

Chapter 1

Introduction

The goal of the language is to write code efficiently and easally while it's still extendable, even at the core language. The language is thereby inspired by Rust, C#, Python and C/C++.

Chapter 2

Keywords

- `fn`
- `rt`
- `infix`
- `let`
- `const`
- `struct`
- `if`
- `when`

Chapter 3

Operators

- +
- -
- *
- /
-

Chapter 4

Grammar

4.1 Comments

```
// I'm a single line comment

/*
 * I'm a multi-line
 * comment
 */function
```

4.1.1 Variables

```
let a = 0;
let b, c = 1;
let d = 2, e = 3;
```

4.1.2 Constants

Constants are immutable variables.

```
const a = 0;
const b, c = 1;
const d = 2, e = 3;
```

4.2 Flow control

4.2.1 if-statement

```
if a == b {
    // code...
}
```

4.2.2 Pattern match

```
when x {  
    0 => print("Hello World");  
    1 => print("Hello John");  
}
```

4.2.3 Variable assignment

4.3 typing

4.3.1 Structs

```
struct Structure {  
    a, b: int;  
    c: float;  
}
```

4.3.2 Enums

```
enum Seasons {  
    winter,  
    spring  
    summer,  
    autumn  
}
```

4.4 Literals

4.4.1 Strings

4.5 Routines

4.5.1 Callables

Callables are functions or routines that can be called from somewhere else.

```
rt print_hello(){  
    print("Hello")  
}
```

4.5.2 Functions

Functions cannot, so do not have side-effects

```
fn sum(lhs: int, rhs: int) {  
    return a + b;  
}
```

4.5.3 Shorthand notation

```
rt print_hello => print("Hello");  
fn sum(a: int, b: int) => a + b;
```

4.5.4 Infix

```
infix fn + (a, b)
```

Chapter 5

Examples

5.1 Hello World

```
rt main() => print("Hello World!");
```

5.2 Operator Overloading

Operator overloading can be achieved by defining the operator as an Infix callable.

```
struct Vector [int:2];
infix fn + (rhs: Vector, lhs: Vector) => [ lhs[0] + rhs[0], lhs[1] + rhs[1] ]

rt main() {
    const a = Vector[1, 2];
    const b = Vector[3, 4];
    print(a + b); // Vector[4, 6]
}
```

5.3 Custom operators

```
infix fn % (rhs: int, lhs: int) => (rhs / lhs) + rhs - (rhs / lhs);

rt main() {
    print(5 % 2); // 1
}
```

5.4 Member callable declaration

```
struct Vector [int:2];
```

```
member rt Vector::print() => print(print[0], print[1]);

rt main() {
    const a = Vector[1, 2];
    const b = Vector[3, 4];
    print(a + b); // Vector[4, 6]
}
```