The Sailfish Programming Language: An Implementation Retrospective, the Manual, and a Formal Language Specification

The product of an Independent Study undertaken during Spring 2019
by Robert Durst under the guidance of Professor Dale Skrien.

1. Some Acknowledgements

First and foremost, thanks to Professor Dale Skrien for the advice and general guidance throughout the entirety of this study. Without someone to keep me on track or guilt me about the lack of a test suite, this project may never have compiled even the simplest of programs correctly. Also, thanks to my parents for pretending to understand what I was describing on the phone as I periodically rambled on about weird nuances of C++ and my friends Jonny and John for letting me borrow their coffee makers on late nights.

Contents

2. Background: How Sailfish Came to Be.

2.1. **The Backstory.** After a few rather unsuccessful weeks of playing around with the AngularJs framework during the summer of 2017, the start of my gap year away from Colby, I began exploring options for the upcoming Fall. I was interested in finding a learning opportunity, somewhere I could learn practical coding skills and learn them well. I had experience learning things on my own: with only about a year of CS classes under my belt, I picked up Swift and built a simple application for the Colby College school cafe. However, during an iOS mobile application internship the following summer, it became apparent my Swift skills were rather fragile, built not on a strong foundation of understanding, but on a basis of learning just enough to stay afloat. Later I would go on to embarrass myself in a summer 2017 internship interview, unable to define the simplest of concepts in Swift. Thus, for the Fall of 2017, I wanted to REALLY know something, from the bottom up. Eventually I discovered the Horizons School of Technology, a sixteen week full-stack JavaScript, coding bootcamp.

This bootcamp turned out to be exactly what I was looking for. After sixteen weeks of coding nearly 70 hours of JavaScript per week, I became quite good at JavaScript. I had obtained the strong foundation I desired. This turned out to be an incredible asset. As I transitioned from coding bootcamp to internship to full-time developer during the Spring/Summer of 2018, I began exploring more languages; at work I programmed in Go and on the side I learned Rust. While neither of these were particularly like JavaScript, in fact not at all, having learned a language so comprehensively, I had an idea of what it took to learn a new programming language. I developed side projects and wrote snippets in each of the languages, comparing and contrasting. Developing a love for Rust, I listened to the *The New Rustacean Podcast* on the way to work and attended RustConf 2018 in Portland, Oregon.

Coming back to Colby in the Fall, I now had a desire to learn more languages, and even a budding passion to create my own. However, at this point, I had no idea what it meant to create my own language. I Googled around a few times, but the terms parser and code generation were quite foreign. After completing a basics of Programming Languages class in the Fall of 2018, a class where we learned about the compilation process and self-studied the basics of three languages on our own (I chose Haskell, C, and C++). After this class I had a basic idea of what it meant to create my own language. Thus, I approached Dale Skrien, my CS361/CS461 Object Oriented Design professor, and asked if he would be my advisor for a programming language independent study. Intrigued, he accepted. The one requirement was that I go beyond his CS461 curriculum (in this class we would also be developing a compiler, however it was under the guise of learning good OOD principles). From this, Sailfish was born.

2.2. **Brainstorming – What was Not Meant to Be.** My initial idea for a programming language was something that would be like Solidity (a smart contract language for the Ethereum blockchain) but for Stellar. Quickly realizing Stellar had no concept of a VM, or any sort of environment to execute such a language, I then considered writing something for the Ethereum Virtual Machine (EVM), a machine sort of like the Java Virtual Machine (JVM) in that it defined a bytecode instruction set. However, after a week of indecision about whether to target the EVM or the newer webassembly based EVM, I decided to write a general purpose language. This general purpose language was going to have numerous functional features, bootstrap, and have a reference compiler written in Haskell. Unfortunately, after reading through some example Haskell compiler source code, I realized my lack of experience with Haskell would be an issue, so I decided to utilize C++. Furthermore, after releasing **Sailfishc 0.1.0** I came to the conclusion bootstrapping would be hard without any sort of io capabilities or any basic collection data structures (all of which were left out of the initial version of Sailfish). So, I finally ended up with Sailfish as it stands today, a C-like language that mostly, sort of works, with hints of functional features (no loops and some LISP function syntax).

3. A Retrospective on the Implementation of Sailfishc

The following section is a retrospective on my semester long attempt to implement a Sailfish reference compiler, Sailfishc. The section describes my basic approach, its successes, its shortcomings, and a list of limitations in the current version of the compiler. After reading this section, the reader should understand most of the design decisions of Sailfishc and its current state. This is helpful for the curious onlooker, the future contributor, and the compiler engineer (hopefully someone interested in implementing a second reference compiler themselves).

3.1. **Implementation #1.** Following good object oriented design principles, the initial implementation largely mimicked the codebase from the Bantam Java compiler I was simultaneously working on in my Object Oriented Design Class: each phase of the compiler was a separate class (or comprised of multiple classes), each node in the Abstract Syntax Tree (AST) was an object, the Symbol Table was an object, etc. By setting up the code this way, I was able to utilize various design principles to make the compiler code easier to read, easier to reason about, and easier to implement. An example pattern I used was the *Visitor Pattern*, a pattern that allowed for traversal of the AST once it was created. Thus, for semantic analysis and code generation, I had a very convenient method for doing type checking or code generation at each node.

While this implementation resulted in **Sailfishc 0.1.0**, it was far from an acceptable, or even correct program. Below I will describe some of the most notable issues I encountered.

3.1.1. *Issue 1: Incorrect Grammar.* For the most basic Sailfish source texts, the compiler worked well. However, due to issues in the design of the grammar certain correct Sailfish programs did not pass Semantic Analysis.

Consider the following section of the grammar:

Exponentiation := Expression '' Expression**
MultDivMod := Expression ['*' — '/' — '%'] Expression
Arithmetic := Expression ['+' — '-'] Expression
Comparison := Expression ['>', '>=', '<', '<='] Expression
Equivalence := Expression ['!=', '=='] Expression
LogicalComparison := Expression ['and' — 'or'] Expression
Assignment := Expression '=' Expression
FunctionCall := Expression '(' [Identifier] (',' Identifier)*')'
ExpressionOnlyStatement := Expression

Not only does this grammar not really work well with an LL(1) recursive descent parser, it also is unable to differentiate between functional calls and parenthesized expressions. Therefore, **Sailfishc 0.1.0** seriously limited the logical expression possibilities of the language.

