Itai's programming language

Contents

Basic syntax

Statements are followed by a semicolon (`;`).

Expressions can be grouped inside parentheses (`()`) for precedence.

Literals

Integers: `<number>` for example: `12` .

Floats: `<integer>.<integer>` for example: `3.14` . if no decimal point is provided, `.0` is added. so `12` becomes `12.0` .

characters: `'<character>'` for example: `'A'` . only ASCII characters are supported.

strings: `"<string>"` for example: `"Hello, World!"` . only ASCII characters are supported.

Operators

Math: addition: `+` , substraction: `-` , multiplication: `*` , division: `/` , modulo (remainder): `%` .

Bitwise: and: `&` , or: `|` , not: `!` , xor: `^` .

Asignment: `=` .

Comparison: `==` , `!=` , `<` , `>` , `<=` , `>=` .

Control flow: and: `&&` , or: `||`

prefix and postfix `+` and `-`

```
var int a=10;
// '+' operator
var b = a++; // 10, a is 11
var c = ++a; // 12, a is 12

// '-' operator
var d = a--; // 12, a is 11
var e = --a; // 10, a is 10
```

Comments

- Start with `//` until end of line.
- Between `/*` and `*/` (doesn't work as part of an expression, for eample: `*x/*y` means dereference `x` and `y` and divide `x` by `y`).

Importing libraries

```
import library
from library import thing
```

Variables

```
var <type> <name>;
var <type> <name> = <value>;
```

The type can be detected by the compiler automatically:

```
var <name> = <value>;
```

Constants

```
const <type> <name> = <value>;
```

The type doesn't have to be specified:

```
const <name> = <value>;
```

Examples

```
// regular variables
var int num = 10;
var num2 = 5;

// constants
const int NUM = 10;
const NUM2 = 5;
```

Arrays

Initializing:

If all the array is filled in declaration, there is no need to specify the size.

```
var <type> <name>[<size>];
var type <name>[size] = {<comma separated elements>};
var <name>[<size>] = {<comma separated elements>};
```

accessing elements:

```
// with type declaration:
// var int array[] = {1, 2, 3, 4, 5};
var array[] = {1, 2, 3, 4, 5};
// accesing elements
```

```
array[0]; // 1
array[1]; // 2

// writing to elements
array[0]=0;
array[0]; // 0
```

Types

`type` - Used to save types. **usage:** assignment: `var type type_int = int`, comparison: `type_int == int`.

Boolean

`bool` - 1 byte, holds `true` or `false` (which are 1 or 0 respectively).

Any number that isn't 0 is true, 0 is false.

Numbers

`int` - 4 byte integer.

`float` - 32 bit floating point number.

`size` - 32 or 64 bit depending on the architecture.

Larger sizes are available by appending the size to the end:

`int` sizes:

`2`, `4`, `8`, `16` (bytes).

`int` is an alias for `int4`.

`float` sizes:

`32`, `64` (bits).

`float` is an alias for `float32`.

unsigned numbers

`int` and `size` are signed by default. you can make them unsigned by prefixing them with `u`:

unsigned int: `uint`.

unsigned size: `usize`.

You use `u` in combination with `int` sizes:

unsigned 2 byte int: `uint2`.

unsigned 8 byte int: `uint8`.

unsigned 16 byte int: `uint16`.

Text

`char` - 1 byte, can hold only ASCII characters.

`str` - Constant string, alias for a fixed size constant `char` array.

Other

`struct` - A Structure can hold any fixed size type inside. used to group variables that belong to the same thing together. Variables defined inside a `struct` don't need to be declared with the `var` keyword.

`enum` - An Enumeration (enum) is a bunch of constants in a single place. they can only be numbers. the first number is 0 by default.

`struct` and `enum` example

```
// a struct
struct name {
        int a;
        float b;
        str name;
};

// an enum
enum name {
        A, // 0
        B, // 1
        C=10 // 10
};
```

Custom types

You can create a custom type with the `deftype` (define type) keyword:

```
deftype oldtype newtype;
```

for example:

```
deftype usize unsigned_size;
```

You can also use `structs` and `enums`:

```
deftype enum {
        CAT,
        DOG
} AnimalType;
// AnimalType is now a type that can only be CAT or DOG (0 or 1 respectively).

deftype struct {
        String name;
        uint age;
        AnimalType type;
} Animal;
// 'Animal' is now a type.
```

```
// usage
var Animal dog;
dog.name="doggy";
dog.age=2;
dog.type=DOG;
```

Casting

Casting between types is done by enclosing the value you want to cast in parentheses prefixed by the type you want to cast to.

```
var int a=10;
var float b=float(a); // a is converted to a float, so it's now 10.0
// a more elegant way
var c=float(a); // no type name duplication
```

Pointers and references

Pointers and references are supported.

They work for basic types, functions and objects (classes).

Pointers:

```
var a = 10;
var *ptr_to_a = &a;
// to access the value in 'a'
var b = *ptr_to_a; // 10
```

References:

```
var a = 10;
var &ref_to_a = a;
// to access 'a'
var b = ref_to_a; // exactly the same as 'var b = a'
```

Loops

There are two loop types: `while` and `for`.

