Contents

The Satie programming language

Introduction

Satie is envisioned as programming language especially suited for manipulation of large text masses. Ideal for implementing programming and text editors of tomorrow. Its capabilities extend beyond just constructing editors; Satie also excels as a scripting language for creating editor plugins and customizations. However, the true essence of Satie lies in its versatility. It is a purpose-built language and Virtual Machine (VM) that boasts a high degree of generality, making it adaptable to a wide array of applications.

Satie owes a great deal to the individuals behind the Erlang and D programming languages, as well as all those who stand with them and support their efforts.

All rise. Here is a tribute and a premature Satie example:

```
$ cat tribute.sa
import std.stdio : writeln
import std.lists

export fn main(args) {
    ?numberOfTributes = args[1],
    ?jobs = startTributes(numberOfTributes),
    lists.foreach(fn (job) {
        job <| "Standing on the shoulders of giants"
    }, jobs)
}

fn startTributes(numberOfTributes, n = 0, jobs = []) {
    if n < numberOfTributes {
        ?job = spawn fn () {
            receive {
```

```
                case ?message {
                   writeln("$n: $message")
                }
            }
        },
        startTributes(numberOfTributes, n + 1, job ~ jobs)
    } else {
        jobs
    }
}
$ sac tribute.sa
$ sa build/tribute 100000
0: Standing on the shoulders of giants
1: Standing on the shoulders of giants
2: Standing on the shoulders of giants
3: Standing on the shoulders of giants
...
99999: Standing on the shoulders of giants
```

Source: tribute.sa

That said.

Design choices

The following design choices have been made (in some sort of order):

Concurrent Oriented Programming (COP)

- Satie is built on a custom multi-core virtual machine (VM) that offers robust support for time-sliced green threads, here after referred to as *jobs*. These jobs adhere to a *share-nothing* semantics and depend exclusively on message passing for communication. This architecture simplifies the process of designing and implementing highly concurrent, massively scalable soft real-time systems. It emphasizes fault tolerance and high availability, making it easier to reason about such systems.
- Jobs have the capability to establish *monitors* and *links* (bidirectional monitors) with one another. This feature enables the creation of supervisor jobs, which can start and restart children jobs should they unexpectedly fail.

Functional and immutable

- Satie is a pure functional language featuring native persistent datatypes at its core. All data within Satie is immutable, and the persistent datatypes are custom-built to handle large amounts of data efficiently. However, there are limits to Satie's purity; no monads.

A fusion of application and scripting language

- Satie aims to strike a balance between a fully-fledged application language and a scripting language, as highlighted by the reasons mentioned above (and below). This may be evident considering the choice of semantics, syntax, type system, support for object-orientation (from now on called *encapsulation*), and other aspects.

Dynamically typed and type inference

- Satie is dynamically typed and offers a concise set of basic and composite types. Its compiler employs Hindley-Milner type inference to deduce the types of variables, expressions, and functions.
- Semantic and syntactic considerations have been made to facilitate the future addition of a gradual type system.
- Satie's dynamic type system utilizes a Garbage Collection (GC) mechanism that is carefully designed to perform garbage collection on a job basis. This approach is intended to prevent the GC process from becoming a "stop-the-world" activity.

A streamlined syntax using curly braces

- Satie, as a compact language, features a clean, regular, and minimalist syntax that aligns with the tradition of curly braces languages. This includes the use of well-known reserved words, syntax conventions, and scoping rules.
- The principle of least surprise has been a guiding factor in Satie's development. However, in Satie, everything is an expression; there are no statements (and consequently, no semicolons). Due to this design choice, certain syntactic constructs might appear unorthodox, yet they embody a regular and consistent syntax.
- The syntax of the D and Erlang programming languages has heavily influenced Satie's design where applicable. Another key principle has been to create a syntax that is both familiar and visually appealing, though ultimately, you'll be the judge of that. Satie uses only a few keywords and has an even smaller set of operators. The complete syntax is formally defined as a Parsing Expression Grammar (PEG) in Appendix B: PEG grammar.

Pattern matching in its core

- Satie incorporates pattern matching as a fundamental feature, with the `=` operator primarily serving pattern matching rather than traditional assignment (emphasizing the absence of mutable updates). Essentially, ev-

erything can be matched and deconstructed using pattern matching in conjunction with the `=` operator.

- Pattern matching is also employed by the `switch` expression (described below). Furthermore, the `receive` expression (also described below) utilizes pattern matching to perform selective reception of messages in a job's mailbox.

A custom-built Virtual Machine (VM)

- Satie operates on a custom-built virtual machine (VM), which includes a multi-core and time-slicing job scheduler. This scheduler runs multiple instances, one for each job, of a custom-built register machine.
- The Virtual Machine (VM) and its scheduler are designed to have a small memory footprint. Each job, when initially started, allocates 1KB for its heap and stack (excluding the program code).
- The VM is standalone and has minimal dependencies, making it easily portable to systems with restricted capabilities.

Functional encapsulation

- Effort has been invested in introducing a purely functional and encapsulating `struct` definition. This allows for the grouping of member variables and functions using familiar C++/Java member modifiers such as `public`, `private`, `const`, `this` references and more.

Many features are are by design excluded from Satie, including:

- Pointers
- Type specifications
- Exceptions
- Pre-processor and macros
- Variadic function parameters
- Mutexes (not needed)
- Mutability
- Currying
- Inheritance
- Monads

There are most likely other features you will miss as well.

Overall structure

A Satie file is identified by a `.sa` suffix and constitutes a Satie *module* named after the basename of its filename. Why complicate things? A module begins with a series of import definitions, followed by a mix of `enum`, `interface`, `struct`, and `fn` (function) definitions, which can be arranged in any order.

`struct`, `enum`, and `interface` definitions are **only** permitted at the top level of a module. However, function definitions can be nested within other function definitions freely.

In an application, exactly one of the Satie modules must define a single exported main function.