For example, the following was deemed incorrect:

```
(10 + 9) < (1 + 2) or (true and true)
```

Unable to figure out these basic cases, I ended up cutting out some less trivial constructs and thus no collection types made it into **Sailfishc 0.1.0**.

3.1.2. *Issue 2: Transpilation.* Unfortunately, even for code that did pass semantic analysis, there was no guarantee that it would transpile correctly

(this issue still plagues Sailfishc today). When initially deciding to transpile to C, I believed it would be trivial – the original version of Sailfish seemed very C-like. However, the deeper I got into transpilation, the more I realized I had diverged from what is easily possible to express in C. Take the following example:

```
Uat Foo {
    Foo foo
    int i
}

Ufn Foo {
    fun getSelf
    <- void
    -> Foo
    {return own.foo}

    fun getI
    <- void
    -> int
    {return own.i}
}

start {
    dec Foo a new Foo { i: 0, foo: empty }
    dec Foo b new Foo { i: 1, foo: a }
    dec Foo c new Foo { i: 2, foo: b }

    dec int i = c...getSelf(void).foo...getI()
}
```

This will not even come close to transpiling correctly because of the way I create user defined types in C. The basic process is:

(1) Generate a struct that corresponds to the **Uat** section of the UDT declaration.
(2) Generate a method for each function in the **Ufn** section of the UDT declaration, passing in a reference to the above struct type.
(3) For the declaration of the UDT variable itself, malloc enough memory for the struct, then initialize all the variables on the struct (since I do this, I force the user to give a definition for every UDT attribute).

Thus, the transpiler must be able to properly pass in references to each struct as its methods are called, and correctly place all the arrows in the right places. While this is not necessarily the most complex task, since I originally thought that all the transpilation would be trivial, I did not setup enough helper data structures to keep track of what struct name goes where and which need arrows.

3.1.3. *Issue 3: Memory Leaks.* Since this was my first large C++ project, and I did not have much experience with the language, I had an exceptional amount of memory leaks and in general did not utilize the language correctly or to its fullest. Three of the most obvious offenders were:

(1) Raw pointers
(2) No abstract classes
(3) Rolling my own basic data structures (like a stack)

3.1.4. *Issue 4: Sailfish Had No Purpose.* While this is of course just a toy language, an experiemental language I was using to learn more about compilers, it did not yet have a story or a reason for existence. When asked *"why Sailfish?"* I could not answer with anything besides *"why not?"* To make the language more interesting, it needed a more intentional design, or feature set that could explain its reason for existence.

3.2. **Implementation #2.** Most of the limitations of **Sailfishc 0.1.0** were serious enough to require significant refactors to solve. Since I had about a month left in my independent study and I found myself throwing out large portions of **Sailfishc 0.1.0** as I tried to fix it, I decided to completely restart from scratch. While this implementation ended up with its own rather lengthy list of limitations as well, it turned out to be a significant improvement over the first implementation.

3.2.1. *Solution 1: A New Grammar.* When constructing the new grammar, I sought to keep the basic structure of the old grammar, but make it LL(1) and make it easier to reason about types. To solve both of these issues, I attached some semantic meaning to the production rules via Syntax Directed Translations (SDT's). Since I used a top down recursive descent parser, there are two grammar attributes I can add to the SDT's, *synthesized* and *inherited*. Synthesized attributes are those which are a summation of attributes of child members while inherited attributes are a result of attributes passed down from parent members. Since I am using recursive descent, this meant that synthesized attributes were return values and inherited attributes were method parameters, or arguments. Here is a similar section of the above grammar with SDT's:

$E_0 \rightarrow T_0 E_1$	$E_0.syn = E_1.syn$ $E_1.inh = T_0.syn$
$E_1 \rightarrow$ ['**'] $T_1\ E_0$	$E_1.syn =$ new Node(["**"], $E_1.inh$, $E_2.inh$) $E_2.inh = T_1.syn$
$E_1 \rightarrow E_2$	$E_1.syn = E_2.syn$ $E_2.inh = E_1.inh$
$E_2 \rightarrow$ ['*'|'/'|'%'] $T_2\ E_0$	$E_2.syn =$ new Node(["*"|"/"|"%"], $E_2.inh$, $E_3.inh$) $E_3.inh = T_2.syn$
$E_2 \rightarrow E_3$	$E_2.syn = E_3.syn$ $E_3.inh = E_2.inh$
$E_3 \rightarrow$ ['+'|'-'] $T_3\ E_0$	$E_3.syn =$ new Node(["+"|"-"], $E_3.inh$, $E_4.inh$) $E_4.inh = T_3.syn$
$E_3 \rightarrow E_4$	$E_3.syn = E_4.syn$ $E_4.inh = E_3.inh$
$E_4 \rightarrow$ ['<'|'>'|'<='|'>='] $T_4\ E_0$	$E_4.syn =$ new Node(["<"|">*"|"<="|">="], $E_5.inh$) $E_5.inh = T_4.syn$

FIGURE 1. Improved Grammar

As you can see here, I utilize synthesized attributes and inherited attributes to pass types between nodes that need to know about them. To understand the usefulness of this, consider the following parse tree:

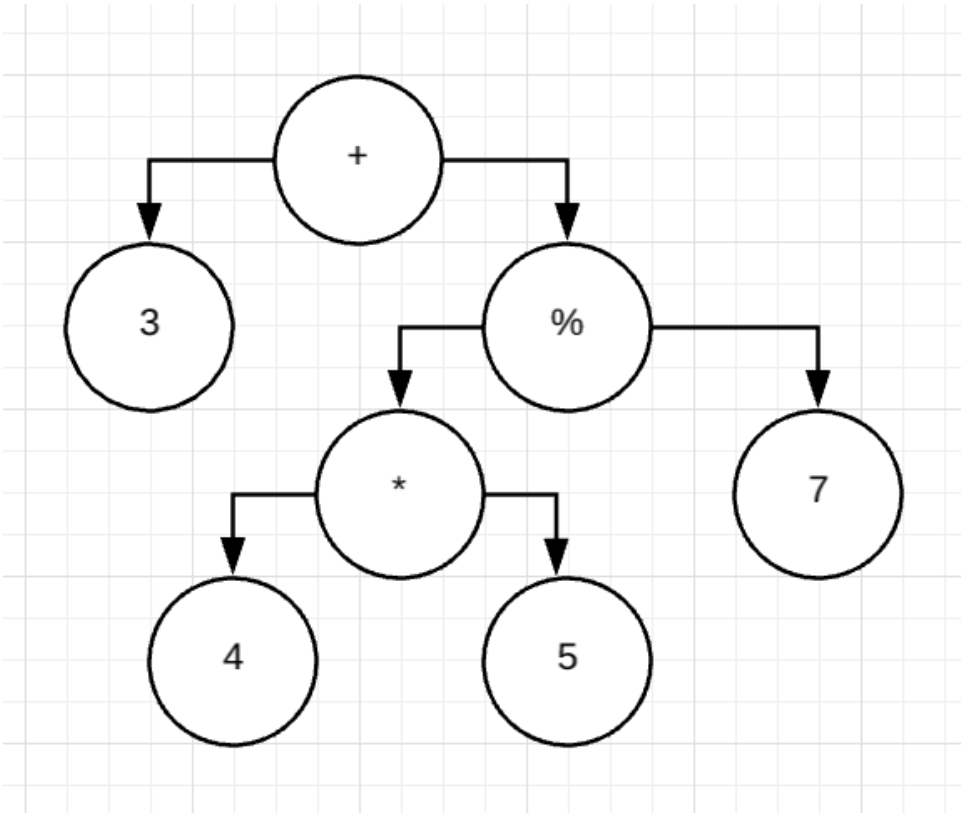

FIGURE 2. Parse Tree for 3 + (4 * 5) % 7

Here, according to the above grammar, each operation node would know the types of both of its expressions because these types are *evaluated* via synthesis up the various branches of the tree. Thus, when I needed to do semantic analysis it was easy since I could quickly check that each type made sense for the given node's operation. See below:

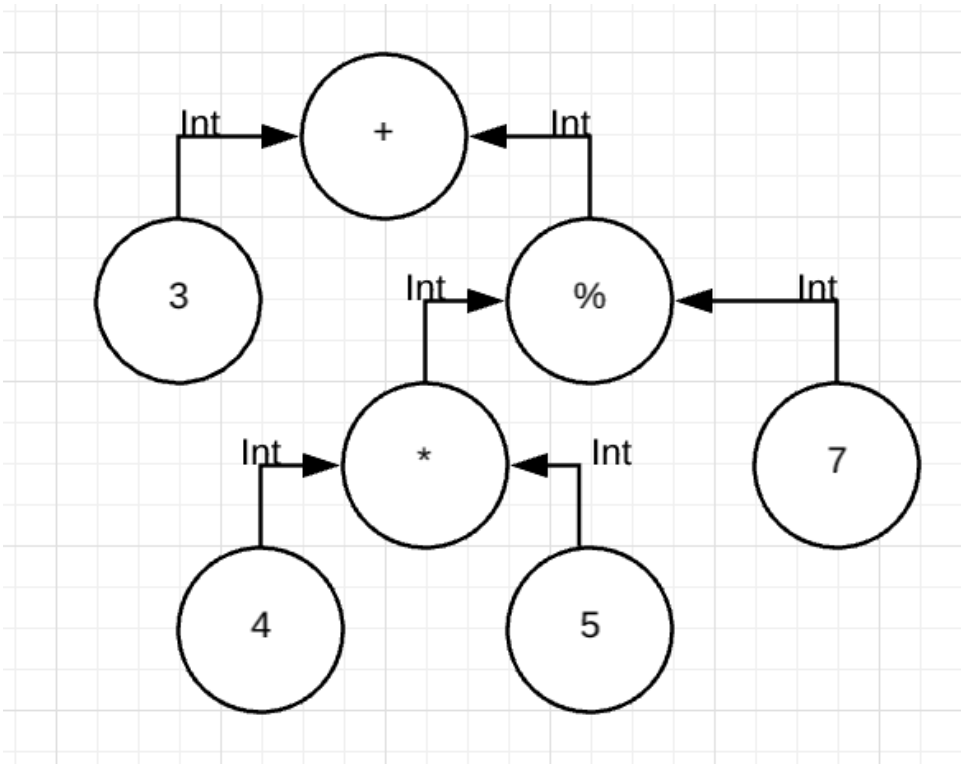

FIGURE 3. Parse Tree Showing Synthesized Attributes

By utilizing these attributes, and correctly designing a LL(1) grammar, I was easily able to reason about the correctness of the structure of a Sailfish program and expand the grammar to include more useful features like collections and parenthesized expressions.

3.2.2. *Solution 2: Better, Safer C++.* After reading through Bjarne Stroustrup's *A Tour of C++* I utilized the following to improve my C++:

(1) smart pointers instead of raw pointers
(2) type aliases instead of long type names
(3) auto type inference
(4) stdlib collections and algorithms wherever possible
(5) templates to reduce duplication

3.2.3. *Solution 3: Sailfish Has a Story.* A significant change (not backwards compatible) from **Sailfishc 0.1.0** to **Sailfishc 0.2.0** was the separation of UDT's from the rest of the program. Now UDT's are required to be in their own file, one per file, and take on the name of the file. The main logic from the program resides in a script file. The script file imports the UDT's it

needs and then combines these types with logic to do useful things. There is only one script file per program – the script file contains the initial execution body where the program begins execution. Therefore, I can now say that idiomatic Sailfish separates logic from type definitions. So why use Sailfish? Sailfish is useful for forcing programmers to adopt a mindset of isolated type definition and type declaration. By doing so, programmers can create types that are modular, readable, and reusable, making it natural for library creators to share types with their users (one day, imports will not just take file paths, but git links).

3.3. **Sailfishc 0.3.0: Single Traversal, Many Issues.** The most interesting implementation decision of implementation 2 was to remove the AST; removal of the AST was the result of an aggregation of design decisions and not necessarily an intentional feature.

