Source code

# main file (src.js)

// ╔════════════════════════════════════════════════════════╗

// ║ \_\_\_ \_\_\_\_ \_\_\_\_ \_ ║

// ║ / \_ \ / \_\_\_| \_ \ \_\_\_ \_ \_\_| | ║

// ║ | | | | | | |\_) | / \_ \ '\_\_| | ║

// ║ | |\_| | |\_\_\_| \_ < | \_\_/ | | | ║

// ║ \\_\_\_/ \\_\_\_\_|\_| \\_\ \\_\_\_|\_| |\_| ║

// ║ \_\_\_ \_ \_ ║

// ║ |\_ \_|\_ \_\_ | |\_ \_\_\_ \_ \_\_ \_ \_\_ \_ \_\_ \_\_\_| |\_ \_\_\_ \_ \_\_ ║

// ║ | || '\_ \| \_\_/ \_ \ '\_\_| '\_ \| '\_\_/ \_ \ \_\_/ \_ \ '\_\_| ║

// ║ | || | | | || \_\_/ | | |\_) | | | \_\_/ || \_\_/ | ║

// ║ |\_\_\_|\_| |\_|\\_\_\\_\_\_|\_| | .\_\_/|\_| \\_\_\_|\\_\_\\_\_\_|\_| ║

// ║ |\_| ║

// ║ made by Patrick Williams ║

// ║ for his OCR Computer Science NEA ║

// ╚════════════════════════════════════════════════════════╝

//

// main repository: https://github.com/patrickWilliams07/ocr-erl-interpreter

// nea repository: https://github.com/patrickWilliams07/ocr-erl-nea

// website URL: tbd.

////////////////

// SYMBOL TABLE

////////////////

class SymbolTable {

constructor(){

this.table = []

}

async get(token){

for (let item of this.table){ // Checks if in current table

if (item.name == token.name){

return await item.value

}

}

if (this != Evaluator.global){ // If current table isnt global, checks global

for (let item of Evaluator.global.table){

if (item.name == token.name){

return await item.value

}

}

}

return new IdentifierError(token, "was not declared") // Not defined

}

async set(token, newValue){

for (let item of this.table){ // Checks own table onlky

if (item.name == token.name){

return await item.set(token, newValue)

}

}

this.table.push(new Symbol(token)) // If doesn't exist, will create

return await this.table[this.table.length - 1].set(token, newValue)

}

push\_native\_subroutine(name, subroutine, tag=null){

let symbol = new Symbol(new Identifier(null, null, name, true))

if (tag == null){

symbol.value = new subroutine(null, null)

} else {

symbol.value = new subroutine(null, null, tag)

}

this.table.push(symbol)

}

}

class Symbol {

constructor(token){

this.name = token.name

this.value = null

this.constant = token.constant // Is a constant

this.declared = false // If it is a constant, has it been declared

}

set(token, newValue){

if (token.constant){ // Check if is now a constant

this.constant = true

this.declared = false

}

if (!this.constant){ // Normal variable

this.value = newValue

return null

}

if (this.declared){ // Constant and already defined

return new IdentifierError(token, "is a constant and has already been defined")

}

this.declared = true // Defining a constant

this.value = newValue

return null

}

}

////////////////

// TOKEN

////////////////

class Token {

constructor(position, line){

this.position = position

this.line = line

}

}

////////////////

// ARITHMETIC

////////////////

class BinaryOperator extends Token{

constructor(position, line){

super(position, line)

this.left = null

this.right = null

this.leftValue = null

this.rightValue = null

}

async evaluate(){

let leftValueHolder = await this.left.evaluate()

this.rightValue = await this.right.evaluate()

this.leftValue = leftValueHolder

if (this.contains\_type(Error)){

return this.leftValue instanceof Error ? this.leftValue : this.rightValue

}

if (this.contains\_type(Return)){

return new EvaluationError((this.leftValue instanceof Return ? this.leftValue : this.rightValue), "Cannot operate on subroutine with no return")

}

return this.calculate()

}

// Ensures the left data type is left argument, and right data type is right argument

strict\_type\_check(leftType, rightType){

if (this.leftValue instanceof leftType && this.rightValue instanceof rightType){

return true

}

return false

}

// Ensures both left and right are instances of one of the arguments each

loose\_type\_check(){

let leftDone = false

let rightDone = false

for (let type of arguments){

if (this.leftValue instanceof type){

leftDone = true

}

if (this.rightValue instanceof type){

rightDone = true

}

}

return leftDone && rightDone

}

// Ensures either left or right are instances of one of the arguments

contains\_type(){

for (let type of arguments){

if (this.leftValue instanceof type || this.rightValue instanceof type){

return true

}

}

return false

}

}

class Add extends BinaryOperator{

calculate(){

if (this.contains\_type(FloatType) && this.loose\_type\_check(FloatType, IntegerType)){ // Contains at least one float with potential integers

return new FloatType(this.position, this.line, this.get\_result()) // Definitely a float return

}

if (this.strict\_type\_check(IntegerType, IntegerType)){ // Contains two integers

return new IntegerType(this.position, this.line, this.get\_result()) // Definitely an integer return for addition

}

if (this.strict\_type\_check(StringType, StringType)){ // Contains two strings

return new StringType(this.position, this.line, this.get\_result()) // Concatenation

} // ERRORS

if (this.leftValue instanceof StringType){ // String + ?

return new TypeError(this, `Cannot concatenate type ${this.rightValue.typeAsString} with String, expected String`)

}

if (this.leftValue instanceof IntegerType || this.leftValue instanceof FloatType){ // Number + ?

return new TypeError(this, `Cannot add type ${this.rightValue.typeAsString} to ${this.leftValue.typeAsString}, expected Integer or Float`)

} // ELSE

return new TypeError(this, `Cannot combine types ${this.leftValue.typeAsString} and ${this.rightValue.typeAsString}`)

}

get\_result = () => this.leftValue.value + this.rightValue.value

}

class Minus extends BinaryOperator{

calculate(){

if (this.contains\_type(FloatType) && this.loose\_type\_check(FloatType, IntegerType)){ // Contains at least one float with potential integers

return new FloatType(this.position, this.line, this.get\_result()) // Definitely a float return

}

if (this.strict\_type\_check(IntegerType, IntegerType)){ // Contains two integers

return new IntegerType(this.position, this.line, this.get\_result()) // Definitely an integer return for addition

} // ERRORS

return new TypeError(this, `Cannot subtract type ${this.rightValue.typeAsString} from ${this.leftValue.typeAsString}, expected Integers or Floats`)

}

get\_result = () => this.leftValue.value - this.rightValue.value

}

class Multiply extends BinaryOperator{

constructor(position, line){

super(position, line)

}

calculate(){

if (this.contains\_type(FloatType) && this.loose\_type\_check(FloatType, IntegerType)){ // Contains at least one float with potential integers

return new FloatType(this.position, this.line, this.get\_result()) // Definitely a float return

}

if (this.strict\_type\_check(IntegerType, IntegerType)){ // Contains two integers

return new IntegerType(this.position, this.line, this.get\_result()) // Definitely an integer return for addition

} // ERRORS

return new TypeError(this, `Cannot mutltiply type ${this.leftValue.typeAsString} with ${this.rightValue.typeAsString}, expected Integers or Floats`)

}

get\_result = () => this.leftValue.value \* this.rightValue.value

}

class Divide extends BinaryOperator{

calculate(){

if (this.rightValue.value === 0){

return new EvaluationError(this.leftValue, "Cannot divide by zero")

}

if (this.contains\_type(FloatType) && this.loose\_type\_check(FloatType, IntegerType)){ // Contains at least one float with potential integers

return new FloatType(this.position, this.line, this.get\_result()) // Definitely a float return

}

if (this.strict\_type\_check(IntegerType, IntegerType)){ // Contains two integers

let result = this.get\_result() // Must check as can produce a float result

return String(result).includes('.') ? new FloatType(this.position, this.line, result) : new IntegerType(this.position, this.line, result)

} // ERRORS

return new TypeError(this, `Cannot divide type ${this.rightValue.typeAsString} by ${this.leftValue.typeAsString}, expected Integes or Floats`)

}

get\_result = () => this.leftValue.value / this.rightValue.value

}

class Exponent extends BinaryOperator{

calculate(){

if (isNaN(this.get\_result())){

return new EvaluationError(this.leftValue, "Cannot raise a negative number to this power")

}

if (this.contains\_type(FloatType) && this.loose\_type\_check(FloatType, IntegerType)){ // Contains at least one float with potential integers

return new FloatType(this.position, this.line, this.get\_result()) // Definitely a float return

}

if (this.strict\_type\_check(IntegerType, IntegerType)){ // Contains two integers

let result = this.get\_result() // Must check as can produce a float result

return String(result).includes('.') ? new FloatType(this.position, this.line, result) : new IntegerType(this.position, this.line, result)

} // ERRORS

return new TypeError(this, `Cannot divide type ${this.rightValue.typeAsString} by ${this.leftValue.typeAsString}, expected Integers or Floats`)

}

get\_result = () => this.leftValue.value \*\* this.rightValue.value

}

class Modulus extends BinaryOperator{

calculate(){

if (this.rightValue.value === 0){

return new EvaluationError(this.leftValue, "Cannot take moduluo by zero")