Here follows yet another premature Satie example but read it through and an explanation will follow:

```
import std.stdio : writeln

enum Color {
    red
    green
    blue
}

interface Iterator {
    public fn next()
    public fn hasNext()
}

struct ColorIterator : Iterator {
    private colors
    private graffiti

    this(colors) {
        this(colors: colors, graffiti: "Kilroy was here")
    }

    public fn next() {
        if (!hasNext()) {
            false
        } else {
            #(this(colors: colors.rest()), colors.first())
        }
    }

    public fn hasNext() {
        !colors.isEmpty()
    }
}

export fn main() {
    ?colors = [Color.red, Color.red, Color.blue, Color.green],
    ?iterator = struct ColorIterator(colors),
```

```
    fn iterate(iterator) {
        if (iterator.hasNext()) {
            #(?iterator, ?color) = iterator.next(),
            writeln("Color: $color"),
            iterate(iterator)
        }
    },
    iterate(iterator)
}
```

Source: color.sa

A plain and fairly boring color iterator, and a main function that iterates over a list of colors. Still, it manages to show the overall structure of a Satie application.

The short story without any preamble:

A Satie module can define as many top-level functions as needed, whether exported or not. However, it's advisable to define functions as member functions within structs to prevent cluttering of the global function namespace.

Only functions marked with the `export` keyword can be imported by other modules.

`struct`, `enum`, and `interface` definitions can be imported by other modules without any restrictions though.

`#( ... )` defines a tuple, and placing a question mark before a variable informs the compiler that the variable should be treated as unbound, regardless of its previous state. Omitting the question mark prompts the compiler to ensure that the variable is already bound. At runtime, a check is conducted to verify that the bound variable matches the rvalue.

The `this` constructor calling itself might initially seem confusing. However, this is the final step a constructor must perform to actually initialize its member variables. The `:` notation is a way to call a function with **named** parameters, and a constructor can call itself using its member variables as named parameters. This self-referential mechanism is an integral part of Satie's support for functional encapsulation.

That covers the basics, but naturally, many aspects are left unexplained at this stage. The rest is in the gory details.

Building and executing

The compiler is named `sac`, and the bytecode it generates can be executed using the `sa` runner. This process was outlined in the introduction chapter above and is reiterated here for clarity:

```
$ sac tribute.sa
```

```
$ find .
.
./tribute.sa
./build
./build/tribute.sab
$ sa build/tribute 100000
0: Standing on the shoulders of giants
1: Standing on the shoulders of giants
2: Standing on the shoulders of giants
3: Standing on the shoulders of giants
...
99999: Standing on the shoulders of giants
```

That is it.

An application can also be composed of a multitude of modules within a directory hierarchy, but the build process remains the same. For instance, consider an application named `zarah` consisting of these modules:

```
$ cd zarah
$ find .
.
./src
./src/main.sa
./src/utils
./src/utils/httpclient.sa
./src/database
./src/database/backup.sa
./src/database/utils
./src/database/tablestore.sa
```

Here is how the application can be compiled and executed:

```
$ sac src/main.sa
$ find .
.

./build
./build/main.sab
./build/utils
./build/utils/httpclient.sab
./build/database
./build/database/utils
./build/database/backup.sab
./build/database/tablestore.sab
./src
./src/main.sa
./src/utils
```

```
./src/utils/httpclient.sa
./src/database
./src/database/backup.sa
./src/database/utils
./src/database/tablestore.sa
$ sa build/main
```

Note: The `sac` compiler can be configured to use alternative directories for `src/` and `build/`.

By default, the `sac` compiler follows module dependencies starting from `main.sa` and automatically compiles those modules as well. However, it can be configured not to follow these dependencies, to ignore missing modules, or to skip modules that cannot be compiled. For more detailed information about the `sac` compiler and the `sa` runner, refer to their respective manual pages.

The interactive shell

The `sa` runner is also capable of launching an interactive shell:

```
$ sa --shell
Satie <0.3.1> (press ctrl-c to abort)
0> _
```

In the shell expressions can be evaulated and the status of an already executing application can be inspected, i.e a shell can be made to connect to an already executing `sa` runner instance. Read more about this in the `sa` runner's manual page.

The gory details

Comments

Everything after `//` and to end of line and within `/* ... */` are considered comments.

Types

Basic types

`bool` : A boolean value

`int` : On a 64-bit machine, a signed integer value is handled natively if it fits within 61 bits. If it exceeds this limit, it is transparently represented as an arbitrary-precision bignum. Similarly, on a 32-bit machine, an integer must fit within 29 bits, and so on, adapting to the architecture's capabilities.

`float` : A floating-point value must conform to the machine's bit capacity. On a 64-bit machine, a float must fit within 61 bits, while on a 32-bit machine, it

must fit within 29 bits, and so on, aligning with the specific architecture.

`char` : A 32-bits Unicode code point value

`function` : A function reference

`job` : A job reference

`enum`: An enumeration value

Composite types

`string` : An immutable sequence of UTF-8 encoded characters

`tuple` : A fixed sized sequence of values of any type

`list`: A list of values of any type

`map`: A mapping between a key of any type and a value of any type

`struct` : A unit of encapsulation for member variables and member functions

`buf` : A buffer to efficiently manipulate large amount of characters

Type management

All values can be type checked in run-time using the functions `isBool`, `isInt`, `isFloat`, `isChar`, `isFunction`, `isJob`, `isEnum`, `isString`, `isList`, `isMap`, `isStruct`, `isBuf` and `typeof`.

Look at these examples:

```
?a = 3.14,
a.isInt(),      // false
a.isFloat(),    // true
a.typeof()      // "float"
```

Operators that operate on `int` and `float` values require the operands to be of the same type. No implicit numeric conversion is performed. However, a cast operator is available to facilitate casting between int and float values.

Grok this:

```
?a = 3,
?b = 93326215443944152681,
?c = 3.0,
?d = b / cast(int)c + a,     // 3110873848131471360 3
d + c                        // Error!
```

The `inspect` function offers more run-time type information:

```
enum Foo {          //  Defined in bar.sa
  a = 3.14
  b
```

```
}

?c = Foo.a,
?d = [1, "foo"],
c.inspect(),        // [ "type" : "enum",
                    //   "name": "a",
                    //   "value": 3.14,
                    //   "owner" : #("bar", "Foo") ]
d.inspect()         // [ "type" : "list", "length": 2 ]
```

Note: In the above example the value attached to the enumeration constant would typically be accessed using `c.value`, and the list length would be referred to as `d.length`.

