language that compiles to JavaScript (PureScript, 2020). It aims to be a Haskell clone for the JavaScript ecosystem.

The language features most of what Func hopes to achieve. However, it aims towards a more academic crowd, while Func aims to be approachable for anyone coming from JavaScript.

PureScript has been crucial for developing the compiler, mostly by providing inspiration and ideas for the code generation process.

2.3.4. Coffescript

CoffeeScript was a reasonably popular compile-to-js language that appeared in 2009. and wanted to make programming in JavaScript more practical. As stated by the author, its aim was to *expose the good parts of JavaScript in a simple way* (Ashkenas, 2009). The language initially saw a great deal of success but was ultimately made obsolete by ES2015. The ES2015 update of JavaScript's standard natively solved many of the problems that actec as CoffeeScript's main selling points.

Regardless of its irrelevance in the present day, CoffeeScript still managed to influence Func's syntax (e.g., the lambda expression described in 6.2).

2.4. Nim

As described by its author, Nim is a statically typed compiled systems programming language that combines successful concepts from mature languages like Python, Ada, and Modula (Rumf, 2016).

Nim is an exciting modern programming language, but most of its impressive properties have little to do with this thesis's scope. While it did inspire some of Func's syntax (e.g., named function definitions explained in 6.8 and operator references described in 6.2.3), its main contribution is that it was the first language that got me interested in programming language design.

3. Design Goals and Concepts

This Chapter details the motivation behind Func and justifies most of the language's design decisions. Several syntactic decisions, however, are deferred until Chapter 6 because they depend either on the language's fundamental model established in Chapter 4 or its type system constructed in Chapter 5.

Section 3.1 explains the reasoning behind Func's general design goals. Section 3.2 sets the path for implementing those goals by deciding on Func's essential theoretical properties. Based on the goals defined in 3.1 and the properties set in 3.2, Section 3.3 chooses the most appropriate paradigm for the language. Finally, Section 3.4 goes through different features of various programming languages and selects the ones appropriate for Func, considering the limitations imposed by the previous three sections.

3.1. General Design Goals

Func's primary design goal is to provide a more powerful, type-safe layer on top of JavaScript. Power, in this context, primarily refers to the power of abstraction – the language should allow programmers to think at a higher abstraction level compared to when programming directly in JavaScript. Func aims to be a general-purpose language focused on data processing and describing functional relationships.

Subsection 3.1.1 explains why I decided to design a syntactically and semantically new language instead of creating a static preprocessor for JavaScript (as Flow does) or build on top of the existing language (as TypeScript does). Since type safety is one of the primary motivators for Func, Subsection 3.1.2 lists the type system's desired properties and explains how they relate to common pitfalls.

3.1.1. Why a New Language

The industry has demonstrated a demand for fixing JavaScript's flaws on numerous occasions, and there has been a steep increase in projects working towards this goal

in recent years. Some projects attempted to mitigate the problems with preprocessors and linters (e.g., Flow, ESLint). Others approached it by designing new languages on top of JavaScript (e.g., TypeScript). Finally, there were even attempts at implementing technologies meant to replace the entire ecosystem (e.g., Dart[1]). The second approach turned out to be the most lucrative one by a considerable margin – TypeScript has all but taken over the entire ecosystem in just eight years (TypeScript, 2014).

TypeScript owns most of its success to the design decision of exclusively adding features and improvements to JavaScript without changing anything about the existing language (i.e., all valid JavaScript must be valid TypeScript). However, this decision also imposed substantial limitations, preventing Microsoft from changing some of JavaScript's most problematic traits – a problem that would not exist had they decided to implement an entirely new language. Flow, TypeScript's competitor on the market, suffers the same limitations.

Other projects, such as CoffeeScript, PureScript, and Elm, did decide to create an entirely new language. Unfortunately, these projects were either made obsolete soon after gaining traction or never saw much success at all. CoffeeScript gradually started falling out of use with the appearance of ES2015. With this release, most of CoffeeScript's added features became redundant. PureScript and Elm are both purely functional languages and, as such, only manage to attract mostly academic audiences.

Finally, the JavaScript ecosystem offers an abundance of presentational client-side libraries and frameworks, such as React [2], Angular [3], Vue [4], or even jQuery [5]. These technologies concentrate on presenting data to the user and providing abstractions over the Document Object Model (DOM). While very useful and influential in their own right, they do not attempt to correct JavaScript's shortcomings. They only try to make application development in the existing language more pleasant. Again, this is for a good reason, as any framework's popularity depends on fast adoption by the industry.

All mentioned projects almost exclusively focus on enhancing JavaScript as a web language. There have not been many attempts to use JavaScript in the way programmers use Python (i.e., data processing and manipulation, scientific calculations). This space is still mostly undiscovered and could be an excellent fit for the language: JavaScript and Python possess similar semantics, JavaScript is generally more performant, and

[1]The Dart team later decided to change their strategy and compile to JavaScript due to the project being criticised for fragmenting the web (Bak & Lund, 2020).

[2]The React project is available at `https://reactjs.org/`

[3]The Angular project is available at `https://angular.io/`

[4]The Vue project is available at `https://vuejs.org/`

[5]The jQuery project is available `https://jquery.com/`

integrating Python with JavaScript's ecosystem is challenging at the very least.

Considering the reasons above, I have decided to design a new programming language that:

- Compiles to JavaScript and easily integrates with JavaScript ecosystem
- Emphasizes JavaScript's strengths and mitigates its flaws
- Provides a cleaner interface to the language's powerful semantics, both syntactically and semantically
- Operates in the space of data processing and scientific computation

3.1.2. Type Safety

JavaScript's flexible type system provides no safety guarantees before execution time. I consider this undesirable for reasons elaborated in Section 5. The industry backs this claim as well – it has demonstrated the need for a type-safe language in the ecosystem through projects such as TypeScript, Flow, PureScript, and Elm.

Other successful programming languages (e.g., Java or C#), while providing static analysis with more type safety [6], do not have type systems as reliable as some state-of-the-art functional languages, such as Haskell and ML. The type system's unreliability is a general trend present in many languages, and this Chapter makes an effort to describe what the term implies in more detail. Type systems of most major programming languages exhibit two main issues (Fiedler, 2021): *unsoundness* and *undecidability*.

Unsoundness

The first problem type checkers often share is their *unsoundness*. Consider the following Java code (Norswap, 2014):

```
1 Dog[] dogs = new Dog[] { new Dog() };
2 Animal[] animals = dogs;
3 animals[0] = new Cat();
```

[6] The literature defines type safety as a binary property of a language as a whole, most often synonymous with type soundness. However, since this Chapter aims to provide a high-level overview of the language, it uses the term more loosely to mean "a set of language features that help prevent type errors," which roughly corresponds to the term's implied meaning in the non-academic programming community. The more such features a language has, the "more type-safe" it is.

Assuming that both `Dog` and `Cat` extend `Animal`, this program successfully satisfies the type checker. However, line 3 crashes with a type error during run-time. Such erratic behavior is possible because Java, like most mainstream languages, does not have a *sound type system* (Fiedler, 2021) [7]. Some type systems strive for soundness and accidentally lose it by adding a new feature with unforeseen implications (Elm, 2018). Others intentionally compromise on soundness in exchange for productivity. For example, TypeScript opted to sacrifice soundness in favor of practicality and easy adoption. The language designers even explicitly list *a sound or "provably correct" type system* as one of TypeScript's non-goals (TypeScript, 2014). Therefore, it makes sense to take a different approach with Func and strive for a provably sound type system, one similar to those found in ML and Haskell.

Undecidability

The second problematic but common property of popular type systems is their *undecidability*. Consider an example in Java (Norswap, 2014):

```
1  public class Crash {
2      static class L<T> {}
3      static class C<P> extends L<L<? super C<C<P>>>> {}
4      L<? super C<Byte>> crash = new C<Byte>();
5  }
```

The presented code crashes the compiler with a stack overflow error – the type checker gets trapped inside an infinite loop when it attempts to verify that the code satisfies all type constraints. This behavior is not caused by a bug in the compiler but is a consequence of the type system's inherent undecidability [8]. Most widely-used type checkers, including Java's, are undecidable (Fiedler, 2021). Some language designers focus on maximizing the type system's expressive power without paying much attention to its mathematical properties, often resulting in the type system becoming undecidable by accident[9]. Sacrificing decidability for expressiveness is a legitimate trade-off – undecidable behavior generally occurs as a rare edge case, and most users never experience it. TypeScript was designed with this sentiment in mind and soon ended up with a very expressive but undecidable type system (Fiedler, 2021). I do not

[7] Soundness is formally defined in 5.2.2, this Chapter considers it from a mostly informal standpoint.

[8] This Chapter only explores the practical implications of undecidability. Section 5.2.1 later formally defines the term.

[9] Such an accident recently happened to Swift (Pestov, 2020).

consider the type system's decidability as necessary as its soundness. Regardless, I will attempt to implement a decidable type checker for Func. I do not expect a significant loss in expressiveness on account of maintaining decidability, considering there are type systems that are both decidable and highly expressive (e.g., Haskell).

Conclusion

Func aims to have a mathematically provable type system with the following properties:

1. **Decidability** - The type checker always decides whether the program has type errors in finite time.

2. **Soundness** - A program that successfully type-checks at compile-time is guaranteed not to have any type errors at run-time (i.e., there are no false positives).

3. **Completeness**- The type checker accepts all well-typed programs (i.e., there are no false negatives).

I consider the first two properties to be strong requirements. Completeness is desirable but is not a requirement (as long as its absence does not hinder the language's capabilities[10]). All mathematical terms used to describe the type system are formally defined in Section 5.2. They are later proven for Func in Section 5.4.

3.2. Exploring Language Properties

This Chapter looks at the properties of several popular programming languages today and throughout history. I will examine each property to see whether it proves desirable and sensible in the context of Func.

The design is aided (but also constrained) by JavaScript's semantics and pragmatics (e.g., single-threaded run-times). It is technically possible to forgo JavaScript's semantics altogether and build an entirely different model on top of them. However, such an approach not only goes against the initial design goal of providing a cleaner interface to JavaScript but would also result in a waste of resources – JavaScript is expressive enough to naturally support just about any useful paradigm (more on this in 3.3). Therefore, I have decided to embrace the underlying language and take its semantic properties into account when making design decisions.

[10]It is important to distinguish the completeness of a type system as a whole from the completeness of its type inference algorithm. The distinction is made clear in Section 5.2.

This Chapter looks at the language from a relatively high-level perspective (i.e., like most programmers would in their day-to-day work). Most areas and concepts are described very pragmatically, without paying much attention to academic terminology and taxonomies. The subsequent chapters describe some of the mentioned concepts formally and in more detail.

Many arguments I make when justifying my decisions could be dismissed as "appealing to the people." However, since I aim to build a language with properties and features proven to be successful in the community, I consider popularity highly relevant.

Subsection 3.2.1 decides on various properties of Func's type system by exploring relevant design decisions and their consequences in other languages. Subsection 3.2.2 decides whether Func's semantics will be strict or non-strict. Subsection 3.2.3 explains why Func must be a compiled language. Subsection 3.2.4 analyzes memory management and garbage collection. Finally, Subsection 3.4 goes through different features from various programming languages and picks those appropriate for Func.

3.2.1. The Type System

When programmers discuss type systems, they mostly categorize them using a handful of vaguely defined dimensions. Each segment explores one of them.

Statically or Dynamically Checked

The first dimension recognizes whether the language checks the types predominantly statically or dynamically (i.e., during run-time or compile-time). Examples of languages that use what people would consider a dynamic type system are Python, JavaScript, Elixir, and Ruby. Statically typed languages include C, Java, Haskell, and TypeScript. Since statically checked languages cause type errors at compile-time, while dynamically checked ones cause them at run-time, the former are safer and more robust than the latter.

The main arguments for using dynamically checked languages are expressiveness, flexibility, readability (less code), and better writing speeds. However, utilizing these selling points' full potential is often discouraged and considered a bad programming practice. For example, programmers generally advise against using the same variable to hold values of different types, even though dynamic languages allow it. As the codebase starts increasing in size, missing type annotations stop adding to readability and start detracting from it. Code writing speed only matters in quick one-time scripts meant to

be run once and forgotten. In all projects with longer lifespans, developers spend a lot more time reading code than they spend writing it.

Therefore, it appears that statically typed languages almost always turn out to be a better choice, with the exception being small scripts where specifying types would pose an unnecessary overhead and hinder productivity (e.g., shell scripting). Whenever building a complex system, I would recommend choosing a statically checked language. The JavaScript ecosystem has all but proven this point. Originally meant to provide flexibility and expressiveness, JavaScript was designed as a dynamically typed language. The client-side application space has grown tremendously since then, and JavaScript, along with its dynamic type system, was proven inadequate and difficult to scale. Today, developers rarely start building large projects using plain JavaScript. Most companies use a transpiled language (e.g., TypeScript) or a static type checker (e.g., Flow). Further proof comes from Python's ecosystem, where the `typing` module introduced in version 3.5 has been very well received by the community.

Verdict: With the above arguments taken into consideration, I have decided to design and implement Func as a statically checked language.

Manifest or Inferred

Assuming the language features static type checking (decided and justified in the previous segment), it remains to decide whether to use a manifest typing scheme or implement type inference.

Manifest typing, also known as explicit typing, requires the programmer to write type annotations for each variable. It is by far the prevalent approach, as most mainstream statically typed languages generally rely on explicit type annotations, with only occasional constructs supporting type inference. Examples of languages predominantly relying on manifest typing include C, Java, and Go. To understand manifest typing, consider an example in Go:

```
1  func square(x int32) int32 {
2      return x * x
3  }
4
5  square(5)   // works
6  square("5") // type error
```

The function `square` has to accept a 32-bit integer. The compiler knows this thanks to the programmer's type annotation and raises a type error on line 6.

Type inference, also known as implicit typing or type reconstruction, is a mechanism for automatically detecting the types of expressions in programming languages (Hegemann, 2019). A language with type inference can require varying amounts of type annotations. Some languages require the programmer to annotate most of the code, providing type inference only when defining local variables. One such example is Go (Fiedler, 2021). Other languages can infer types in most cases but sometimes require the programmer's input. TypeScript, for example, features a very advanced type inference engine. The programmer only has to annotate the types of function parameters and occasionally step in when the compiler struggles to infer a complicated expression. Finally, some languages feature full type inference – they can infer all types automatically, type annotations are entirely optional and used only for documentation purposes. Having such a powerful inference engine puts several constraints on the language's design. Precisely, most compilers that work on this level base their type inference algorithms on different forms of lambda calculus (Pierce, 2002), which is why full type inference almost always entails a functional programming language. Haskell, ML, and Flow all feature full type inference. To understand type inference, consider the same example in JavaScript with Flow:

```
1  function square(x) {
2      return x * x
3  }
4
5  square(5)   // works
6  square("5") // type error
```

Since multiplication is an operation defined only for numbers, the compiler infers that parameter `x` must be a number. It also knows that the result of multiplying two numbers is a number, meaning that function `square` necessarily has the type $Number \rightarrow Number$. At this point, the compiler can check lines 5 and 6 and detect the error.

As mentioned when discussing the disadvantages of dynamically checked languages, type annotations are often useful – they document the code and allow the programmer to ensure it works as intended. When working with a third-party library, it is helpful to have type signatures for its public API instead of being forced to infer the types from the library's implementation.

However, manifest typing also creates verbosity and prolongs development time. When explaining Go's design decisions, Rob Pike nicely described the problem using a quote from Dick Gabriel (Pike, 2010):

> *I'm always delighted by the light touch and stillness of early programming languages. Not much text; a lot gets done. Old programs read like quiet conversations between a well-spoken research worker and a well-studied mechanical colleague, not as a debate with a compiler. Who'd have guessed sophistication brought such noise?*

The lecturer referred to then-used statically typed server-side languages, such as C++ and Java. As the program's complexity grows, type annotations become increasingly difficult to understand and can even start taking away from its overall readability. Trivial type annotations create unnecessary noise. For example, consider the following Java code:

```
1  String name = "John";
2  Map<String, Integer> nameMap = new HashMap<Integer, String>();
3  Integer index = map.get(name);
```

Now compare it to the equivalent code in Go, a language that supports type inference in variable definitions:

```
1  name := "John"
2  nameMap := make(map[string]int)
3  index := nameMap[name]
```

I consider the Go version more readable – all types Java requires the programmer to annotate explicitly are trivial to infer for both humans and computers. In such cases, explicitly written types provide no additional or helpful information. The programming community generally agrees with this statement, which explains why modern versions of Java also allow type inference in variable definitions:

```
1  var name = "John";
2  var nameMap = new HashMap<Integer, String>();
3  var index = map.get(name);
```

Verdict: Taking all of type annotations' presented benefits and shortcomings into consideration, I have decided to focus on type inference. However, programmers will also have the option to annotate their code, either for documentation purposes or to restrict inferred types.

Nominal or Structural

Another significant attribute of a type system is whether it is structural or nominal. All types have names and structures, and a static type checker can use either of those when comparing types. *Nominal typing* (i.e., name-based type system) is checking types against names, whereas *structural typing* (i.e., structural-based type system) is checking types against structure (Flow, 2020b).

Nominal typing is the more popular discipline. Java, Swift, and Haskell all primarily use nominal typing. Consider the following Haskell code:

```
1  data A = A Int
2  data B = B Int
3
4  a:: A
5  a = B 3
```

Line 5 produces a type error even tough types `A` and `B` are structurally equivalent. This is a feature of nominal type systems. Java exhibits similar behavior – the compiler does not allow interchangeable use of different classes, regardless of their internal structure.

Despite still being the less popular option, structural type systems have started gaining traction in recent years, mostly thanks to Go and TypeScript. These two structurally typed languages cemented their respective roles as notable technologies in server-side and client-side development. Consider the following TypeScript code:

```
1  type A = { n: number; }
2  type B = { n: number; }
3
4  const a: A = { n: 3 };
5  const b: B = a
```

Line 5 does not produce a type error. Since types `A` and `B` have the same structure, the compiler treats them as the same type.

The main arguments favoring nominal type systems come down to intuition (i.e., humans differentiate things by their names) and preventing accidental type equivalence (i.e., semantically unrelated types accidentally having the same structure). However, these properties come with a cost of reduced flexibility. When using nominal type systems, a type `T` can only be a subtype of type `U` if the programmer explicitly declares so in its definition. It is also not possible to create supertypes without modifying the

existent subtypes. Structural type system posses no such limitations, thus reducing dependencies and increasing flexibility (Pierce, 2002).

Verdict: The implications of choosing between a nominal and a structural type system are significantly less profound than those introduced by the choices made in previous segments. Therefore, I have decided not to insist on either option before designing the language. I will choose the one that better fits the model imposed by other constraints during the design process.[11]

Strong or Weak

Programming languages are often colloquially referred to as being *strongly typed* or *weakly (loosely) typed*. There are no precise technical definitions of these terms. Still, the categorization most often relates to how the language handles type conversions and whether it provides a mechanism for bypassing its type system.

Since these terms are poorly defined, programmers generally do not them as clear-cut categories but rather as a specter. For example, Haskell is generally considered more strongly typed than Java, while Java's type system is considered stronger than C's.

The phrases are not only poorly defined, but they are also wrong – the properties they describe have nothing to do with the "strength" of the type checker, but rather with the nature of the backing run-time (Krishnamurthi, 2015). This claim is enforced by the fact that the language's "weakness" does not depend on whether the types are checked statically or dynamically (e.g., Python is considered to have a stronger type system than C despite being dynamically typed).

Despite all issues with their meaning, these terms still enjoy widespread use. Since this Chapter aims to describe and justify the design decisions from the user's perspective, I will explore them in more detail by examining the two traits that most often label a language as weakly typed: *implicit type conversions* and *type system bypassing*.

Implicit Type Conversion Type conversion means converting a value from one type to another (Pierce, 2002). The term often used to denote explicit type conversion is *type casting*[12]. Implicit type conversion (i.e., conversion forced by the way a value is used, performed automatically by the compiler) is generally referred to as *coercion*.

[11] As described in Section 5.3, a nominal type system turned out to be a better fit.

[12] When talking about languages outside the C family (most notably ones from the ALGOL family), conversion and coercion refer to distinctly different concepts. *Conversion* refers to changing a value from one representation to another. *Cast* refers to explicitly telling the computer how to view and treat the value, without changing its representation.

Most programming languages support coercion, at least to some extent. Notable examples of more conservative languages include C, C++, Java, and C#. Most of the time, these languages will complain or throw an error when the type does not fit the context, but they will allow conversions they deem "intuitive enough." For example, Java will require that all expressions inside an `if` statement evaluate to a `boolean` value, but will happily perform automatic autoboxing on primitive types, as well as silently convert integers to floating-point numbers. Other languages, such as JavaScript and Perl, take this idea a step further and define procedures for converting a value of any type to any other type depending. The language then implicitly calls a conversion procedure appropriate for the context. The following example demonstrates coercion in JavaScript:

```
1  const str = "2" + 3; // "23"
2  const num = "2" * 3; // 6
```

The variable `str` contains the value `"2"` of type `string`, while the variable `num` contains the value `6` of type `number`. When adding two different types, JavaScript, in general, coerces both values into strings. On the other hand, the multiplication operator forces a numeric context and, in general, coerces all operands into numbers[13].

