

CatLang Design Doc

Alic Szecsei

July 8, 2018

Contents

1	Background	2
2	Functions	2
3	Typing	2
3.1	Type Unions	3
4	Casting	3
4.1	Type Checking	3
4.2	Safe Casting	3
4.3	Unsafe Casting	4
5	Native Types	4
5.1	Primitives	4
5.2	Any	5
5.3	Optionals	5
5.4	Collections	6
5.4.1	Arrays	6
5.4.2	Strings	7
5.4.3	Structs	7
5.5	Functions	8
5.6	Pointers	10
5.7	Enums	11
6	Build System	11

1 Background

My goal with CatLang is to produce a programming language well-suited for low-level programming (such as game development) which includes many of the features common in more modern programming languages. I hardly propose that CatLang is a universal fix, but it seems like an engaging venture for my specific uses (and personal opinions about what a “good programming language” looks like).

A few notes about specific opinions I hold: *kind* type-safety is important. Compilers should be able to infer as much as possible. Functions should be first-class; purely functional languages are too rigid, especially when performance is critical, but enabling programmers to use *aspects* of functional languages seems the best of both worlds.

2 Functions

Functions can be defined in a C-like fashion:

```
function timesTwo(num: int) -> int {  
    return num * 2;  
}
```

This is syntactically equivalent to a second, more functional approach:

```
const timesTwo = (num: int) -> int {  
    return num * 2;  
}
```

3 Typing

CatLang uses strong typing, although some typing can be inferred. As an example:

```
const timesTwo = (num) -> {  
    return num * 2;  
} // ERROR! Argument types cannot be inferred  
  
const timesTwo = (num: int) -> {  
    return num * 2;  
} // OK!  
  
const PI = 3.14159265; // OK!
```

3.1 Type Unions

Types can be composed through unions; this means that a value can be one of the listed types. For example:

```
const falseIfEven = (num : int) -> {  
    if (num % 2 == 0) {  
        return false;  
    }  
    return num;  
}
```

This method will return an object of type `bool | int`; that is to say that it may be either a `bool` or an `int`, and CatLang cannot deterministically know which one it will be. Any further uses of the returned value must allow for *both* possibilities, or CatLang will throw an error.

4 Casting

Casting is a way to check the type of an instance or to treat an instance of one type as another. There is no implicit casting in CatLang.

4.1 Type Checking

Type checking is performed with the `is` operator:

```
const x: int | string = otherFunction();  
if (x is int) {  
    print("This is an int!");  
} else {  
    print("This is a string!");  
}
```

4.2 Safe Casting

Safe casting returns an optional type, equivalent to `type | null`; safe casting must be evaluated at run-time. If run-time type information indicates that the variable's *actual value* is of the casted type, the cast will succeed. Otherwise, the cast will return `null`.

```
const x: int | string = otherFunction();  
const myFunc = (i: int) -> {
```

```

        // Something
    }

    myFunc(x); // ERROR!
    myFunc(x as int); // ERROR!
    if (let castX = x as int) {
        myFunc(castX); // OK
    }

```

The reason this last example works is that an assignment operator returns the object that was assigned; the last if statement is thus equivalent to:

```

let castX = x as int;
if (castX) {
    myFunc(castX);
}

```

4.3 Unsafe Casting

Unsafe casting is used to force a cast from one value to another. While this can be used to avoid type lookups at run-time, it is not recommended for standard use. This is equivalent to a C-style cast.

```

const x: long = 5000000000;
const y = x as! int; // 705032704

```

5 Native Types

5.1 Primitives

1. Integer Numbers

- (a) **S8** (1 byte, signed)
- (b) **U8** (1 byte, unsigned)
- (c) **S16** (2 bytes, signed)
- (d) **U16** (2 bytes, unsigned)
- (e) **S32** (4 byte, signed)
- (f) **U32** (4 byte, unsigned)
- (g) **S64** (8 byte, signed)

- (h) `U64` (8 byte, unsigned)
- (i) `char` (equivalent to U8)
- (j) `int` (equivalent to S32)
- (k) `long` (equivalent to S64)

2. Booleans

- (a) `bool` (1 byte)

3. Floating-Point Numbers

- (a) `float` (4 bytes)
- (b) `double` (8 bytes)

4. Unvalued

- (a) `null`

5.2 Any

The `any` type is the direct equivalent to C's `void*` type.