All values can be converted to string representation using the `toString` function.

Try this:

```
?a = 3.14,
?b = [Foo.a : 42, "bar": #(fn () { x + x}, [1, 2,3])],
a.toString(),       // "3.14"
b.toString()        // "[Foo.a : 42, "bar": fn/0]"
```

Identifiers

An identifier is a case-sensitive sequence of characters that begins with an ASCII letter or an underscore, followed by any number of ASCII letters, underscores, or digits. This is denoted by the regular expression `^[[:alpha:]_][[:alnum:]_]*$`. Identifiers are used for various elements such as keywords, variables, function names, struct names, and enum names.

Note: By design, only strings may contain Unicode characters. However, this restriction could be reconsidered and potentially lifted if compelling reasons arise.

Keywords

24 special identifiers cannot be used in user code. These reserved identifiers are exclusive to the language's internal syntax and functionality.

```
import
true
false
enum
in
fn
export
if
else
```

```
switch
default
struct
interface
public
private
readonly
const
this
spawn
monitor
link
receive
timeout
self
```

Literals

Boolean literal

`true` or `false`

Integral literal

An integral literal can be represented as decimal, octal and hexadecimal values:

```
4,              // Decimal format
017,            // Octal format
0xffff,         // Hexadecimal format
0b101010100     // Binary format
```

Floating-point literal

A floating point literal represents a real number that includes a fractional part. Its notation is similar to that used in other languages like C or Java:

```
1.0,
.666e2
```

Character literal

A character literal is a Unicode code point value enclosed within single quotation marks. Internally, it consists of four bytes:

```
'A',
' ',
'\u03c9'    //
```

Function literal

A function literal follows the same syntax as regular function definitions, but without a function name:

```
?sum = fn (x, y) { x + y },
sum(1, 2)                    // 3
```

Job literal

Job literals are opaque.

Enumeration literal

An enumeration is a named constant, and it is always defined within a enumeration definition. An enumeration literal is represented as a dot-separated sequence, consisting of the enumeration name followed by the constant name:

```
enum Color {
    red
    green
    blue
}

Color.red                    // An enumeration literal
```

String literal

A string literal is represented as an immutable UTF-8 encoded sequence of Unicode characters, enclosed within double quotation marks. Escape sequences are interpreted in these double quoted strings. Additionally, there is raw strings, which are also enclosed within double quotation marks but prefixed with the letter `r`. In raw strings, escape sequences have no meaning, and all characters are parsed verbatim.

Like this:

```
?a = "foo ",
a[3],                // ' '
?b = r"foo\nbar"     // b.length == 8
```

Tuple literal

A tuple literal is represented as comma separated, fixed size sequence of values of any type. This sequence is enclosed between a leading `#(` and a trailing `)`:

```
#("foo", 3.14, #("bar", fn (x) { x + 1}))
```

List literal

A list literal is represented as comma-separated sequence of values, which can be if any type enclosed within square brackets:

```
?a = [3.14, "foo", 1816381],
?b = a[1 = 42, 2 = "bar"]      // b == [3.14, 42, "bar"]
```

Note: Only existing list entries can be updated this way.

Map literal

A map literal is represented as comma-separated sequence of key-value pairs, where the key and value can be of any type and are separated by a `:` character. This sequence is enclosed within square brackets:

```
?a = ["foo" : 12, 3.14 : 981237198192378 ],
a[3.14: 4711, 2 : 4]                          // ["foo" : 12, 3.14: 4711, 2 : 4],
```

Struct literal

A struct literal is represented as a semicolon-separated sequence of member-value pairs. Each pair consists of a member (which is an identifier) and a value of any type, separated by a `;` character. This sequence is enclosed within square brackets.

```
struct Foo P {
    public foo = 4711
    public bar = "foo"
    public zonk = #(42, 3.14)
}

?a = struct Foo(),
[foo ; ?b,
 bar ; ?c] = a,
b,                            // 4711
c                             // "foo"
```

Expressions

In Satie every element is treated as an expression, with the exception of top-level definitions. These include `import`, `struct`, `interface`, and `enum`.

Among all expressions, the bind expression, marked by the `=` operator, stands out for its ability to bind variables to values. Instead of variable assignment, unbound variables are bound to values. An unbound variable is identified by a leading `?` character, and a naked variable (without a `?`) must be bound before use.

Like this:

```
?a = 42,
a = 42,
a = "foo",     // Error!
b = 4711       // Error!
```

In the example above, `a = "foo"` leads to an error, where `a` is expected to be `42`, not `"foo"`. Such mismatches are typically used as assertions or may indicate a software bug. A runtime error halts the execution of the job where the mismatch occurred. For more details on concurrency and jobs, refer to the Concurrency section.

Additionally, in the example above, `b = 4711` would result in a compiler error because `b` has not been previously bound.

The pattern matching capability of bind expressions can be utilized to deconstruct composite values into their primitive components.

Grok this:

```
fn foo(x) {
  #(4711, x + x, "bar")
},
?a = 1,
#(a, ?b, 1) = #(1, 2, 1),     // b == 2
#(?a, b, ?c) = foo(1),        // a == 4711 && c == "bar"
#(a, a, a) = c                // Error!
```

Pretty nifty.

Note: In the examples above, we used expressions that have not been explained yet. For now, it's sufficient to understand that `#(1, 2, 1)` represents a fixed-size tuple.

The `switch` and `receive` expressions, which are described below, also employ pattern matching. This feature enables them to deconstruct and select specific elements. Further details about these expressions can be found in the subsequent sections.

Furthermore, the `=` operator is restricted to be used as a standalone expression and is not permitted within more complex expressions. This restriction is in place to maintain clarity and simplicity in coding practices. Your understanding and cooperation are appreciated.