}

if (this.contains\_type(FloatType) && this.loose\_type\_check(FloatType, IntegerType)){ // Contains at least one float with potential integers

return new FloatType(this.position, this.line, this.get\_result()) // Definitely a float return

}

if (this.strict\_type\_check(IntegerType, IntegerType)){ // Contains two integers

return new IntegerType(this.position, this.line, this.get\_result()) // Definitely an integer return

} // ERRORS

return new TypeError(this, `Cannot take modulus of type ${this.leftValue.typeAsString} modulo ${this.rightValue.typeAsString}, expected Integers or Floats`)

}

get\_result = () => this.leftValue.value % this.rightValue.value

}

class Quotient extends BinaryOperator{

calculate(){

if (this.rightValue.value === 0){

return new EvaluationError(this.leftValue, "Cannot do quotient division by zero")

}

if (this.contains\_type(FloatType) && this.loose\_type\_check(FloatType, IntegerType)){ // Contains at least one float with potential integers

return new FloatType(this.position, this.line, this.get\_result()) // Definitely a float return

}

if (this.strict\_type\_check(IntegerType, IntegerType)){ // Contains two integers

return new IntegerType(this.position, this.line, this.get\_result()) // Definitely an integer return

} // ERRORS

return new TypeError(this, `Cannot take the quotient of type ${this.leftValue.typeAsString} and ${this.rightValue.typeAsString}, expected Integers or Floats`)

}

get\_result = () => Math.floor(this.leftValue.value / this.rightValue.value)

}

////////////////

// EQUALS

////////////////

class Equals extends BinaryOperator{

async evaluate(){

this.rightValue = await this.right.evaluate()

if (this.leftValue instanceof Error){

return this.leftValue

}

if (this.rightValue instanceof Error){

return this.rightValue

}

return await this.left.set(this.rightValue)

}

}

////////////////

// LOGICAL

////////////////

class And extends BinaryOperator{

calculate(){

if (this.strict\_type\_check(BooleanType, BooleanType)){ // Contains two Booleans

return new BooleanType(this.position, this.line, this.get\_result()) // Expected result

} // ERRORS

return new TypeError(this, `Cannot use AND on type ${this.rightValue.typeAsString} with ${this.leftValue.typeAsString}, expected Booleans`)

}

get\_result = () => this.leftValue.value && this.rightValue.value

}

class Or extends BinaryOperator{

calculate(){

if (this.strict\_type\_check(BooleanType, BooleanType)){ // Contains two Booleans

return new BooleanType(this.position, this.line, this.get\_result()) // Expected result

} // ERRORS

return new TypeError(this, `Cannot use OR on type ${this.rightValue.typeAsString} with ${this.leftValue.typeAsString}, expected Booleans`)

}

get\_result = () => this.leftValue.value || this.rightValue.value

}

class Not extends Token{

constructor(position, line){

super(position, line)

this.child = null

}

async evaluate(){

let childValue = await this.child.evaluate()

if (childValue instanceof Error){

return childValue

}

if (childValue instanceof BooleanType){

return new BooleanType(this.position, this.line, !childValue.value)

}

return new TypeError(this, `Cannot use NOT on type ${childValue.typeAsString}, expected Boolean`)

}

}

class ComparisonOperator extends BinaryOperator{

constructor(position, line, tag){

super(position, line)

this.tag = tag

}

calculate(){

if (this.loose\_type\_check(IntegerType, FloatType) || this.strict\_type\_check(StringType, StringType)){ // Contains two Booleans

return new BooleanType(this.position, this.line, this.get\_result()) // Expected result

}

if (this.strict\_type\_check(BooleanType, BooleanType)){

if (this.tag == "==" || this.tag == "!="){

return new BooleanType(this.position, this.line, this.get\_result())

}

return new TypeError(this, `Cannot compare two Booleans with comparator '${this.tag}', only '==' or '!='`)

}

// ERRORS

return new TypeError(this, `Cannot compare type ${this.rightValue.typeAsString} against ${this.leftValue.typeAsString}`)

}

get\_result(){

switch (this.tag){

case "==":

return this.leftValue.value == this.rightValue.value

case ">":

return this.leftValue.value > this.rightValue.value

case ">=":

return this.leftValue.value >= this.rightValue.value

case "<":

return this.leftValue.value < this.rightValue.value

case "<=":

return this.leftValue.value <= this.rightValue.value

case "!=":

return this.leftValue.value != this.rightValue.value

}

}

}

////////////////

// DATA TYPES

////////////////

class DataType extends Token{

constructor(position, line, value){

super(position, line)

this.value = value

}

evaluate() {

return this

}

}

class IntegerType extends DataType{

get typeAsString(){

return "Integer"

}

display(){

return String(this.value)

}

cast\_to\_type(type){ // FROM INTEGERS

switch (type){

case IntegerType:

return this

case FloatType:

return new FloatType(this.position, this.line, this.value)

case BooleanType: // true when not 0, false when 0

return new BooleanType(this.position, this.line, this.value != 0)

case StringType:

return new StringType(this.position, this.line, this.display())

}

}

}

class FloatType extends DataType{

get typeAsString(){

return "Float"

}

display(){

if (String(this.value).includes('.')){

return String(this.value)

}

return String(this.value) + ".0"

}

cast\_to\_type(type){ // FROM FLOAT

switch (type){

case IntegerType:

return new IntegerType(this.position, this.line, Math.floor(this.value))

case FloatType:

return this

case BooleanType: // true when not 0, false when 0

return new BooleanType(this.position, this.line, this.value != 0)

case StringType:

return new StringType(this.position, this.line, this.display())

}

}

}

class BooleanType extends DataType{

get typeAsString(){

return "Boolean"

}

display(){

return this.value ? "True" : "False"

}

cast\_to\_type(type){ // FROM BOOLEAN

switch (type){

case IntegerType:

case FloatType: // 1 when true, 0 when false

return new type(this.position, this.line, this.value ? 1 : 0)

case BooleanType:

return this

case StringType:

return new StringType(this.position, this.line, this.display())

}

}

}

class StringType extends DataType{

get typeAsString(){

return "String"

}

display(){

return this.value

}

cast\_to\_type(type){ // FROM STRING

switch (type){

case IntegerType: // Can be converted to a number, no full stops

if (isNaN(Number(this.value)) || this.value.includes('.')){

return new TypeError(this, `Cannot cast ${this.value} to type Integer`)

}

return new IntegerType(this.position, this.line, Number(this.value))

case FloatType: // Can be converted to a number

if (isNaN(Number(this.value))){

return new TypeError(this, `Cannot cast ${this.value} to type Float`)

}

return new FloatType(this.position, this.line, Number(this.value))

case BooleanType: // "True" or "False" are accepted, rest are errors

switch (this.value){

case "True":

return new BooleanType(this.position, this.line, true)

case "False":

return new BooleanType(this.position, this.line, false)

default:

return new TypeError(this, `Cannot cast ${this.value} to type Float, expected "True" or "False"`)

}

case StringType:

return this

}

}

}

////////////////

// ARRAYS

////////////////

class ArrayCall extends Token{

constructor(position, line){

super(position, line)

this.callee = null

this.indexes = []

}

async evaluate(ignoreLastIndex=false){

let callee = await this.callee.evaluate() // Ensures callee is valid datatpye

if (callee instanceof Error){

return callee

}

if (!(callee instanceof ArrayType)){

return new TypeError(this, `Can only access value from type Array, not ${callee.typeAsString}`)

}

let evaluatedIndexes = []

for (let i = 0; i < this.indexes.length - (ignoreLastIndex ? 1 : 0); i++){ // converts ASTS to values

let result = await this.indexes[i].evaluate()

if (result instanceof Error){

return result

}

evaluatedIndexes.push(result)

} // returns array of data types

return await callee.get\_index(evaluatedIndexes)

}

async set(newValue){

let outerArray = await this.evaluate(true)

if (outerArray instanceof Error){

return outerArray

}

if (!(outerArray instanceof ArrayType)){

return new EvaluationError(outerArray, "Array does not have this many dimensions")

}

let finalIndex = await this.indexes[this.indexes.length-1].evaluate()

if (finalIndex instanceof Error){

return finalIndex

}

if (!(finalIndex instanceof IntegerType)){ // index must be integer

return new SyntaxError(finalIndex, "Integer value required as index")

}

if (finalIndex.value < 0 || finalIndex.value >= outerArray.arrayLength) { // ensure in range

return new EvaluationError(finalIndex, `Array index not in range from 0 to ${outerArray.arrayLength - 1}`)

}

if (outerArray.type == null){

outerArray.type = newValue.typeAsString

}

else if (newValue.typeAsString != outerArray.type){

return new TypeError(newValue, `Cannot add item of type ${newValue.typeAsString} to array of ${outerArray.type}`)

}

outerArray.contents[finalIndex.value] = newValue

return null

}

}

class ArrayType extends Token {

static nullValue = {typeAsString : "EmptyArrayValue"}

constructor(position, line){

super(position, line)

this.creationAsts = [] // the initial asts to define the array in the line of code

this.isEmptyCreation = true // determines wether the array is predefined or not

this.contents = [] // the actual values of the array which is dynamically updated

this.type = null // the data type of the array

this.arrayLength = null

this.identifier = null

}

get typeAsString(){

return "Array"