As I began rewriting Sailfish from scratch, I decided to do away with the plethora of AST node types and instead utilize just two nodes: a non-terminal inner tree node and a terminal leaf node. These nodes would have an enum that described the type of the node and also a value field that captured relevant information, such as 3 for an integer type of leaf node. As I began to construct the AST with these nodes during parsing, I realized that some nodes need lists of children, while others only have a single child. To save myself from expanding to multiple different types of inner tree nodes, I utilized a recursive binary tree design. To demonstrate what I mean, consider the following code snippet:

```
import foo : "foo.fish"
import bar : "bar.fish"
import baz : "baz.fish"

start {
    # some useful code that does stuff
}
```

This would result in the following tree:

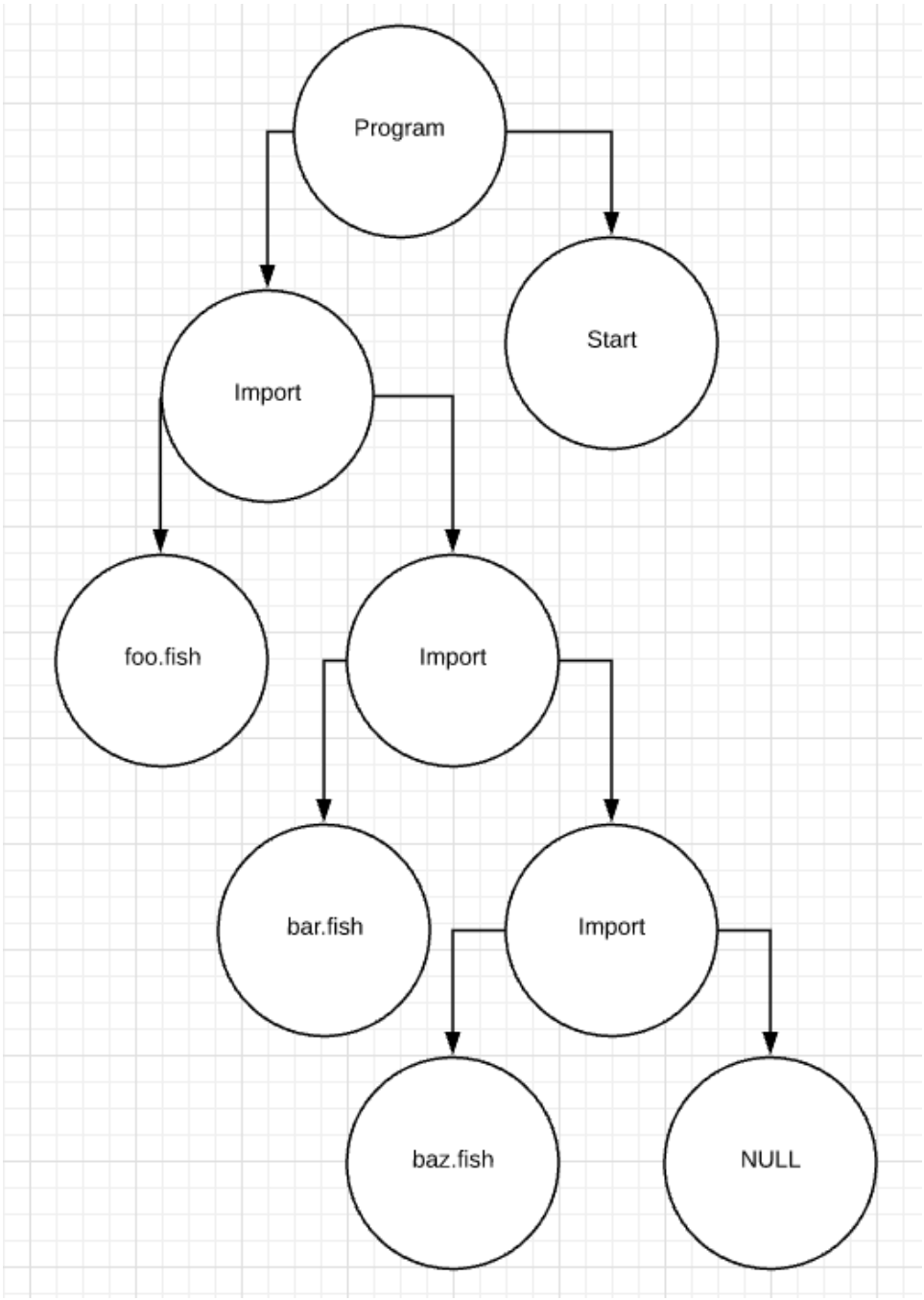

FIGURE 4. Recursive Binary AST

Initially, I thought this was a clever – I figured out how to design a tree that describes my program structure without the hundreds of lines necessary to

define numerous different node types. However, I quickly realized I had made a serious design mistake. One of the purposes of the design of the previous tree – node type for each production and aggregation of children into lists – was that it worked well with the Visitor Pattern; my new Binary AST did not. Thus, instead of concocting a work around, I decided to fold the parsing, semantic analysis, and code generation into a single traversal. Since this all happened at once, I realized I did not actually need an AST since every compiler phase was finished following the initial traversal of the tokens.

While this resulted in a code deduction of nearly 50% (5000+ lines), it ultimately resulted in an ugly mess of spaghetti code. Since all the phases of compilation were combined into one:

(1) I could not separately test each phase
(2) Errors propagated up through the phases
(3) Refactoring often broke more than it fixed since everything was so tightly coupled
(4) The code was hard to reason about since every method did numerous things
(5) I did not use C++ correctly – this was a blatant violation of good OOD principles

Thus, many of the gains from the improvements mentioned in the previous section were lost. While implementation 2 was certainly an improvement over implementation 1, it still needs a lot of work before it functions completely correctly.

3.4. **Sailfish Limitations: A Laundry List.** There exist a number of known issues with the Sailfishc compiler. Some of these issues are limitations of the languages, others are simply results of incorrect implementation. Some, or all of these should be solved in a third implementation of the language.