This produces an error:

```
main() {
    ?a = 42,
    a + (?c = 42 + c) + a    // Error!
}
```

Block expression -- `{ a, b, c, ... }`

The block expression consists of a sequence of expressions separated by commas and enclosed in curly braces. These expressions are evaluated sequentially and establish a lexical scope. A variable bound within this scope is visible to all subsequent expressions in the same scope. However, it remains invisible outside of this scope and overshadows any identifier with the same name introduced outside the scope. The value of the last expression in the sequence is what the block ultimately returns.

Take a look at this:

```
main() {                    // A function block starts here
    ?a = 42,
    ?d = {                  // Local block starts here
        ?b = a + 1,
        ?a = b
     }
    // b is not defined
    // a = 42
    // d = 43
}
```

If expression -- `if`, `elif`, `else`

The `if` expression is straightforward and does not require additional clarification:

```
?a = 4,
?b = if a == 4 {
         42
     } elif {
         c,
         d
     },
b         // 42
```

Switch expression -- `switch`, `case`, `default`

The `switch` expression employs pattern matching not only to dispatch between its case paths but also to perform deconstruction, as introduced in the Expressions section.

A simple case path selection:

```
?a = 42,
switch a {
    case "foo" {
        "No!"
```

```
    }
    case 42 {
        "Yes!"
    }
}
```

A combined case path selection and tuple deconstruction:

```
?a = #("bar", 4711),
switch a {
    case "foo" {
        "Darn!"
    }
    case #("bar", ?c) {
        c
    }
    default {
        "No one picks me!"
    }
}
```

No more no less.

Note: The `switch` expression does not include a fall-through mechanism, and the use of the `default` keyword is optional. The absence of switch fall-through makes the world a little bit better. YMMV.

Enumeration -- `a.b`

The enumeration definition introduces named constants, improving code readability and maintainability. A named constant can optionally have a value of any type.

Like this:

```
enum Color {
    red = #(255, 0, 0)
    green = #(0, 255, 0)
    blue = #(0, 0, 255)
}

Foo.red,              // An enumeration literal
Foo.red.value         // #(255, 0, 0)
```

Function -- `fn`

`foo(a, b)` calls the function `foo` with a list of expression arguments separated by commas. These arguments are evaluated from left to right before invoking the function. `foo` can refer to either a named function definition or a variable

bound to a function literal. Functions can be overloaded, and their parameters can have default values.

Grok this:

```
fn foo(a, b, c = 0) {
  c;
  d;
}

fn foo(a = 1) {
  a;
}
```

Default values are assignable only to trailing parameters, and such parameters (as seen in the example above) can be omitted in function calls. Furthermore, a function call must use either positional parameters exclusively or named parameters exclusively. The following function calls illustrate equivalent function calls:

```
foo(2, 6),
foo(2, 6, 0),
foo(a: 2, b: 6),
foo(b: 6, a: 2)
```

Functions can be defined within functions:

```
fn foo(a, b, c = 0) {
    fn bar(d) {
        d
    },
    bar(a)
}
```

Functions are treated as first-class citizens:

```
import stdio.lists

export fn main() {
    ?l = [1, 3, 2],
    ?f = fn (x, y) { x > y },
    lists.sort(l, f)
}
```

String -- " ... "

A string is an immutable sequence of UTF-8 encoded characters. String interpolation is supported, and despite the UTF-8 encoding, random access to individual characters in a string is possible.

Grok this:

```
?a = 3.0,
?b = "foo $a is not ${a + 1.0}"  // b == "foo 3.0 is not 4.0" (interpolation)
?a = "foo",
?b = "bar",
?c = a ~ b,                      // c == "foobar"
?c = a ~ '\u03c9'                // c == "foo "
c[3] == ' '                      // true
r"foo\nbar"                      // A raw string
r.length == 9                    // true
```

Tuple -- `#( a, b, c, ... )`

Not much needs to be said about tuples: they are simply tuples.

Like this:

```
?a = 42,
?b = #(4711, #(a, [1, 2])),
#(_, #(?a, [_, c])) = b,
a,                               // 4711
c                                // 2
```

List -- `[ a, b, c, ... ]`

A list can hold elements of any type, and list slicing makes it easy to work with portions of a list. The expression `a[i .. j]` returns a slice of the list starting at index `i` and ending at the index `j - 1`. Both `i` and `j` can be any valid expression, and the keyword `$` represents the list's length.

Grok this:

- `a[i]` returns the i:th element.
- `a[2 = "foo"]` evaluates to a new list with element 2 set to "foo".

Study this:

```
?a = [],                  // An empty list
?a = [1, 2, 3, 4, 5],     // A rebind of variable a
a.first(),                // 1
a.rest(),                 // [2, 3, 4, 5]
?b = a[1 .. 3],           // b == [2, 3, 4]
b.length == 3,            // true
?c = a[2 .. $ - 1],       // c == [3, 4, 5]
?d = b ~ c,               // d == [2, 3, 4, 3, 4, 5]
d[1 = 42],                // [2, 42, 4, 3, 4, 5]
a[2 = 23],                // [1, 2, 23, 4, 5],
?b = 4711 ~ b,            // b == [4711, 2, 3, 4]
?f = a[$ / 2 .. $ - 1]    // f == [3, 4, 5],
```

```
f.delete(2),              // [3, 4]
?g = a,
g == a                    // true
```

Map -- `[ a : b, ... ]`

A map can be viewed as a function that creates a mapping between keys and values, both of which can be of any type.

Grok this as well:

```
?a = [:],                         // En empty map
?a = ["a": 1.0, "b": "foo"],
a["a": "bar"],                    // ["a": "bar", "b": "foo"],
a[42 : 4711],                     // ["a": "bar", "b": "foo", 42: 4711],
?c = a ~ ["pi": 3.14]             // c == [ "a" : 1.0, "b" : "foo", "pi" : 3.14 ]
c.length == 3,                    // true
a.delete("a"),                    // [ "b" : "foo" ]
a.keys,                           // ["a", b] || ["b", "a"]
a["a"],                           // 1.0
a.values                          // [1.0, "foo"] || ["foo", 1.0]
?d = ["a": 1.0, "b": "foo"],      //
a == d                            // true
```

Encapsulation -- `struct a { ... }`

A struct in Satie goes beyond a traditional C struct. It is immutable and inherently functional, serving as a unit of encapsulation for member variables and functions.