}

display(){

console.log("printed")

let displayed = []

for (let item of this.contents){

if (item == ArrayType.nullValue){

displayed.push("<Empty>")

} else if (item instanceof StringType){

displayed.push(`"${item.display()}"`)

} else {

displayed.push(item.display())

}

}

if (this.type == "Array"){

return `[ ${displayed.join(",\n ")} ]`

}

return `[ ${displayed.join(", ")} ]`

}

async get\_index(indexes){

if (indexes.length == 0){

return this

}

let index = indexes[0]

if (!(index instanceof IntegerType)){ // index must be integer

return new SyntaxError(index, "Integer value required as index")

}

indexes.shift() // remove current index from index dimensions

if (index.value < 0 || index.value >= this.arrayLength) { // ensure in range

return new EvaluationError(index, `Array index not in range from 0 to ${this.arrayLength - 1}`)

}

if (indexes.length == 0){ // If this is current dimensions

return this.contents[index.value]

}

if (this.type != "Array"){ // Only do further dimensions if array

return new EvaluationError(indexes[0], "Array does not have this many dimensions")

}

return await this.contents[index.value].get\_index(indexes)

}

async evaluate(){

let result = this.isEmptyCreation ? await this.evaluate\_empty\_array() : await this.evaluate\_defined\_array()

if (result instanceof Error){

return result

}

if (this.identifier != null) {

await this.identifier.set(this)

}

return this

}

async evaluate\_defined\_array(){ // ensuring identifier is assigned with values

let arrayLengths = null // for 2d arrays only

this.contents = []

for (let item of this.creationAsts){

let result = await item.evaluate()

if (result instanceof Error){

return result

}

if (this.type == null){

this.type = item.typeAsString

} else if (this.type != item.typeAsString){ // not aligning types

return new TypeError(item, "Array can only be defined with a single data type")

}

if (item instanceof ArrayType){ // only if 2d array

if (arrayLengths == null){

arrayLengths = item.arrayLength

} else if (arrayLengths != item.arrayLength) { // lengths dont match up

return new EvaluationError(item, "All sub-arrays must be the same length")

}

}

this.contents.push(result)

}

}

async evaluate\_empty\_array(){

let currentLength = await this.creationAsts[0].evaluate() // this index

if (currentLength instanceof Error){

return currentLength

}

if (!(currentLength instanceof IntegerType)){ // ensure is integer

return new TypeError(currentLength, `Array length must be of type Integer, not ${currentLength.typeAsString}`)

}

this.arrayLength = currentLength.value

if (this.creationAsts.length == 1){ // if last dimension, fill with null

this.contents = Array(currentLength.value).fill(ArrayType.nullValue) // empty item

return

}

this.type = this.typeAsString // set type to array

for (let i = 0; i < currentLength.value; i++){ // next dimension

let newArray = new ArrayType(currentLength.position, currentLength.line)

newArray.creationAsts = this.creationAsts.slice(1) // ignores first index

let result = await newArray.evaluate\_empty\_array()

if (result instanceof Error){

return result

}

this.contents.push(newArray) // creates own contents

}

console.log("built")

}

}

////////////////

// VARIABLES AND FUNCITONS

////////////////

class Identifier extends Token{

constructor(position, line, name, constant=false){

super(position, line)

this.name = name

this.constant = constant

this.global = false

}

async evaluate(){

return await Evaluator.currentScope.get(this)

}

async set(newValue){

if (this.global){

return await Evaluator.global.set(this, newValue)

}

return await Evaluator.currentScope.set(this, newValue)

}

}

class Call extends Token{

constructor(position, line){

super(position, line)

this.callee = null

this.argumentsAsts = []

}

async evaluate(){

if (this.callee instanceof Property){ // if a property

return this.callee.call(this) // calls without evaluation

}

let callee = await this.callee.evaluate() // Ensures callee is valid datatpye

if (callee instanceof Subroutine){

return await callee.call(this)

}

if (callee instanceof Error){

return callee

}

if (callee instanceof DataType){

return new TypeError(this.callee, `Type ${callee.typeAsString} cannot be called`)

}

return new TypeError(this.callee, "Cannot call this")

}

}

class Subroutine extends Token {

static nullReturn = {typeAsString : "an empty Subroutine Return value"}

get typeAsString(){

return "Subroutine"

}

}

class UserDefinedSubroutine extends Subroutine {

static callStackSize = 0

constructor(position, line, tag){

super(position, line)

this.tag = tag

this.parameters = []

this.contents = []

this.identifier = null

}

display(){

return `<subroutine: ${this.identifier.name}>`

}

async call(call){

if (UserDefinedSubroutine.callStackSize >= 1500){

return new EvaluationError(call, "Call stack exceeded maximum size of 1500")

}

UserDefinedSubroutine.callStackSize++

if (this.parameters.length != call.argumentsAsts.length){

return new EvaluationError(call, `Subroutine expected ${this.parameters.length} arguments, ${call.argumentsAsts.length} given`)

}

let argumentsValues = []

for (let i = 0; i < call.argumentsAsts.length; i++){ // First get argument values in old scope

let result = await call.argumentsAsts[i].evaluate()

if (result instanceof Error){

return result

}

argumentsValues.push(result)

}

let previousScope = Evaluator.currentScope // Then change

Evaluator.currentScope = new SymbolTable()

for (let i = 0; i < argumentsValues.length; i++){ // Then sets parmaters in new scope

await this.parameters[i].set(argumentsValues[i])

}

let result = await new Evaluator().evaluate\_many\_asts(this.contents)

if (result instanceof Error){

return result

}

let returnValue = Subroutine.nullReturn

if (result instanceof Return){

if (result.child != null){

returnValue = await result.child.evaluate()

}

}

Evaluator.currentScope = previousScope

UserDefinedSubroutine.callStackSize--

return returnValue

}

async evaluate(){

await this.identifier.set(this)

}

}

////////////////

// NATIVE FUNCTIONS

////////////////

class Print extends Subroutine {

display(){

return "<native subroutine: print>"

}

async call(call){

let output = [] // output

if (call.argumentsAsts.length == 0){ // Ensure 1 or more parameters

return new EvaluationError(call, "print expected 1 or more arguments, 0 given")

}

for (let item of call.argumentsAsts){

let result = await item.evaluate()

if (result instanceof Error){

return result

}

if (!(result instanceof DataType || result instanceof ArrayType ||

result instanceof Subroutine)){

return new EvaluationError(item, `Can only print type String, not ${result.typeAsString}`)

}

output.push(result.display())

}

outputPrint(output.join(' '))

await wait()

return Subroutine.nullReturn

}

}

class Input extends Subroutine {

static enterPressed = false

display(){

return "<native subroutine: input>"

}

async call(call){

if (call.argumentsAsts.length > 1){ // Ensure 1 parameter

return new EvaluationError(call, `input expected 0 or 1 arguments, ${call.argumentsAsts.length} given`)

}

let message = ""

if (call.argumentsAsts.length == 1){

let result = await call.argumentsAsts[0].evaluate()

if (result instanceof Error){

return result

}

if (!(result instanceof StringType)){

return new EvaluationError(result, `Can only output type String, not ${result.typeAsString}`)

}

message = result.display() // Outputs input message

}

outputPrint(message, false) // displays message

let before = $("#output").val() // records contents of textarea before

Input.enterPressed = false // resets last key in case previously enter was entered

$("#output").attr("readonly", false) // allows typinh

$("#output").focus() // focuses cursor on window

while (!Input.enterPressed && Evaluator.active) { // waits for enter key

let length = $("#output").val().length // current length of contents

if (length < before.length){ // if something was deleted

$("#output").val(before) // revert

}

if ($("#output").prop("selectionStart") < before.length){ // if cursor moved

$("#output").prop("selectionStart", before.length+1) // move back

}

await wait() // wait

}

$("#output").attr("readonly", true) // no longer can type

if (!Evaluator.active){

if ($("#output").val().length > 0){

outputPrint('', true) // newline for abort message

}

return new Abort()

}

let after = $("#output").val() // value afterwards to get contents

return new StringType(call.position, call.line, after.slice(before.length, -1))

}

}

class Random extends Subroutine {

display(){

return "<native subroutine: random>"

}

async call(call){

if (call.argumentsAsts.length != 2){

return new EvaluationError(call, `random expected 2 arguments, ${call.argumentsAsts.length} given`)

}

let min = await call.argumentsAsts[0].evaluate()

if (min instanceof Error){

return min

}

let max = await call.argumentsAsts[1].evaluate()

if (max instanceof Error){

return max

}

if (min instanceof IntegerType && max instanceof IntegerType){

return new IntegerType(call.postition, call.line, Math.floor(Math.random() \* (max.value - min.value + 1) + min.value))

}

if (min instanceof FloatType && max instanceof FloatType){

return new FloatType(call.position, call.line, Math.random() \* (max.value - min.value) + min.value)

}

return new TypeError(call, `Cannot use random on type ${min.typeAsString} with ${max.typeAsString}, expected two Integers or two Float`)

}

}

class TypeCast extends Subroutine {

constructor(position, line, tag){

super(position,line)

this.tag = tag

}

display(){

switch (this.tag){

case IntegerType:

return "<native subroutine: int>"

case FloatType:

return "<native subroutine: float>"

case BooleanType:

return "<native subroutine: bool>"

case StringType:

return "<native subroutine: str>"