(1) Mutual recursion is unsupported.
(2) Impossible to import UDT's within UDT files.
(3) Name collisions exist in code generation.
(4) No concept of UDT constructors.
(5) No concept of UDT private/static members.
(6) Double dispatch does not transpile correctly.
(7) Declaring more than one attribute accessor within a function call does not transpile correctly.
(8) Functions are not first class.
(9) A majority of basic functionality not supported due to limited standard library (io, string methods, list methods, etc.)
(10) String and UDT comparison does not work due to transpilation error.

4. The Manual: Let's Write some Sailfish!

This comprehensive manual should be enough to get anyone up and running Sailfish. However, it is aimed mostly at bilingual programmers. Thus, if this is your first time coding, this manual may not be enough to get Sailfish up and running.

4.1. **Installation.** Sailfish can be installed in two ways, either by building from source, or via installation script. Both of these are available on the Sailfish Github organization page: https://github.com/sailfish-lang.
Note: *Sailfish requires a modern version of gcc and cmake to run, so make sure to download the latest versions of these as well. Also, Sailfish has never been tested on Windows, so if you are trying this, good luck!*

4.1.1. *Testing Installation.* To test your sailfish installation, go to an appropriate directory that has access to a compiled version of Sailfishc – if you installed via the script this will be anywhere, if you built from source, this will likely be in the root of the repo or in a build directory – and run:

```
$: ./sailfishc
```

If properly installed, you will see the following help text:

```
                Welcome to the Sailfish Compiler
The Sailfish is hungry and wants to compile some code.
To get started, consider one of the following commands:

        sailfishc [filename]

        sailfishc --help

        sailfishc --version
```

4.2. **Hello World.** Hello World in Sailfish presents the most basic syntax necessary to compile a program in the language. First, you will notice a start body (called *initial execution body* in Sailfish land), or the keyword start followed by curly braces. This is the equivalent of a main function in C, C++, Go, or Java. Inside this, you will see the *printStr* method, a built-in method which takes in a string and prints it to the console. Also notice the lack of semi-colons. Semi-colons are not just elided here, they are banned! The compiler will yell at you if you try to slide them in to any of your Sailfish source files. The last thing to note is that this code exists in a file named helloworld.fish. The name of the file is insignificant for now, however, it must end in *.fish*.

```
start {
    printStr("Hello World!")
}
```

To compile this, file, type the following command into your terminal:

```
$: ./sailfishc helloworld.fish
```

You should see:

```
Hello World!
```

Congratulations! You're now a *Sailer*– one who codes in Sailfish.

4.3. **The Basics.**

4.3.1. *Primitives.* There are four basic primitive types in Sailfish:

- Boolean, denoted *bool*
- Integer, denoted *int*
- Float, denoted *flt*
- String, denoted *str*

Currently, these utilize C semantics, where boolean utilizes 1 for true and 0 for false and strings convert to char*. There is more on this in the Implementation section.

4.3.2. *Lists.* Each primitive type has a corresponding list type. Lists in Sailfish are homogeneous, variable length collections. The current version of Sailfish only supports primitive typed lists. List types are denoted by a type, enclosed by brackets such as: *[int]*.

4.3.3. *Variable Declarations.* All variables are statically typed and initialized. Thus, the only place to define a new variable is in a code block, or within the formal arguments for a function. This leads to a generally more type safe language and for the most part, a language void of void (more will be discussed on this in coming sections). A variable declaration statement has five parts: the *dec* keyword, the type specifier, an identifier naming the type, an equals sign, and an initialization expression whose evaluated value is assigned to the new variable.

Here is an example of a string declaration:

```
dec str foo = "Hello World!"
```

Here is an example of a list of ints declaration:

```
dec [int] foos = [1,2,3,4,5]
```

Note: user defined type declarations are slightly different and will be discussed in the user defined type section.

4.3.4. *Basic Operations.* Sailfish supports the following, which evaluate exactly as you would expect:

- Arithmetic Operators: *+, +=, -, -=, *, *=, **, /, /=*
- Comparison Operators: *==, !=, >, >=, <, <=*
- Unary Operators: *–, ++, !*
- Logical Operators: *and, or*

4.3.5. *Comments.* Comments are python style (*use the #*) and are limited to only single line as of 0.3.0. There is really no reasoning behind this except that my VIM plugin wouldn't function properly with multi-line comments and that I have not needed multi-line comments in my Sailfish programs thus far. Multi-line comments is a feature I would expect in an upcoming release.

4.3.6. *Conditionals.* Sailfish supports conditionals via a *tree statement.* A tree statement is essentially syntactic sugar for a list of conditional branches. Each branch contains a conditional, enclosed within pipes, and a body within curly brackets that executes if the condition is true.

Basic structure:

```
Tree (
    ( | CONDITIONAL | { STATEMENTS } )
    ... more branches ...
)
```

Example (fizzbuzz):

```
Tree (
    ( | (i % 15) == 0 | { printStr("FizzBuzz" } )
    ( | (i %  5) == 0 | { printStr("Fizz" } )
    ( | (i %  3) == 0 | { printStr("Buzz" } )
    ( | true          | { printInt(i)}))
```

4.3.7. *Function Definitions.* Sailfish is a semi-functional language with hopes that one day it will be a fully functional language of the likes of Haskell, OCAML, or LISP. Right now the only functional thing it does is force recursion by way of not including loops. However, as it stands, functions are still an integral part of Sailfish – the only way to operate on data without utilizing a built in operator.

Semantically, functions in sailfish are very C-like (the exception being they are not first class), starting with the *fun* keyword, a type signature, and a function body. Syntactically they are very LISP-like, requiring a plethora of parenthesis. Functions may only be defined before an initial execution body in a program's script file, or in the *Ufn* section of a user defined type definition.

Basic structure:

```
(fun NAME (INPUTS)(RETURN_TYPE) {
    ... some code to do cool things ...
})
```

Example (simple addition):

```
(fun add(int x, int y)(int) {
    return x + y
})
```

So the careful reader probably has some questions. *How can I specify a void return? How can I specify no required inputs? Are there overloaded functions?* For void input and output, these are stated explicitly. When calling these functions, an explicit void is required. See below.

```
(fun useless(void)(void) {
    # do nothing
})

# call this useless construction
useless(void)
```

In regards to overloading, none exists. Yep, sorry! No overloading, no partial application, no currying, no immediately invoked function expression, no higher order functions, no variadic functions. These things are wonderfully fantastic, however they are features I am saving for a much stabler release of Sailfish (limitations are described in section *1.6*).