Lets dive into an example right away:

```
struct Foo {
    public a = 1
    private b = 2
    readonly c = 3
    public const d = 4
    private const e = 5

    this(a, g) {          // Optional constructor
        this(a: a, b: g)
    }

    ~this() {             // Optional destructor
        // Do nothing for now
    }

    public fn foo(a) {
```

```
        this.a + b;
    }

    private fn bar(b) {
        b + d
    }
}
```

The example above revisits familiar concepts from Java, including member variables and functions, access specifiers and modifiers, along with a constructor and destructor, and the use of the this reference. No novel elements have been introduced here.

Interestingly, the constructor is designed to call itself for the final initialization of its member variables. It's noteworthy how the constructor can use its member variables as named parameters in this self-referential call.

To instantiate a struct `Foo` do this:

```
?a = struct Foo(2, 1)
```

A struct can opt to implement certain mandatory interfaces. An *interface* specifies the member variables and functions that the struct must provide. The structure of an interface definition is as follows:

```
interface Bar {
    public zippo
    public fn bonk()
}
```

A struct that opts to implement this interface is structured as follows:

```
struct Foo : Bar {
    public zippo = 8
    public fn bonk() {
        // Delve into the nature of God
    }
    ...
}
```

A struct can implement several interfaces using a comma separated sequence of interfaces:

```
struct Foo : Bar, Bonk {
    ...
}
```

Do the following to define a set of constants:

```
struct Math {
    public const PI = 3.1;
    public const SQUARE2 = math.sqrt(2);
```

```
}
```

Note: This struct must be instantiated somewhere in the code as there is no `static` modifier in Satie.

Buf(fer) -- `buf`

A buffer is an opaque, persistent data type designed for the efficient manipulation of large amounts of characters. It is well-suited for a programming language aimed at simplifying the implementation of programming editors and more.

Note: This section is currently under consideration and the details are not yet finalized.

API overview:

1. Creation and Initialization
    - `create()`: Initializes an empty buf.
    - `fromString(string)`: Creates a buf from a given string.
    - `fromFile(filePath, lazyLoad)`: Initializes a buf from a file, with optional lazy loading for large files.
2. Reading and Access
    - `charAt(index)`: Returns the character at a specified index.
    - `substring(startIndex, endIndex)`: Retrieves a substring from the buffer.
    - `length()`: Provides the length of the text in the buffer.
3. Text Modification
    - `insert(index, string)`: Inserts a string at the specified index.
    - `delete(startIndex, endIndex)`: Deletes text between given indices.
    - `replace(startIndex, endIndex, string)`: Replaces a segment of text with a new string.
4. Text Selection and Clipboard Operations
    - `select(startIndex, endIndex)`: Selects text between given indices.
    - `cut(startIndex, endIndex)`: Cuts (removes and copies) the selected text.
    - `copy(startIndex, endIndex)`: Copies the selected text.
    - `paste(index, string)`: Pastes the copied text at the specified index.
5. Search and Navigation
    - `indexOf(substring, startIndex)`: Finds the index of the first occurrence of a substring.

- `lastIndexOf(substring, startIndex)`: Locates the last occurrence of a substring.
- `moveCursorTo(index)`: Moves the cursor to a specified index for navigation purposes.

6. Text Transformation
    - `toUpperCase()`: Converts all text in the buffer to uppercase.
    - `toLowerCase()`: Converts all text in the buffer to lowercase.
    - `map(function)`: Applies a specified function to each character in the buffer.
7. Undo Mechanism
    - `undo()`: Reverts the buffer to its previous state, utilizing a history of operations or states.
8. Concurrency
    - `asyncInsert(index, string)`: Asynchronously inserts a string at a specified index.
    - `asyncDelete(startIndex, endIndex)`: Asynchronously deletes text between given indices.
    - `asyncReplace(startIndex, endIndex, string)`: Asynchronously replaces text in a specified range.
9. Utility Functions
    - `toString()`: Converts the buf to a standard string for output or display.
    - `serialize()`: Serializes the buf for storage or transmission.
    - `deserialize(serializedData)`: Constructs a buf from serialized data.
10. Advanced Editing
    - `batch(operations)`: Performs multiple operations in a single step for efficiency.
11. File Handling and Lazy Loading
    - `loadMore()`: Incrementally loads more content from the file if lazy loading is enabled.
12. Pending additions
    - `isEmpty()`: Checks if the buffer is empty.
    - `trim()`: Removes whitespace from the beginning and end of the text.
    - `split(separator)`: Splits the buffer into a list of bufs based on a separator.
    - `merge(buffers)`: Combines multiple buf instances into one.

Design Considerations:

- *Immutability*: Each operation creates a new buf instance, preserving the original and adhering to the principles of functional programming.
- *Efficiency and Scalability*: The design is optimized for common text editing operations, with considerations for handling large files through lazy loading.
- *Clipboard Operations*: The addition of select, cut, copy, and paste functionalities provides essential editing capabilities.
- *Extensibility*: The design allows for future extensions, such as language-specific features or plugins.
- *Undo Mechanism*: The undo functionality is natural and integral, implemented via a history of states or operations.
- *Concurrency Support*: Concurrency support is vital for collaborative editing scenarios, requiring thread-safe operations and potential conflict resolution strategies.
- *Lazy Loading*: This feature enables efficient handling of large files, loading content as needed rather than all at once.

With these functionalities and considerations, the `buf` datatype becomes a comprehensive and robust tool for building a programming editor, offering a wide range of functionalities required for text editing and manipulation in a functional programming environment.

Concurrency

Any function can be executed concurrently as a job using the `spawn` keyword.

Grok this:

```
fn sum(l) { l.first() + sum(l.rest() },
?a = [1, 2, 3],
job = spawn sum(a)
```

In the example above, `spawn` returns a job reference and it can be used to send messages to the job using the `<|` operator:

```
job <| #(timeout, 1000)
```

A message sent to a job is placed in its mailbox and can be retrieved using the receive keyword.