}

}

async call(call){

if (call.argumentsAsts.length != 1){

return new EvaluationError(call, `${this.tag} expected 1 argument, ${call.argumentsAsts.length} given`)

}

let old = await call.argumentsAsts[0].evaluate()

if (old instanceof Error){

return old

}

return await old.cast\_to\_type(this.tag)

}

}

class Asc extends Subroutine {

display(){

return "<native subroutine: ASC>"

}

async call(call){

if (call.argumentsAsts.length != 1){

return new EvaluationError(call, `asc expected 1 argument, ${call.argumentsAsts.length} given`)

}

let character = await call.argumentsAsts[0].evaluate()

if (character instanceof Error){

return character

}

if (!(character instanceof StringType)){

return new TypeError(character, `Expected type String, not type ${character.typeAsString}`)

}

if (character.value.length != 1){

return new TypeError(character, "Expexted single character, as string of length 1")

}

return new IntegerType(call.position, call.line, character.value.charCodeAt(0))

}

}

class Chr extends Subroutine {

display(){

return "<native subroutine: CHR>"

}

async call(call){

if (call.argumentsAsts.length != 1){

return new EvaluationError(call, `chr expected 1 argument, ${call.argumentsAsts.length} given`)

}

let code = await call.argumentsAsts[0].evaluate()

if (!(code instanceof IntegerType)){

return new TypeError(code, `Expected type Integer, not type ${code.typeAsString}`)

}

return new StringType(call.position, call.line, String.fromCharCode(code.value))

}

}

////////////////

// RETURN AND TEMPLATE

////////////////

class Return extends Token {

constructor(position, line){

super(position, line)

this.child = null

}

async evaluate(){

if (this.child == null){

return null

}

return await this.child.evaluate()

}

}

// For generic keywords like const, global which don't need a unique object

class TemplateKeyword extends Token{

constructor(position, line, tag){

super(position, line)

this.tag = tag

}

}

////////////////

// IF AND SWITCH

////////////////

class IfStatement extends Token{

constructor(position, line){

super(position, line)

this.cases = []

this.elseCase = null

}

async evaluate(){

for (let ifCase of this.cases){

let result = await ifCase.condition.evaluate()

if (result instanceof Error){

return result

}

if (!(result instanceof BooleanType)){

return new TypeError(result, `Condition must be type Boolean, not ${result.typeAsString}`)

}

if (result.value === true){

return ifCase.contents

}

}

if (this.elseCase != null){

return this.elseCase.contents

}

return null

}

}

class SwitchStatement extends Token{

constructor(position, line){

super(position, line)

this.comparison = null

this.cases = []

this.elseCase = null

}

async evaluate(){

let comparison = await this.comparison.evaluate()

let comparisonTypeAsString = comparison.typeAsString

if (comparison instanceof Error){

return comparison

}

for (let switchCase of this.cases){

let result = await switchCase.condition.evaluate()

if (result instanceof Error){

return result

}

if (comparisonTypeAsString == result.typeAsString && comparison.value == result.value){

return switchCase.contents

}

}

if (this.elseCase != null){

return this.elseCase.contents

}

return null

}

}

class Case extends Token{

constructor(position, line){

super(position, line)

this.condition = null

this.contents = []

}

}

class ElseCase extends Token{

constructor(position, line){

super(position, line)

this.contents = []

}

}

////////////////

// LOOPS

////////////////

class Loop extends Token{

constructor(position, line){

super(position, line)

this.contents = []

}

evaluate(){

return this.contents

}

}

class While extends Loop{

constructor(position, line){

super(position, line)

this.condition = null

}

async evaluate\_condition(){

let condition = await this.condition.evaluate()

if (condition instanceof Error){

return condition

}

if (!(condition instanceof BooleanType)){

return new TypeError(condition, `Condition must be type Boolean, not ${condition.typeAsString}`)

}

return condition.value

}

reset = () => null

}

class Do extends Loop{

constructor(position, line){

super(position, line)

this.condition = null

this.firstPassComplete = false

}

async evaluate\_condition(){

if (!this.firstPassComplete){

this.firstPassComplete = true

return true

}

let condition = await this.condition.evaluate()

if (condition instanceof Error){

return condition

}

if (!(condition instanceof BooleanType)){

return new TypeError(condition, `Condition must be type Boolean, not ${condition.typeAsString}`)

}

return !condition.value

}

reset(){

this.firstPassComplete = false

}

}

class For extends Loop{

constructor(position, line){

super(position, line)

this.firstPassComplete = false

this.variable = null

this.variableValue = null

this.assignment = null

this.finish = null

this.finishValue = null

this.step = null

this.stepValue = null

this.increasing = null

}

async first\_pass(){

let assignment = await this.assignment.evaluate()

if (assignment instanceof Error){

return assignment

}

let result = await this.variable.evaluate()

if (!(result instanceof IntegerType)){

return new TypeError(result, "Starting value is not an Integer")

}

this.variableValue = new IntegerType(this.variable.position, this.variable.line, result.value)

this.finishValue = await this.finish.evaluate()

if (this.finishValue instanceof Error){

return this.finishValue

}

if (!(this.finishValue instanceof IntegerType)){

return new TypeError(this.finishValue, "Final value is not an Integer")

}

this.stepValue = await this.step.evaluate()

if (this.stepValue instanceof Error){

return this.stepValue

}

if (!(this.stepValue instanceof IntegerType)){

return new TypeError(this.stepValue, "Step value is not an Integer")

}

if (this.variableValue.value <= this.finishValue.value && this.stepValue.value > 0){

this.increasing = true

} else if (this.variableValue.value >= this.finishValue.value && this.stepValue.value < 0){

this.increasing = false

} else if (this.stepValue.value == 0){

return new EvaluationError(this.step, "Step must have non-zero value")

} else {

return new EvaluationError(this.step, "Step value must align with bounds of for loop")

}

this.firstPassComplete = true

}

async evaluate\_condition(){

let newValue

if (this.firstPassComplete){ //increasing value

newValue = this.variableValue.value + this.stepValue.value

} else {

let result = await this.first\_pass()

if (result instanceof Error){

return result

}

newValue = this.variableValue.value

}

let condition = this.increasing ? newValue <= this.finishValue.value : newValue >= this.finishValue.value

if (condition){ // only change identifier if another loop will occur

this.variableValue.value = newValue

await this.variable.set(this.variableValue)

}

return condition

}

reset(){

this.firstPassComplete = false

}

}

////////////////

// PROPERTIES

////////////////

class Property extends Token {

constructor(position, line){

super(position, line)

this.callee = null

}

evaluate(){

return new EvaluationError(this, "This method should be called using '()'")

}

call(call){

return new EvaluationError(call, "This property should not be called and does not require '()'")

}

}

class Length extends Property {

async evaluate(){

let callee = await this.callee.evaluate()

if (callee instanceof Error){

return callee

}

if (callee instanceof StringType){

return new IntegerType(callee.position, callee.line, callee.value.length)

}

if (callee instanceof ArrayType){

return new IntegerType(callee.position, callee.line, callee.arrayLength)

}

return new TypeError(callee, `Type ${callee.typeAsString} has no property length`)

}

}

class Substring extends Property {

async call(call){

let string = await this.callee.evaluate() // evaluates what it is a property of

if (string instanceof Error){

return string

}

if (!(string instanceof StringType)){ // ensures it is a string

return new TypeError(string, `Type ${string.typeAsString} has no substring method`)

}

if (string.value.length == 0){

return new EvaluationError(string, "Cannot take substring of an empty string")

}

if (call.argumentsAsts.length != 2){ // Ensure 1 or more parameters

return new EvaluationError(call, `Substring expected 2 arguments, ${call.argumentsAsts.length} given`)

}

let index = await call.argumentsAsts[0].evaluate() // first argument

if (index instanceof Error){

return index

}

if (!(index instanceof IntegerType)){

return new TypeError(index, `Index argument must be type Integer, not ${index.typeAsString}`)

}

let length = await call.argumentsAsts[1].evaluate() // second argument

if (length instanceof Error){

return length

}

if (!(length instanceof IntegerType)){

return new TypeError(length, `length argument must be type Integer, not ${length.typeAsString}`)

}

if (index.value < 0 || index.value >= string.value.length){ // index must be in range of string

return new EvaluationError(index, `Value of index must be in range of string, from 0 to ${string.value.length-1}`)

}

if (length.value <= 0){ // length must be more than 1

return new EvaluationError(length, "Length must be 1 or greater")

}

if (index.value + length.value > string.value.length){ // length must be compatible with the index

return new EvaluationError(length, `Substring must be in range of string.\nFor position ${index.value}, substring length must be between 1 and ${string.value.length - index.value}`)

}

return new StringType(this.position, this.line, string.value.substring(index.value, index.value + length.value))

}

}

class LeftOrRight extends Property {

constructor(position, line, tag){

super(position, line)

this.tag = tag

}

async call(call){

let string = await this.callee.evaluate() // evaluates what it is a property of

if (string instanceof Error){

return string

}

if (!(string instanceof StringType)){ // ensures it is a string

return new TypeError(string, `Cannot use ${this.tag} on type ${string.typeAsString}`)