4.3.8. *Performing Iteration.* **Recursion.**
If recursion is scary, I highly recommend *The Little Lisper* by Daniel Friedman and Matthias Felleisen. This book teaches the basics of recursion via a subset of scheme in a highly interactive and enjoyable way. Don't want to spoil it, but by the end you're doing some really cool things.

Recursion Example (print all items in a list):

```
(fun printList([int] is, int index, int endIndex)(void) {
    Tree (
        ( | index == endIndex | ) { return })
        ( | true | { printInt(is[index]) }))

    # recurse
    printInt(is, index ++, endIndex)
})
```

4.4. **User Defined Types.** As in most general purpose languages, Sailfish has a construct for defining custom data types. These custom data types allow the user to aggregate related data and define useful operations on it. A main focus of idiomatic Sailfish is to isolate all user defined types such that they are testable, reusable, sharable, and modular, enforcing some good coding practices.

While user defined types are defined in *.fish* files just like script files, you can have multiple UDT files per project. These UDT's take the name of their file when being imported to the script and include two sections, an attribute section defined within a *Uat* body and a function section defined within a *Ufn* body.

Basic structure:

```
Uat {
    ... some awesome attributes ...
}

Ufn {
    ... some meaningful methods ...
}
```

The implementation of UDT's in this manner was largely motivated by the admirable simplicity of Rust's structs. Structs in Rust have an attribute definition section and then an optional method implementation section. By writing code like this, it enforces a physical and psychological separation of attributes from methods. Thus, when using and designing UDT's like Rust structs, the programmer thinks of the type more like a type in the real sense of a type: *a category of data with its associated operations.*

4.4.1. *This, Accessing A UDT's Own Properties:* In Sailfish, the *own* keyword is the same as the *this* keyword in many other languages – it is used to access the internals of a UDT within the body of an UDT's method.

4.4.2. *Attribute and Method Access:* Since everything in Sailfish is public by default (think structs in C++), it all, for better or worse, can be accessed externally. To do so, the programmer utilizes the *dot accessor notation*:

```
# access attribute bar on UDT foo
foo.bar
# access method bar on UDT foo
foo...bar(void)
```

You may notice that a side effect of structuring UDT's as so allows for attributes and methods to share names. Good catch! This is an intentional

design decision. While there is no clear benefit yet from this decision, it leaves the door open for future innovation.

4.4.3. *Declaring a new UDT:.* To declare a new UDT, you utilize a very similar syntax to a normal declaration, however, you must define an initial value for each attribute. This is accomplished in a very similar manner to defining structs in C, C++, or Go.

Basic Structure for some UDT Foo with an int attribute named *a* and a str attribute named *b*:

```
dec Foo f = new Foo { a: 10, b: "bar" }
```

4.5. **A Complete Example.**

4.5.1. *File Types.* As briefly mentioned in previous sections, there are two types of files in Sailfish, UDT files and a script file. Both reside in *.fish* files, however there may be infinitely many UDT files, but only one script file. The purpose of the script file is to combine many different types with some logic to do useful things. Thus, the program begins execution within the script file, pulling UDT information from associated UDT's whom were imported into the script via *import* statements.
Note: when importing UDT's, the filename must match the import name.

An example of import may be seen in the following example.

4.5.2. *Counter, An Example Program.* The following program contains a UDT, Counter, and a script that utilizes this Counter to implement a simple countdown.

Counter.fish (UDT file):

```
Uat {
    int count
}

Ufn {
    (fun decrement(void)(void) {
        --own.count
    })

    (fun increment(void)(void) {
        ++own.count
    })

    (fun count(void)(int) {
        return own.count
    })
}
```

sail.fish (script file):

```
import Counter : "Counter.fish"

(fun countdown(Counter c)(void) {
    # base case
    Tree ((  | c...count(void) == 0 | {
        printStr("Blast off!")
        return
    }))

    # display current count
    printInt(c...count(void))

    # decrement the count
    c...decrement()

    # recurse
    countdown(c)
})

script {
    dec Counter counter = new Counter { count: 10 }
    countdown(counter)
}
```

Execution output:

```
10
9
8
7
6
5
4
3
2
1
Blast off!
```

5. Language Specification

The following section is pertinent for anyone who wants to work on Sailfishc or who wants to implement a reference compiler of their own. Of course I should say checkout **Sailfishc 0.3.0**, the reference compiler. However, there are some obvious and less obvious gaps between the formal definition of Sailfish and what is accepted and generated by the Sailfishc compiler.

5.1. **Lexicon.** In Sailfish, neither semi-colons not new lines are used to separate values. Thus, semi-colons are illegal characters, and newlines do not matter. However, for parsing and lexing spacing does matter. For example, the following two lines are not equivalent:

+ ++ – results in add followed by unary add
+++ – results in unary add followed by add

The following are reserved keywords:

- Ufn
- Uat
- start
- Tree
- empty
- void
- int
- bool
- flt
- str
- new
- return
- import
- dec
- fun
- own
- and
- or
- [int]
- [str]
- [bool]
- [flt]

The following are binary operators:
+, -, *, \, +=, -=, \=, *=, %, >, >=, <, <=, !=, ==, =

The following are unary operators:
++, –, !

The following are punctuation:

`_, ',', {, }, (, ), ., ..., |, :`

Sailfish comments are only single-line and begin with a #

5.2. **Grammar.** The following is the most up to date grammar for the Sailfish programming language as of the time of writing this document. It was last updated April 2019 and may not reflect the current state of the language.