Coercion, especially the more liberal kind, has a bad reputation in the programming community. Languages that feature it are often criticized and ridiculed, and the majority of projects written in weakly typed languages use linters to prevent programmers from using the feature. This constraint is introduced under the rationale that *if a feature is sometimes dangerous, and there is a better option, you should always use the better option.* (Crockford, 2008), and coercion can be both dangerous and unpredictable. We can further classify coercion into *implicit* coercion and *explicit coercion*..

Implicit coercion refers to hidden type conversions with non-obvious side effects that implicitly occur from other actions. In other words, implicit coercion is any conversion that is not obvious. Explicit coercion refers to type conversions that are obvious and explicit (Simpson, 2015) (i.e., forcing the language to perform an implicit conversion in a way obvious to the reader). Explicit coercion is not received as poorly as implicit coercion but is still often considered undesirable. Naturally, the difference between the two terms is mainly semantic and inherently subjective, and, since they both rely on the same mechanism, there is no need to explore them further.

[13]For more precise coercion rules, see the official specification at `https://262.ecma-international.org/11.0/#sec-addition-operator-plus`.

Despite all the bad press it gets, coercion does have its benefits, particularly when used explicitly. It is a flexible, fast, and convenient way of changing types built into the language itself. However, much like the dynamic type systems explored in one of the earlier segments of 3.2.1, history has proven that coercion tends to get in the way more often than it helps. An excellent indication is that almost all languages featuring it (Perl, JavaScript, C) were designed more than 25 years ago, while modern languages (e.g., Go, Rust, Swift) generally opt for "the stricter, the better" philosophy. For instance, the Go programming language does not even allow implicit conversion from a 32-bit integer to a 64-bit one.

Verdict: Func will not allow implicit type conversions. The mechanism's drawbacks greatly outweigh its benefits. Func's design goal is to provide a cleaner interface to JavaScript, and the one thing most of the JavaScript derivatives agree on is minimizing implicit type behavior.

Bypassing the Type System Some languages, the most prominent one being C, employ type-safety in limited contexts while at the same time providing mechanisms to bypass the type system altogether. In C, a programmer can accomplish this using unchecked casts and void pointers.

Rust contains a more modern and higher-level example of this mechanism. By default, the language ensures full memory and type safety at compile-time. It does, however, allow the programmer to write potentially dangerous code (e.g., dereferencing raw pointers) provided they use the `unsafe` keyword. With `unsafe`, the programmer forgoes Rust's memory safety guarantees to achieve the flexibility needed for low-level systems programming (Rust, 2018).

Another modern example comes from TypeScript. The programmer can opt-out of the language's type system by declaring a variable's type to be `any`, or by not declaring anything at all – as this is the language's default behavior[14]. Of course, TypeScript had to be designed like this – it was aiming to be a superset of JavaScript, meaning that its compiler had to accept all valid JavaScript programs. Func has no such limitation and can afford to be more strict.

These issues do not fall under the implicit type system discussion since the programmer must explicitly use unsafe mechanisms. However, they nevertheless contribute to the language's overall type "weakness."

Verdict: Func will provide no way of bypassing its type system. One of its design

[14] Almost all typescript projects do not rely on this and add the `-noImplicitAny` flag to the compiler, thus forcing the programmer to be explicit when bypassing the type system.

goals is providing a type-safe layer on top of JavaScript. Including a backdoor is not required by its anticipated use cases and would prevent Func's type system from being decidable and sound.

3.2.2. The Semantics

A function f is said to be non-strict if, when applied to a non-terminating expression, it also fails to terminate. A non-terminating expression, also called a bottom expression ($\perp$), is an expression that does not return a normal value, either because it causes an unrecoverable error (e.g., division by zero) or because it loops infinitely (Haskell, 1998). Formally, function f is strict if and only if $f(\perp) = \perp$. A non-strict function is a function that is not strict.

A programming language has strict semantics if and only if all user-defined functions [15] are strict. Such languages are also called *strict programming languages*. A *non-strict programming language* is any language that is not strict (i.e., users can define non-strict functions). Most of the widely used programming languages fall into the strict category (e.g., Python, JavaScript, Go, Java). Consider an example in Python:

```
1  def bottom(n):
2      return bottom(n)
3
4  def f(x):
5      return 10
6
7  f(bottom(1))
```

The function `f` does not internally use its parameter. The interpreter nonetheless evaluates the argument, causing an infinite loop. This example demonstrates Python's strict semantics and applies to most programming languages. The same code in Haskell behaves differently:

```
1  bottom n = bottom n
2  f n = 10
3  f (bottom 1)
```

[15] It is crucial to emphasize that the definition focuses exclusively on user-defined functions. Strict languages often allow non-strict built-in functions in the form of logical and conditional operators, a pattern also known as short-circuit evaluation.

Since `f` does not use the parameter's value internally, the language chooses not to evaluate it. Although `bottom 1` does cause an infinite loop on evaluation, the program immediately terminates with a non-bottom value, 10. In Haskell, the constraint $f(\bot) = \bot$ does not generally hold. Therefore, Haskell is a non-strict language.

Non-strict languages are mainly interesting because they allow lazy evaluation (i.e., expressions do not get evaluated before they are used to produce a value) (Hudak, 1989). This feature can be useful and harmful, depending on the specific use case. Programming languages used in the industry are predominantly strict (e.g., Go, Python, JavaScript, Java), while non-strict languages mainly find their uses in academic environments (e.g., Haskell, Miranda, Clean). This trend suggests that strict languages are better suited for practical use cases.

Verdict: Due to the practical considerations presented above, I have decided to make Func a strict language. The decision is reinforced by the fact that Func's primary compiler target is JavaScript, which also features strict semantics. As noted in the introduction to Section 3.2, I believe that embracing JavaScript's underlying properties makes more sense than attempting to work around them.

3.2.3. Compiled or Interpreted

If Func wants to satisfy its design goal of having a statically checked type system, it must be a compiled language. The code will be statically checked and compiled to executable JavaScript.

As far as compiling the generated code is concerned, JavaScript's specification does not define whether it should be compiled or interpreted. The official language overview describes the language in an interpreter-agnostic way (ECMA, 2015) and leaves the decision to specific implementation. The text will not be considering the generated code's behavior any further.

3.2.4. Memory Safety and Management

Since Func aims to be a high-level language, it is difficult to make a case for manual memory management. This feature is useful mainly in low-level and performance-sensitive programming, none of which fit the language's anticipated use cases. Of course, supporting real manual memory management was never really an option since JavaScript already features a mark-and-sweep garbage collector (MDN, 2020).

Verdict: Due to its high-level use cases and limitations imposed by the JavaScript run-time, Func will be a garbage-collected language.

3.3. Programming Paradigms

3.3.1. An Introduction to Programming Paradigms

A programming paradigm is an approach to programming a computer-based on mathematical theory or a coherent set of principles (Van Roy, 2012). In less formal terms, each paradigm offers its particular way of describing computation.

Some paradigms are more appropriate for specific problems than others, depending on the relationship between the problem and the paradigm's underlying model. For example, protocols and sequential algorithms are best described by an imperative language since the imperative paradigm perceives computation as a series of steps and mutations. The functional paradigm expresses computation through functions, types, mappings, and composition. Its essence is that programs are a combination of expressions (Allen & Moronuki, 2018). Therefore, it is very well suited for mathematical problems. Finally, when faced with many related data abstractions carrying an internal state, it is appropriate to approach the problem through an object-oriented perspective (Van Roy, 2012). The underlying model of computation for imperative languages is the Turing machine, while functional languages build on top of λ-calculus (further explored in Section 4.1).

A paradigm is a purely theoretical construct defined by a set of programming concepts. Each language realizes one or more paradigms, and most languages generally support at least two of them. JavaScript supports even more, with the three most clear-cut paradigms being functional, procedural, and prototype-based object-oriented programming. JavaScript's multi-paradigm foundation is the key reason why the language so generously lends itself to transpiling. Not all languages support multiple paradigms. A famous example of a single-paradigm (pure) language is Haskell, which exclusively supports functional programming.

3.3.2. Choosing the Appropriate Paradigm(s) for Func

Since Func aims to provide a clean way to define data transformations and functional relationships (a decision explained in 3.1.1), it should describe computation in a declarative way, as one would when using mathematical notation. The syntax should thus have to be heavily expression-based. The language should use statements only to state invariants (e.g., variable definitions, type declarations) and not as a means of describing computation. Expression-based syntax, declarative mathematical notation, and a high abstraction level all steer the language towards the functional paradigm, perhaps even

purely functional. I believe it makes sense to initially design the language as purely functional and add side effects when and if they prove themselves necessary.

Considering practical limitations (i.e., underlying paradigms supported by JavaScript), the language could have been either functional, object-oriented, or procedural. JavaScript imposes no limitations on the language's design in this regard – any of the three mentioned paradigms would have worked, including the ultimately selected functional paradigm.

Verdict: Func will be a purely functional language based on the λ-calculus[16].

3.4. Exploring Language Features

This section goes through specific language features proven to be generally well-received by the programming community and selects those deemed appropriate for Func's paradigm and use cases. Most of these features have been present for several decades, but some can be considered relatively new.

3.4.1. First-class Functions

Christopher Strachey coined the term first-class function to describe languages that treat functions as first-class citizens (Strachey, 1967). In more formal terms, a language is said to have first class functions if it treats functions the same way it treats any other value (i.e., allows passing them as arguments, assigning them to variables, storing them in data structures, returning them from functions, etc.) (Wikipedia contributors, 2020). For a language with first-class functions, a function is nothing more than an ordinary variable with a function type.

First-class functions are a necessary concept for functional programming. Other related terms, like callbacks or higher-order functions[17], all require the language to have first-class functions. All functional and most high-level multi-paradigm languages have these features, the examples being Haskell and ML for the former, and Python and JavasScript for the latter category. In fact, according to Douglas Crockford, first-class functions and lexically-scoped closures are JavaScript's most indispensable abstraction mechanisms (Crockford, 2008).

[16] More precisely, its fundamental model (Chapter 4) is essentially the non-typed λ-calculus with just enough extensions to make it practically usable, while its type system shifts the perspective towards a Turing-complete extension of the simply typed λ-calculus (Chapter 5).

[17] A callback is a function passed to another function (Crockford, 2008). A higher-order function is a function taking functions as arguments or returning them (Wikipedia contributors, 2020)

There is not a real downside to supporting first-class functions. Languages can choose to omit the feature if it does not fit with their intended paradigm (as Java does, for example), but it is primarily a matter of perspective, not a trade-off.

An important thing to consider from a practical perspective is that JavaScript already supports fist-class functions, meaning that Func can get them "for free."

Verdict: Since Func is a functional language, first-class functions are a necessity. JavaScript makes heavy use of the concept, meaning that the feature perfectly fits with Func's underlying semantics and should be trivial to implement.

3.4.2. Currying

Currying refers to transforming a function that takes multiple arguments into a sequence of functions each taking a single argument. It is very common in purely functional programming languages (e.g., Haskell) because it closely relates to λ-calculus (discussed in Section 4.1). If Func ends up being a purely functional language, currying will be mandatory. If not, I believe it still makes sense to implement a primitive for currying functions, as it significantly simplifies partial application.

Verdict: Func will include support for currying, either natively or through a special primitive.

3.4.3. Built-in Primitives and Literals

It is important to distinguish between *primitive values* and *literals*. For example, JavaScript includes literal notation for objects and arrays, but neither is a primitive value. On the other hand, the language provides no way of writing `symbol` literals, despite them being primitive values. Haskell and Python provide a literal notation for non-primitive tuples. This type does not even exist in JavaScript[18]

Primitive Values

Virtually all modern programming languages support a range of built-in primitive values. Java has booleans, characters, integers, unsigned integers, long unsigned integers, and many others. At the time of writing, JavaScript only has seven of them: `string`, `number`, `bigint`, `boolean`, `undefined`, `null`, and `symbol`. Since Func compiles to JavaScript, it makes sense to consider using the types JavaScript has to offer.

[18]Technically, JavaScript programmers sometimes refer to fixed-length arrays as tuples, but it is hardly the same thing.

Adding booleans is the easiest to justify, as they are present in virtually all programming languages.

JavaScript's `number` type is an IEEE 754-2008 double-precision floating point number. Many feel that this is not a reasonable default number format due to its precision problems. However, it is so widespread at this point that it most likely is not worth fighting, and implementing any other default number type in Func would pose a significant challenge. When precision is needed, one can use big integers (at least when performing integer calculations).

JavaScript does not have a character type, but one can easily be simulated using strings. Such a simulation is not an ideal solution in terms of efficiency, but performance is not of great importance, at least not at this point.

Symbols are very specific to JavaScript's inner workings and will most likely struggle to find a use case in Func.

Douglas Crockford once jokingly said that *people argue about whether a language should even have a bottom value, but no one thinks it should have two of them* (Crockford, 2008). JavaScript, however, does have two: `null` and `undefined`. Modern languages are increasing efforts towards banishing bottom values. I believe Func should follow in their footsteps.

Verdict: Func will include booleans, numbers, bigints, and strings (potentially as characters). It will not include symbols, `null` and `undefined`.

Literals for Common Structures

It makes sense to include literal notation for all primitive values. In addition to these literals, it also makes sense to add types proven practical and useful. More specifically: arrays, named records, and tuples. Adding support for more complex literals (e.g., dictionaries, sets) is also worth considering, but perhaps not in this early phase.

Verdict: Func will support tuple literals, array literals, and record literals. In the future, I will consider adding literals for other common data structures, such as sets and maps.

3.4.4. Pattern Matching

Pattern matching is another concept originating from functional programming languages, the most prominent example being Haskell. It is an especially expressive construct that fits nicely with algebraic data types and the declarative programming style. To understand pattern matching, consider a function for calculating factorials in Haskell:

```
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

The language attempts to "match" the argument with the given options, from top to bottom. The application `factorial 0` matches the first line (representing the base case), while all other arguments match the second generic line (and make a recursive call).

Although made famous primarily by functional programming languages, pattern matching also found its use in other paradigms (e.g., Rust is a multi-paradigm, mostly imperative language that features it). I do not see any downsides to implementing pattern matching, aside from the construct being notoriously challenging to type check correctly (Hudak et al., 2007).

Verdict: Since Func prioritizes the type system's provable properties, I will consider implementing pattern matching only after accomplishing all primary design goals. Pattern matching would surely make a handy feature, assuming I can implement it correctly.

3.4.5. Polymorphism

The term *polymorphism* refers to a range of language mechanisms that allow a single part of a program to be used with different types in different contexts. Type systems that allow a single piece of code to be used with multiple types are called polymorphic systems (Pierce, 2002). The term is generally divided into three major classes: *parametric polymorphism*, *ad hoc polymorphism*, and *subtyping* (Strachey, 1967).

Parametric Polymorphism

Parametric polymorphism allows a single piece of code to be typed "generically" [19] (Pierce, 2002). A parametrically polymorphic function handles values identically without depending on their type (i.e., the types *do not* affect the underlying logic). For example, a function for reversing a list only requires that its argument is a list, it does not care about the types of values inside the list. This is how a parametrically polymorphic (i.e., generic) `reverse` function might look in TypeScript:

[19] Which is why object-oriented languages sometimes use the term *generics* when referring to this concept.

```
function reverse<T>(array: T[]): T[] {
    return [...array].reverse();
}
```

The function `reverse` is said to be parametereized by `T` for all values of `T` (i.e., the type variable `T` can be substituted for any type without impacting the function's behavior).

Parametric polymorphism saves the programmer from defining the same procedure for all possible types. Not having this feature impairs the language significantly (as many Go programmers will most likely confirm).

Verdict: Func should strive to achieve parametric polymorphism. It is a useful feature with no apparent downsides.

Ad Hoc Polymorphism

In simple terms, ad hoc polymorphism allows the polymorphic construct to behave differently depending on the types it operates on (i.e., the types *do* affect the underlying logic). A classic example of such behavior is operator overloading. Consider an example in Java:

```
String string = "some" + "string";
int integer = 1 + 2;
double decimal = 1.5 + 2.8;
```

All three lines make a call to the same operator. However, adding strings does not constitute the same operation as adding integers (the same applies to adding double-precision floating-point numbers). Each call performs a different operation on its arguments.

Ad hoc polymorphism allows flexibility and increases expressiveness. It does, however, come with several downsides. Heavily overloading the same symbol in unorthodox ways makes the code more difficult to read. Another problem is handling the construct at a language level – designing a provably correct type system that exhibits ad hoc polymorphism in a reasonable manner has proven to be challenging, and a satisfying solution did not exist until relatively recently (Hudak et al., 2007). The Haskell committee finally solved the problem by inventing type classes (Wadler & Blott, 1989), a current state-of-the-art way to integrate ad hoc polymorphism with a provably correct type system. Type classes are a non-trivial extension to the Hindley-Milner type systems used in Haskell and other functional programming languages.

Verdict: Since Func aspires to be a functional language, it should support ad hoc polymorphism through type classes. This is a handy feature whose downsides are easily mitigated by responsible use.

Subtyping

Subtyping (or subtype polymorphism) is generally found in object-oriented languages and is often considered an essential feature of the object-oriented paradigm (Pierce, 2002). The term has to do with types being related to other types by the ability to substitute one for the other. This ability to substitute types can be defined with an explicit annotation (in nominal type systems) or through implicit structural requirements (in structural type systems). Mechanisms that leverage subtyping include virtual methods, double dispatch, and other generally object-oriented concepts.

The main arguments against subtyping are similar to those against ad hoc polymorphism – implicit and potentially unclear dynamic behavior.

Verdict: Since subtyping is almost endemic to object-oriented programming and Func will not be an object-oriented language, it will initially not feature subtyping.

4. Designing the Core Language

Chapter 3 explored Func's use cases and enumerated its desired properties. Given the language's planned usage in representing logical and mathematical operations, the logical choice was designing Func as a functional language. The next natural decision is basing Func on λ-calculus – a formal system that serves as a foundation to all functional programming languages (e.g., ML and Haskell).

Section 4.1 gives a short overview of lambda calculus and explains its foundational role in functional programming. Section 4.2 builds the language's core model. It does this by representing the terms from λ-calculus and its extensions as an abstract syntax tree (AST). The AST then serves as a starting point for defining the rest of the language – its syntax, semantics, and code generation. All these modules depend exclusively on the AST (i.e., the AST is the only shared dependency between different parts of the compiler)[1].

4.1. The λ-calculus

Alonzo Church introduced the λ-calculus in the 1930s as a way of formally defining computation (i.e., a mathematical process that in a finite amount of steps produces a result) (van Bakel, 2001). Like Turing machines, it formalizes the concept of effective computability, thus determining which classes of problems are solvable (Allen & Moronuki, 2018).

The λ-calculus has three basic components, or *lambda terms*: expressions, variables, and abstractions (Allen & Moronuki, 2018). The following three rules construct valid lambda terms [2]:

[1] The implementation of Func's AST can be found here.

[2] The thesis does not include a strictly formal definition of lambda calculus and its operations. It instead presents a mid-level overview of the system, which should be sufficient for understanding Func's semantics and type system. Formal definitions of all mentioned terms are available in the literature (e.g., (van Bakel, 2001))

1. A *variable* x is a valid lambda term.

2. If M is a lambda term and x is a variable, then $(\lambda x.M)$ is also a lambda term called *an abstraction*.

3. if M and N are lambda terms, then $(M \cdot N)$[3] is also a lambda term called *an application*.

When writing lambda terms, it is a common practice to omit letfmost and outermost parentheses, and abbreviate repeated abstractions (e.g., $(\lambda x.(\lambda y.(x \cdot y)))$ thus becomes $\lambda xy.x \cdot y$ [4]). The system defines two fundamental operations for achieving computation:

1. α-conversion - Renaming the bound variables in an expression. It is used to avoid name collisions and and does not change the abstraction's semantics:

$$\lambda x.M \rightarrow_{\alpha} \lambda y.M[x \mapsto y] \tag{4.1}$$

2. β-reduction - Applying a function to an argument:

$$(\lambda x.M) \cdot N \rightarrow_{\beta} M[x \mapsto N] \tag{4.2}$$

where $M[x \mapsto N]$ denotes a substitution operation (i.e., it stands for "substitute all occurrences of variable x inside term M with term N").

The λ-calculus is Turing complete and serves as a fundamental model of all functional programming languages – a purely functional language *is* the λ-calculus with some (mostly syntactic) extensions. Computations in λ-calculus are performed through a series of β-reductions (i.e., by applying functions to arguments). α-conversions mainly exist as a way of formalizing variable scoping and shadowing. They are mostly irrelevant for the rest of the discussion.

4.2. Constructing the Abstract Syntax Tree

When discussing the syntax of programming languages, it is useful to distinguish two levels of structure (Pierce, 2002):

– **The surface syntax** - The characters that programmers read and write, also called the external language.

[3] Most of the literature omits the dot, writing $M\ N$ instead of $M \cdot N$. I decided to leave the dot for clarity.

[4] This is an abstraction for performing function application. It applies the function x to the variable y.

– **The abstract syntax** - The internal representation of programs as labeled trees (Abstaract Syntax Trees or ASTs), also called the internal language.

This Section focuses on the abstract syntax – it describes Func's fundamental constructs and shows their respective representations in the AST. All grammars and syntactic notation introduced here serve only for describing Func's internal model and do not necessarily correspond with its actual syntax. All other aspects of the language, including its surface syntax covered in Chapter 6, build on top of the internal model this Section defines.

As decided in Chapter 3, Func is an expression-based language built on top of the λ-calculus. Therefore, almost all AST nodes represent expressions. Most of them directly correspond to lambda terms. The rest represent custom additions included to simplify the language and improve its usability.

There are six distinct expression-level nodes: *literals*, *identifier references*, *lambdas*, *applications*, *let expressions*, and *conditionals*. Figure 4.1 represents a generic expression node (i.e., a placeholder for any other expression-level node). The AST also includes two statement-level node: *the declaration* and *the function definition*. Finally, *the model* serves as a root node for all programs written in Func.

Subsection 4.2.1 explores the nodes that come directly from the λ-calculus (i.e., identifier references, lambdas, and applications). Subsection 4.2.2 lists and justifies Func's custom extensions (i.e., literals, let expressions, and conditionals). It also shows how these additions essentially do not change the language's theoretical model (i.e., the λ-calculus).

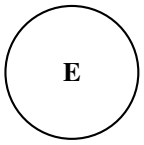


Figure 4.1: The AST representation of a generic expression. No such node exists in the AST – it is only used as wildcard to make the some of the figures more concise.

4.2.1. Representing the λ-calculus

Since Func is based on the λ-calculus, most AST nodes are merely representations of different lambda terms. These nodes include *identifier references*, *lambdas*, and *applications*.

Identifier references

Identifier references are expression-level AST nodes representing what λ-calculus refers to as variables.

As a theoretical system, λ-calculus does not differentiate between variables appearing in abstraction heads from those appearing in their bodies. However, from a practical standpoint, it makes sense to interpret variables in two different ways depending on the context:

1. In abstraction heads, variables are names. They denote where inside the body we should substitute for the argument during β-reduction (i.e., evaluation).

2. In abstraction bodies, variables are lambda terms (i.e., expressions)

Only the second case qualifies as an identifier reference. Since Func does not interpret parameter names as expressions, the first case does not exist inside the AST as a stand-alone node. Parameter names do not evaluate to anything. They simply serve as instructions for performing substitutions (more on this when discussing lambdas). Of course, the names are still saved as strings inside lambda nodes. Figure 4.2 shows an example of an identifier reference node.

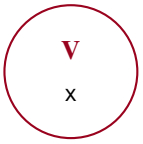


Figure 4.2: The AST representation of the lambda term x.

Lambdas

Lambdas are expression-level AST nodes that represent abstractions from λ-calculus, to whom they are semantically equivalent [5].

In practical terms, lambdas represent anonymous unary functions (unary because the λ-calculus limits all abstractions to take only a single argument). Functional programming languages then simulate multi-argument lambdas through currying. The expression $\lambda x.e$ can be interpreted as "given the operand x, the function evaluates and returns the expression e."

[5] I have decided to use the name "lambda" instead of "abstraction" because it is the ingrained way to call anonymous functions when talking about programming languages. The term abstraction can have a much more general meaning and might confuse the reader.

Lambdas consist of two parts: the *head* and the *body*. The head is merely a parameter name. The language uses it when evaluating expressions, but the head itself is never evaluated (only bound to the argument's evaluation result). The body is an expression (i.e., an AST node) evaluated whenever the lambda is applied. Figure 4.3 shows how lambdas look in the abstract syntax tree.

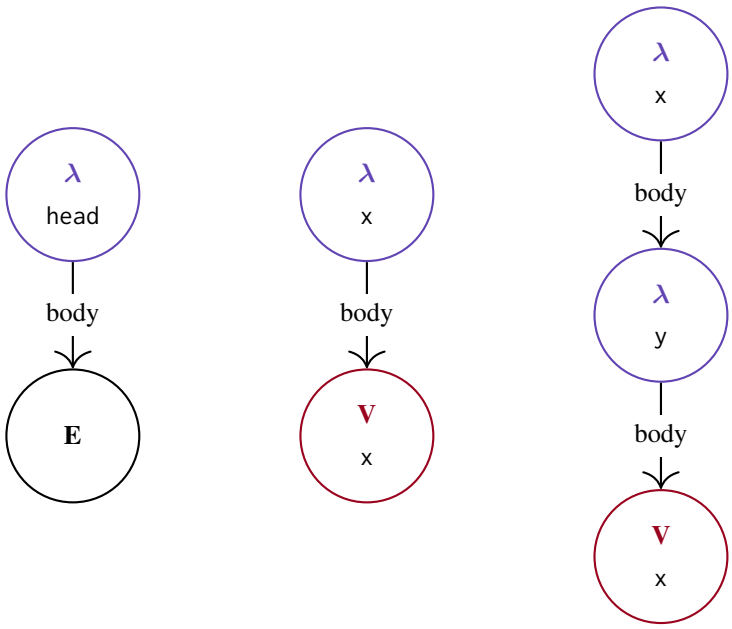


Figure 4.3: The AST representations of a generic lambda node (left), and concrete lambda terms $\lambda x.x$ (middle) and $\lambda x.\lambda y.x$ (right).

Applications

Applications are expression-level AST nodes that represent λ-calculus applications. An application takes two lambda terms, M and N, and produces a new term, the application of M to N. Term M can be interpreted as a function and term N as its argument. Applications in λ-calculus directly map to function applications in programming languages. Figure 4.4 shows the AST representation of an application.

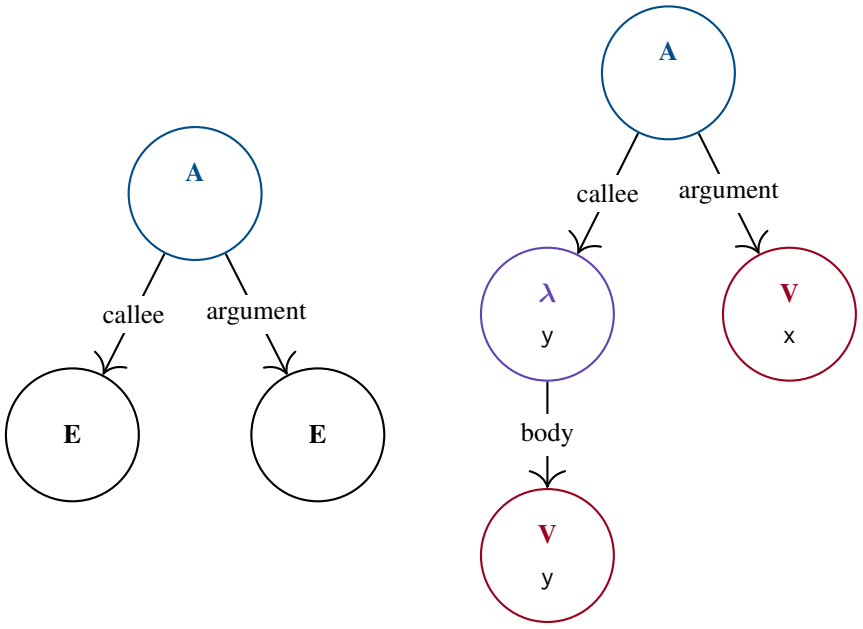


Figure 4.4: The AST representations of a generic application node (left) and the concrete lambda term $(\lambda y.y) \cdot x$ (right).

4.2.2. Extending the λ-calculus

Despite being Turing complete, λ-calculus does not make a practical programming language in and of itself. Its three main limitations are:

1. **The lack of useful values** (e.g., numbers, booleans, characters) - λ-calculus does not define such constructs, they have to be encoded using abstractions.

2. **The lack of shared named constants** - All referenced variables must be bound in the head of one of the enclosing lambdas.

3. **The lack of a type system** - λ-calculus does not posses a notion of types or of type safety.

These inadequacies, in an effort to turn lambda calculus into a practical programming language, led to the development of various patterns and similar but enhanced formal systems (e.g., the simply-typed λ-calculus[6]). Most functional programming languages, including Func, use these extensions to make programming cleaner and more efficient.

[6]The simply-typed λ-calculus "enhances" λ-calculus only in terms of type safety. The untyped λ-calculus is actually more powerful due to the simply-typed λ-calculus not being Turing-complete (Pierce, 2002).

This Subsection shows how Func resolves the first two points. It enumerates extensions to the λ-calculus I added into Func to address its lack of values and named constants. It then explains and justifies each extension's purpose and develops its representations inside the AST. The third point (the lack of type safety) is addressed in Chapter 5 when discussing the language's type system.

Let Expressions

The let expression is a standard extension to the λ-calculus present in many functional programming languages, including Func.

It is a purely syntactic addition to λ-calculus aiming to account for its lack of named constants without affecting its semantics. Since a Func program can only be an expression at this point (the AST does not yet possess the notion of a statement), it is possible to emulate named constants in program P using three simple steps:

1. Set aside a name for the constant (e.g., c) and decide on its desired value (e.g., V).

2. Wrap the program P with a lambda whose head is the chosen name c. The program now becomes $\lambda c.P$ (i.e., the name c becomes bound inside the program P).

3. Finally, apply the created lambda to the desired value, resulting in $(\lambda c.P) \cdot V$.

In short, to use c to mean V in P, one can say $(\lambda c.P) \cdot V$. The let expression is nothing more than a syntactically cleaner way to express this pattern:

$$(\lambda c.P) \cdot V \quad \equiv \quad \texttt{let } c = V \texttt{ in } P \tag{4.3}$$

Both expressions shown above can be read as "evaluate the expression V and bind the name c to the resulting value while evaluating expression P" (Pierce, 2002). Since let expressions simplify programming without changing the language's underlying model, I can safely add them to the AST. Figure 4.5 shows how the let expression looks in the AST and compares it with its pure λ-calculus equivalent[7].

[7] In reality, let expressions are not fully semantically equivalent to applications as the text here suggests. They carry one important semantic implication which is explained in detail in 5.4.3

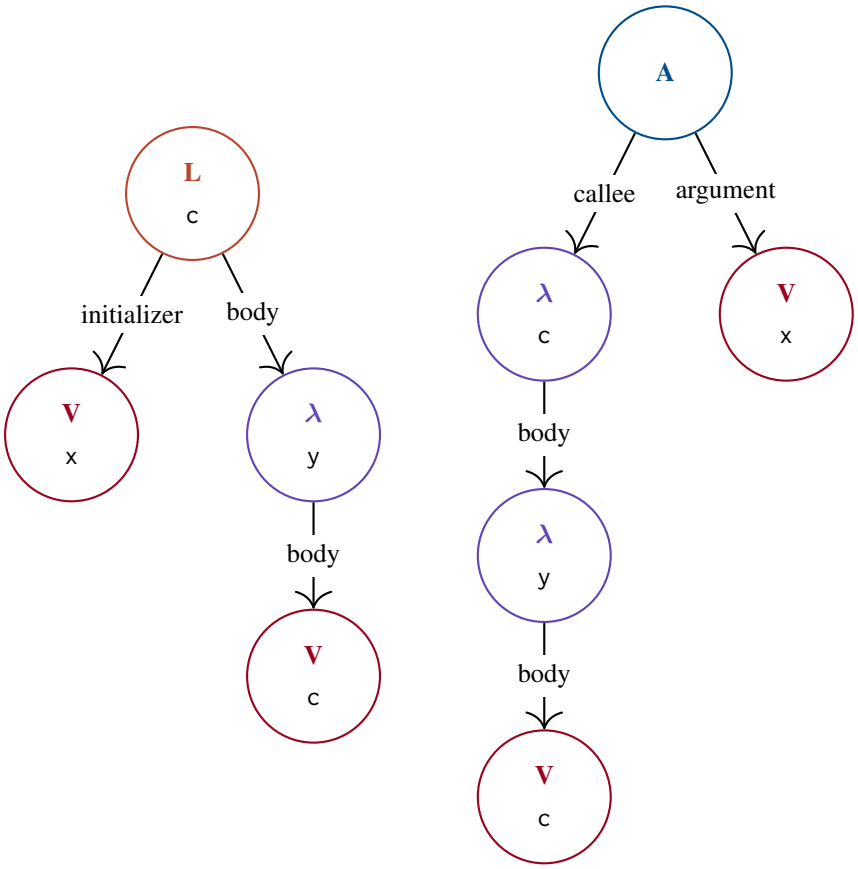


Figure 4.5: The AST representations of the expression `let` $c = x$ `in` $\lambda y.c$ is shown on the left. Its semantically equivalent lambda term, $(\lambda c.\lambda y.c) \cdot x$, is shown on the right.

Literals

Pure λ-calculus does not have a notion of values and literal. However, it is possible to easily encode such constructs using nothing but pure lambda terms: variables, lambdas, and applications. Equation 4.4 shows how to encode boolean values inside λ-calculus. These encodings are called Church booleans (Pierce, 2002).

$$\begin{aligned} true &= \lambda t.\lambda f.t \\ false &= \lambda t.\lambda f.f \end{aligned} \tag{4.4}$$

While they may seem unintuitive at first, the terms $true$ and $false$ can be viewed as representing boolean values in the sense that they can be used to encode logical operations (shown in equation 4.5) (Pierce, 2002). For simplicity and readability purposes, the equations assume support for named constants included by the previous

Segment.

$$\begin{aligned} and &= \lambda m.\lambda n.m \cdot n \cdot false \\ or &= \lambda m.\lambda n.m \cdot true \cdot n \\ not &= \lambda m.m \cdot false \cdot true \end{aligned} \tag{4.5}$$

These abstractions behave in the same way regular boolean operations would. Equation 4.6 shows the result of applying and to $true$ and $false$.

$$\begin{aligned} and \cdot true \cdot false &\equiv (\lambda m.\lambda n.m \cdot n \cdot false) \cdot true \cdot false \\ &\equiv \underline{(\lambda m.\lambda n.m \cdot n \cdot (\lambda t.\lambda f.f))} \cdot (\lambda t.\lambda f.t) \cdot (\lambda t.\lambda f.f) \\ &\to_\beta \underline{(\lambda n.(\lambda t.\lambda f.t) \cdot n \cdot (\lambda t.\lambda f.f)) \cdot (\lambda t.\lambda f.f)} \\ &\to_\beta (\lambda t.\lambda f.t) \cdot (\lambda t.\lambda f.f) \cdot (\lambda t.\lambda f.f) \\ &\to_\alpha^* \underline{(\lambda t_1.\lambda f_1.t_1) \cdot (\lambda t_2.\lambda f_2.f_2)} \cdot (\lambda t_3.\lambda f_3.f_3) \\ &\to_\beta \underline{(\lambda f_1.\lambda t_2.\lambda f_2.f_2) \cdot (\lambda t_3.\lambda f_3.f_3)} \\ &\to_\beta \lambda t_2.\lambda f_2.f_2 \\ &\to_\alpha \lambda t.\lambda f.f \\ &\equiv false \end{aligned} \tag{4.6}$$

It is possible to encode numbers, pairs, or any other value in a similar manner. Encoding values (e.g., numbers) also enables encoding functions operating on those values (e.g., $>$, $<$, $+$, $-$), as was the case with booleans and basic logical operations. This thesis will not go into more detail about encoding literals in λ-calculus. However, more examples, proofs, and explanations are available in the literature (Pierce, 2002).

From a practical standpoint, it makes sense to include top-level support for value literals, both in the language's syntax and semantics. In fact, most general-purpose functional programming languages do precisely that. Since it is possible to encode any value using nothing but pure λ-calculus, adding literals to the language does not infringe the underlying model's properties. Func features conventional support for values, their literals, and a handful of built-in functions that operate on them. The language semantically supports only four types of literals: booleans, numbers, big integers, characters, and arrays. All four have their respective nodes in the AST, as shown in Figure 4.6. Syntactically, the language also supports tuple literals and string literals (and plans to support named record literals in the future). However, to maintain the type system's proven properties, these are implemented as purely syntactic constructs and do not exist in the internal language. The proof of the type inference algorithm in Subsection 5.4.3 explains why such an implementation was necessary. In contrast, Subsection 6.2.3 elaborates further on their exact relationship with the internal model.

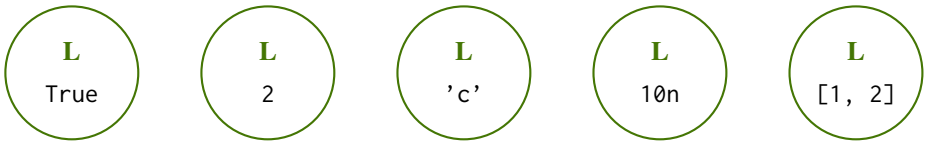


Figure 4.6: The AST representations of the boolean literal `True`, the number literal `2`, the character literal `'c'`, the big integer literal `10n`, and the array literal `[1, 2]`.

Conditionals

Conditional expressions (or statements) are fundamental building blocks of any programming language, functional or otherwise. As with literals, it is possible to encode them using nothing but pure λ-calculus. Equation 4.7 shows how to encode a conditional expression assuming the availability of Church booleans explored in the previous Segment:

$$cond = \lambda b.\lambda m.\lambda n.b \cdot m \cdot n \tag{4.7}$$

The head of the first lambda represents a boolean value. The two remaining heads represent the values returned if the boolean value is true or false, respectively. Equation 4.8 demonstrates how the term reduces to its second argument when the condition is $false$. It uses Church booleans and assumes the existence of named constants a and b:

$$\begin{aligned} cond \cdot true \cdot a \cdot b &\equiv \underline{(\lambda b.\lambda m.\lambda n.b \cdot m \cdot n) \cdot (\lambda t.\lambda f.t)} \cdot a \cdot b \\ &\rightarrow_\beta \underline{(\lambda m.\lambda n.(\lambda t.\lambda f.t) \cdot m \cdot n) \cdot a} \cdot b \\ &\rightarrow_\beta \underline{(\lambda n.(\lambda t.\lambda f.t) \cdot a \cdot n) \cdot b} \\ &\rightarrow_\beta \underline{(\lambda t.\lambda f.t) \cdot a} \cdot b \\ &\rightarrow_\beta \underline{(\lambda f.a) \cdot b} \\ &\rightarrow_\beta a \end{aligned} \tag{4.8}$$

Similarly, the term reduces to its third argument when the condition is $false$, as shown here:

$$\begin{aligned} cond \cdot false \cdot a \cdot b &\equiv \underline{(\lambda b.\lambda m.\lambda n.b \cdot m \cdot n) \cdot (\lambda t.\lambda f.f)} \cdot a \cdot b \\ &\rightarrow_\beta \underline{(\lambda m.\lambda n.(\lambda t.\lambda f.f) \cdot m \cdot n) \cdot a} \cdot b \\ &\rightarrow_\beta \underline{(\lambda n.(\lambda t.\lambda f.f) \cdot a \cdot n) \cdot b} \\ &\rightarrow_\beta \underline{(\lambda t.\lambda f.f) \cdot a} \cdot b \\ &\rightarrow_\beta \underline{(\lambda f.f) \cdot b} \\ &\rightarrow_\beta b \end{aligned} \tag{4.9}$$

Equations 4.8 and 4.9 prove that λ-calculus can support conditional expressions natively. Nevertheless, it is far more practical for a programming language to include a new term for the construct, thus supporting it in a cleaner and more efficient way. Taking after ML and Haskell, Func defines a top-level conditional expression: `if` c `then` t `else` f. Term c represents the condition, term t represents the expression evaluated when the condition is true, and term f represents the expression evaluated when the condition is false. Unlike equations 4.7, 4.8, and 4.9, the conditional expression assumes the existence of real boolean values and requires term c to reduce to one of them. Figure 4.7 shows the AST representation of a conditional expression.

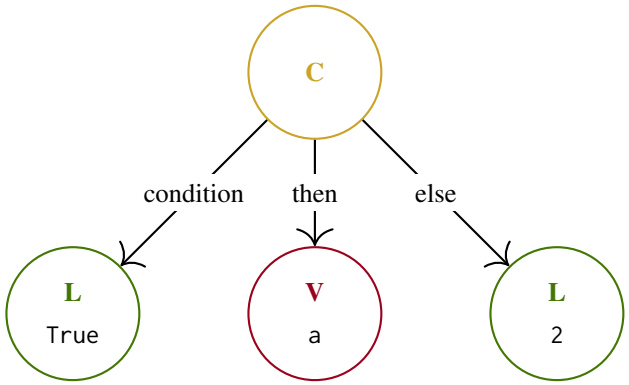


Figure 4.7: The AST representation of the term `if` $True$ `then` a `else` 2.

Declarations, Definitions, and the Module

Func already supports named constants through let expressions introduced in one of the previous Segments. Should a programmer require multiple named constants in their program, they would have to define a chain of nested let expression:

$$\begin{aligned} &\texttt{let } a = e_1 \texttt{ in} \\ &\quad \texttt{let } b = e_2 \texttt{ in} \\ &\quad\quad \texttt{let } c = e_3 \texttt{ in} \\ &\quad\quad\quad \texttt{let } d = e_4 \texttt{ in} \\ &\quad\quad\quad\quad \dots \end{aligned} \tag{4.10}$$

Writing let expression chains is in such a manner is syntactically very impractical. To address the verbosity of nested let expressions, Func introduces variable declarations.

The expression from 4.10 now becomes:

$$\begin{aligned} a &= e_1 \\ b &= e_2 \\ c &= e_3 \\ d &= e_4 \\ &\dots \end{aligned} \tag{4.11}$$

Introducing declarations does not have to impact the semantics established thus far – the language could simply translate them into let expression chains (4.11 becomes 4.10). However, treating declarations as separate semantic constructs improves efficiency, both during compilation and during run-time. For this reason, I have decided to add three more nodes to the AST:

1. **The declaration node** - Regular variable declarations (i.e., assignments of an expression on the right to a variable name on the left).

2. **The function definition node** - Mostly equivalent to declaration nodes but limited to functions. They exist mainly as an implementation detail meant to enable recursion. If a function is recursive, the compiler needs to know about it inside its body. Having a distinct node for function definitions makes this easy to ensure [8].

3. **The module node** - The root AST node introduced to hold all declarations. Simply put, a module represents a full Func program.

With the module node's addition, a Func program stops being a single expression and becomes a list of statements. Naked top-level expressions are not allowed. Figure 4.8 shows AST representations of the newly added nodes.

Since λ-calculus and Func treat functions as first-class values, and since function definition nodes exist solely as an implementation detail, the text does not generally differentiate between declaration nodes and function definition nodes. It instead treats function definition nodes as regular declarations whose left side is a lambda expression. One exception to this general rule is a Segment on function definitions in 6.2.2 which explores their purpose in more detail.

[8] 5.4.3 looks at this construct from a typing standpoint, while the pragmatic perspective is given by 6.2.2.

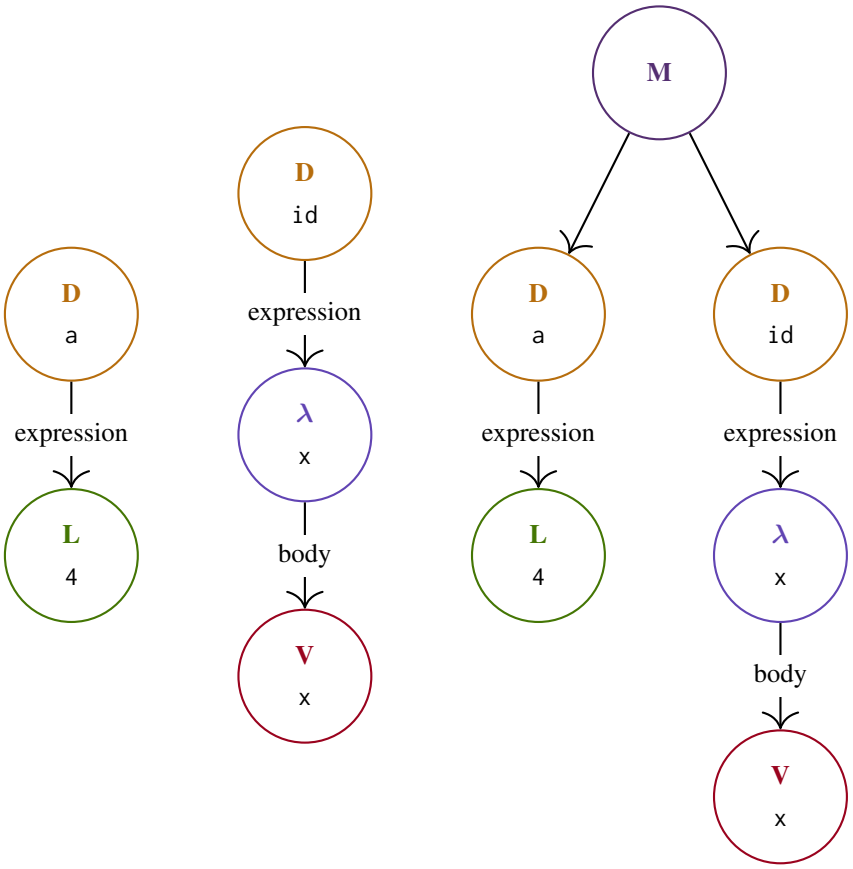


Figure 4.8: The picture on the left shows the AST representation of the declaration $a = 4$. The picture in the middle shows the AST representation of the declaration $id = \lambda x.x$. The picture on the right shows the representation of a program (i.e., module) containing these two declarations.

5. The Type System

This chapter explores Func's type system. One of the language's design goals (stated in 3.2.1) is to have a statically checked type system that is decidable, sound, and ideally complete. For reasons explained in 3.2.1, Func should predominantly rely on type inference while also allowing programmers to annotate their code with optional type signatures.

The type system usually has three components: a language of types, a set of typing rules, and an algorithm that enforces these rules. Most works blur the distinction between the second and the third component. However, they should still be thought of as distinct – the rules provide a declarative description of the system, while the algorithm provides an executable one (Krishnamurthi, 2015).

The internal language discussed in Chapter 4 is based on the untyped λ-calculus and does not possess a notion of types. The code generator does not use types when translating the AST into JavaScript, and JavaScript itself does not have a notion of types during run-time. Therefore, it is best to keep the internal language "untyped" and avoid unnecessary complexity. Be that as it may, the compiler still has to somehow type-check the AST. Therefore, it inspects the AST through the perspective of the simply-typed λ-calculus, a formal system that adds types to the previously established model. However, it is important to note that this does not change the language's foundation from the untyped to the simply-typed λ-calculus. Types exist exclusively from the type checker's perspective – it verifies whether the AST is well-typed, but the compiler does not use the typing information afterward (i.e., the AST remains unchanged).

Section 5.1 gives an overview of the simply typed λ-calculus, the mentioned theoretical extension that adds types to the original model. Section 5.2 formally defines the desired properties of Func's type system. It also accounts for the distinction between the type inference algorithm and the type system as a whole. Section 5.3 introduces Func's type system and sets up a foundation for understanding how Func achieves type safety, it explains the language of types. Finally, section 5.4 describes in detail how Func performs type inference using Milner's Algorithm $\mathcal{W}$. It lists all typing rules and

ultimately provides an algorithm for enforcing them[1].

5.1. The Simply Typed λ-calculus

The simply typed λ-calculus ($\lambda^{\rightarrow}$) is a formal system Alonzo Church introduced as an attempt to avoid paradoxical uses of the untyped (pure) λ-calculus (described in Section 4.1) (Pierce, 2002).

In practice, the simply typed λ-calculus helps programming languages combat the third problem mentioned in Subsection 4.2.2 – the lack of a proper type system. It does this by extending the λ-calculus with a single type constructor ($\rightarrow$) that builds function types. Some of the rules Func uses in its type inference algorithm come directly from the simply typed λ-calculus. However, since the language the base model in several ways, enumerated in Subsection 4.2.2, some rules that the algorithm uses do not exist in the simply typed λ-calculus (e.g., rules for inferring literals and let expressions). All of the rules used to perform type inference are listed and explained in Subsection 5.4.2.

It is important to note that the simply typed λ-calculus cannot have non-terminating expressions, meaning that it is not Turing complete. In practical terms, the system provides no way of defining recursive functions. In the untyped λ-calculus, recursive functions can be defined using the so-called fix combinator (Peyton Jones, 1987). However, the combinator becomes ill-typed inside the simply typed λ-calculus. Programming languages usually overcome this limitation by adding the fix combinator as a primitive to the language, with evaluation rules mimicking the behavior of the untyped fix combinator and a typing rule that captures its intended uses (Pierce, 2002). Semantically, Func does the same thing. From an implementation perspective, the combinator does not actually exist in the language but is ingrained inside the compiler.

5.2. Desired Properties

As described in Subsection 3.1.2, Func's type system must be decidable and sound. If possible, it should also be complete.

When discussing these properties, it is important to distinguish the type inference algorithm from the type system as a whole. Besides formally defining a property, each Segment also explains its meaning within the contexts of both the inference algorithm and the type system, respectively.

[1]The implementation of Func's type system can be found here.

Subsection 5.4.3 later proves that Func's type inference algorithm indeed has these properties.

5.2.1. Decicability

A decision problem is decidable if there exists a method that always produces an answer in finite time. Every problem that is not decidable is undecidable.

A decidable type system always accepts or rejects a program, meaning that its internal inference algorithm must always terminate.

5.2.2. Soundness

An inference rule r is sound if, when applied to a set of premises, derives a formula that is a logical consequence of these premises (Dalbelo Bašić & Šnajder, 2019). Formally, an inference rule r is sound if and only if:

$$P_1, \ldots, P_n \vdash_r C \implies P_1, \ldots, P_n \vDash C \tag{5.1}$$

Informally, a sound algorithm cannot prove anything that is untrue as long as it works with true premises.

A sound type inference algorithm never infers an incorrect type. A sound type system never accepts a program containing type errors (i.e., the types constraints encoded in its semantics are always satisfied).

5.2.3. Completeness

A set of inference rules R is complete if it can be used to derive all logical consequences. Formally, a set of inference rules R is complete if (Dalbelo Bašić & Šnajder, 2019):

$$P_1, \ldots, P_n \vDash C \implies P_1, \ldots, P_n \vdash_R C \tag{5.2}$$

Informally, a complete algorithm can prove everything that is true.

A complete type inference algorithm always infers a type if one exists. A complete type system accepts all programs without type errors.

Because λ-calculus (and by extension, Func) is Turing complete, any type system that is both sound and decidable must also be incomplete (i.e., it must reject some valid programs) (Remy, 2020). To understand why, consider a conditional expression:

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \tag{5.3}$$

Assume that e_2 represents a valid expression, while e_3 contains a type error. If expression e_1 always evaluates to $True$ at run-time, the else branch never executes (i.e., the program is not ill-typed). However, the type checker must still reject the program because determining whether e_1 always evaluates to $True$ might diverge. The static analyzer would essentially have to emulate the Turing-complete run-time, making the problem undecidable. Therefore, a type system that is both sound and decidable is necessarily incomplete (Pierce, 2002).

5.3. Types in Func

The state-of-the-art way for a functional λ-calculus-based language to ensure the properties described in the previous section is by using the Hindley-Milner (HM) type system (Damas & Milner, 1982). HM is a nominal type system for the λ-calculus with parametric polymorphism, featuring a type inference algorithm proven to be both sound and complete (Damas & Milner, 1982). All functional programming languages that aim to have a provably correct type system without requiring type annotations use Hindley-Milner (these include Haskell, PureScript, Flow, Standard ML, OCaml, and many more) [2]. Therefore, it makes much sense for Func to implement HM as well. This section focuses on describing the type system's fundamental concepts and defining the type language, while the next one (Section 5.4) enumerates Func's typing rules and explains how it achieves type inference.