}

if (call.argumentsAsts.length != 1){ // Ensure 1 or more parameters

return new EvaluationError(call, `${this.tag} expected 1 argument, ${call.argumentsAsts.length} given`)

}

let length = await call.argumentsAsts[0].evaluate() // first argument

if (length instanceof Error){

return length

}

if (!(length instanceof IntegerType)){

return new TypeError(length, `Argument must be type Integer, not ${length.typeAsString}`)

}

if (length.value > string.value.length){ // cannot be longer than list length

return new EvaluationError(length, "Length of new string must be less than or equal to the old string's length ")

}

if (length.value <= 0){

return new EvaluationError(length, "Length of new string must be 1 or greater")

}

if (this.tag == "left"){ // LEFT

return new StringType(this.position, this.line, string.value.substring(0, length.value))

} // RIGHT

return new StringType(this.position, this.line, string.value.substring(string.value.length - length.value))

}

}

class UpperOrLower extends Property {

constructor(position, line, tag){

super(position, line)

this.tag = tag

}

async evaluate(){

let string = await this.callee.evaluate()

if (string instanceof Error){ // check for error

return string

}

if (!(string instanceof StringType)){ // must be a string

return new TypeError(callee, `Type ${callee.typeAsString} has no property ${this.tag}`)

} // returning new value

return new StringType(this.position, this.line, this.tag == "upper" ? string.value.toUpperCase() : string.value.toLowerCase())

}

}

////////////////

// FILES

////////////////

class FileStorage {

constructor(){

this.files = []

}

get length(){

return this.w

}

get(fileName) { // returns file based on name

for (let file of this.files){

if (file.name == fileName) {

return file

}

} // null if doesnt exist

return null

}

addNew(fileName) {

let file = this.get(fileName)

if (file == null){ // if it doesnt exist, creates

$(`<div id="${(fileName)}">📎 ${fileName}</div>`).insertBefore( "#add" )

this.files.push(new FileItem(fileName))

}

else { // otherwise it will reset the current file

file.reset()

}

}

getNewName(fileName){

if (this.get(fileName) == null){ // name is not taken

return fileName

}

let i = 1 // iterates until avaliable name is free

let parts = fileName.split('.')

while (this.get(`${parts[0]} (${i}).${parts[1]}`) != null){

i++

}

return `${parts[0]} (${i}).${parts[1]}`

}

forceAddNew(fileName, contents="") {

let name = this.getNewName(fileName)

this.files.push(new FileItem(name, contents))

return name

}

rename(oldName, newName){

let file = this.get(oldName) // get file to rename

let avaliable = this.getNewName(newName)

file.name = avaliable // updates name

return avaliable

}

storeAll() { // updates all the stores

for (let file of this.files){

file.storedContents = file.contents

}

}

delete(fileName){

this.files = this.files.filter( (file) => {

file.name != fileName

})

}

import(fileName) { // only used manually

this.files.push(new FileItem(fileName, fs.readFileSync(fileName, 'utf8')))

}

export(fileName) { // only used manually

fs.writeFileSync(fileName, this.get(fileName).contents)

}

}

class FileItem {

constructor(name, contents=""){

this.name = name

this.contents = contents

this.storedContents = ""

}

reset(){ // resets the contents

this.contents = ""

}

write(toWrite){

this.contents += toWrite

}

rewrite(toWrite){

this.contents = toWrite

}

writeStored(){

this.contents = this.storedContents

}

}

class FileHandler extends Token {

constructor(position, line, file){

super(position, line)

this.file = file

this.contents = file.contents.split('\n')

this.position = -1

this.closed = false

}

get typeAsString(){

return "File"

}

endOfFile(){

return this.position >= this.contents.length -1

}

readLine(){

this.position += 1

if (this.position >= this.contents.length) {

return null

}

return this.contents[this.position]

}

write(toWrite){

this.file.write('\n' + toWrite.replaceAll("\\n", "\n")) // replaces any \ns properly

this.contents = this.file.contents.split('\n')

}

}

class Open extends Subroutine {

async call(call){

if (call.argumentsAsts.length != 1){ // one argument only

return new EvaluationError(call, `open expected 1 argument, ${call.argumentsAsts.length} given`)

}

let fileName = await call.argumentsAsts[0].evaluate()

if (fileName instanceof Error){

return fileName

}

if (!(fileName instanceof StringType)){ // must be string

return new EvaluationError(call, `Expected file name to be type String, not ${fileName.typeAsString}`)

}

let file = Evaluator.files.get(fileName.value) // returns the file

if (file == null) { // does not exist

return new EvaluationError(call, `A file named ${fileName.value} does not exist`)

} // creates new filehandler

return new FileHandler(this.position, this.line, file)

}

}

class NewFile extends Subroutine {

async call(call){

if (call.argumentsAsts.length != 1){

return new EvaluationError(call, `newFile expected 1 argument, ${call.argumentsAsts.length} given`)

}

let fileName = await call.argumentsAsts[0].evaluate() // filename is string value

if (fileName instanceof Error){

return fileName

}

if (!(fileName instanceof StringType)){

return new EvaluationError(call, `Expected file name to be type String, not ${fileName.typeAsString}`)

} // creates new file, addNew manages if it resets the file

if (!validateFileName(fileName.value)){

return new EvaluationError(fileName, "Can only create files with '.txt' or '.csv' extentions, containing a single '.'")

}

Evaluator.files.addNew(fileName.value)

}

}

class ReadLine extends Property {

async call(call){

let file = await this.callee.evaluate() // gets file

if (file instanceof Error){

return file

}

if (!(file instanceof FileHandler)){ // ensures it is a file

return new TypeError(file, `Type ${file.typeAsString} has no readLine method, expected File`)

}

if (file.closed){

return new EvaluationError(this, "File has been closed and can no longer be read")

}

if (call.argumentsAsts.length != 0){ // Ensure no argyments

return new EvaluationError(call, `readLine expected 0 arguments, ${call.argumentsAsts.length} given`)

}

let nextLine = file.readLine()

if (nextLine == null){

return new EvaluationError(this, "Cannot read new line, end of file reached")

}

return new StringType(this.position, this.line, nextLine)

}

}

class EndOfFile extends Property {

async call(call){

let file = await this.callee.evaluate() // gets file

if (file instanceof Error){

return file

}

if (!(file instanceof FileHandler)){ // ensures it is a file

return new TypeError(file, `Type ${file.typeAsString} has no endOfFile method, expected File`)

}

if (file.closed){

return new EvaluationError(this, "File has been closed")

}

if (call.argumentsAsts.length != 0){ // Ensure no arguments

return new EvaluationError(call, `endOfFile expected 0 arguments, ${call.argumentsAsts.length} given`)

}

return new BooleanType(this.position, this.line, file.endOfFile())

}

}

class WriteLine extends Property {

async call(call){

let file = await this.callee.evaluate() // gets file

if (file instanceof Error){

return file

}

if (!(file instanceof FileHandler)){ // ensures it is a file

return new TypeError(file, `Type ${file.typeAsString} has no readLine method, expected File`)

}

if (file.closed){

return new EvaluationError(this, "File has been closed and can no longer be written to")

}

if (call.argumentsAsts.length != 1){ // needs argument of what to write

return new EvaluationError(call, `writeLine expected 1 argument, ${call.argumentsAsts.length} given`)

}

let toWrite = await call.argumentsAsts[0].evaluate()

if (toWrite instanceof Error){

return toWrite

}

if (!(toWrite instanceof StringType)){

return new TypeError(toWrite, `Can only write type String to file, not ${toWrite.typeAsString}`)

}

file.write(toWrite.value)

return Subroutine.nullReturn

}

}

class Close extends Property {

async call(call){

let file = await this.callee.evaluate() // gets file

if (file instanceof Error){

return file

}

if (!(file instanceof FileHandler)){ // ensures it is a file

return new TypeError(file, `Type ${file.typeAsString} has no endOfFile method, expected File`)

}

if (file.closed){ // already closed

return new EvaluationError(this, "File is already closed")

}

if (call.argumentsAsts.length != 0){ // Ensure no arguments

return new EvaluationError(call, `close expected 0 arguments, ${call.argumentsAsts.length} given`)

}

file.closed = true

file.contents = []

return Subroutine.nullReturn

}

}

////////////////

// ERRORS

////////////////

class Error {

constructor(position, line){

this.position = position

this.line = line + 1

this.text = Interpreter.currentText[line]

}

location(){

return `${this.text}\n${' '.repeat(this.position)}^`

}

}

class LexicalError extends Error {

constructor(position, line, description='') {

super(position, line)

this.description = description

}

display(){

return ` 🚨 ERROR @line ${this.line}\nLexical Error: ${this.description}\n${this.location()}`

}

}

class SyntaxError extends Error {

constructor(token, description='') {

super(token.position, token.line)

this.token = token

this.description = description

}

display(){

return ` 🚨 ERROR @line ${this.line}\nInvalid Syntax: ${this.description}\n${this.location()}`

}

}

class EvaluationError extends Error {

constructor(token, description='') {

super(token.position, token.line)

this.token = token

this.description = description

}

display(){

return ` 🚨 ERROR @line ${this.line}\nEvaluation Error: ${this.description}\n${this.location()}`

}

}

class IdentifierError extends Error{

constructor(token, description='') {

super(token.position, token.line)

this.token = token

this.description = description

}

display(){

return ` 🚨 ERROR @line ${this.line}\nIdentifier Error: '${this.token.name}' ${this.description}\n${this.location()}`