5.3 Optionals

CatLang uses an “optional” type, equivalent to `type | null`, to help avoid null-pointer exceptions. Any function that requires a non-Optional value must be enclosed in a conditional to ensure that the Optional value exists, or an error will be thrown.

```
const myPrint = (num?: int) -> {
    if (num) {
        print(num);
    } else {
        print("Nope!");
    }
} // OK!

const myPrint2 = (num?: int) -> {
    print(num);
} // ERROR!
```

If a user wishes to force-unwrap an optional value, they can use the `!` operation to do so. This is, however, not recommended and may lead to null-pointer exceptions.

```
const myPrint = (num?: int) -> {  
    print(num!);  
} // OK!
```

5.4 Collections

CatLang has two ways to combine data: arrays and structs.

5.4.1 Arrays

CatLang's arrays differ from C-style arrays in that they contain information about their length.

```
const finalElement = myArray[myArray.length - 1];
```

Arrays also include convenience methods for a more functional style of programming; namely, `map`, `filter`, `each`, and `reduce`. They are instantiated using a syntax similar to C:

```
const arr = [25]int;
```

Arrays can be either static- or dynamically-sized:

```
const staticArray = [25]int;  
const dynamicArray = [...]int;
```

Arrays of pointers and pointers to arrays are syntactically different:

```
const pointerToArray : *[]int = @myArray;  
const arrayOfPointers : []*int = myArray;
```

Arrays can be iterated over:

```
const arr = []int { 1, 2, 3, 4, 5 };  
for (x in arr) {  
    print(x);  
}  
delete arr;
```

They can also iterate over inner properties of structs; in the following example, loops A and B are equivalent:

```

struct Vector3 {
    x : float;
    y : float;
    z : float;
}
const arr = []Vector3 { ... };
for (xVal in arr.x) {
    print(xVal);
} // A
for (vec in arr) {
    print(vec.x);
} // B

```

5.4.2 Strings

CatLang’s **string** type is a wrapper around **Array<char>**. A notable feature of the language is string interpolation:

```

let name = "";
print("Enter your name: ");
readLine(name);
print("Hello, ${name}!\n");

```

5.4.3 Structs

CatLang’s syntax for defining structures is straightforward:

```

struct Vector3 {
    x : float;
    y : float;
    z : float;
}

```

Structs can also have default values assigned:

```

struct Vector3 {
    x : float = 1.0;
    y : float = 2.0;
    z : float = 3.0;
}

```

To denote an “owned” property in a struct, use the **!** keyword. When a struct is deleted, all owned properties are also deleted.

```

struct Transform {
    position : ! Vector3;
}

```

```

        rotation : ! Quaternion;
    }

```

To allocate memory on the stack, users should declare variables *without* the **new** keyword. If the **new** keyword is used, the variable is allocated on the heap and must be freed later with the **delete** keyword.

```

const v1 = Vector3; // Allocated on the stack.
const v2 = new Vector3; // Allocated on the heap...
delete v2; // ...so must be manually freed.

```

Structs can define how they are stored in arrays, in order to reduce cache misses. Structs default to the “array of structs” schema, but can be swapped to the “struct of arrays” schema using the **SOA** keyword:

```

struct V3A {
    x : float = 1;
    y : float = 2;
    z : float = 3;
}
let v1 = [4]V3A; // Memory will contain 1 2 3 1 2 3 1 2 3 1 2 3

struct V3B SOA {
    x : float = 1;
    y : float = 2;
    z : float = 3;
}
let v2 = [4]V3B; // Memory will contain 1 1 1 1 2 2 2 2 3 3 3 3

```

No matter how these arrays are stored in memory, they are used and referenced the same way within CatLang code.