PRODUCTION	SEMANTIC RULES
Program → *Source*	Program.*syn* = Source.*syn*
Source → *Import Source* | *SourcePart*	Source.*syn* = new Node("SOURCE", Import.*syn*, SourcePart.*syn*)
Import → ['Import' *ImportInfo*]+	Import.*syn* = *List*(ImportInfo.*syn*)
ImportInfo → *UDName* ':' *Location*	ImportInfo.*syn* = new Node("IMPORT", UDName.*syn*, Location.*syn*)
UDName → **Identifier**	UDName.*syn* = Identifier.*syn*
Location → **String**	Location.*syn* = Identifier.*syn*
SourcePart → *UDT*	SourcePart.*syn* = UDT.*syn*
SourcePart → *Script*	SourcePart.*syn* = Script.*syn*
---	---
UDT → *UserDefinedType*	UDT.*syn* = UserDefinedType.*syn*
UserDefinedType → *Attributes Methods*	UserDefinedType.*syn* = new Node("UDT", Attributes.*syn*, Methods.*syn*)
Attributes → 'Uat' '{' *Variable** '}'	Attributes.*syn* = List(Variable.*syn*)
Methods → 'Ufn' '{' *FunctionDefinition** '}'	Methods.*syn* = List(FunctionDefinition.*syn*)
---	---
Script → *Script_*	Script.*syn* = new Node("SCRIPT", Script_.*syn_fund*, Script_.*syn_start*)
Script_ → *FunctionDefinition** *Script_*	Script_.*syn_fund* = List(FunctionDefinition.*syn*)
Script_ → *Start*	Script.*syn_start* = Start.*syn*
FunctionDefinition → '(' 'fun' **Identifier** *FunctionInfo* ')'	FunctionDefinition.*syn* = new Node("FUNCTION_DEFINITION", Identifier.*syn*, FunctionInfo.*syn*)
FunctionInfo → *FunctionInOut Block*	FunctionInfo.*syn* = new Node(FUNCTION_INFO", FunctionInOut.*syn*, Block.*syn*)
FunctionInOut → *FunctionInputs* FunctionOutputs*	FunctionInOut.*syn* = new Node("FUNCTION_IN_OUT", FunctionInputs.*syn*, FunctionOut.*syn*)
FunctionInputs → '(' *Variable* [',' *Variable*]* ')'	FunctionInputs.*syn* = List(Variable.*syn*)
FunctionOutput → '(' *Type* ')'	FunctionOut.*syn* = Type.*syn*
Start → 'start' *Block*	Start.*syn* = Block.*syn*

| --- | --- |
Block → '{' *Statement** '}'	Block.*syn* = List(Statement.*syn*)
Statement → *Tree*	Statement.*syn* = Tree.*syn*
Statement → *Return*	Statement.*syn* = Return.*syn*
Statement → *Declaration*	Statement.*syn* = Declaration.*syn*
Statement → E_0	Statement.*syn* = E_0.*syn*
Tree → '(' *Branch** ')'	Tree.*syn* = List(Branch.*syn*)
Branch → '(' *Grouping Block* ')'	Branch.*syn* = new Node("BRANCH", Grouping.*syn*, Block.*syn*)
Grouping → '|' E_0 '|'	Grouping.*syn* = E_0.*syn*
Return → 'return' *T*	Return.*syn* = T.*syn*
Declaration → 'dec' Variable = T	Declaration.*syn* = new Node("DEC", Variable.*syn*, T.*syn*)

$E_0 \rightarrow T_0 E_1$	$E_0.syn = E_1.syn$ $E_1.inh = T_0.syn$
$E_1 \rightarrow$ ['**'] T_1 E_0	$E_1.syn$ = new Node(["**"], $E_1.inh$, $E_2.inh$) $E_2.inh = T_1.syn$
$E_1 \rightarrow E_2$	$E_1.syn = E_2.syn$ $E_2.inh = E_1.inh$
$E_2 \rightarrow$ ['*'|'/'|'%'] T_2 E_0	$E_2.syn$ = new Node(["*"|"/"|"%"], $E_2.inh$, $E_3.inh$) $E_3.inh = T_2.syn$
$E_2 \rightarrow E_3$	$E_2.syn = E_3.syn$ $E_3.inh = E_2.inh$
$E_3 \rightarrow$ ['+'|'-'] T_3 E_0	$E_3.syn$ = new Node(["+"|"-"], $E_3.inh$, $E_4.inh$) $E_4.inh = T_3.syn$
$E_3 \rightarrow E_4$	$E_3.syn = E_4.syn$ $E_4.inh = E_3.inh$
$E_4 \rightarrow$ ['<'|'>'|'<='|'>='] T_4 E_0	$E_4.syn$ = new Node(["<"|">*"|"<="|">="], $E_5.inh$) $E_5.inh = T_4.syn$
$E_4 \rightarrow E_5$	$E_4.syn = E_5.syn$ $E_5.inh = E_4.inh$
$E_5 \rightarrow$ ['=='|'!='] T_5 E_0	$E_5.syn$ = new Node(["=="|"!="], $E_5.inh$, $E_6.inh$) $E_6.inh = T_5.syn$
$E_5 \rightarrow E_6$	$E_5.syn = E_6.syn$ $E_6.inh = E_5.inh$
$E_6 \rightarrow$ ['and'|'or'] T_6 E_0	$E_6.syn$ = new Node(["and"|"or"], $E_6.inh$, $E_7.inh$) $E_7.inh = T_6.syn$
$E_6 \rightarrow E_7$	$E_6.syn = E_7.syn$ $E_7.inh = E_6.inh$
$E_7 \rightarrow$ ['='] T_7 E_0	$E_7.syn$ = new Node(["="], $E_8.inh$, $E_7.inh$) $E_8.inh = T_7.syn$
$E_7 \rightarrow E_8$	$E_7.syn = E_8.syn$ $E_8.inh = E_7.inh$
$E_8 \rightarrow$ ['!', '++','--,] T_8 E_0	$E_8.syn$ = new Node(["!","++","--"], $E_9.inh$) $E_9.inh = T_8.syn$
$E_8 \rightarrow E_9$	$E_8.syn = E_9.syn$ $E_9.inh = E_8.inh$
$E_9 \rightarrow$ [+=','-=','*=','/='] T_8 E_0	$E_8.syn$ = new Node(["<",">","<=",">="], $E_{10}.inh$) $E_{10}.inh = T_9.syn$
$E_9 \rightarrow E_{10}$	$E_{09}.syn = E_{10}.syn$ $E_9.inh = E_9.inh$
$E_{10} \rightarrow$ *MemberAccess* E_0	$E_{10}.syn$ = MemberAccess.*syn* MemberAccess.*inh* = $E_{10}.inh$
$E_{10} \rightarrow E_{11}$	$E_{10}.syn = E_{11}.syn$ $E_{11}.inh = E_{10}.inh$
$E_{11} \rightarrow$ *new New*	$E_{11}.syn$ = new Node("NEW", $E_{11}.inh$, New.*syn*)
$E_{11} \rightarrow E_{12}$	$E_{11}.syn = E_{12}.syn$ $E_{12}.inh = E_{11}.inh$
$E_{12} \rightarrow$ FunctionCall	$E_{12}.syn$ = FunctionCall.*syn*
$E_{12} \rightarrow T_{12}$	$E_{12}.syn = T_{12}.syn$
---	---