}

}

class TypeError extends Error{

constructor(token, description='') {

super(token.position, token.line)

this.token = token

this.description = description

}

display(){

return ` 🚨 ERROR @line ${this.line}\nType Error: ${this.description}\n${this.location()}`

}

}

class Abort extends Error{

constructor(){

super(null, null)

}

display(){

return " 🚨 Program aborted by user"

}

}

////////////////

// LEXER

////////////////

class Lexer {

static DIGITS = [..."0123456789"]

static LETTERS = [..."qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM"]

constructor(program){

this.allPlaintext = program

this.line = -1

this.currentPlaintext = null

this.position = -1

this.character = null

this.advance\_line()

}

advance\_line(){

this.line += 1

this.currentPlaintext = this.line == this.allPlaintext.length ? null : this.allPlaintext[this.line]

this.position = -1

}

continue(){

this.position += 1

this.character = this.position == this.currentPlaintext.length ? null : this.currentPlaintext[this.position]

}

reverse(){

this.position -= 1

this.character = this.currentPlaintext[this.position]

}

make\_tokens(){

let file = []

while (this.currentPlaintext != null){

this.continue()

let result = this.make\_tokens\_line()

if (result instanceof Error){

return result

}

if (result.length >= 1){

file.push(result)

}

this.advance\_line()

}

return file

}

make\_tokens\_line(){

let tokens = []

while (this.character != null){

// Checking for anb empty space or tab to ignore

if (this.character == ' ' || this.character == "\t"){

// checking for number to create

} else if (Lexer.DIGITS.includes(this.character)) {

let number = this.make\_number()

if (number instanceof Error) {

return number

}

tokens.push(number)

continue

// checking to create identifier

} else if (Lexer.LETTERS.includes(this.character)){

tokens.push(this.make\_identifier())

continue

// checking for quotations for a string

} else if (this.character == '"' || this.character == "'"){

let string = this.make\_string()

if (string instanceof Error){

return string

}

tokens.push(string)

// checks for full stops for a property

} else if (this.character == '.') {

let property = this.make\_property()

if (property instanceof Error){

return property

}

tokens.push(property)

continue

// all arithmetic symbols

} else if (this.character == '+') {

tokens.push(new Add(this.position, this.line))

} else if (this.character == '-') {

tokens.push(new Minus(this.position, this.line))

} else if (this.character == '\*') {

tokens.push(new Multiply(this.position, this.line))

} else if (this.character == '/') {

this.continue()

if (this.character == '/'){ // Comment

return tokens

}

tokens.push(new Divide(this.position-1, this.line))

continue

} else if (this.character == '^') {

tokens.push(new Exponent(this.position, this.line))

// logical operators

} else if (['=','<','>','!'].includes(this.character)){

tokens.push(this.make\_logical\_operator())

continue

// punctuation

} else if (['(', ')',',',':','[',']'].includes(this.character)){

tokens.push(new TemplateKeyword(this.position, this.line, this.character))

// Not recognised gives an error

} else {

return new LexicalError(this.position, this.line, `Unexpected character '${this.character}'`)

}

this.continue()

}

return tokens

}

make\_number(){

let number = []

let fullStops = 0

let position = this.position

while (Lexer.DIGITS.includes(this.character) || this.character == '.'){

number.push(this.character)

if (this.character == '.'){ // for full stops

fullStops += 1 // increment

if (fullStops == 2){ // float followed by .

this.continue()

if (!Lexer.DIGITS.includes(this.character)){ // property check

this.reverse()

return new FloatType(position, this.line, Number(number.join('')))

} // error

return new LexicalError(this.position, this.line, "Only expected one '.' to create Float")

}

this.continue()

if (Lexer.DIGITS.includes(this.character)){ // float case

number.push(this.character)

} else if (this.character == null) { // end case

return new LexicalError(this.position, this.line, "Expected rest of Float to follow '.'")

} else { // property case

this.reverse()

return new IntegerType(position, this.line, Number(number.join('')))

}

}

this.continue()

}

return fullStops == 0 ? new IntegerType(position, this.line, Number(number.join(''))) : new FloatType(position, this.line, Number(number.join('')))

}

get\_name(){

let name = []

while (Lexer.LETTERS.includes(this.character) || Lexer.DIGITS.includes(this.character)){

name.push(this.character)

this.continue()

}

return name.join('')

}

make\_identifier(){

let position = this.position

let name = this.get\_name()

switch (name) {

case "MOD":

return new Modulus(position, this.line)

case "DIV":

return new Quotient(position, this.line)

case "True":

return new BooleanType(position, this.line, true)

case "False":

return new BooleanType(position, this.line, false)

case "AND":

return new And(position, this.line)

case "OR":

return new Or(position, this.line)

case "NOT":

return new Not(position, this.line)

case "if":

return new IfStatement(position, this.line)

case "switch":

return new SwitchStatement(position, this.line)

case "while":

return new While(position, this.line)

case "do":

return new Do(position, this.line)

case "for":

return new For(position, this.line)

case "procedure":

case "function":

return new UserDefinedSubroutine(position, this.line, name)

case "return":

return new Return(position, this.line)

case "const": case "global": case "array":

case "elseif": case "else": case "then":

case "case": case "default":

case "endif": case "endswitch":

case "endwhile": case "until":

case "to": case "step": case "next":

case "endprocedure": case "endfunction":

return new TemplateKeyword(position, this.line, name)

default:

return new Identifier(position, this.line, name)

}

}

make\_logical\_operator(){

let initialCharacter = this.character

this.continue()

if (this.character == '='){

this.continue()

return new ComparisonOperator(this.position-2, this.line, initialCharacter+'=')

}

switch (initialCharacter){

case '=':

return new Equals(this.position-1, this.line)

case '!':

return new LexicalError(this.position-1, this.line, "Unexpected character '!'")

default:

return new ComparisonOperator(this.position-1, this.line, initialCharacter)

}

}

make\_string(){

let string = []

let position = this.position

let quotationMark = this.character

this.continue()

while (this.character != quotationMark && this.character != null){

string.push(this.character)

this.continue()

}

return this.character == null ? new LexicalError(position, this.line, "Unclosed string") : new StringType(position, this.line, string.join(''))

}

make\_property(){

let position = this.position

this.continue()

let name = this.get\_name()

switch (name) {

case "length":

return new Length(position, this.line)

case "substring":

return new Substring(position, this.line)

case "left": case "right":

return new LeftOrRight(position, this.line, name)

case "upper": case "lower":

return new UpperOrLower(position, this.line, name)

case "readLine":

return new ReadLine(position, this.line)

case "endOfFile":

return new EndOfFile(position, this.line)

case "writeLine":

return new WriteLine(position, this.line)

case "close":

return new Close(position, this.line)

default:

return new LexicalError(position, this.line, `No property called '.${name}' exists`)

}

}

}

////////////////

// PARSER

////////////////

class Parser {

constructor(tokens){

this.allTokens = tokens // All of the tokens in the program in an 2D array

this.line = -1 // The current index of tokens in the array

this.currentTokens = null // The corresponding line of tokens in an array

this.position = -1 // The current position in the line

this.token = null // The corresponding token for the position

this.previous = null // The previous token

this.allowReturn = false // True if a function is currently being parsed

this.allowSubroutines = true // False if a subroutine is being parsed

}

advance\_line(){

this.line += 1

this.currentTokens = this.line == this.allTokens.length ? null : this.allTokens[this.line]

this.position = -1

}

continue(){

this.position += 1

this.previous = this.token

this.token = this.position == this.currentTokens.length ? null : this.currentTokens[this.position]

}

reset(){

this.position = -1

this.continue()

}

// Takes in an the classes as arguments and checks if current token is instance of the items

check\_instance() {

for (let item of arguments){

if (this.token instanceof item){

return true

}

}

return false

}

check\_tag(){

if (!(this.token instanceof TemplateKeyword)){

return false

}

for (let item of arguments){

if (this.token.tag == item){

return true

}

}

return false

}

check\_binary\_operator(){

if (this.token instanceof BinaryOperator){

if (this.token instanceof Add || this.token instanceof Minus){

return false

}

return true

}

return false

}

parse\_next(){

this.advance\_line()

if (this.currentTokens == null){

return null

}

this.continue()

return this.parse()

}

parse(){

// Check if there are no tokens

if (this.token == null){

return null

}

// Checks if it a while loop

if (this.token instanceof While){

let result = this.build\_while\_loop(this)

return this.check\_result(result) ? result : new SyntaxError(this.token, "Expeceted nothing to follow 'endwhile'")

}

// Checks if a do until loop

if (this.token instanceof Do){

return this.build\_do\_loop(this)

}

// Check if a for loop

if (this.token instanceof For){

return this.build\_for\_loop(this)

}

// Checks for a subroutine definition

if (this.token instanceof UserDefinedSubroutine){

if (this.allowSubroutines){

return this.build\_subroutine(this)

}

return new SyntaxError(this.token, "Cannot define a subroutine inside another subroutine")

}

// Check if it is a return statement

if (this.token instanceof Return) {

if (this.allowReturn){

return this.return(this)

}

return new SyntaxError(this.token, "Can only use 'return' within functions")