You can exit a loop with the `break` keyword, and jump to it's start with the `continue` keyword. The output of all the following programs will be: `|1|2|3|4|5|`.

The `while` loop

The while loop keeps running until the condition provided is false.

```
// count to 10
var a = 1;
while a <= 10 {
        print("|{}|", i)
        a=a+1;
}
```

The `for` loop

The `for` loop has two versions:
Counter:

```
// count to 10
for var i=1; i<=10; i=i+1 {
        print("|{}|", i);
}
```

Iterator:

```
var array = {1, 2, 3, 4, 5};
// iterate over every element in the array
for element in array {
        print("|{}|", element);
}
```

if / else

```
var a = 10;
if a == 10 {
        print("'a' is 10\n");
} else {
        print("'a' isn't 10\n");
}

// multiple tests
if a == 10 {
        print("'a' is 10\n");
} else if a == 5 {
        print("'a' is 5\n");
} else {
        print("'a' isn't 10 or 5\n");
}
```

switch

```
var a = 10;
switch(a) {
        10 => print("'a' is 10\n");
        5 => print("'a' is 5");
default => // default catches anything that isn't handled by the other cases.
        print("'a' isn't 10 or 5\n");
}

// for declaring variables inside the switch or having more than one line, you can
switch(a) {
        10 => {
                print("'a' is 10\n");
                var b = 5;
                print("'b' is 5\n");
        }
default =>
        print("'a' isn't 10\n");
}
```

```
// you can add labels to the cases and jump to them from a different case
a = 5;
switch(a) {
        10 : ten => print("'a' is 10\n");
default => {
        print("'a' isn't 10, jumping to 10 case...\n");
        break ten;
}
}
// output of above switch will be:
//
// 'a' isn't 10, jumping to 10 case...
// 'a' is 10
```

functions

```
fn name(type parameter) return_type {
        body
}
```

`body` is the body of the function.

`return_type` is the return type of the function. if not provided, the function doesn't return anything.

example

```
fn add(int a, int b) int {
        return a+b;
}
```

Variable argument functions and methods

Variable argument functions work by adding `name...` (name can be any valid variable name) as the last parameter in a function or method. `name` is a `Vector<any>` that contains the arguments.

to get each argument, you can use the `Vector<type T>` methods to get the data appending the `.get()` method of the `any<type T>` class. you can get the type by appending `.type` instead of `.get()`.
The last argument can be accessed using the `pop_front()` Vector method, and the first one using the `pop_back()` Vector method.

Example

```
fn variable_args(args...) {
        var arg1 = args.pop_back().get();
        var last_arg = args.pop_front().get();

        // you can get a regular array of the arguments
        var args_as_array[] = args.to_array();
        // iterate over the array
        for arg in args_as_array {
                print("|{}|", arg);
        }
        // you can also do this
        for arg in args.to_array() {
                print("|{}|", arg);
        }
}
```

Objects

```
class name {
<access modifier>:
        variables, methods
};
```

Members (variables in the object)

Syntax

Same as variables but without the `var` keyword:

```
class Animal {
        int i;
        float f;
        String s
};
```

Methods

Syntax

Same as functions but without the `fn` keyword:

```
class Animal {
        print_sound() {
                // print the sound
        }
        get_name() str {
                // return the name
        }
};
```

`this` and `super`

- `this` is a reference to the current instance of the class, it has to be used to access anything inside the class from inside the class.
- `super` is a reference to the super-class. when used like a function (`super()`) it calls the super-class constructor.

Special methods

- **constructors** - Have to have the same name as the class.

  called when a new instance is created.
- **destructors** - Have to have the same name as the class prefixed with a `~` .

  called when an instance is destroyed (goes out of scope, it's memory freed etc.).

The constructor and destructor can be `private` and `public` .

Example

```
class Animal {
public:
        // constructor
        Animal() {
                // do stuff to initialize
        }
        // destructor
        ~Animal() {
                // do stuff to uninitialize
        }
};
```

Access modifiers

- `public` : available to everyone.
- `private` : avilable only to the class.

  The default is `private` .

They can be used for a “section”, or for single variables/methods:

```
class Animal {
// "section"
private:
        String name;
        String sound;
public:
        // single variable
        private int age;

        Animal(str name, str sound, int age) {
                this.name.from(name);
                this.sound.from(sound);
                this.age=age;
        }
};
```

Inheritance

Inheritance is done by using `<`.
The class inheriting from is called the **super class** or the **parent class**, and the class inheriting from the superclass is called the **child class**.

You can access all the methods and variables in the super class (including private ones) using the `super` keyword.