Proper Types and Type Constructors

Before discussing Func's type system, it is necessary to understand the distinction between *proper types* and *type constructors*. This Segment gives the meaning behind those terms[3].

A *proper type* is any type that can be used to classify an expression (i.e., all expressions evaluate to values of proper types). Examples include $Bool$ (the type of all boolean values) and $Number \to Bool$ (the type of all functions taking a number and returning a boolean). The text uses symbols T or T_i to denote arbitrary proper types.

[2] It is important to note the constraint of not requiring any programmer-supplied type annotations. Suppose the language is willing to compromise this feature. In that case, it can become more powerful while retaining the desired properties, but using a different type system (e.g., System F) (Dader, 2018).

[3] The distinction between proper types and type constructors is formalized by type kinds (i.e., the types of types, type arity specifiers). While the text does not mention nor explore kinds, the reader can find more information in the literature (Pierce, 2002).

A *type constructor* is an abstraction (lambda) at the level of types. It cannot be used as a proper type itself but can create proper types when applied to other proper types (this is an application at the level of types). The most famous type constructor and the only one included in the simply typed λ-calculus is the function type constructor ($\rightarrow$). No expression can have the type $\rightarrow$, but $\rightarrow$ can be used to construct proper function types. For example, applying the type constructor $\rightarrow$ to $Number$ and $Bool$ yields the proper type $Number \rightarrow Bool$, applying it to $Bool$ and $Number \rightarrow Bool$ yields $Bool \rightarrow Number \rightarrow Bool$, and so on. The function constructor binds to the left, meaning that $Bool \rightarrow (Number \rightarrow Bool)$ represents the same proper type as $Bool \rightarrow Number \rightarrow Bool$. Generally, the text uses $C\langle T_1, \ldots, T_n\rangle$ to denote a type constructor C applied to proper types $T_1, \ldots . T_n$. Following this syntax, a function type $T_1 \rightarrow T_2$ would be written as $\rightarrow \langle T_1, T_2\rangle$. However, since the function type constructor is such a central part of the language, it enjoys special infix syntax (as well as its own typing rule, as seen in 5.4.2). Func includes only two more builtin type constructors ($Tuple$ and$Array$), but technically provides semantic support for arbitrary user-defined type constructors[4]. The text uses the symbols C or C_i to denote arbitrary type constructors.

For brevity, the text may refer to proper types as *types*. To avoid confusion, it will always be explicit when referring to type constructors.

5.3.1. Monotypes

Monotypes are types that always designate a particular type (i.e., they are not polymorphic)[5]. Possible monotypes in Func are *base types*, *parametric types*, and *type variables*.

Base types (or type literals) represent simple, unstructured values. At the time of writing, Func has a total of four base types: booleans (`Bool`), characters (`Char`), numbers (`Num`), and bigints (`Bigint`). The text uses the Greek letter ι to denote an arbitrary type literal or a metavariable ranging over all possible type literals.

Parametric types are proper types obtained by applying type constructors, examples in Func include: all possible function types (e.g., $Number \rightarrow Bool$), $Array\langle Bool\rangle$, and $Tuple\langle Number, Bool\rangle$.

For theoretical purposes, it is often useful to abstract away from the details of particular proper types and their operations (Pierce, 2002). Uninterpreted base types or

[4]Though not syntactic support at the time of writing.

[5]This will make more sense after reading about polytypes in the next Subsection.

type variables serve as placeholders for concrete types. They can be instantiated and substituted with other proper types. Thanks to let-polymorphism (explained in the next Subsection, 5.4.3), type variables cannot exist in the program's top-level context outside of type schemes that bind them (more on this in 5.3.2). They can and do serve as proper types inside expressions, created either by instantiating a type scheme or defining an inline lambda. The text uses small Greek letters to denote arbitrary type variables. The same symbols may also denote metavariables ranging over all possible type variables, but the distinction should be clear from the context.

Two monotypes are equal if they have identical terms (Wikipedia contributors, 2021). Two equal parametric monotypes must be built using the same constructor, which is then lifted before comparing the terms. Formally, $C^A\langle T_1^A, \ldots, T_n^A\rangle$ is equal to $C^B\langle T_1^A, \ldots, T_1^B\rangle$ if and only if $C^A = C^B$ and $T_i^A = T_i^B$. For example, type $Array\langle T_1\rangle$ is equal to type $Array\langle T_2\rangle$ if and only if type T_1 is equal to type T_2, while $Tuple\langle T_1\rangle$ and $Array\langle T_2\rangle$ can never be equal, regardless of their inner types.

5.3.2. Type Schemes (Polytypes)

A *type scheme* is a type that possibly quantifies (captures) type variables. Schemes allow Func to exhibit parametric polymorphism, a language feature originally described in Subsection 3.4.5. Consider the identity function, which has the following type scheme:

$$id :: \forall\alpha\ \alpha \to \alpha \tag{5.4}$$

he type variable α can then be instantiated to a number of different types, depending on how a programmer uses the polymorphic function. For example, expression $id \cdot 4$ instantiates the type scheme $\forall\alpha\ \alpha \to \alpha$ into $Number \to Number$ by applying the substitution $[\alpha \mapsto Number]$. The precise meaning of the term *instantiation* will become clear in Subsection 5.4.2 after explaining the type system's instantiation rule.

While type schemes might seem as an unnecessary complication (why not just use the monotype $\alpha \to \alpha$ instead of a type scheme $\forall\alpha\ \alpha \to \alpha$?), the language cannot be parametrically polymorphic without them. Their necessity will become apparent after discussing the type order in Subsection 5.4.1, the generalisation rule in Subsection 5.4.2, and let-polymorphism in Subsection 5.4.3.

Two polytypes are equal up to α-conversion and reordering of their quantifiers. Unused quantifiers can be safely ignored.

5.3.3. The Type Grammar

Assuming a set of type variables α, a set of primitive types ι, and a set of type constructors C, the syntax of monotypes T and type schemes S in Func is given here:

$$\begin{aligned} T &::= \alpha \mid \iota \mid T \to T \mid C\langle T, \cdots, T\rangle \\ S &::= T \mid \forall\alpha\ S \end{aligned} \tag{5.5}$$

Type schemes may or may not quantify type variables (i.e., all valid monotypes are valid type schemes, but not the other way around). In fact, a monotype can be interpreted as a type scheme without any bound variables.

5.4. Type Inference

This Section defines the mechanisms Func uses to perform type inference. Subsection 5.4.1 introduces the mathematical concepts and operations necessary to understand the typing rules and perform the inference algorithm. Subsection 5.4.2 lists and explains the typing rules that serve as the foundation of Func's type system. Finally, Subsection 5.4.3 lays out the algorithm the compiler uses to infer types in a Func program.

5.4.1. Preliminaries

This subsection goes through the definitions and concepts necessary for understanding Func's typing rules and its type inference algorithm. Detailed explanations of all mentioned concepts and proof of their assumed properties are available in the literature (Pierce, 2002).

The Typing Context

Each typing rule describes the local typing behavior of a particular kind of expression. However, since the language permits nested lambdas and named constants, the local rules also need to account for typing assumptions about variables bound somewhere else. A *typing context* (or a type environment) Γ is a mapping of variable names to their type schemes. The claim $\Gamma \vdash e : S$ means "the expression e has type S in context Γ". The context maps names to type schemes and not monotypes because the latter is a subset of the former (as defined by the type grammar in Subsection 5.3.3) (Hegemann, 2019).

Free Type Variables

Quantified variables α_i in a type scheme $\forall\alpha_1 \ldots \alpha_n\ T$ are said to be *bound* inside the monotype T. All unbound type variables in monotype T are called *free type variables*. The definition of free type variables naturally extends to contexts. Equation 5.6 formally defines how to calculate the set of free type variables for monotypes, type schemes, and contexts (using the notation introduced in Section 5.3).

$$\begin{aligned}
\text{free}(\alpha) &= \{\alpha\} \\
\text{free}(\iota) &= \{\iota\} \\
\text{free}(T_1 \to T_2) &= \text{free}(T_1) \cup \text{free}(T_2) \\
\text{free}(C\langle T_1, \ldots, T_n\rangle) &= \cup_{i=1}^{n} \text{free}(T_i) \\
\text{free}(\forall\alpha\ S) &= \text{free}(S) - \{\alpha\} \\
\text{free}(\Gamma) &= \cup_{x:S\in\Gamma} \text{free}(S)
\end{aligned} \tag{5.6}$$

Substitution

A type substitution σ is a finite mapping from type variables to other types. For example, the substitution $[T_1 \mapsto T_2, T_3 \mapsto T_4]$ associates T_1 with T_2 and T_3 with T_4. The domain of a substitution σ is written as $\text{dom}(\sigma)$ and represents the set of all variables appearing on the left side of pairs in σ. The range of a substitution σ is written as $\text{range}(\sigma)$ and represents the set of all variables appearing on the right side of pairs in σ. Equation 5.7 formally defines substitution on types (Pierce, 2002). The definition of substitution on type schemes assumes no name clashes between bound variables and $\text{dom}(\sigma)$ (this is easily ensured by applying α-conversions on quantified variables)[6].

$$\begin{aligned}
\sigma(\alpha) &= \begin{cases} T, & \text{if } (\alpha \mapsto T) \in \sigma \\ \alpha, & \text{if } \alpha \notin \text{dom}(\sigma) \end{cases} \\
\sigma(\iota) &= \iota \\
\sigma(T_1 \to T_2) &= \sigma(T_1) \to \sigma(T_2) \\
\sigma(\forall\alpha\ S) &= \forall\alpha\ \sigma(S) \\
\sigma(C\langle T_1, \ldots, T\rangle) &= C\langle\sigma(T_1), \ldots, \sigma(T_n)\rangle
\end{aligned} \tag{5.7}$$

Substitution extends pointwise to contexts (Pierce, 2002):

$$\sigma(\{\alpha_1 : S_1, \ldots, \alpha_n : S_n\}) = \{\alpha_1 : \sigma(S_1), \ldots, \alpha_n : \sigma(S_2)\} \tag{5.8}$$

[6]What this essentially means is "Do not substitute quantified variables, quantifiers introduces a new naming scope that can shadows any name appearing in the substitution".

If σ_1 and σ_2 are substitutions, their composition $\sigma_1 \circ \sigma_2$ is equivalent to applying σ_1 after σ_2:

$$(\sigma_1 \circ \sigma_2)(X) = \sigma_1(\sigma_2(X)) \tag{5.9}$$

A substitution σ_1 is less specific (or more general) than a substitution σ_2, written as $\sigma_2 > \sigma_1$, if $\sigma_2 = \sigma_3 \circ \sigma_1$ for some substitution σ_3 (Pierce, 2002).

Type Order

As mentioned when describing type schemes in 5.3.2, the identity function has type $\forall\alpha\ \alpha \to \alpha$. It can then be instantiated into more specific types, such as $Number \to Number$ or $(\alpha \to \beta) \to (\alpha \to \beta)$, but not $Bool \to Number$. The most general type of this function is $\forall\alpha\ \alpha \to \alpha$. It is possible to obtain all other types by applying substitutions to the most general type. For example, applying the substitution $[\alpha \mapsto Number]$ on $\alpha \to \alpha$ gives $Number \to Number$ (the substitution is applied to the scheme's monotype part, the quantifier only indicates that substitution is allowed). Of course, no substitution can instantiate the scheme to $Bool \to Number$.

Formally, a type scheme S_1 is less specific (or more general) than type scheme S_2, written as $S_1 > S_2$, if $S_1 = \sigma(S_2)$ for some substitution σ (Pierce, 2002).

The relation $<$ defines a partial order on the set of all type schemes, with the smallest element being $\forall\alpha\ \alpha$ (Pierce, 2002).

5.4.2. Typing Rules

This Subsection presents and explains Func's typing rules. Some rules come directly from the simply typed λ-calculus (i.e., T-VAR, T-ABS, T-APP). Other rules describe extensions to the λ-calculus and were originally provided by Milner (Milner, 1978) or Damas and Milner (Damas & Milner, 1982) together with their definition of Algorithm $\mathcal{W}$(these include T-LET, T-IF, T-INST, T-GEN). Finally, Func, like most other languages, adds a family of rules T-LIT for typing literals.

The table 5.1 lists the notation used in rule definitions and the description of the algorithm.

Symbol	Meaning
$\lambda x.e$	An abstraction with the head x and the expression body e
$e_1 \cdot e_2$	An application of expression e_1 to expression e_2
Γ	The typing context
T	A type, as defined by grammar 5.5
S	A type scheme, as defined by grammar 5.5
$e : S$	Expression e has type scheme S
$\Gamma, x : S$	The typing context extended with the claim $e : S$. The claim on the right takes precedence.
$e : S \in \Gamma$	The context Γ contains the claim $e : S$
$e : S \notin \Gamma$	The context Γ does not contain the claim $e : S$
$P_1, \ldots P_n \vdash C$	The set of premises $P_1, \ldots, P_n$ deductively entails the conclusion C
$\frac{P_1 \quad ... \quad P_n}{C}$	Equivalent to $P_1, \ldots P_n \vdash C$

Table 5.1: Notation for describing Func's typing rules.

Typing Identifier References

Identifier references represent variables. Since variables are pure lambda terms, their typing rule comes directly from the simply typed λ-calculus. The variable typing rule is by far the most straightforward one – an identifier reference simply has whatever type scheme it is currently assumed to have:

$$\frac{x : S \in \Gamma}{\Gamma \vdash x : S} \qquad \text{(T-VAR)}$$

Meaning: If context Γ contains the claim that identifier x has type scheme S, **then** identifier x has type scheme S in context Γ.

Typing Lambdas

Since lambdas (abstractions) also belong to the set of original lambda terms, their typing rule comes directly from the simply typed λ-calculus as well.

Assuming that the lambda's argument x is known to be of type T_1, it is clear that the type of its result will be the type of the body e_2, where occurrences of x inside e_2 are assumed to denote terms of type T_1. The typing rule T-ABS formally expresses this intuition (Pierce, 2002):

$$\frac{\Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \lambda x.e_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

Meaning: If expression e_2 has type T_2 in context Γextended with the premise that x has type T_1, **then** lambda $\lambda x.e_2$ has type $T_1 \to T_2$ in context Γ.

Typing Applications

Finally, the simply typed λ-calculus includes the typing rule for applications. Applying an expression of type $T_1 \to T_2$ to an argument of type T_1 has to result in a value of type T_2. Formally:

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1 \cdot e_2 : T_2} \quad \text{(T-APP)}$$

Meaning: If expression e_1 has type $T_1 \to T_2$ in context Γ**and** expression e_2 has type T_1 in context Γ, **then** application $e_1 \cdot e_2$ has type T_2 in context Γ.

Typing Let Expressions

Since let expressions are an extension to the λ-calculus, the simply typed λ-calculus does not offer a rule for typing them. Luckily, Damas and Milner included let expressions in their original type inference algorithm for ML (Damas & Milner, 1982). Therefore, I can freely use it in Func and rely on its proven properties.

Assuming that the initializing expression e_1 has type T_1, the type of the let expression's result will be the type of its body e_2, where occurrences of x inside e_2 are assumed to denote terms of type T_1 (Pierce, 2002). This is how the reasoning looks when expressed formally:

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : T_2} \quad \text{(T-LET)}$$

Meaning: If expression e_1 has type T_1 in context Γ**and** expression e_2 has type T_2 in context Γ extended with the premise that x has type T_1, **then** expression $\texttt{let } x = e_1 \texttt{ in } e_2$ has type T_2 in context Γ.

The Generalisation Rule

The generalization rule also comes directly from the original paper (Damas & Milner, 1982). It exists to translate monotypes into type schemes, thus enabling parametric

polymorphism. To justify the need for type schemes, consider the following code without the generalization rule:

$$id = \lambda a.a$$
$$x = id \cdot 3$$
$$y = id \cdot True$$

id has type $\alpha \rightarrow \alpha$. Expression $id \cdot 3$ requires substitution $[\alpha \mapsto Number]$, while expression $id \cdot True$ requires substitution $[\alpha \mapsto Bool]$. These two substitutions cannot be satisfied simultaneously, making the program ill-typed. The type system must somehow quantify (capture) the variable α in the type of id to be polymorphic. It is allowed to quantify any variable α that is not free in the context, thus making the scheme polymorphic in terms of α:

$$\frac{\Gamma \vdash e : S \qquad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall \alpha \; S} \qquad \text{(T-GEN)}$$

Meaning: If expression e has type scheme S in context Γ **and** variable α is not free inside context Γ, **then** expression e also has the more general type scheme $\forall \alpha \; S$ in context Γ[7].

The Instantiation Rule

To use a polymorphic function created by T-GEN, the language must instantiate it (as shown in 5.4.1). Damas and Milner included the instantiation rule to complement the generalization rule. A type scheme can always be specialized (i.e., instantiated to a more specific scheme):

$$\frac{\Gamma \vdash e : S_1 \qquad S_1 > S_2}{\Gamma \vdash e : S_2} \qquad \text{(T-INST)}$$

Meaning: If expression e has type scheme S_1 in context Γ**and** S_1 is more general than S_2, **then** expression e has the more specific type scheme S_2 in context Γ.

Typing Conditionals

Func added conditional expressions to its core model in 4.2.2. They are an extension to λ-calculus (i.e., the simply typed λ-calculus does not include a rule for typing them). Fortunately, the conditional expressions' semantics are reasonably straightforward. Although Damas and Milner do not describe an appropriate typing rule in their original

[7]However, the compiler must be careful when applying this rule. Overusing it can easily make the type system undecidable, as explained in the Segment about let-polymorphism in Subsection 5.4.3.

paper (Damas & Milner, 1982), Milner does so in one of his earlier works (Milner, 1978).

The condition e_1 must evaluate to a value of type $Bool$, while branches e_1 and e_2 must both evaluate to values of the same type T (Pierce, 2002):

$$\frac{\Gamma \vdash e_1 : Bool \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : T} \tag{T-IF}$$

Meaning: If e_1 has type $Bool$ in context Γ **and** e_2 has type T in context Γ **and** e_3 has type T in context Γ, **then** expression `if` e_1 `then` e_2 `else` e_3 has type T in context Γ.

Typing Literals

Damas's and Milner's original paper does not explore inferring literal expressions. Most theoretical work examines literals by analyzing their encodings inside the λ-calculus. Since this thesis primarily focuses on practical applications and since typing literal expressions is trivial, the rules are included here. Literals for all base types can be inferred directly from the AST. For example, booleans only adhere to two obvious typing rules:

$$\frac{\varnothing}{\Gamma \vdash True : Bool} \qquad \frac{\varnothing}{\Gamma \vdash False : Bool} \tag{T-LIT}$$

Meaning: The value $True$ has type $Bool$ in any context. The value $False$ has type $Bool$ in any context.

For other base types (i.e., $Number$, $Bigint$, $Char$), we can imagine an infinite family of similar rules ranging over all possible values of said type. All such rules are tautologies (i.e., they do not compromise the system's proven properties).

It is possible to define a rule for built-in parametric types as well. Tuples impose no limitations on their inner types (i.e., a tuple of size n may conatain n values with n different types):

$$\frac{\Gamma \vdash e_1 : T_1 \quad \cdots \quad \Gamma \vdash e_n : T_n}{\Gamma \vdash (e_1, \ldots, e_n) : Tuple\langle T_1, \ldots, T_n \rangle} \tag{T-TUP}$$

Meaning: If expressions e_1, ..., e_n have respective types T_1, ..., T_n in context Γ, **then** tuple $(e_1, \ldots, e_n)$ has type $Tuple\langle T_1, \ldots, T_n \rangle$ in context Γ.

Arrays, on the other hand, require all inner expressions to have the same type:

$$\frac{\Gamma \vdash e_1 : T \quad \cdots \quad \Gamma \vdash e_n : T}{\Gamma \vdash [e_1, \ldots, e_n] : Array\langle T \rangle} \tag{T-ARR}$$

Meaning: If expressions e_1, ..., e_n all have type T in context Γ, **then** array $[e_1, \ldots, e_n]$ has type $Array\langle T \rangle$ in context Γ.

In reality, the type inference algorithm does not use rules T-TUP and T-ARR. They are listed here for thoroughness. To avoid extending the original algorithm and risk

losing its proven properties, Func interprets tuples and arrays in a slightly different but equivalent way (described in 5.4.3).

5.4.3. Performing Type Inference

The algorithm Func uses for type inference is mostly equivalent to Milner's Algorithm $\mathcal{W}$. Robin Milner invented $\mathcal{W}$ for ML. However, the algorithm has since become the de-facto standard for type inference in functional languages[8]. Func's variant of $\mathcal{W}$ extends it only with rules for inferring literal expressions. To adequately describe the algorithm, it is first necessary to describe the operations it internally uses.

Unification

Algorithm $\mathcal{W}$ makes heavy use of Robinson's unification algorithm U which, given a pair of types T_1 and T_2, returns a substitution σ_u or fails:

1. if $\mathrm{U}(T_1, T_2)$ returns σ_u, then σ_u unifies T_1 and T_2.
2. if σ_1 unifies T_1 and T_2, then $\mathrm{U}(T_1, T_2)$ returns some substitution σ_u and there is another substitution σ_2 such that $\sigma_1 = \sigma_2 \circ \sigma_u$[9].

Furhter, σ_u only involves type variables present in T_1 and T_2 (Damas & Milner, 1982).

Unique Type Variables

The type assignment algorithm also relies on a function called UNIQUETYPEVAR. This function, whenever called, produces a unique type variable name. It exists mainly as an implementation detail for avoiding name clashes.

Let-polymorphism

Let-polymorphism (also ML-style and Damas-Milner polymorphism) is a feature introduced in the first dialect of ML (Gordon et al., 1978). It allows a single function to be used with different types in different contexts (i.e., makes it parametrically polymorphic). It forms the basis of powerful generic libraries of commonly used structures in most successful functional programming languages (e.g., Haskell, ML) (Pierce, 2002).

[8] Most modern implementations actually use Algorithm $\mathcal{J}$, which is just an efficient simulation of $\mathcal{W}$ (Milner, 1978). No one seems to know how and why Milner came up with the names $\mathcal{W}$ and $\mathcal{J}$ for his algorithms (Grant, 2011)

[9] In other words, U always returns the most general unifier.

Assuming expression `let` $x = e_1$ `in` e_2, let-polymorphism stands for applying the rule T-GEN to all allowed type variables inside the type e_1 before setting it as x's type inside the context, thus turning the type of x into a polymorphic type scheme. For example, consider the following expression:

$$\texttt{let } id = \lambda x.x \texttt{ in } (id \cdot 1, id \cdot True) \tag{5.10}$$

Without let-polymorphism, the expression is ill-typed (see T-GEN for a more detailed explanation). However, with let-polymorphism, the expression is valid and reduces to a value of type $Tuple\langle Number, Bool\rangle$ [10].

Formally, a closure of a type T with respect to context Γ, written $\overline{\Gamma}(T)$, is defined as

$$\overline{\Gamma}(T) = \forall \alpha_1, \ldots, \alpha_n \, T, \tag{5.11}$$

where $\alpha_1, \ldots, \alpha_n$ represent all type variables occuring free in T, but not in Γ (i.e, $\alpha_i \in \text{free}(T) - \text{free}(\Gamma)$) (Damas & Milner, 1982).

The Type Assignment Algorithm

Algorithm 1 documents Func's type inference algorithm. It does not include procedures for typing declarations and function definitions as they are essentially equivalent to the procedure used to implement T-LET.

One crucial detail is that inferring the types of recursive functions requires an extra step. When the compiler encounters a recursive definition of a function (e.g., $f = e$, where e represents the function's body), it simply adds the claim $f : t_a$ to the context Γ and proceeds to infer the type of expression e, resulting in t_b. To obtain the type of function f, all that the compiler has to do is unify types t_a and t_b. This procedure can be viewed as a pragmatic typing rule for the fix combinator mentioned during the discussion on the simply-typed λ-calculus's Turing completeness in Section 5.1. The idea for typing the combinator also comes from Milner, who proved it in his original paper (Milner, 1978).

[10] ML, Haskell and Func all limit the application of T-GEN to variables bound in let expressions. The main reason being that allowing generalization on all identifier references would turn the underlying type system into System F, which is not decidable (Wells, 1994).

Algorithm 1 The type assignment Algorithm $\mathcal{W}$

Require: A context Γ and an AST node e

Ensure: A substitution σ and a type scheme S such that $\sigma(\Gamma) \vdash e : S$

function W(Γ, e)
  if $e = x$ **and** $x : \forall\alpha_1, \ldots, \alpha_n\ T \in \Gamma$ **then**
    $\sigma \leftarrow [\,]$
    $T' \leftarrow [\alpha_i \mapsto \text{UNIQUETYPEVAR}](T)$ ▷ Applying T-INST
    return (σ, T')
  else if $e = e_1 \cdot e_2$ **then** ▷ Applying T-APP
    $\beta \leftarrow \text{UNIQUETYPEVAR}$
    $(\sigma_1, T_1) \leftarrow \text{W}(\Gamma, e_1)$
    $(\sigma_2, T_2) \leftarrow \text{W}(\sigma_1(\Gamma), e_2)$
    $\sigma_u \leftarrow \text{U}(\sigma_2(T_1), T_2 \rightarrow \beta)$
    return $(\sigma_u \circ \sigma_2 \circ \sigma_1, \sigma_u(\beta))$
  else if $e = \lambda x.e$ **then** ▷ Applying T-ABS
    $\beta \leftarrow \text{UNIQUETYPEVAR}$
    $\Gamma_x \leftarrow \Gamma, x : \beta$
    $(\sigma_1, T_1) \leftarrow \text{W}(\Gamma_x, e)$
    return $(\sigma, \sigma_1(\beta) \rightarrow T_1)$
  else if $e = \texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2$ **then** ▷ Applying T-LET
    $(\sigma_1, T_1) \leftarrow \text{W}(\Gamma, e_1)$
    $\Gamma_2 \leftarrow \sigma_1(\Gamma)$
    $\Gamma_x \leftarrow \Gamma_2, x : \overline{\Gamma_2}(T_1)$ ▷ Applying T-GEN
    $(\sigma_2, T_2) \leftarrow \text{W}(\Gamma_x, e_2)$
    return $(\sigma_2 \circ \sigma_1, T_2)$
  else if $e = \texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3$ **then** ▷ Applying T-IF
    $(\sigma_1, T_1) \leftarrow \text{W}(\Gamma, e_1)$
    $\sigma_b \leftarrow \text{U}(T_1, Bool)$
    $(\sigma_2, T_2) \leftarrow \text{W}((\sigma_b \circ \sigma_1)(\Gamma), e_2)$
    $(\sigma_3, T_3) \leftarrow \text{W}((\sigma_2 \circ \sigma_b \circ \sigma_1)(\Gamma), e_3)$
    $\sigma_u \leftarrow \text{U}(T_3, \sigma_3(T_2))$
    return $(\sigma_u \circ \sigma_3 \circ \sigma_2 \circ \sigma_b \circ \sigma_1, \sigma_u(T_3))$
  else if e is a primitive type **then** ▷ Applying T-LIT
    return the appropriate type literal i for e
  else
    fail
  end if
end function

Proving the Algorithm

As mentioned at the start of the Section, the algorithm described above is mostly equivalent to Milner's original Algorithm $\mathcal{W}$. Robin Milner proved the algorithm's soundness (Milner, 1978) and later, together with his student Luis Damas, proved its completeness (Damas & Milner, 1982). This Section proves that the original algorithm's properties also apply to this variant. It is only necessary to prove the procedure for inferring literals, as all other procedures come directly from the original paper.

As stated during their definition in 5.4.2, the rules for inferring primitive types are tautologies and do not constitute a meaningful change to the algorithm (i.e., they are complete and sound by their definition). All that remains is to express array literals and tuple literals through already proven constructs. Array literals can be fully represented by:

1. An empty array with type scheme $\forall\alpha\ Array\langle\alpha\rangle$
2. A function for extending arrays with type scheme $\forall\alpha\ \ \alpha\ \rightarrow\ Array\langle\alpha\rangle\ \rightarrow\ Array\langle\alpha\rangle$

The first point is sound and complete by definition, while the second point nicely fits within the already established typing rules (i.e., T-ABS, T-APP, and T-INST).

Func internally treats tuples in a similar way. For example, a function for constructing a tuple of size 2 has type scheme:

$$\forall\alpha\forall\beta\ \alpha \rightarrow \beta \rightarrow Tuple\langle\alpha, \beta\rangle$$

A function for constructing a tuple of size 3 has type scheme:

$$\forall\alpha\forall\beta\forall\gamma\ \alpha \rightarrow \beta \rightarrow \gamma \rightarrow Tuple\langle\alpha, \beta, \gamma\rangle$$

And so on[11].

Such an interpretation of parametric type value literals avoids adding special cases to Milner's original algorithm, thus enabling Func to rely on its proven properties (Damas & Milner, 1982) [12].

Therefore, I can conclude that type inference in Func is both sound and complete. Since the inference algorithm always produces an answer, the type system is sound and decidable. However, the type system as a whole cannot be complete due to the theoretical limitation explored in Subsection 5.2.3.

[11] This is why Func limits tuple sizes to five. All tuple sizes are different types and have to be hardcoded into the inference engine separately.

[12] In other words, it saves me from having to prove that my extensions do not impact the algorithm's soundness and completeness.

6. The Syntax

Syntax is the *form* of a language. It focuses on concrete notations used to encode the language's phrases (Turbak et al., 2008). This definition describes the surface syntax (i.e., the langauge as seen by the programmer). Func's internal syntax and the difference between the two were discussed in Chapter 4).

Section 6.1 makes several high-level decision about the language's general syntactic properties (e.g. whitespace and case sensitivity). Section 6.2 goes through all of the language's AST nodes defined in Chapter 4 and shows how the surface language represents them. It also lists Func's purely syntactic features (i.e., syntactic sugar) and demonstrates how they map into the core language.

6.1. Desired Syntactic Properties

Here I discuss some of the general syntactic choices I made when designing Func. More specific choices concerning particular language constructs are explained and justified throughout Section 6.2.

The discussion here is very similar in form to the one in Section 3.2, but was postponed until this Chapter because it depends on individual design choices made during the earlier phases of the design process (i.e., in Chapter 4).

6.1.1. Case Sensitivity

Most general-purpose programming languages are case sensitive, meaning that they treat `NAME` and `name` as different identifiers. Case sensitivity is a popular feature because it improves readability and ensures consistency in naming. Case sensitive languages generally come with naming conventions that define this property's use by prescribing naming rules (e.g., types are capitalized, constants are uppercase). Some languages go a step further by either enforcing these conventions at the compiler level or even using them to convey semantic meaning. For example, capitalization in Go decides whether a

module exports a name or not:

```
package math

func SquareSum(a float64, b float64) float64 {
    return square(a) + square(b)
}

func square(a float64) float64 {
    return a * a
}
```

When the user imports the package `math`, they will be able to access `math.SquareSum` but not `math.square`.

There have been recent efforts to move away from case sensitivity in general-purpose programming, with Nim being the most prominent language promoting this stance. In fact, Nim even goes a step further and employs style insensitivity (i.e., `na_me`, `NaMe`, `NAME`, and `N_a_m_E` all represent the same identifier). The team behind Nim argues that *Identifiers which only differ in case are bad style* and that *Case insensitivity is widely considered to be more user friendly* (Rumf, 2016). Regardless of what one may think about the mentioned claims, there is no denying that Nim's style insensitivity caused a considerable backlash in the community, thus significantly impacting the language's adoption rate (Picheta, 2018).

Verdict: Since Func aims to reflect design choices proven to be popular and useful, it will be case and style sensitive. Ideally, it will enforce capitalization in type names and prevent it in identifier names.

6.1.2. Significant Whitespace

Significant whitespace has been a controversial subject in the programming language community for decades. All languages assign semantic meaning to whitespace to some extent – if for nothing else, then for separating keywords from identifiers. What causes debate is when a language assigns much meaning to whitespace, either by relying on inline whitespace to differentiate between operations inside expressions, adhering to the off-side rule, or using line breaks to separate statements.

Whitespace Sensitivity

For an example of what is most often referred to as whitespace sensitivity, consider an example in Ruby:

```
1 f[1]
2 f [1]
```

The first line calls the function `f` with no arguments and then calls the method `[]` on the returned value, passing `1` as an argument. The second line applies the function `f` to the array `[1]`. Such behavior has proven to be very confusing for programmers. Of course, whitespace still has to carry *some* semantic meaning (e.g, `if isTrue` should be treated differently than `ifisTrue`), but it is best to minimize its importance. This claim especially holds in expression-based languages – most functions are length expressions, and the lack of whitespace sensitivity plays a significant role in their readability because it allows liberal formatting.

Verdict: Func will try to minimize whitespace sensitivity inside expressions.

The Off-side Rule

A programming language has indentation-based syntax (also called the off-side rule) if it uses indentation to express code blocks. Python is probably the most prominent example of a language adhering to the off-side rule. JavaScript, C, and Java are all *free-form languages*, meaning that they use whitespace characters only as token delimiters. Because context-free grammars cannot express indentation rules, the off-side rule significantly complicates the parsing process. Fortunately, since Func is an expression-based language that does not possess a notion of code blocks, I do not need to decide whether to implement indentation-based syntax.

Verdict: Func's current design does not have a notion of blocks. The discussion on expressing them does not apply to the language at this stage.

Statement Separators

Func only supports statements at the top level and inside let expressions (added in Section 4.2.2). Separating statements by newlines instead of semicolons has become a popular choice among modern language designers. For example, Rob Pike and his team decided to adopt it in Go (Pike, 2010). Several modern JavaScript frameworks (e.g., Vue) decided to abuse the language's automatic semicolon insertion feature (ASI) to remove all semicolons from their codebases.

Although the lack of semicolons (or other explicit statement separators) makes the language more restricted and less "free-form," it is generally perceived as a useful feature – programmers almost always write statements on separate lines, and most languages still allow multiple statements on the same line by including optional semicolons (e.g., Python, Go).

Verdict: Func will allow both line breaks and semicolons as statement separators. The preferred way of separating statements will be writing them on separate lines. However, the programmer will also be able to write multiple statements on the same line by delimiting them with semicolons.

6.2. Defining the Surface Sytnax

This Section explains how the programmer's code translates into the internal language defined in Chapter 4.

Due to many low-level complications (e.g., operator associativity, lexical grammar), it is tedious to formally describe the language's grammar when examining AST nodes one at a time. Instead, the text takes an example-oriented approach[1].

Subsection 6.2.1 describes the surface syntax for expression-level AST nodes. These represent expressions that come directly from the λ-calculus, as well as some extensions (see Chapter 4 for a full reference). Subsection 6.2.2 describes the syntax for representing the remaining, statement-level AST nodes (i.e., function definitions, declarations, modules). Finally, Subsection 6.2.3 examines syntactic language constructs that appear in the surface syntax but do not appear in the AST (i.e., syntactic sugar). It shows how they look like in the code and demonstrates their translation into the internal language.

6.2.1. Expression-level Syntax

This Subsection covers Func's expression-level syntax. None of the nodes here constitute a valid Func program because, due to the introduction of statements in 4.2.2, Func does not allow naked top-level expressions.

Variables and Identifier References

Variables (i.e., identifier declarations) and identifier references (i.e., expressions referencing declared variables) adhere to the same surface syntax rules. In other words, the

[1] A formal definition of Func's grammar can be found in the GitHub repository. The parser was generated using pegjs.

programmer can reference a variable directly by its name. This claim does not hold for all programming languages. For example, when referencing Bash variables, one has to prefix the variable's name with a dollar sign. Since JavaScript (as most other modern high-level languages) does not impose such limitations, neither does Func.

Valid identifiers consist of numbers, letters, and underscores. They must start with either a letter or an underscore. Figure 6.1 shows examples of valid identifier references and their corresponding AST nodes.

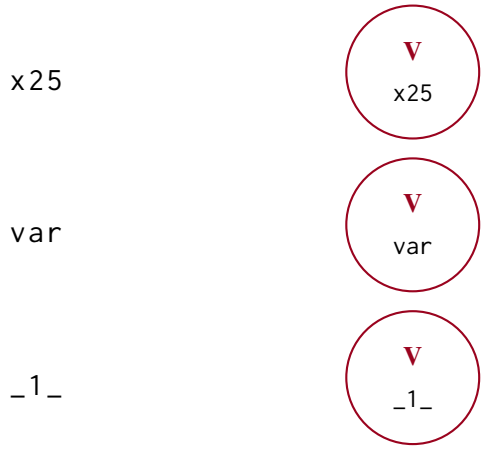


Figure 6.1: Several valid identifier references on the left with their corresponding AST nodes on the right.

Anonymous Functions (Lambdas)

Func's syntax for anonymous functions was inspired mainly by CoffeeScript. Lambdas in Func do not allow omitting parentheses around parameters and use a single arrow (`->`) instead of a double arrow (`=>`). Parentheses are mandatory regardless of how many arguments the lambda takes. This is not the case in JavaScript, as it allows omitting parentheses in arrow functions when they only take a single argument. I have decided to sacrifice this liberty in favor of consistency. I have chosen the single arrow (`->`) over a double arrow (`=>`) because of its traditional use in describing functional relationships (both in theoretical work and in languages such as Haskell).

Lambdas can be defined as taking multiple arguments in a C-like fashion. However, semantically, all lambdas are curried (i.e., transformed into accepting only a single argument). In this respect, Func behaves exactly like the λ-calculus and does not support multi-argument functions. Figure 6.2 demonstrates what happens internally when the programmer defines a lambda that takes two arguments. I justify adhering to the C-like

syntax (as opposed to the λ-calculus-based syntax seen in Haskell and ML) in the next segment, which discusses function applications.

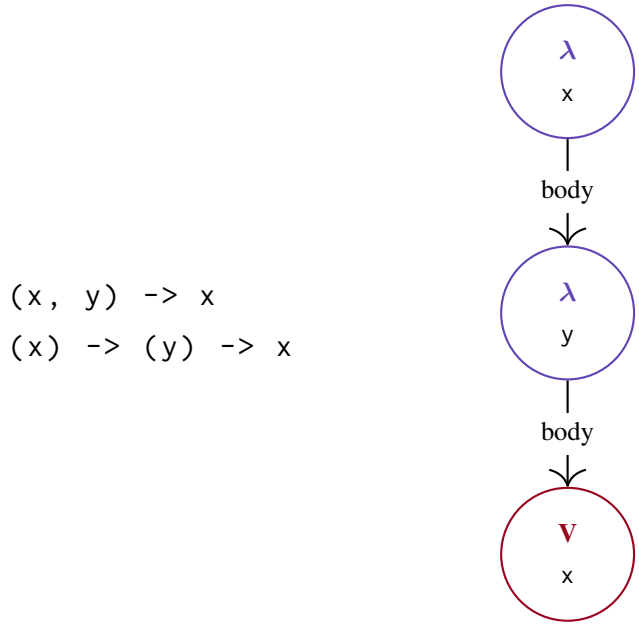


Figure 6.2: The two expressions on the left are fully equivalent (i.e., multi-argument functions can be viewed as syntactic sugar for a curried function chain). The AST on the right represents both of them.

Function Applications

Function applications also take their syntax directly from JavaScript. Purely functional languages generally use spaces for signifying function application to emphasize their similarity to the λ-calculus (e.g., ML, Haskell, PureScript). However, Func keeps JavaScript's C-like postfix call syntax, the main reasons being its familiarity and its aesthetic similarity to mathematics. The function's name, followed by its arguments enclosed in a pair of parentheses, is an almost universal way to represent function calls, present in almost all popular programming languages and traditional mathematical notation. While λ-calculus-like syntax would perhaps make more sense regarding the underlying model, it would also require getting used to, as it is generally considered strange by many programmers (especially those coming from the JS ecosystem).

Having a C-like function call syntax means that there are many syntactically different ways to call the same function (i.e., the programmer is free to group arguments arbitrarily). Figure 6.3 shows this property.

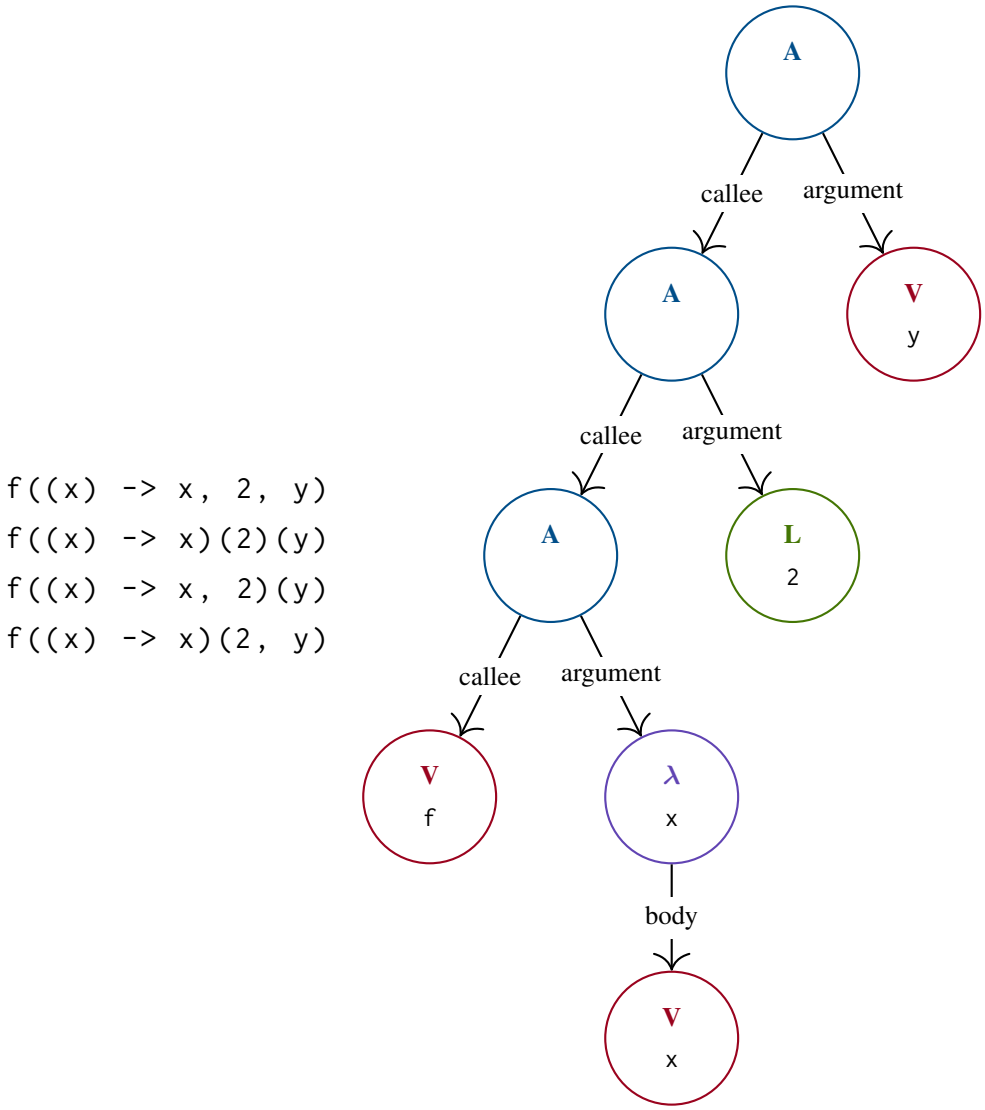


Figure 6.3: There are a total of four ways to apply a function that "takes three arguments." They are shown on the left. All four expressions result in the same AST, which is shown on the right. In reality, `f` is not a function that takes three arguments, but rather three curried functions that each take a single argument.

Literals

At the time of writing, Func includes syntactic support for six kinds of literals: booleans, numbers, characters, arrays, tuples, and strings[2]. This segment considers only the literals that directly correspond to the internal language (i.e., booleans, numbers, characters, and arrays). The remaining two are purely syntactic additions and are thus covered by

[2]Although the internal language also supports big integer literals, the syntax, at the time of writing, does not. The full list of semantically supported literals is available in 4.2.2

Subsection 6.2.3.

All literals have a traditional look that depends on their type. The lexical grammar for numeric literals is equivalent to JavaScript's (i.e., it supports decimals, scientific notation, leading dots, etc.)[3]. The lexical grammar for boolean literals consists of only two constant strings, `True`, and `False`. Character literals are enclosed in single quotes to differentiate from string literals[4]. Array literals look the same as they do in JavaScript. Figure 6.4 shows examples of built-in literals with their corresponding AST nodes.

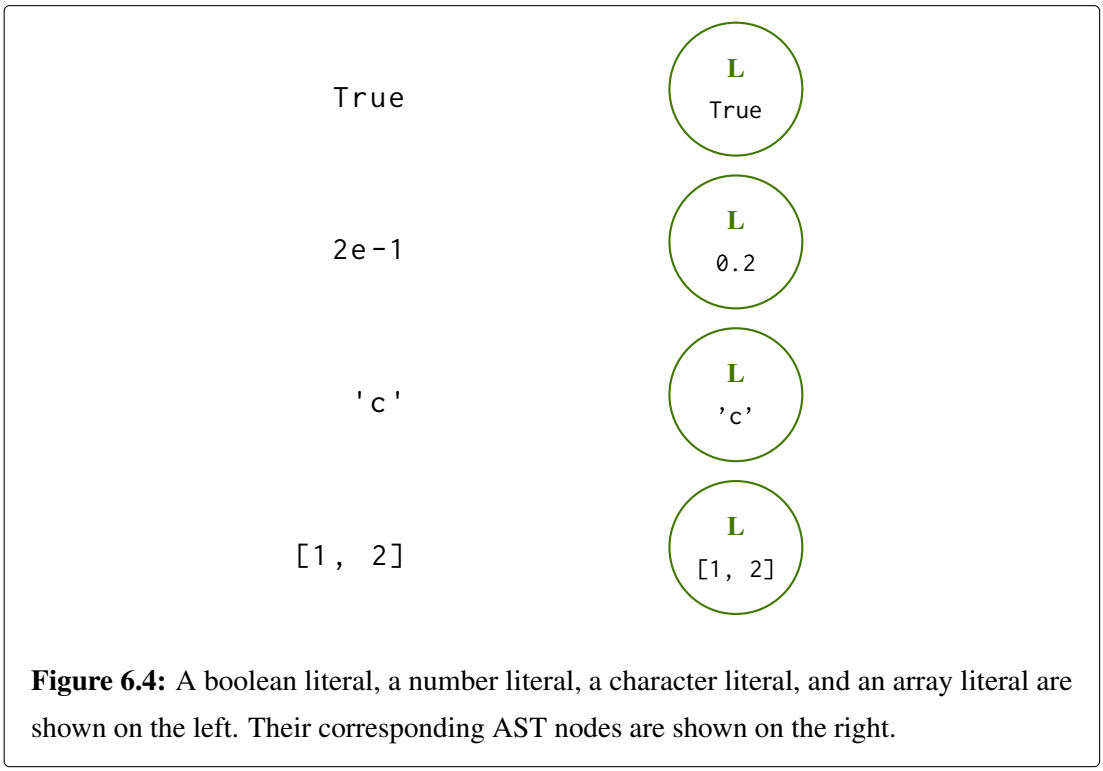


Figure 6.4: A boolean literal, a number literal, a character literal, and an array literal are shown on the left. Their corresponding AST nodes are shown on the right.

Let Expressions

The let expression syntax introduced in 4.2.2 to describe the theoretical model directly corresponds to the expression's surface syntax. JavaScript does not have let expressions, so Func takes its syntax for let expression from Haskell. The same syntax is used in PureScript, OCaml, Standard ML, and most other functional languages. Figure 6.5 shows a let expression and its AST.

[3]It does not support octal and hexadecimal literals at the time of writing.

[4]String literals are syntax sugar for character arrays, see 6.2.3 for more info

```
let id = (x) -> x in id
```

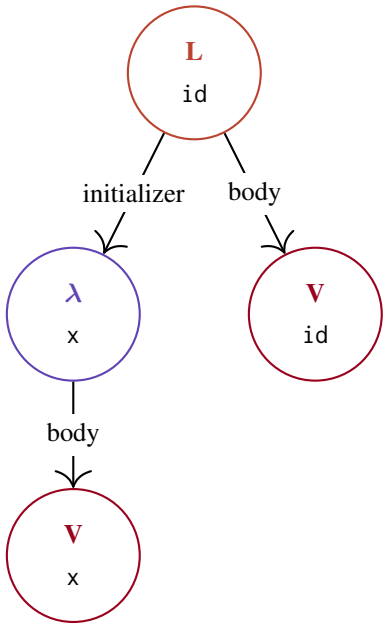


Figure 6.5: A valid let expression is shown on the left. Its AST equivalent is shown on the right.

Conditionals

Conditionals in Func have a slightly different form than one might expect – they share their syntax with Python's conditional expression. The more conventional option was copying Haskell's syntax (i.e., `if x then y else z`). Nonetheless, I have decided to base the syntax on Python because I liked the expression's symmetry (the condition in the middle, the if-branch on the left, the else-branch on the right) and its terseness. With clever use of whitespace, chained conditional expressions written in this form look like case statements and are very readable. Figure 6.6 shows how the parser translates a conditional expression into the AST.

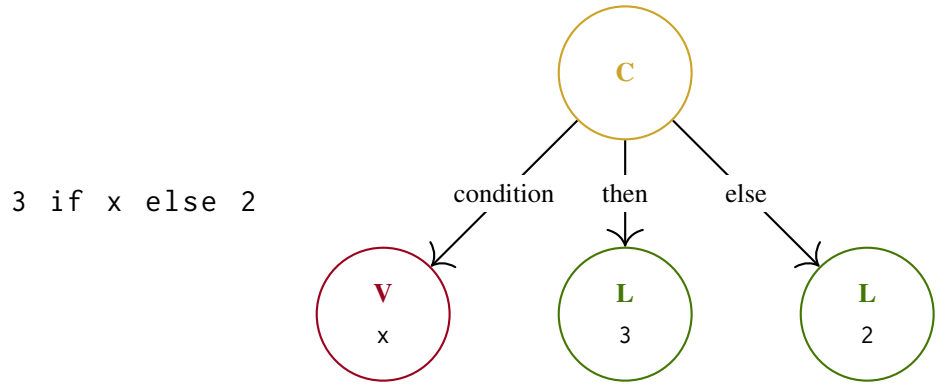


Figure 6.6: A simple conditional expression is shown on the left. Its representation in the AST is shown on the right.

6.2.2. Statement-level Syntax

This subsection explores the syntax of the language's three statement-level nodes: declarations, function definitions, and modules. Declarations and function definitions are the only two nodes allowed at the top-level of a Func program. The module node collects all the statement nodes and represents a full program. Every semantically valid statement constitutes a valid Func program on its own (i.e., the module has only a single child).

Declarations

Declarations in Func are relatively minimalistic. The language does not require any keywords when declaring variables. It is enough to assign an expression to a name, as shown in Figure 6.7.

```
pi = 3.14
id = (x) -> x
t = True
```

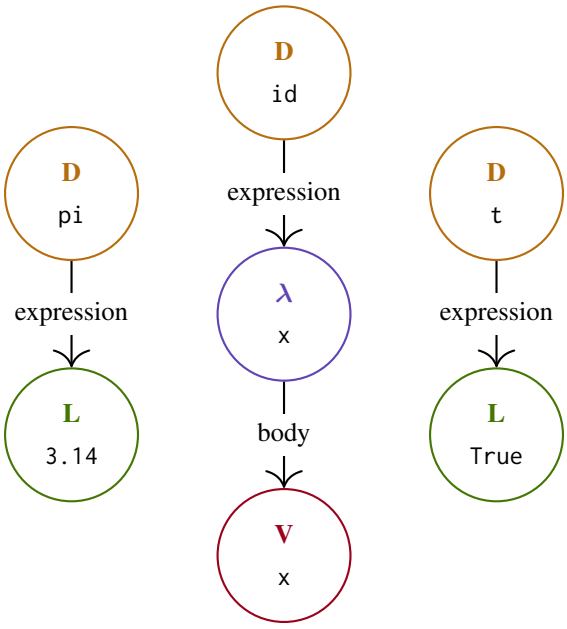


Figure 6.7: Several top-level declarations are shown on the left. Their corresponding ASTs are shown on the right.

Function Definitions

As mentioned in 4.2.2, a function defined with a function definition is almost entirely equivalent to a function defined with a declaration (i.e., by assigning a lambda to a name). In fact, the generated code displays no differences between them whatsoever (as demonstrated in Subsection 7.8). The only difference between the two statements is in their semantics – function definitions can recurse, while anonymous functions assigned to variables cannot.

This property was more necessary than it was a design choice. The compiler must know about the function's name inside its body for recursion to be possible. A possible solution is to make the compiler push the declared name into the context before processing the declaration's right-hand side. This solution would indeed enable recursion but would also create new issues. Consider the following Func code[5]:

```
1 fact = (n) -> 1 if n == 0 else n * fact(n - 1)
```

Upon encountering the declaration, the compiler adds then name `fact` to the context and proceeds to process the expression on the right. When it encounters the identifier

[5] I have not yet explained how Func supports binary operators (i.e., ==, *, -), but assume it does the sake of argument.

`fact` inside the function's body, it recognizes it and does not produce a name error. However, performing this action on each declaration would open the door to a whole new class of errors. For example:

```
1  x = x + 1
```

The code does not make semantic sense, but the compiler would deem it correct. The generated JavaScript code would then fail at run-time with a reference error[6]. To avoid such a scenario, the compiler must somehow know that it is processing a statement defining a function, which is why Func supports recursion only in named function definitions. Section 1 already explored recursion in Func from a type-checking perspective. The text will continue to ignore the AST difference between declarations and function definitions. Figure 6.8 shows a function definition and its virtually equivalent representation in the AST. I have chosen to start function definitions with the keyword `func` because people inherently treat functions differently than all other values, despite them being first-class citizens. I believe that function definitions starting with keywords are much more natural for humans to parse and reason about. With that said, I have felt that `func` is long enough to be clear and short enough to avoid being tedious to write. The word `function` is much longer but does not provide any new information for the programmer.

[6]Funnily enough, GHC accepts this expression. Executing it results in an infinite loop. This is a consequence of Haskell's non-strict semantics.