}

// Checks if an if statement is being built

if (this.token instanceof IfStatement){

let result = this.build\_if\_statement(this)

return this.check\_result(result) ? result : new SyntaxError(this.token, "Expeceted nothing to follow endif")

}

// Check for switch statements

if (this.token instanceof SwitchStatement){

let result = this.build\_switch\_statement(this)

return this.check\_result(result) ? result : new SyntaxError(this.token, "Expeceted nothing to follow endif")

}

// Check if the first token is a binary operator

if (this.check\_binary\_operator()){

return new SyntaxError(this.token, "Expected value")

}

// Check if there is a tagged variable assignment

if (this.check\_tag("const", "global")){

let result = this.assignment(this)

return this.check\_result(result) ? result : new SyntaxError(this.token, "Expected operator")

}

if (this.check\_tag("array")){

return this.create\_array(this)

}

// Check if elseif or endif is used not at end of line

if (this.check\_tag("endif", "elseif", "else")){

return new SyntaxError(this.token, "Needs to follow 'if' statement")

}

// Check for switch keywords used outside of switch

if (this.check\_tag("endswitch", "case", "default")){

return new SyntaxError(this.token, "Needs to follow 'switch' statement")

}

// Check if it is a normal assignment

if (this.token instanceof Identifier){

this.continue()

if (this.token instanceof Equals){

this.reset()

let result = this.assignment(this)

return this.check\_result(result) ? result : new SyntaxError(this.token, "Expected operator")

}

if (this.check\_tag('[')){

do {

this.continue()

} while (!this.check\_tag(']'))

this.continue()

if (this.token instanceof Equals){

this.reset()

let result = this.assignment(this)

return this.check\_result(result) ? result : new SyntaxError(this.token, "Expected operator")

}

}

this.reset()

}

// Left over case is just an expression or a statement

let result = this.statement\_chain(this)

return this.check\_result(result) ? result : new SyntaxError(this.token, "Expected operator")

}

check\_result(result){

if (result instanceof Error || this.token == null){

return true

}

return false

}

////////////////

// ARITHMETIC

////////////////

parse\_arguments\_or\_parameters(self, bracket){

self.continue()

if (self.token == null){ // ensures does not end at [

return new SyntaxError(self.previous, `Expected first value or '${bracket}'`)

}

if (self.check\_tag(bracket)){

self.continue()

return []

}

let argumentsOrParameters = []

let argument = self.statement\_chain(self)

if (argument instanceof Error){

return argument

}

argumentsOrParameters.push(argument)

if (self.token == null){

return new SyntaxError(self.previous, `Expected '${bracket}' or ',' to follow first value`)

}

while (self.check\_tag(',')){

self.continue()

if (self.token == null || self.check\_tag(bracket)){

return new SyntaxError(self.previous, "Expected next value to follow ','")

}

argument = self.statement\_chain(self)

if (argument instanceof Error){

return argument

}

argumentsOrParameters.push(argument)

}

if (self.check\_tag(bracket)){

self.continue()

return argumentsOrParameters

}

if (self.token == null){

return new SyntaxError(self.previous, `Expected '${bracket}' or ',' to follow value`)

}

return new SyntaxError(self.token, `Expected '${bracket}' or ',' followed by additional argument`)

}

calls\_or\_property(self, factor){ // accepts old factor

if (self.token instanceof Property){

let property = self.token

property.callee = factor // makes old factor the callee of the property

self.continue()

return property

} // otherwise it is a function call or an array call

let functionCall = self.token.tag == '(' // true if an array call

let call = new (functionCall ? Call : ArrayCall)(self.token.position, self.token.line) // creates new call

let argumentsAsts = self.parse\_arguments\_or\_parameters(self, functionCall ? ')' : ']') // parses args

if (argumentsAsts instanceof Error){

return argumentsAsts

}

if (functionCall) { // different names as very different purpioses

call.argumentsAsts = argumentsAsts

} else {

call.indexes = argumentsAsts

}

call.callee = factor // makes old factor callee

return call

}

factor(self){

let result

if (self.check\_instance(DataType, Identifier)){

result = self.token

self.continue()

} else if (self.check\_tag('(')){

result = self.parse\_brackets(self, ')', self.statement\_chain)

} else if (self.check\_instance(Add)){ // Add unary operator

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Incomplete input")

}

result = self.factor(self)

} else if (self.check\_instance(Minus)){ // Minus unary operator

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Incomplete input")

}

result = self.factor(self)

if (result instanceof Error){

return result

}

result.value = -result.value

} else { // Two operators in a row

return new SyntaxError(self.token, "Expected value")

}

while (self.check\_tag('(', '[') || self.token instanceof Property){ // adding calls or properties

result = self.calls\_or\_property(self, result) // updating

if (result instanceof Error){

return result

}

}

return result

}

exponent(self){

return self.parse\_binary\_operator(self, self.factor, [Exponent])

}

term(self){

return self.parse\_binary\_operator(self, self.exponent, [Multiply, Divide, Modulus, Quotient])

}

expression(self){

let result = self.parse\_binary\_operator(self, self.term, [Add, Minus])

if (self.check\_instance(IntegerType, FloatType, Identifier)){

return new SyntaxError(self.token, "Expected operator")

}

return result

}

statement(self){

if (this.check\_binary\_operator()){ // starts with binary operator

return new SyntaxError(self.token, "Expected value")

}

let left = self.expression(self) //left hand side

if (left instanceof Error || !(self.token instanceof ComparisonOperator)){

return left // returns if error or next is not comparison

}

let result = self.token // middle

self.continue()

if (self.token == null){ // nothing after comparison

return new SyntaxError(result, "Incomplete input")

}

if (this.check\_binary\_operator()){ // starts with binary operator

return new SyntaxError(self.token, "Expected value")

}

let right = self.expression(self) //right hand side

if (right instanceof Error){

return right

}

result.left = left

result.right = right

return result // returns comparison

}

not\_statement(self){

if (self.token instanceof Not){

let notToken = self.token

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Incomplete input")

}

let result = self.not\_statement(self)

if (result instanceof Error){

return result

}

notToken.child = result

return notToken

}

return self.statement(self)

}

statement\_chain(self){

return self.parse\_binary\_operator(self, self.not\_statement, [And, Or])

}

assignment(self){

let tag = null

if (self.check\_tag("const", "global")){

tag = self.token.tag

self.continue()

}

if (!(self.token instanceof Identifier)){

return new SyntaxError(self.token, "Expected identifier")

}

self.token.constant = tag == "const"

self.token.global = tag == "global"

let variable = self.factor(self)

let equals = self.token

if (equals == null){

return new SyntaxError(variable, "Expected '=' to follow identifier")

}

if (!(equals instanceof Equals)){

return new SyntaxError(equals, "Expected equals")

}

self.continue()

if (self.token == null){

return new SyntaxError(equals, "Expected expression to follow '='")

}

equals.right = self.statement\_chain(self)

if (equals.right instanceof Error){

return equals.right

}

equals.left = variable

return equals

}

////////////////

// ARRAYS

////////////////

create\_array(self){

let array = new ArrayType(self.token.position, self.token.line)

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Expected identifier to follow 'array'")

}

if (!(self.token instanceof Identifier)){

return new SyntaxError(self.token, "Expected identifier after 'array'")

}

array.identifier = self.token

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Expected = or [] to follow identifier")

}

if (self.check\_tag('[')){

let errorToken = self.token

array.creationAsts = self.parse\_arguments\_or\_parameters(self, ']')

if (array.creationAsts instanceof Error){

return array.creationAsts

}

if (array.creationAsts.length == 0){

return new SyntaxError(errorToken, "Array declaration must contain at least 1 dimension")

}

if (self.token != null){

return new SyntaxError(self.token, "Expected no tokens after array declaration")

}

return array

}

if (self.token instanceof Equals){

self.continue()

if (!self.check\_tag('[')){

return new SyntaxError(self.token == null ? self.previous : self.token , "Expected array to follow '='")

}

return self.create\_defined\_array(self, array)

}

}

create\_defined\_array(self, array){

array.isEmptyCreation = false

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Expected first value to follow '['")

}

if (self.check\_tag(']')){

return new SyntaxError(self.token, "Cannot create arrays with length 0")

}

let result

let arrayLength = 0

if (self.check\_tag('[')){

result = self.create\_defined\_array(self, new ArrayType(self.token.position, self.token.line))

} else {

result = self.statement\_chain(self)

}

if (result instanceof Error){

return result

}

arrayLength += 1

array.creationAsts.push(result)

if (self.token == null){

return new SyntaxError(self.previous, "Expected ']' or ',' to follow first value")

}

while (self.check\_tag(',')){

self.continue()

if (self.token == null || self.check\_tag(']')){

return new SyntaxError(self.previous, "Expected next item in array to follow ','")

}

if (self.check\_tag('[')){

result = self.create\_defined\_array(self, new ArrayType(self.token.position, self.token.line))

} else {

result = self.statement\_chain(self)

}

if (result instanceof Error){

return result

}

arrayLength += 1

array.creationAsts.push(result)

}

if (self.check\_tag(']')){

array.arrayLength = arrayLength

self.continue()

return array

}

if (self.token == null){

return new SyntaxError(self.previous, `Expected ']' or ',' to follow value`)

}

return new SyntaxError(self.token, `Expected ']' or ',' followed by additional argument`)