Like this:

```
receive {
    case #(?job, ?result) {
        stdio.writeln("Job $job sent result $result")
        result
    }
    timeout 1000 {
        42
    }
```

```
}
```

Note: The `timeout` keyword is optional and the timeout unit is in milliseconds.

The mailbox is unbounded in size but can be restricted using the `std.concurrency` library function `setMaxMailboxSize`:

Like this:

```
concurrency.setMaxMailboxSize(job, 64, OnCrowding.block)
```

In the example above the job's mailbox is restricted to contain at most 64 messages, and if a sending job hits that threshold it is automatically blocked in the `<|` operator waiting for the mailbox to shrink.

As an alternative, `OnCrowding.ignore` can be used to specify that overflowing messages should be ignored.

The `OnCrowding` constant can also be replaced with a function that returns `false` if overflowing messages should be ignored or `true` if the sending job should be blocked in the `<|` operator.

The concurrency keyword `self` refers to the job which user code currently runs within.

The `std.concurrency` module also provides these functions:

- `monitor(job)` : Send a message `#(JobMonitor.died, job, reason)` to me if a job dies
- `link(job)` : Send a message `#(JobMonitor.died, job, reason)` to me if a job dies. Do the same to the linked job if I die. A link is a bidirectional monitor.
- `kill(job)`

The `spawn` keyword can take an extra `monitor` or `link` specifier to create a monitor or link during the actual spawn. Look how this is done in the upcoming example.

A small concurrency example

Here follows a small concurrent example that spawn jobs to compute Ackermann function values for the parameters `m = 3, n = 1 .. 10`.

The `main` function uses member functions in an Ackermann struct to start 10 jobs and then waits for all jobs to send a result back as a message.

Study this and do not despair:

```
import std.jobs : OnCrowding, Job
import std.stdio
import std.lists

export fn main() {
```

```
  ?ackermann = struct Ackermann(),
  ?ackermann = ackermann.startJobs(3, 10),
  ackermann.waitForJobs()
}

struct Ackermann {
    private jobs = []

    public fn startJobs(m, n, i = 0, startedJobs = []) {
        if i < n {
            fn computeAckermann(parentJob, m, n) {
                ?result = ackermann(m, n),
                parentJob <| #(self, m, n, result)
            },
            ?job = spawn monitor computeAckermann(self, m, i),
            job.setMaxMailboxSize(job, 4, OnCrowding.block),
            startJobs(m, n, i + 1, job ~ startedJobs)
        } else {
            this(jobs: startedJobs)
        }
    }

    public fn waitForJobs() {
        fn waitForJobs(jobs) {
            if jobs.length > 0 {
                receive {
                    case #(?job, ?m, ?n, ?result) {
                        stdio.writeln("ackermann($m, $n) = $result"),
                        waitForJobs(jobs.delete(job))
                    }
                    case #(Job.died, ?job, ?reason) {
                        stdio.writeln(
                            "Oh no! Compute job $job died: $reason")
                    }
                }
            } else {
                this(jobs: [])
            }
        },
        waitForJobs(jobs)
    }

    private fn ackermann(m, n) {
        if m == 0 {
            n + 1
        } elif n == 0 {
```

```
            ackermann(m - 1, 1)
        } else {
            ackermann(m - 1, ackermann(m, n - 1))
        }
    }
}
```

Source: ackermann.sa

Directory hierarchy of modules

A directory hierarchy of modules is a nice way to organize code and in the example below a module uses the `foreach` and `writeln` functions from the standard libray module hierarchy:

```
import std.stdio
import std.lists

export fn main(args) {
  lists.foreach(fn (arg) { stdio.writeln("$arg") }, args)
}
```

Note how the module names must be specified in the call to the call to `foreach` and `writeln`, i.e. nothing is automatically imported into the module namespace.

It is possible to import enumerations, interfaces, structs and functions into a module namespace:

```
import std.stdio : writeln
import std.lists : foreach

export fn main(args) {
  foreach(fn (arg) { writeln("$arg") }, args)
}
```

In the Building and Executing section above the *zarah* project was introduced and it had the following directory hierarchy of modules:

```
$ cd zarah
$ find .
.
./src
./src/main.sa
./src/utils
./src/utils/httpclient.sa
./src/database
./src/database/backup.sa
./src/database/utils
./src/database/tablestore.sa
```

The modules `std.lists` and `database.utils.lists` have the same module name. To resolve this, an import alias can be used.

Like this:

```
import std.stdio : writeln
import std.lists
import dlists = database.utils.lists
import database

export fn main(args) {
  lists.foreach(fn (arg) { writeln("$arg") }, args)  // as before
  dlists.removeReference(fn (staleReference) {
      database.removeReference(staleReference)
  }, database.getStaleReferences())
}
```

That is it.

Appendix A: Operator precedences

Operators in decreasing order of precedence:

Expression	Description
a.b	Field access
a(b, c)	Function call
a[i]	List indexing
a[b .. c]	List slicing (see ”List Literal” above)
a[a = b]	List setter (see ”List Literal” above)
a[a : b]	Map setter (see ”Map Literal” above)
a[a ; b]	Struct setter (see ”Struct Literal”)
-a	
+a	
!a	
~a	Bitwise complement
<|	Send message
cast(t)a	Cast expression
a ^^ b	Exponentiation
a * b	
a / b	
a % b	Modulus
a + b	
a - b	
a ~ b	Concatenation
a « b	
a » b	

Expression	Description
a »> b	Unsigned right shift
a in b	Map membership
a == b	Equality
a < b	
a <= b	
a > b	
a >= b	
a | b	
a ^ b	Bitwise xor
a & b	
a && b	Logical and
a || b	
a = b	