Fields of a struct that begin with an underscore are private fields and are only accessible to functions within the struct’s namespace. All other fields are public.

```

struct MyStruct {
    _hiddenVar : string;
    publicVar : string;
}

```

5.5 Functions

Global functions are defined without any namespace.

```

const timesTwo = (num: int) -> {
    return num * 2;
}
const value = timesTwo(4);

```


Static functions are attached to a struct, but not an instance of that struct. They *cannot* use the keyword `this` in their parameter list.

```
Vector3::dot = (first: Vector3, other: Vector3) -> {  
    return first.x * second.x + first.y * second.y + first.z  
        * second.z;  
}  
const v1 = Vector3;  
const v2 = Vector3;  
const dot = Vector3::dot(v1, v2);
```

Instance functions are attached to an instance of a struct. They *must* include the keyword `this` as the first parameter.

```
Vector3::toString = (this) -> {  
    return "(${this.x}, ${this.y}, ${this.z})";  
}  
const v1 = Vector3;  
print(v1.toString());
```

There is one special instance function: if the `constructor` function is declared, it is called whenever a new instance of the struct is created. If a constructor is declared, it *must* be called; using the default constructor-less instantiation will raise an error.

```
Vector3::constructor = (this, x: float, y: float, z: float) -> {  
    this.x = x;  
    this.y = y;  
    this.z = z;  
}  
const v1 = Vector3(2, 3, 4); // OK  
const v2 = new Vector3(1, 4, 9); // OK  
const v3 = Vector3; // ERROR!
```

Functions attached to a struct, whether static or instance, are treated as immutable and cannot be redefined. The only way to overload a function is through the use of optional types.

Generic functions look similar to generics in other languages:

```
const contains = <T>(arr: []T, value: T) -> {  
    for (x in arr) {  
        if (x == value) {  
            return true;  
        }  
    }  
    return false;  
}
```

When generic functions are called, if the generic type can be inferred, it can be dropped:

```
const arr = []int { 1, 2, 3, 4, 5 };
const result = contains(arr, 4);
```

However, if the generic type cannot be inferred, it must be made explicit:

```
const arrayFactory = <T>(count: number) -> {
    return new [count]T;
}
const arr = arrayFactory(10); // ERROR!
const arr = arrayFactory<int>(10); // OK!
```

Functions can also be passed in to other functions, for a more functional style of programming. An example `map` function might look like:

```
const map = <InT, OutT>(arr: []T, fn: (InT)->OutT) -> {
    result = new [arr.count]OutT;
    for (x = 0; x < arr.count; x++) {
        result[x] = fn(arr[x]);
    }
    return result;
}
```

5.6 Pointers

Pointers can be retrieved from any struct or primitive. Optionals do not have pointers, although you can have an optional pointer. Functions can be called on pointers using the standard dot syntax, unlike C++'s arrow operator.

To retrieve the pointer to an object, use the `@` operator. To retrieve an object from a pointer, use the `*` operator.

```
const myFunc = (x: *Vector3) -> {
    // This takes a pointer to a Vector3 struct
    print(x.toString()); // prints x as normal
}

const v = Vector3;
myFunc(@v);
```

Ownership works similarly with pointers:

```
struct Transform {
    position : !*Vector3;
    rotation : !*Quaternion;
}
```

5.7 Enums

Enum types are declared similarly to structs:

```
enum Alignment {  
    Left, // 0  
    Right, // 1  
    Center, // 2  
}  
const a = Alignment::Left;  
const val = a as U16;
```

By default, enums are wrappers around **U16** values, starting at 0. However, they can be manually associated with other values:

```
enum Alignment: U32 {  
    Left, // 0  
    Right = 12, // 12  
    Center // 13  
}
```

If the type an enum wraps is not able to be incremented (for example, a string) then the associated values of all enum values must be declared.

6 Build System

NOTE: THIS IS WORK-IN-PROGRESS