```
func apply(f, x) = f(x)
```

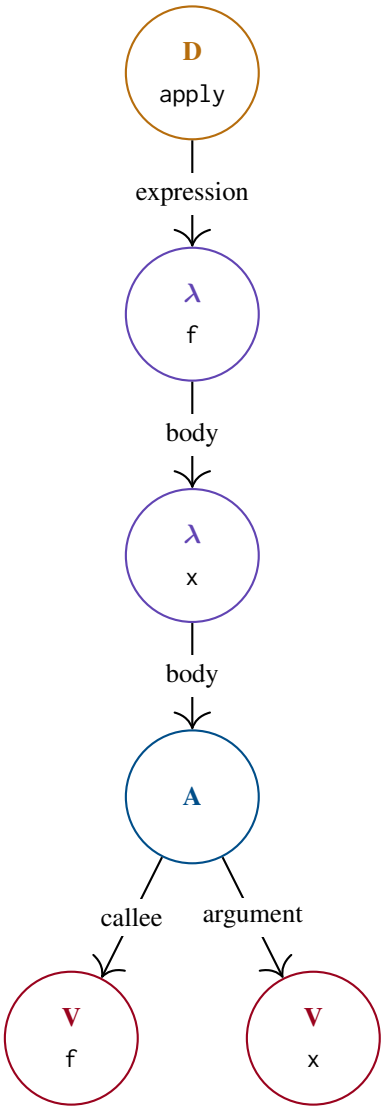


Figure 6.8: The function definition shown on the left, for the purposes of this paper, corresponds to the AST shown on the right.

The Module

Finally, the module node serves as the root of the AST for every program written in Func. A module simply gathers all top-level statements under a common root node, as shown in Figure 6.9. Statements can be separated with line breaks, semicolons, or both at the same time.

```
id =(x) -> x
e = 2.72; id2 = id
```

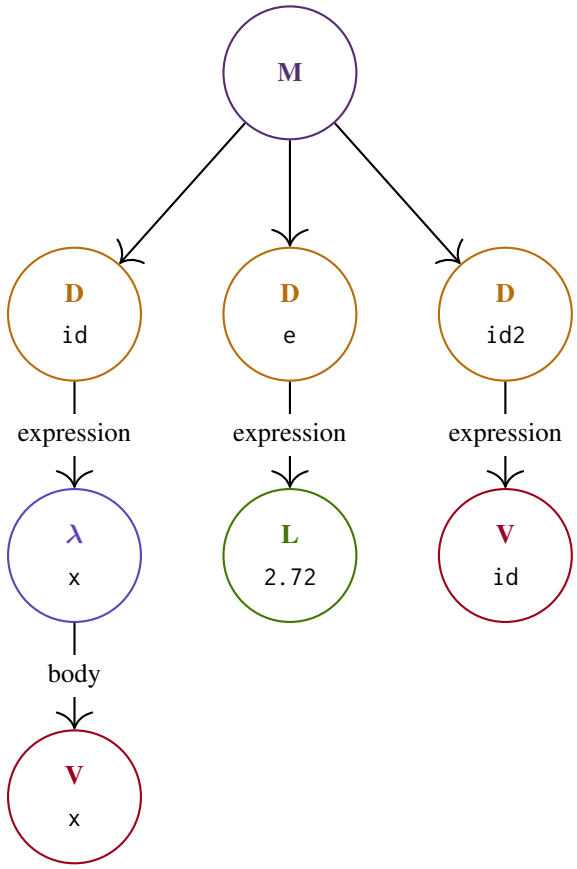


Figure 6.9: A simple but fully valid Func program is written on the left. Its corresponding AST is shown on the right.

6.2.3. Syntactic Additions

Func supports several programming constructs that do not directly map into the AST. Their typing and behavior can easily be represented through more fundamental operations present in the core language. Such constructs are called derived forms or *syntactic sugar*, while the process of replacing a derived form with its lower-level definition is called *desugaring* (Pierce, 2002). Syntactic sugar allows adding useful features to the language's surface syntax (i.e., the language used by the programmer) without adding complexity to its internal model (Pierce, 2002). Keeping the internal model as minimal as possible helps prove theorems about the language (e.g., type safety, which was proven in 5.4.3) and considerably simplifies code generation. In other words, it pays off to keep the AST as simple as possible.

Strings

Strings are the simplest form of syntax sugar in Func. They are merely a simplified way of representing arrays of characters. This is by no means a new concept and is used almost everywhere, from C to Haskell. Essentially, the value stored in `s1` is fully equal to the value stored in `s2`:

```
1 s1 = "sugar"
2 s2 = ['s', 'u', 'g', 'a', 'r']
```

The compiler parses `s1` as `s2`, which then maps directly into the AST, as documented in Subsection 6.2.1.

Operators

Operators play a vital role in virtually all modern programming languages. An operator is a special symbol used to denote an operation or, in the context of programming languages, a function call (Hudak, 1989). Binary operators represent functions taking two arguments and are typically written using infix syntax. Unary operators represent single argument functions and are traditionally written as prefixes.

In Func, operators are semantically regular functions written in a syntactically different way. As such, the parser can easily translate them into the AST. Of course, since Func only supports unary functions, all functions representing binary operators are curried. Currying opens a semantic possibility of partially applying binary operators, but that requires special syntax (operator referencing is described in one of the following segments of 6.2.3). All operators follow the same associativity and precedence rules as they do in JavaScript[7]. Func adds an operator for concatenating arrays (++), which comes from Haskell and is not available in JavaScript. The concatenation operator shares its associcativity and precedence with regular summation operators (i.e., + and -). Figure 6.10 shows how the language parses a simple sum.

[7] JavaScipt's operator precedence and associativity table is available on MDN.

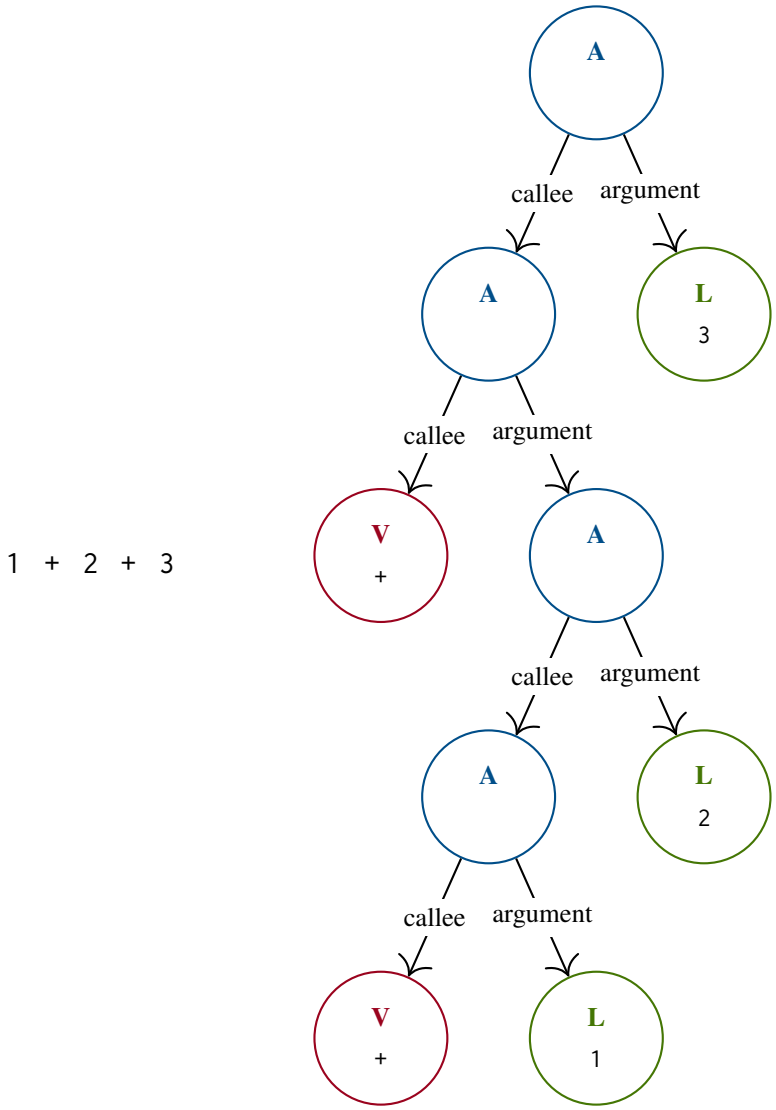


Figure 6.10: The AST of a simple sum. Since the binary operator `+` is curried and left associative, the adders bind to the left, one by one. The expression `1 + 2 + 3` can be interpreted as `plus(plus(1)(2))(3)`.

The Composition Operator

Function composition is an operation that takes two functions f and g and produces a third function h such that $h(x) = f(g(x))$. It is a standard mathematical operation that nicely fits inside functional programming languages.

Mathematicians denote function composition with a circle ($\circ$), but since the symbol is difficult to access on most keyboard layouts, programming languages generally chose a different way of representing it. For example, Haskell uses a dot (`.`), F# represents it

using >>, while PureScript chooses <<<. Like Haskell, Func also uses a dot for writing function compositions.

Translating function compositions into the core language is relatively straightforward. Consider the following program in Func:

```
1  c1 = f . g . h
2  c2 = ((x) -> x * 2) . f
3  c3 = (c1 if a else c2) . g
```

The code shows three different function compositions: one constructed using only identifier references (`c1`), and two constructed using other kinds of expressions (`c2` uses a lambda, while `c3` uses an inline conditional expression). The compiler can desugar all of them into anonymous functions[8]:

```
1  c1 = (y) -> f(g(h(y)))
2  c2 = (y) -> ((x) -> x * 2)(f(y))
3  c3 = (y) -> (c1 if a else c2)(g(y))
```

Since the desugared code is nothing more than a regular lambda with function applications in its body, it maps into the AST directly following the rules for lambdas defined in Subsection 6.2.1.

The Pipeline Operator

The pipeline is a construct used to perform a sequence of operations on a given value efficiently. It is very similar to function composition. The main inspiration for the operator's semantics comes from the Unix shell. Func is not the first general-purpose language to feature a pipeline operator—for example, both F# and Elixir support pipelines. In fact, the operator's syntax and semantics in Func are most similar to those it has in F#. To improve the generated code's efficiency, I have decided to implement this operator as a derived form instead of a primitive. The same applies to the function composition from the previous segment.

The input to each function in a pipeline is the previous function's output. In other words, the compiler can interpret a pipeline as several chained function applications. For example, the code:

[8] In reality, the compiler chooses a special name for the lambda's parameter, `$0`. Since Func's lexical grammar does not allow dollar signs in identifiers, this fully prevents name clashes. The compiler also uses dollar signs to prevent name clashes with JavaScript's reserved words.

```
pipeResult = 2 |> square  |> double  |> toString
```

is translated as:

```
pipeResult = toString(double(square(2)))
```

The construct has now become a series of function applications and thus has a direct representation in the internal language (shown in Subsection 6.2.1).

Operator References

As mentioned in one of the previous segments, operators in Func are functions. However, their special syntax prevents them from being proper first-class values – the programmer cannot treat them as they would treat a normal function because it would cause a parsing error. This limitation has proven to be very inconvenient in some cases. For example, assuming that the programmer has a `reduce` function with its usual semantics at their disposal, this is how they would have to calculate the sum of all array members:

```
array = [1, 2, 3, 4]
sum = reduce((x, y) -> x + y, array)
```

The lambda on the second line exists solely because the programmer has to wrap the operator `+` inside a function before passing it to `reduce`. To remove the need for such meaningless abstractions, Func provides a way to reference operators directly – by enclosing them in bakcticks[9]. Consider the same code that uses an operator reference instead of a wrapper lambda:

```
array = [1, 2, 3, 4]
sum = reduce(`+`, array)
```

Operator references are full-fledged expressions and do not come with any syntactic limitations. Since they solve a problem that does not exist in the language's semantics, translating them into the AST is trivial – the parser treats them as identifier references, as shown in Figure 6.11.

[9]The backtick syntax was inspired by Nim.

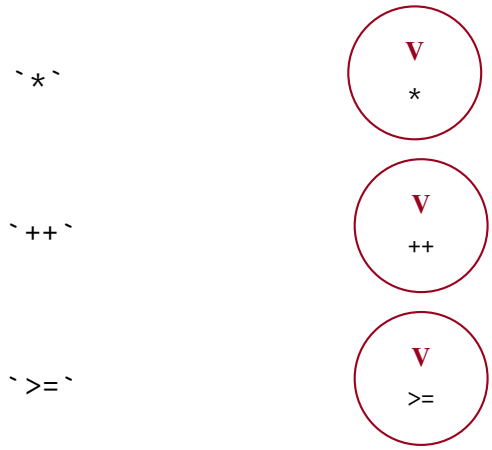


Figure 6.11: Several valid operator references are shown on the left. Their corresponding AST nodes are shown on the right. The parser simply treats operator references as identifier references to builtin functions named with non-alphanumeric symbols, while accounting for their infix syntax.

Besides enabling direct operator referencing, backticks also address the previously mentioned syntactic restriction of partially applying binary operators. More precisely, Func's semantics allows applying binary operators to a single argument (this is semantically nothing more than a regular function application), but its syntax does not. By enclosing a binary operator inside backticks, the programmer can also bind an expression to it on either side. For example, assuming that the function `map` is available and operates under its usual semantics, this is how doubling all numbers in an array looks without using backticks:

```
1  numbers = [1, 2, 3, 4]
2  doubles = map((n) -> n * 2, numbers)
```

And this is how it looks with backticks:

```
1  numbers = [1, 2, 3, 4]
2  doubles = map(`*2`, numbers)
```

Partial operator application is simple to translate into the AST when the bound argument is on the left – it is enough to partially apply the operator. The semantics have always allowed this. However, when binding the right argument, the compiler has to insert a wrapper lambda (not all operators are commutative):

```
fourDiv1 = `4/`
fourDiv2 = `/`(4)

divFour1 = `/4`
divFour2 = (y) -> y / 4
divFour3 = (y) -> `/`(y, 4)
```

The declaration of `fourDiv1` is equivalent to the declaration of `fourDiv2`. Declarations `divFour1`, `divFour2`, and `divFour3` are all mutually equivalent. `fourDiv2` and `divFour3` both directly map into the AST using the already established rules.

Array Spreads

The array spread operator aims to make working with array literals more pleasant. The operator is present in JavaScript (`[...x, y]`), Python (`[*x, y]`), and many other languages. It works by unpacking the inner array and interpolating it inside the enclosing array. In the following code, variables `a1` and `a2` contain the same value:

```
inner = [2, 3]
a1 = [1, *inner, 4, *[5, 6], 7]
a2 = [1, 2, 3, 4, 5, 6, 7]
```

Desugaring the array spread literals is a matter of representing them as function applications. Func does this with the help of a built-in operator for array concatenation (mentioned in the segment that discusses binary operators). The operator is called `++` and has type $\forall\alpha\ Array\langle\alpha\rangle \to Array\langle\alpha\rangle \to Array\langle\alpha\rangle$. The expressions assigned to `b1` and `b2` below are semantically equivalent:

```
inner = [2, 3]
b1 = [1, *inner, 4]
b2 = [1] ++ (inner ++ [4])
```

The compiler interprets all array spreads as expressions analogous to `b2`, which reduces them to binary operator applications. From here, it is simple to turn them into regular function applications (as described in the segment about desugaring operators in 6.2.3).

7. Code Generation

This Chapter explains how Func's internal model maps into JavaScript. More specifically, it demonstrates how to turn the AST defined in 4 into executable JavaScript code. The text assumes familiarity with JavaScript. It will not explain any language constructs besides IIFEs.

The code generation process only requires the AST. It does not in any way depend on Func's concrete syntax, meaning that the language's syntactic grammar can vary entirely independently from the generated code. The same applies to the type system – the type checker only verifies the AST. It does not change it in any way. Each compiler module can be easily changed without affecting other modules, as long as it does not alter the AST. For example, it is possible to introduce an entirely different type system without adapting the parser or the code generator. In fact, Func could theoretically forgo type checking altogether and generate code directly from the unchecked AST produced by the parser. This change would effectively turn Func into a dynamically typed language with run-time behavior identical to JavaScript's. To add a new compiler target (e.g., WASM, x86 assembly), it is enough to define and implement the appropriate code generator (i.e., a procedure for mapping the AST into the target language). The compiler currently only defines the code generator for JavaScript, which this Chapter describes.

Before examining specific language constructs and their respective representations, it is useful to say a few words about *Immediately Invoked Function Expressions (IIFEs)*, one of JavaScript's most useful idioms. Section 7.1 explains the semantics of Immediately Invoked Function Expressions and explains their contribution in transpiling expression-based languages. Section 7.2 goes through all AST nodes defined in 4 and explains how the compiler translates each one into JavaScript[1].

[1]The implementation of Func's code generator can be found here.

7.1. Immediately Invoked Function Expressions (IIFEs)

Functions are first-class values in JavaScript, and the language provides two ways of defining them – with declarations or with expressions. Function declarations (also called function statements) are not relevant for the discussion and will not be further explored. On the other hand, function expressions can, by their definition, be a part of any other expression. This property becomes especially useful if a function expression is invoked immediately.

Immediately Invoked Function Expressions, or IIFEs for short, are a fundamental JavaScript idiom primarily used to simulate a lexical scope using JavaScript's function scoping. However, what makes them excellent for transpiling is their ability to turn statement-level code into expression-level code, essentially providing a mechanism to turn arbitrarily long JavaScript programs, consisting of many statements, into full-fledged expressions. Consider a contrived example:

```
1  const x = 1 + 2 + (function() {
2      const rand = Math.random();
3      if (rand > 0.5) {
4          return 10;
5      } else {
6          return -10;
7      }
8  }()) + 3 + 4;
```

Here, an IIFE was used to embed sequential statement-based code inside a sum. No matter how complex or long a JavaScript program is, it can always be made into an expression using an IIFE.

This idiom is a potent tool used by virtually all modern transpilers. It allows them to harness the full expressive power of JavaScript for simulating a particular language construct because, in the end, they can trivially transform their code into an expression, which is the most fundamental part of the language. It can then naturally compose these wrapping expressions with other expressions and statements.

7.1.1. The Implications for Func

Func is an expression-based language. Some of the expressions it supports (or may support in the future) do not have their equivalent in JavaScript (e.g., the let expression). However, thanks to IIFEs, the code generator can use whatever means necessary to

represent arbitrary language constructs without the restriction of having to transpile expressions exclusively into expressions. If it has to use statements to simulate a particular expression, it can. All it has to do to ensure its usability at an expression level is wrap the resulting code inside an IIFE. As evident by the next Section, Func makes heavy use of this mechanism.

7.2. Generating Code from the AST

Func's code generator works recursively. It starts from the root node (i.e., the module) and works its way down. Whenever it encounters a child expression node, it calls the appropriate subroutine to transpile it. In the code snippets, a recursive call that transpiles expression `expr` is denoted using angled brackets, as `<expr>` [2].

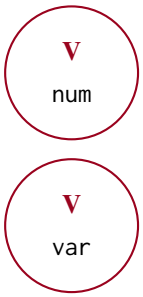
Subsection 7.2.1 documents the transpilation of expression level nodes, while Subsection 7.2.2 shows how the transpiler generates code for statements.

7.2.1. Transpiling Expressions

Identifier References

Since valid identifier names in Func are a proper subset of valid identifier names in JavaScript, the transpiler translates most of them verbatim. It must only make sure to change the names which correspond to words that carry special meaning in JavaScript. These include the language's keywords (i.e., `function`, `const`), literals (e.g., `undefined`, `null`), future reserved words (e.g., `enum`), future reserved words in strict mode (e.g., `interface`, `implements`), old reserved words (e.g., `abstract`, `boolean`), and sometimes reserved words (e.g., `await`, `yield`). These names are prefixed with two dollar signs inside the generated code (`$$`) to avoid breaking the JavaScript interpreter. Figure 7.1 lists several examples.

[2] Interpret it as "transpile the node `expr` and put the resulting string here."

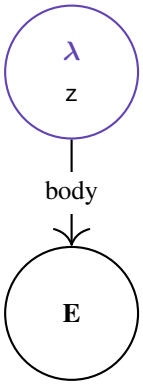



```
num
$$var
```

Figure 7.1: AST nodes representing identifier references are shown on the left. The generated code is shown on the right. The first row shows a standard identifier reference. The second row shows how the compiler treats a name that clashes with one of JavaScript's reserved words.

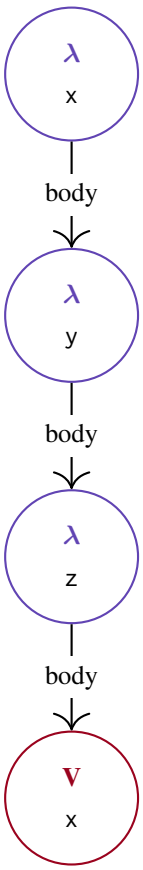
Lambdas

Since Func curries all functions, the compiler transforms multi-argument functions into their curried form, as shown in Figure 7.2. Figure 7.3 shows the code generated for a specific lambda. Of course, the parser did the real transformation when constructing the AST. All the code generator has to do is define a procedure for transpiling unary function since only those can exist in the AST.