Example

```
// the parent class
class Animal {
private:
        String name, sound;
public:
        Animal(str name, str sound) {
                this.name.from(name);
                this.sound.from(sound);
        }
        get_name() String {
                return this.name; // calling this method makes a copy of this.name
        }
        get_sound() String {
                return this.sound;
        }
};

// the child class
class Dog < Animal {
public:
        private String color;
        Dog(str name, str color) {
                // calling the constructor of the superclass
                super(name, "Woof!");
                this.color.from(color);
        }
        get_color() String {
                return this.color;
        }
        to_String() String {
                String s;
                s.from(super.name.to_str()); // can also use super.get_name().to_st
```

```
                s.append(" is a dog of color ");
                s.append(this.color.to_str());
                s.append(" that makes the sound ");
                s.append(super.sound.to_str())
                // s is now "<name> is a dog of color <color> that makes the sound
                // sound will always be "Woof!" in the Dog class
                // because that's what we passed to the superclass constructor in t

                return s;
        }
};
```

The child class `Dog` has access to everything in the super class `Animal` using the `super` keyword.

Templates

Templates make writing one function for many different types possible. templates work with functions, classes and methods.

The compiler creates a version of the function for every provided type.

```
// declaring a template function
// multiple types can be declared:
// fn add<type A, type B, type C>
fn add<type T> (T a, T b) T {
        return a+b;
}

// calling a template function
add<int>(1, 2); // 3
add<float>(1.5, 3.5); // 5
// the following function is generated by the compiler for the above call:
//
// fn add(float a, float b) float {
//     return a+b;
// }
```

You can make a template type only work with specific types by putting the types in parentheses after the template type declaration:

```
// again, you can declare multiple types:
// fn add<type N(int, float), type S(str, String), type U(uint, usize)>
fn add<type T(int, float)> (T a, T b) T {
        return a+b;
}

add<int>(1, 4); // works.
add<char>('a', 'b'); // doesn't work, compilation error.
```

Templates in a class:

```
// declaring a template class
class any_type<type T> {
        T value;
public:
        any_number(T value) {
                this.value=value;
        }
        set(T value) {
                this.value=value;
        }
        get() T {
                return this.value;
        }
};

// creating instances of a template class
any_type<int> integer(12);
any_type<uint> unsigned_int(3);
int i = integer.get() // 12
uint u = unsigned_int.get(); // 3
```

Dynamic memory allocation

New objects can be created in the heap using the `new` keyword and freed using the `delete` keyword:

```
// create a new String in the stack
var String *s = new String;
// you have to dereference 's' to use it, but you don't have to as the compiler doe
*s.from("Hello, World!");
// free it
delete s;

// You don't have to specify the type and make it a pointer as the compiler does it
var str_in_heap = new String;
// The compiler dereferences for you.
str_in_heap.from("Hello, World!");
delete str_in_heap;
```

`new` returns a pointer to the memory address where the allocated memory starts.
Arrays in the heap can be created using `make<type T>(usize size)`:

```
// make an int array of size 10
var array = make<int>(10);
```

A heap allocated object will be automatically freed when it goes out of scope and has no pointers or references.

Standard library

Available without importing

Functions

`print(str fmt, args...)` - Print `fmt` to standard out (`stdout`).

- Suports formatting similar to the rust `println!()` macro: `print("variable 'i' is: {}", i);`.

Smart types

`String` - A wrapper around a dynamic array of chars. this is what you have to use for mutable strings.

Methods for `String`:

- `from(str s)` - add the contents of `s` into the `String`.
- `to_str() str` - convert the contents of the `String` into a `str` and return it.
- `len() usize` return the length of the string in the `String`.
- `substr(usize start, usize end) String` - return a new `String` containing the contents between `start` and `end` in the original `String`.
- `append(str s)` - Append `s` to the `String`.
- `clone() String` - Create a copy of the `String`. equivalent of `String.substr(0, String.len())`.

  Example:

```
var String s;
s.from("Hello, World!");
s.len(); // 13
var hello = s.substr(0, 5); // Hello
hello.append(s.substr(8, 12).to_str()); // append 'World'
// hello is now 'Hello World'
var s2 = s.clone(); // s2 is 'Hello, World!'
```

`Vector<type T>` - A dynamic array that can be any type.

Methods for `Vector<type T>`:

- `push_front(T data)` - Put 'data' in the front of the array.
- `push_back(T data)` - Put 'data' at the back of the array.
- `push_to(uint index, T data)` - Put 'data' at 'index'.
- `pop_front() T` - Remove the front value.
- `pop_back() T` - Remove the back value.
- `pop_from(uint index) T` - Get the value at 'inedx' and return it.
- `size() usize` - Get the size of the array.
- `is_empty() bool` - Check if the `Vector` is empty.
- `to_array() T` - Convert the array into a regular array of type 'T' and return it.
- `dump()` - Print the entire contents of the array.

  Example:

```
Vector<int> v;
v.push_front(1);
v.dump(); // front [1] back

v.push_back(2);
v.dump(); // [1,2]

v.pop_back(); // 2
v.push_front(2);
v.dump(); // [2,1]

var int_array = v.to_array(); // [2,1]
```

`any<type T>` - A class that can hold any type (as long as it supports using the assignment (`=`) operator).

Methods for `any<type T>` :

- `set(T value)` - Set the value.
- `get() T` - Get the value (return it).
- `type() type` - Returns the type being used.