Appendix B: PEG grammar

```
#
# Top level structure
#

Program <- _ (Imports __)? TopLevelDefs EOF
TopLevelDefs <- TopLevelDef (__ TopLevelDef)*
TopLevelDef <- StructDef / InterfaceDef / EnumDef / FunctionDef

Imports <- Import (__ Import)*
Import <- "import" __ (ModuleAlias _ "=" _)? _ ModulePath
ModuleAlias <- Identifier
ModulePath <- Identifier ("." Identifier)* (_ ":" _ ImportedEntities)?
ImportedEntities <- Identifier (_ "," _ Identifier)*

#
# Expression
#

Expr <- BindExpr
BindExpr <- (Literal / UnboundVariable / Identifier) (_ "=" _ Expr) / SendExpr
SendExpr <- ("self" /
             ControlFlowExpr /
             SpawnExpr /
             Identifier /
             "(" _ Expr _ ")") (_ "<|" _ Expr) / LogicalOrExpr
LogicalOrExpr <- LogicalAndExpr (_ "||" _ LogicalAndExpr)*
LogicalAndExpr <- BitwiseAndExpr (_ "&&" _ BitwiseAndExpr)*
```

```
BitwiseAndExpr <- BitwiseXorExpr (_ "&" _ BitwiseXorExpr)*
BitwiseXorExpr <- BitwiseOrExpr (_ "^" _ BitwiseOrExpr)*
BitwiseOrExpr <- LargerThanEqualExpr (_ "|" _ LargerThanEqualExpr)*
LargerThanEqualExpr <- LargerThanExpr (_ ">=" _ LargerThanExpr)*
LargerThanExpr <- LessThanEqualExpr (_ ">" _ LessThanEqualExpr)*
LessThanEqualExpr <- LessThanExpr (_ "<=" _ LessThanExpr)*
LessThanExpr <- IsNotExpr (_ "<" _ IsNotExpr)*
IsNotExpr <- IsExpr (_ "!is" _ IsExpr)*
IsExpr <- NotEqualExpr (_ "is" _ NotEqualExpr)*
NotEqualExpr <- EqualExpr (_ "!=" _ EqualExpr)*
EqualExpr <- InExpr (_ "==" _ InExpr)*
InExpr <- UnsignedRightShiftExpr (_ "in" _ UnsignedRightShiftExpr)*
UnsignedRightShiftExpr  <- RightShiftExpr (_ ">>>" _ RightShiftExpr)*
RightShiftExpr <- LeftShiftExpr (_ ">>" _ LeftShiftExpr)*
LeftShiftExpr <- ConcatenateExpr (_ "<<" _ ConcatenateExpr)*
ConcatenateExpr <- MinusExpr (_ "~" _ MinusExpr)*
MinusExpr <- PlusExpr (_ "-" _ PlusExpr)*
PlusExpr <- ModulusExpr (_ "+" _ ModulusExpr)*
ModulusExpr <- DivideExpr (_ "%" _ DivideExpr)*
DivideExpr <- MultiplicateExpr (_ "/" _ MultiplicateExpr)*
MultiplicateExpr <- ExponentiationExpr (_ "*" _ ExponentiationExpr)*
ExponentiationExpr <- CastExpr (_ "^^" _ CastExpr)*
CastExpr <- "cast" _ "(" _ ("int" / "float") _ ")" _ SendMessageExpr /
            SendMessageExpr
SendMessageExpr <- BitwiseComplementExpr (_ "*" _ BitwiseComplementExpr)*
BitwiseComplementExpr <- "~" _ NotExpr / NotExpr
NotExpr <- "!" _ UnaryPlusExpr / UnaryPlusExpr
UnaryPlusExpr <- "+" _ UnaryMinusExpr / UnaryMinusExpr
UnaryMinusExpr <- "-" _ PostfixExpr / PostfixExpr
PostfixExpr <- PrimaryExpr _ ("." _ (ControlFlowExpr / Identifier) /
                              "(" _ Args? _ ")" /
                              "[" _ Expr _ "]")*

PrimaryExpr <- "this" /
               "self" /
               "$" /
               Literal /
               ControlFlowExpr /
               SpawnExpr /
               StructExpr /
               UnboundVariable /
               Identifier /
               "(" _ Expr _ ")"

Literal <- BooleanLiteral /
           NumberLiteral /
```

```
            CharacterLiteral /
            StringLiteral /
            FunctionLiteral /
            TupleLiteral /
            (Identifier _)? ListLiteral /
            (Identifier _)? MapLiteral /
            StructLiteral

BooleanLiteral <- "true" / "false"

NumberLiteral <- FloatingPoint / Integral
Integral <- HexIntegral / BinaryIntegral / OctalIntegral / DecimalIntegral
FloatingPoint <- DecimalPointLeading / DecimalPointTrailing
DecimalPointLeading <- [0-9]* "." [0-9]+ ExponentPart?
DecimalPointTrailing <- [0-9]+ ExponentPart
ExponentPart <- [eE] [+-]? [0-9]+
DecimalIntegral <- [0-9]+
OctalIntegral <- "0" [0-7]+
HexIntegral <- "0x" [0-9a-fA-F]+
BinaryIntegral <- "0b" [01]+

CharacterLiteral <- "'" ( Escape / NonQuoteChar ) "'"
Escape <- "\\" ( [abfnrtv'"\\] /
                 "x" HexDigit HexDigit /
                 "u" HexDigit HexDigit HexDigit HexDigit /
                 "U" HexDigit HexDigit HexDigit HexDigit
                     HexDigit HexDigit HexDigit HexDigit /
                  OctalDigit /
                  OctalDigit OctalDigit /
                  OctalDigit OctalDigit OctalDigit )
HexDigit <- [0-9a-fA-F]
OctalDigit <- [0-7]
NonQuoteChar <- [^']

StringLiteral <- RegularString / RawString
RegularString <- '"' ( EscapeSequence / [^"] )* '"'
EscapeSequence <- "\\" [btnvfr"\\]
RawString <- 'r"' [^"]* '"'

FunctionLiteral <- "fn" _ "(" _ Params? _ ")" _ BlockExpr

TupleLiteral <- "#(" _ Exprs? _ ")"
Exprs <- Expr (_ "," _ Expr)*

ListLiteral <- "[" _ Exprs? _ "]" /
               "[" Expr _ ".." _ Expr "]" /
```

```
              "[" _ IndexValues _ "]"
IndexValues <- IndexValue (_ "," _ IndexValue)*
IndexValue <- DecimalIntegral _ ":" _ Expr

MapLiteral <- "[:]" / "[" _ KeyValues? _ "]"
KeyValues <- KeyValue (_ "," _ KeyValue)*
KeyValue <- (Literal / Identifier) _ ":" _ Expr

StructLiteral <- "[" _ MemberValues? _ "]"
MemberValues <- MemberValue (_ "," _ MemberValue)*
MemberValue <- Identifier _ ";" _ Expr

ControlFlowExpr <- IfExpr / SwitchExpr / ReceiveExpr / BlockExpr

IfExpr <- "if" __ Expr _ BlockExpr
          (_ "elif" __ Expr _ BlockExpr)*
          (_ "else" _ BlockExpr)?

SwitchExpr <- "match" __ Expr _ "{"
             (_ "case" __ Expr _ BlockExpr)+ _ "}"

ReceiveExpr <- "receive" _ "{"
               (_ "case" __ Expr _ BlockExpr)+
               (_ "timeout" _ DecimalIntegral _ BlockExpr)? _ "}"

SpawnExpr <- "spawn" (__ "monitor" / "link")? __ Expr

StructExpr <- "struct" _ Identifier _ "(" _ Args? _ ")"

UnboundVariable <- "?" _ Identifier

Identifier <- [a-zA-Z_][a-zA-Z_0-9_]*

#
# Struct definition
#

StructDef <- "struct" __ Identifier _ ( ":" _ Interfaces _)?
                  "{" _ StructMembers _ "}"
Interfaces <- Identifier (_ "," _ Identifier)*
StructMembers <- StructMember (_ StructMember)*
StructMember <- Constructor / Deconstructor / MemberFunction / MemberVariable
Constructor <- "this" _ "(" _ Params? _ ")" _ BlockExpr
Deconstructor <- "~this" _ "(" _ Params? _ ")" _ BlockExpr
MemberFunction <- MemberAccess _ FunctionDef
MemberAccess <- "public" / "private"
```

```
MemberVariable <- (MemberAccess (_ "const")? / "readonly") _ Identifier
                  (_ "=" _ Expr)?

#
# Interface definition
#

InterfaceDef <- "interface" __ Identifier _ "{" _ InterfaceMembers _ "}"
InterfaceMembers <- InterfaceMember (_ InterfaceMember)*
InterfaceMember <- InterfaceMemberFunction / InterfaceMemberVariable
InterfaceMemberFunction <- MemberAccess _ InterfaceFunction
InterfaceFunction <- "fn" _ Identifier _ "(" _ Params? _ ")"
InterfaceMemberVariable <- (MemberAccess (_ "const")? / "readonly") _ Identifier

#
# Enumeration definition
#

EnumDef <- "enum" __ Identifier _ "{" _ EnumValues _ "}"
EnumValues <- EnumValue (__ EnumValue)*
EnumValue <- Identifier (_ "=" _ Expr)?

#
# Function definition
#

FunctionDef <- ("export" _)?
               "fn" __ Identifier _ "(" _ Params? _ ")" _ BlockExpr
Params <- NonDefaultParams _ "," _ DefaultParams /
          NonDefaultParams /
          DefaultParams
NonDefaultParams <- NonDefaultParam (_ "," _ NonDefaultParam)*
NonDefaultParam <- Identifier !(_ "=")
DefaultParams <- DefaultParam (_ "," _ DefaultParam)*
DefaultParam <- Identifier _ "=" _ Expr

BlockExpr <- "{" _ BlockLevelExprs _ "}"
BlockLevelExprs <- BlockLevelExpr (_ "," _ BlockLevelExpr)*
BlockLevelExpr <- FunctionDef / Expr

Args <- PositionalArgs / NamedArgs
PositionalArgs <- !NamedArg Expr (_ "," _ Expr)*
NamedArgs <- NamedArg (_ "," _ NamedArg)*
NamedArg <- Identifier _ ":" _ Expr

#
```

```
# Misc
#

#_ <- WS*
#__ <- WS+
_ <- (WS / Comments)*
__ <- (WS / Comments)+
WS <- [ \t\r\n]
Comments <- SingleLineComment / BlockComment
SingleLineComment <- "//" (!EOL .)* EOL?
EOL <- "\r\n" / "\n" / "\r"
BlockComment <- "/*" (!"*/" .)* "*/"
EOF <- _ !.
```