MemberAccess → *AttributeAccess*	MemberAccess.*syn* = AttributeAccess.*syn* AttributeAccess.*inh* = MemberAccess.*inh*
MemberAccess → *MethodAccess*	MemberAccess.*syn* = MethodAccess.*syn* MethodAccess.*inh* = MemberAccess.*inh*
AttributeAccess → '.' **Identifier**	AttributeAccess.*syn* = new Node("ATTRIBUTE_ACCESS", AttributeAccess.*inh*, Identifier.*syn*)
MethodAccess → '...' *FunctionCall*	MethodAccess.*syn* = new Node("MEMBER_ACCESS", MethodAccess.*inh*, FunctionCall.*syn*)
FunctionCall → '(' [**Identifier** [',' **Identifier**]*] ')'	FunctionCall.*syn* = List(Identifier.*syn*)
---	---
New → *UDTDec*	New.*syn* = UDTDec.*syn*
UDTDec → '{' [**UDTDecItem** [',' **UDTDecItem**]*] '}'	UDTDec.*syn* = List(UDTDecItem.*syn*)
UDTDecItem → **Identifier** ':' *Primary*	UDTDecItem.*syn* = new Node("UDT_DEC_ITEM", Identifier.*syn*, Primary.*syn*)
---	---
T → *Primary*	T.*syn* = Primary.*syn*
T → '(' E_0 ')'	T.*syn* = E_0.*syn*
Primary → **Boolean**	Primary.*syn* = Boolean.*syn*
Primary → **Number**	Primary.*syn* = Number.*syn*
Primary → **String**	Primary.*syn* = String.*syn*
Primary → **Identifier**	Primary.*syn* = Identifier.*syn*
Primary → **List**	Primary.*syn* = List.*syn*
Boolean → true | false	Boolean.*syn* = **boolean.lexval**
Number → Integer	Number.*syn* = Integer.*syn*
Number → Float	Number.*syn* = Float.*syn*
Integer → [0-9]+	Integer.*syn* = **integer.lexval**
Float → [0-9]+'.'[0-9]+	Float.*syn* = **float.lexval**
String → '"' [^\"]* '"'	String.*syn* = **string.lexval**
Identifier → [a-ZA-Z][a-ZA-Z0-9_]*	Identifier.*syn* = **identifier.lexval**
List →'[' . ']'	List.*syn* = **list.lexval**
ListType → "[int]" | "[flt]" | "[bool]" | "[str]"	lListType.*syn* = **listtype.lexval**
Type → Identifier	Type.*syn* = Identifier.*syn*
Type → ListType	Type.*syn* = ListType.*syn*

5.3. **The Standard Library.** As with most programming languages, Sailfish's reference compiler *Sailfishc* ships with a standard library of common, useful methods. These rather primitive methods are impossible to implement in Sailfish itself and thus are implemented in Sailfish's target language, C. The table on the following page is a comprehensive overview of these methods.

Method Name Output	Input Description
appendListInt [int] combined_list	[int] list_a, [int] list_b, int size_a, int size_b Append list b to the end of list a, returning the combined list c.
appendListStr [str] combined_list	[str] list_a, [str] list_b, int size_a, int size_b Append list b to the end of list a, returning the combined list c.
appendListBool [bool] combined_list	[bool] list_a, [bool] list_b, int size_a, int size_b Append list b to the end of list a, returning the combined list c.
appendListFlt [flt] combined_list	[flt] list_a, [flt] list_b, int size_a, int size_b Append list b to the end of list a, returning the combined list c.
deleteAtIndexInt [int] modified_list	[int] list, int size, int index Delete the value at a given index and return a new modified list.
deleteAtIndexStr [str] modified_list	[str] list, int size, int index Delete the value at a given index and return a new modified list.
deleteAtIndexBool [bool] modified_list	[bool] list, int size, int index Delete the value at a given index and return a new modified list.
deleteAtIndexFlt [flt] modified_list	[flt] list, int size, int index Delete the value at a given index and return a new modified list.
getAtIndexInt int value	[int] list, int index Return the value of a list at a given index.
getAtIndexStr str value	[str] list, int index Return the value of a list at a given index.
getAtIndexBool bool value	[bool] list, int index Return the value of a list at a given index.
getAtIndexFlt flt value	[flt] list, int index Return the value of a list at a given index.
setAtIndexInt [int] modified_list	[int] list, int index, int value Set the value of a list at a given index, returning the new modified list.
setAtIndexStr [str] modified_list	[str] list, int index, str value Set the value of a list at a given index, returning the new modified list.
setAtIndexBool [bool] modified_list	[bool] list, int index, bool value Set the value of a list at a given index, returning the new modified list.
setAtIndexFlt [flt] modified_list	[flt] list, int index, flt value Set the value of a list at a given index, returning the new modified list.
printInt void	int value Print the given int.
printStr void	str value Print the given str.
printBool void	bool value Print the given bool.
printFlt void	flt value Print the given flt.

References

[1] *Compilers: Principles, Techniques, and Tools*; Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
[2] *Bantam Java Compiler Project Lab Manual*; Marc L. Corliss, David Furcy, E Christoper Lewis and modified by Dale Skrien
[3] *A Tour of C++* Bjarne Stroustrup
[4] *The Solidity Programming Language*
[5] *The Python Programming Language*