```
function (name) {
    return <body>;
}
```

Figure 7.2: A generic lambda node is shown on the left. The generated JavaScript code is shown on the right. The transpilation then continues recursively inside the lambda's body.



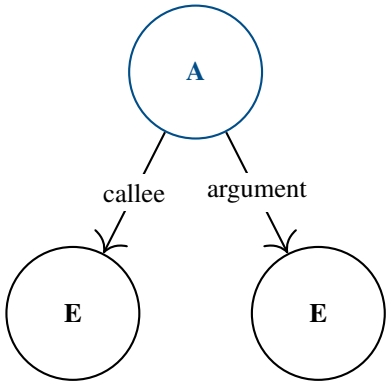

```
function (x) {
    return function (y) {
        return function (z) {
            return x;
        };
    };
}
```

Figure 7.3: This example shows how the transpiler handles a specific lambda. The lambda's AST node on the left is transpiled into the JavaScript code on the right.

Applications

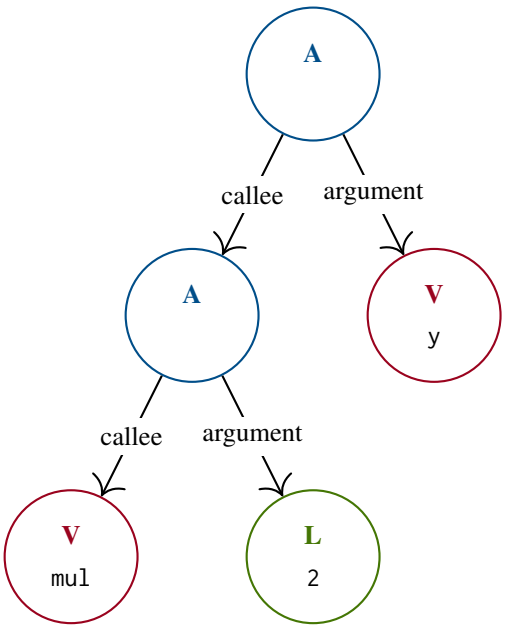
Since all functions are curried, applications have to be applied one argument at a time. Figure 7.4 shows how the compiler transpiles a generic application, while Figure 7.5 shows a specific example.

If an operation can be represented in JavaScript using operators, the compiler translates it accordingly. For example, it translates the expression `1 * (2 + 3) - 4` into `1 * (2 + 3) - 4`. Since the internal model does not differentiate between regular functions and operators, the code generator performs pattern matching on the AST to recognize which expression constitutes an operator call. Not all operators in Func necessarily have a corresponding operator in JavaScript. One such example is the concatenation operator (++). Similarly, an operator in JavaScript does not have to be an operator in Func.




```
<callee>(<argument>)
```

Figure 7.4: A generic application is shown on the left. It maps to the code shown on the right.

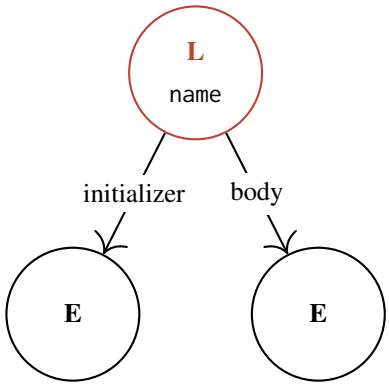



```
mul(2)(y)
```

Figure 7.5: When transpiling functions "taking multiple arguments", the arguments have to be applied one by one.

Let Expressions

Unlike all expressions discussed so far, let expressions do not have their equivalent in JavaScript. Luckily, the compiler can use an IIFE (described in Section 7.1) to implement them smoothly. Figure 7.6 shows how to translate a generic let expression.

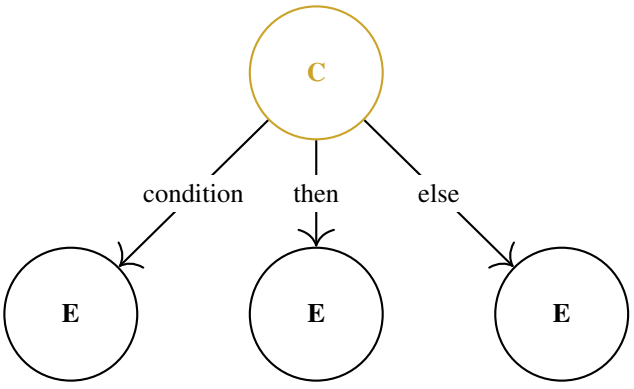



```
(function () {
    const name = <initializer>;
    return <body>;
}())
```

Figure 7.6: A generic let expression is shown on the left. The generated JavaScript code is shown on the right.

Conditionals

Conditionals could be transpiled directly into expressions using JavaScript's ternary operator. However, I have instead decided to use traditional if statements because it makes the generated code more readable. Since the generated code uses if statements, the compiler must finish by wrapping it inside an IIFE. Figure 7.7 shows how it approaches a generic conditional expression.



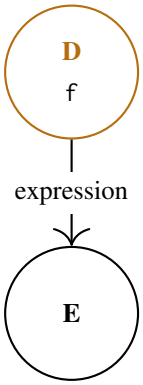

```
(function () {
    if (<condition>) {
        return <then>;
    } else {
        return <else>;
    }
}())
```

Figure 7.7: A generic conditional expression is shown on the left. The generated JavaScript code is shown on the right.

7.2.2. Transpiling Statements

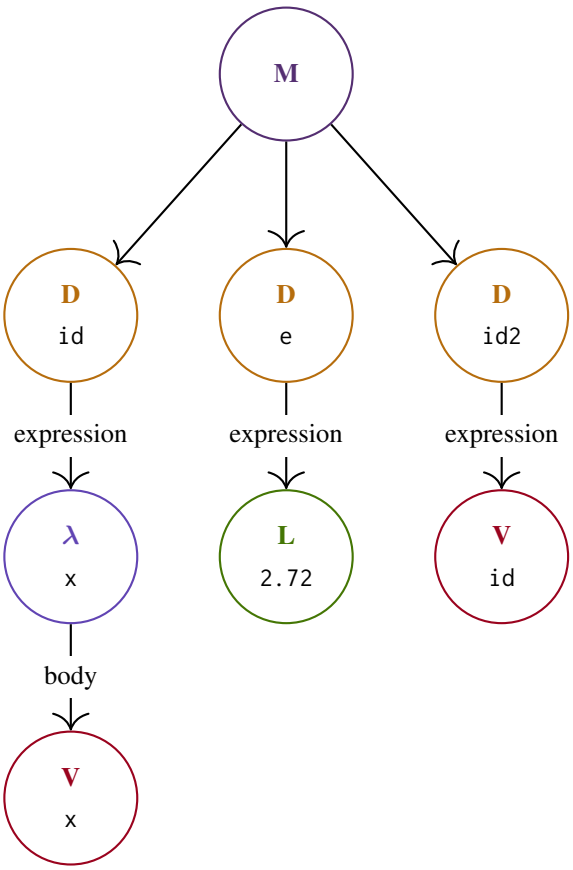
Transpiling statements from Func to JavaScript is much simpler than transpiling expressions, mostly because Func's statement syntax is very similar to JavaScript's. Since

Func is a purely functional language, all identifiers in the generated code are declared as constants (i.e., using JavaScript's `const` keyword). Figure 7.8 shows how to transpile a generic declaration. Figure 7.9 shows how to transpile a specific simple module with three declarations.




```
const f = <expression>;
```

Figure 7.8: A generic declaration is shown on the left. The generated JavaScript code is shown on the right.




```
const id = function (x) {
    return x;
};

const e = 2.72;

const id2 = id;
```

Figure 7.9: Transpiling modules comes down to transpiling statements and joining the results with line separators. The module AST is shown on the right. The generated code is shown on the left.

8. Future Work

The text so far documented the design of a fully functional general-purpose programming language. However, the presented design did not cover some of the initially planned features and design goals. This Chapter lists the missed features and gives a short overview of plans for further developing the language.

Syntactic Support for Big Integers

As mentioned when defining the internal language in 4.2.2, Func includes semantic support for big integers (i.e., they exist in the AST, and the type system knows how to treat them). The run-time can trivially support the feature since JavaScript includes big integers natively. The only thing big integers do not yet have is syntactic support (i.e., there is no way to write a big integer literal in Func). The syntax for big integers, while trivial to add, will not be of much use until Func starts supporting ad hoc polymorphism – all arithmetic operators currently have monomorphic type signatures and work only with numbers. The current plan is to differentiate big integers from numbers the same way JavaScript does it, by including a trailing `n`:

```
1  a = 15
2  b = 15n
```

In the code above, `a` would have type `Number`, while `b` would have type `Bigint`.

User-defined Types

Func does not give programmers a way to define their custom data types. The plan is to add algebraic data types to the language, basing their syntax and semantics on Haskell. The compiler already includes semantic support for custom type constructors, but it would have to be extended to include union types and product types.

Named Records

As stated when discussing literals for common structures in 3.4.3, the initial design plan included literals for named records. Named records are collections of named fields holding values of possibly different data types and are present in many programming languages (e.g., C has structs, JavaScript has objects). The thesis did not implement named records in any way (semantically or syntactically). The plan is to implement records in a way semantically similar to Haskell. The transpiler would translate them into JavaScript objects.

Ad Hoc Polymorphism

Func originally planned to support ad hoc polymorphism through type classes, as described during the initial discussion on the concept in 3.4.5. Type classes a clean, modern approach to ad hoc polymorphism originally designed for Haskell (Wadler & Blott, 1989). They are an extension to the Hindley-Milner type system, which Func already uses. Ad hoc polymorphism is vital for increasing the language's overall usability, and there are still plans to support it using type classes in a future iteration of the language.

Optional Type Annotations

A segment in Subsection 3.2.1 decided to focus Func's type system on type inference while also enabling optional type annotations for documentation and specification purposes. Func indeed has full type inference (i.e., no type annotations are necessary). However, it does not have the promised ability to specify types manually. There are still plans to implement the feature. It is simple to support it semantically, but I did not yet decide its syntax.

Pattern Matching

Pattern matching, explained in detail in 3.4.4, is a declarative language construct used to express function mappings concisely. Func still plans to support pattern matching. Since it is relatively challenging to implement the construct, and Func is already fully functional, pattern matching was postponed until a future version of the language.

9. Conclusion

This thesis's goal was to develop Func – a high-level, general-purpose programming language that emphasizes JavaScript's expressive power while minimizing the impact of its harmful properties. The language was meant to be used for data processing, mathematical calculations, and describing functional relationships. These use cases best fit the functional programming paradigm. Func's internal language was thus based on the λ-calculus.

Func attempted to solve the lack of type safety in JavaScript with the provably correct Hindley-Milner type system. To accomplish this, the type system had to interpret the core language in terms of the simply typed λ-calculus extended with several vital constructs such as the fixed point operator. The language was kept simple enough to ensure its eligibility for Milner's type inference Algorithm $\mathcal{W}$. Thanks to this, Func has provably sound and complete type inference, and a decidable and sound type system.

Func's syntax aimed to provide a cleaner interface to JavaScript's semantics. Due to the language being based on the λ-calculus, its syntax mostly consists of expressions. Thanks to the expression-based syntax, I removed most of JavaScript's syntactic noise. The core language also proved powerful enough to easily support several high-level derived forms, such as binary operators and pipelines.

Func code successfully translates into executable JavaScript. However, the compiler's modular implementation makes code generation fully independent of all other modules (excluding the AST). The compiler could thus be easily extended to support targets other than JavaScript.

Finally, I can conclude that the design process and the compiler's implementation have been successful. Func is a fully functioning high-level general-purpose programming language.

BIBLIOGRAPHY

Christopher Allen & Julie Moronuki. *Haskell Programming from First Principles*. Department of Computing, Imperial College London, 2018.

Jeremy Ashkenas. Initial commit of the mystery language, 2009. URL https://github.com/jashkenas/coffeescript/commit/8e9d637985d2dc9b44922076ad54ffef7fa8e9c2.

Jeff Atwood. The principle of least power, 2019. URL https://blog.codinghorror.com/the-principle-of-least-power/.

Lars Bak & Kasper Lund. Flow, 2020. URL https://news.dartlang.org/2015/03/dart-for-entire-web.html.

David Cassel. Brendan Eich on creating JavaScript in 10 days, and what he'd do differently today, 2018. URL https://thenewstack.io/brendan-eich-on-creating-javascript-in-10-days-and-what-hed-do-differently-today/.

Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

Dader. What are the strongest known type systems for which inference is decidable? Computer Science Stack Exchange, 2018. URL https://cs.stackexchange.com/q/90980. URL:https://cs.stackexchange.com/q/90980 (version: 2018-05-13).

Bojana Dalbelo Bašić & Jan Šnajder. Automated reasoning, 2019.

Luís Damas & Robin Milner. Principal type-schemes for functional programs. *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, stranica 207–212, 01 1982. doi: 10.1145/582153.582176. URL http://steshaw.org/hm/milner-damas.pdf.

ECMA. *Annotated ECMAScript 5.1*, 2015. URL https://es5.github.io/#x4.2.

Elm. *Elm Decidability Bug*, 2018. URL https://github.com/elm/core/issues/960.

Ben Fiedler. Typing is hard, 2021. URL https://3fx.ch/typing-is-hard.html.

Flow. *Flow*, 2020a. URL https://flow.org/.

Flow. *Nominal & Structural Typing*, 2020b. URL https://flow.org/en/docs/lang/nominal-structural/.

M. Gordon, R. Milner, L. Morris, M. Newey, & C. Wadsworth. A metalanguage for interactive proof in lcf. U *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, stranice 119–130. ACM, 1978. doi: 10.1145/512760.512773.

Ian Grant. The Hindley-Milner type inference algorithm, 02 2011.

Haskell. *A Gentle Introduction to Haskell - Functions*, 1998. URL https://www.haskell.org/tutorial/functions.html.

Christoph Hegemann. Type inference from scratch, 2019. URL https://www.youtube.com/watch?v=ytPAlhnAKro.

J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 1969. URL https://www.jstor.org/stable/1995158.

Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, Rujan 1989. ISSN 0360-0300. doi: 10.1145/72551.72554. URL https://doi.org/10.1145/72551.72554.

Paul Hudak, John Hughes, Simon Peyton Jones, & Philip Wadler. A history of Haskell: Being lazy with class. U *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, stranica 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937667. doi: 10.1145/1238844.1238856. URL https://doi.org/10.1145/1238844.1238856.

IDG. Microsoft augments JavaScript for large-scale development. https://thenewstack.io/brendan-eich-on-creating-javascript-in-10-days-and-what-hed-do-differently-today/, 2012.

Shriram Krishnamurthi. *Safety and Soundness*, 2015. URL https://papl.cs.brown.edu/2014/safety-soundness.html.

MDN. *Memory Management*, 2020. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management#mark-and-sweep_algorithm.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17, 348-375 (1978)*, 1978. URL https://homepages.inf.ed.ac.uk/wadler/papers/papers-we-love/milner-type-polymorphism.pdf.

Norswap. Is c# type system sound and decidable?, 2014. URL https://stackoverflow.com/questions/23939168/is-c-sharp-type-system-sound-and-decidable.

Slava Pestov. Swift type checking is undecidable, 2020. URL https://forums.swift.org/t/swift-type-checking-is-undecidable/39024.

Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, January 1987. URL https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/.

Dominik Picheta. Should we get rid of style insensitivity?, 2018. URL https://forum.nim-lang.org/t/4388.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

Benjamin C. Pierce. Types a la Milner, 2012. URL https://www.youtube.com/watch?v=carP8i6YSZs.

Rob Pike. Another Go at language design, 2010. URL https://youtu.be/7VcArS4Wpqk?t=178.

PureScript. *PureScript*, 2020. URL https://www.purescript.org/.

Didier Remy. Type systems for programming languages, 2020. URL http://cristal.inria.fr/~remy/mpri/cours.pdf.

Andreas Rumf. Why is it case/underscore insensitive?, 2016. URL https://github.com/nim-lang/Nim/wiki/Unofficial-FAQ#why-is-it-caseunderscore-insensitive.

Rust. Unsafe Rust, 2018. URL https://262.ecma-international.org/11.0/#sec-addition-operator-plus.

Kyle Simpson. *You Don't Know JS: Types & Grammar - 1st Edition*. Independently Published, 2015. URL https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/types%20&%20grammar/README.md#you-dont-know-js-types--grammar.

Christopher Strachey. Fundamental concepts in computer programming. U *Fundamental Concepts in Computer Programming*, 1967.

Fraklin Turbak, David Gifford, & Mark A. Sheldon. *Design Concepts in Programming Languages*. The MIT Press, 2008.

TypeScript. *TypeScript Design Goals*, 2014. URL https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals.

Steffen van Bakel. *Type Systems for Programming Languages*. Department of Computing, Imperial College London, 2001.

Peter Van Roy. Programming paradigms for dummies: What every programmer should know. U *Programming Paradigms for Dummies: What Every Programmer Should Know*, 04 2012.

P. Wadler & S. Blott. How to make ad-hoc polymorphism less ad hoc. U *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, stranica 60–76, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897912942. doi: 10.1145/75277.75283. URL https://doi.org/10.1145/75277.75283.

Philip Wadler. Monads for functional programming. U Manfred Broy, urednik, *Program Design Calculi*, stranice 233–264, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-662-02880-3.

J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. U *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, stranice 176–185, 1994. doi: 10.1109/LICS.1994.316068.

Wikipedia contributors. First-class function — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=First-class_function&oldid=990928406, 2020. [Online; accessed 18-February-2021].

Wikipedia contributors. Hindley–milner type system — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Hindley%E2%80%93Milner_type_system&oldid=1007263779, 2021. [Online; accessed 18-February-2021].

Designing a Novel Programming Language by Analysing the Industry's Current State

Abstract

JavaScript's lack of type safety, combined with many possible sources of erratic behavior, has long been a source of frustration for programmers. This thesis aims to tackle the mentioned problems by designing and implementing a new programming language that addresses JavaScript's issues and can work within the same ecosystem. The thesis introduces Func, a brand new high-level, general-purpose, purely functional programming language that features a clean, expression-based syntax with ML-style polymorphic type inference. It is transpiled directly into JavaScript. Func's intended use cases include data processing, mathematical calculations, and describing functional relationships.

Keywords: Func, lambda calculus, Hindley-Milner, Damas-Milner, Hindley-Milner-Damas, ML, Haskell, JavaScript, AST, algorithm W, IIFE

Razvoj modernog programskog jezika na temelju trenutnog stanja industrije

Sažetak

Nedostatak sigurnosti sustava tipova u programskom jeziku JavaScript, u kombinaciji s mnogim potencijalnim uzrocima nepredvidiva ponašanja, već dugo predstavlja izvor frustracija u programerskoj zajednici. Ovaj rad nastoji spomenute probleme riješiti dizajnom i implementacijom novog programskog jezika koji popravlja nedostatke JavaScripta i može raditi unutar njegovog postojećeg ekosustava. Rad predstavlja Func, potpuno novi funkcijski programski jezik visoke razine i opće namjene. Jezik ima jasnu sintaksu temeljenu na izrazima te koristi algoritam za polimorfnu rekonstrukciju tipova temeljen na programskom jeziku ML. Func se prevodi izravno u JavaScript. Planirano je korištenje jezika u obradi podataka, matematičkim izračunima, i opisivanju funkcijskih odnosa

Ključne riječi: Func, lambda račun, Hindley-Milner, Damas-Milner, Hindley-Milner-Damas, ML, Haskell, JavaScript, AST, algoritam w, IIFE