Appendix C: A todo list example

```
import std.stdio : writeln
import std.lists

struct TodoItem {
    private description
    private completed

    this(description) {
        this(descriptor: descriptor, completed: false)
    }

    public fn markCompleted() {
        this(completed: true)
    }

    public fn toString() {
       if completed {
           "[x] "
       } else {
           "[ ] "
       }
    }
}

struct TodoList {
    private items = [:]

    public fn addItem(tag, description) {
        ?item = struct TodoItem(tag, description),
```

```
        this(items: [tag : description] ~ items)
    }

    public fn markItemCompleted(tag) {
        ?item = items[tag].markCompleted(),
        this(items: item ~ items.delete(tag))
    }

    public fn displayItems() {
        lists.foreach(fn () { writeln(item.toString())}, items)
    }
}

export fn main() {
    fn loopUntilQuit(todoList) {
        ?input = stdio.readLine(stdio.Stream.stdin),
        if input.command == "add" {
            ?todoList = todoList.addItem(input.description),
            loopUntilQuit(todoList)
        } elif input.command == "complete" {
            ?todoList = todoList.markItemCompleted(input.index),
            loopUntilQuit(todoList)
        } elif input.command == "show" {
            todoList.displayItems(),
            loopUntilQuit(todoList)
        } elif input.command == "quit" {
            0
        } else {
          stdio.writeln("Unknown command: $input.command"),
          loopUntilQuit(todoList)
        }
    },
    ?todoList = struct TodoList(),
    loopUntilQuit(todoList)
}
```

Source: todo.sa