}

////////////////

// IF AND SWITCH

////////////////

// elseif

if\_condition(self){

let ifToken = new Case(self.token.position, self.token.line)

self.continue() // Continues past initial token

if (self.token == null){

return new SyntaxError(ifToken, "Expected condition after 'if")

}

let result = self.statement\_chain(self) // Gets statement to check

if (result instanceof Error){

return result

}

if (self.check\_tag("then")){ // Checks if line finishes with then

ifToken.condition = result

self.continue()

if (self.token == null){

return ifToken

}

return new SyntaxError(self.token, "Unexpected token after 'then'") // Tokens after then

}

return new SyntaxError(self.previous, "Expected 'then' to follow condition")

}

// else

else\_condition(self){

let elseToken = self.token

self.continue()

if (self.token != null){

return new SyntaxError(self.token, "Expected newline after 'else'")

}

return new ElseCase(elseToken.position, elseToken.line)

}

// case

switch\_condition(self){

let caseToken = new Case(self.token.position, self.token.line)

self.continue() // Continues past initial token

if (self.token == null){

return new SyntaxError(caseToken, "Expected value after 'case'")

}

let result = self.statement\_chain(self) // Gets statement to check

if (result instanceof Error){

return result

}

if (self.check\_tag(':')){ // Checks if line finishes with then

caseToken.condition = result

self.continue()

if (self.token == null){

return caseToken

}

return new SyntaxError(self.token, "Expected newline after ':'") // Tokens after then

}

return new SyntaxError(self.previous, "Expected ':' to follow expression")

}

// default

default\_condition(self){

let defaultToken = self.token

self.continue()

if (!self.check\_tag(':')){

return new SyntaxError(defaultToken, "Expected ':' after 'default'")

}

self.continue()

if (self.token != null){

return new SyntaxError(self.previous, "Expected newline after 'else")

}

return new ElseCase(defaultToken.position, defaultToken.line)

}

// if

build\_if\_statement(self){

let mainStatement = self.token // Stores the entire chain

let currentCase = self.if\_condition(self) // Creating first if case

if (currentCase instanceof Error){

return currentCase

}

self.advance\_line() // Advances line

return this.generic\_if\_or\_switch(self, mainStatement, currentCase)

}

// switch

build\_switch\_statement(self){

let mainStatement = self.token // Holds switch statement

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Expected expression after 'switch'")

}

let result = self.statement\_chain(self) // For switch statement comparison

if (result instanceof Error){

return result

}

mainStatement.comparison = result

if (self.token == null){

return new SyntaxError(self.previous, "Expected ':' to follow expression")

}

if (!self.check\_tag(':')){

return new SyntaxError(self.token, "Expected ':' after expression")

}

self.continue()

if (self.token != null){

return new SyntaxError(self.token, "Expected newline following ':'")

}

self.advance\_line()

if (self.currentTokens == null){

return new SyntaxError(self.previous, "Expected case after switch declaration")

}

self.continue()

if (!self.check\_tag("case")){

return new SyntaxError(self.token, "Expected 'case' keyword to define case")

}

let currentCase = self.switch\_condition(self) // Creating first switch case

if (currentCase instanceof Error){

return currentCase

}

self.advance\_line() // Advances line

return this.generic\_if\_or\_switch(self, mainStatement, currentCase)

}

// called after first case's expression has been parsed

// first expression is statement token

// current case is the first case already parsed, ready to add contents

generic\_if\_or\_switch(self, mainStatement, currentCase){

let isIf = mainStatement instanceof IfStatement

let conditionFunction = isIf ? self.if\_condition : self.switch\_condition

let elseFunction = isIf ? self.else\_condition: self.default\_condition

let statementName = isIf ? "if" : "switch"

let caseTag = isIf ? "elseif" : "case"

let elseTag = isIf ? "else" : "default"

let endTag = "end" + statementName

let allowCases = true

// main loop

while(self.currentTokens != null){

self.continue()

// CHECK FOR CASE TAG: either elseif or case

if (self.check\_tag(caseTag) && allowCases){

mainStatement.cases.push(currentCase) // Pushes old if case

currentCase = conditionFunction(self, self.token) // Creates new if case

if (currentCase instanceof Error){

return currentCase

}

} // CHECK FOR ELSE TAG: either else or default

else if (self.check\_tag(elseTag) && allowCases){

mainStatement.cases.push(currentCase) // Pushes old if case

currentCase = elseFunction(self) // new ending case

if (currentCase instanceof Error){

return currentCase

}

allowCases = false // No more cases

}

else if (self.check\_tag(endTag)){ // Check for ending

if (allowCases){

mainStatement.cases.push(currentCase) // Pushes final if case

} else { // OR

mainStatement.elseCase = currentCase // Adds else case

}

self.continue()

return mainStatement // complete

}

else {

let result = self.parse() // Default

if (result instanceof Error){

return result

}

currentCase.contents.push(result) //Add AST to the currents contents

}

self.advance\_line()

}

return new SyntaxError(mainStatement, `Expected ${endTag} at end of ${statementName} statement`) // Not complete

}

////////////////%

// LOOPS

////////////////

build\_while\_loop(self){

let whileToken = self.token

self.continue()

if (self.token == null){

return new SyntaxError(whileToken, "Expected condition after 'while'")

}

let condition = self.statement\_chain(self)

if (condition instanceof Error){

return condition

}

whileToken.condition = condition

self.continue()

if (self.token != null){

return new SyntaxError(self.token, "Expected no tokens after condition")

}

self.advance\_line()

while (self.currentTokens != null){

self.continue()

if (self.check\_tag("endwhile")){

this.continue()

return whileToken

}

let result = self.parse()

if (result instanceof Error){

return result

}

whileToken.contents.push(result)

self.advance\_line()

}

return new SyntaxError(whileToken, "Expected 'endwhile' to close loop")

}

build\_do\_loop(self){

let doToken = self.token //do

self.continue()

if (self.token != null){

return new SyntaxError(this.token, "Expeceted nothing to follow 'do'")

}

self.advance\_line()

while (self.currentTokens != null){ // adding contents

self.continue()

if (self.check\_tag("until")){ // finishing loop

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Expected condition after 'until'")

}

let condition = self.statement\_chain(self) // get condition

if (condition instanceof Error){

return condition

}

doToken.condition = condition

if (self.token != null){

return new SyntaxError(self.token, "Expected no tokens after condition")

}

return doToken // returned

}

let result = self.parse() // default case

if (result instanceof Error){

return result

}

doToken.contents.push(result)

self.advance\_line()

}

return new SyntaxError(doToken, "Expected 'until' to close loop")

}

build\_for\_loop(self){

let forToken = self.token // for

self.continue()

if (self.token == null){ // ensures tokena fter for

return new SyntaxError(forToken, "Expected assignment after 'for'")

}

if (self.token instanceof TemplateKeyword){

return new SyntaxError(self.token, "Expected identifier")

}

let result = self.assignment(self) //assignment

if (result instanceof Error){

return result

}

forToken.assignment = result

forToken.variable = result.left

if (self.token == null){

return new SyntaxError(self.previous, "Expected 'to' to follow assignment")

}

if (!self.check\_tag("to")){ // ensures next token is to

return new SyntaxError(self.token, "Expected 'to'")

}

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Expected expression after 'to'")

}

result = self.expression(self) // final value

if (result instanceof Error){

return result

}

forToken.finish = result

if (self.token == null){ // no step keyword

forToken.step = new IntegerType(self.previous.position, self.previous.line, 1)

} else {

if (!self.check\_tag("step")){ // ensures next token is step

return new SyntaxError(self.token, "Expected 'step' or end of line")

}

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Expected expression after 'step'")

}

result = self.expression(self) // step value

if (result instanceof Error){

return result

}

forToken.step = result

if (self.token != null){

return new SyntaxError(self.token, "Expected end of line")

}

}

self.advance\_line()

while (self.currentTokens != null){ // loop over contents

self.continue()

if (self.check\_tag("next")){

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, `Expected '${forToken.variable.name}' after 'next'`)

}

if (!(self.token instanceof Identifier)){

return new SyntaxError(self.token, `Expected identifier named '${forToken.variable.name}'`)

}

if (self.token.name != forToken.variable.name){

return new SyntaxError(self.token, `Expected identifier to be named '${forToken.variable.name}'`)

}

self.continue()

if (self.token != null){

return new SyntaxError(self.token, `Expected no tokens after '${forToken.variable.name}'`)

}

return forToken

}

let result = self.parse() // default contents

if (result instanceof Error){

return result

}

forToken.contents.push(result)

self.advance\_line()

}

return new SyntaxError(forToken, `Expected 'next ${forToken.variable.name}' to close loop`)

}

////////////////

// OTHER

////////////////

build\_subroutine(self){

self.allowReturn = self.token.tag == "function"

self.allowSubroutines = false

let subroutineToken = self.token // defining token

self.continue()

if (self.token == null ){ // function name

return new SyntaxError(subroutineToken, `Expected identifier to follow ${subroutineToken.tag} decleration`)

}

if (!(self.token instanceof Identifier)){

return new SyntaxError(self.token, `Expected identifier`)

}

subroutineToken.identifier = self.token

self.continue()

if (self.token == null){

return new SyntaxError(self.previous, "Expected '(' to follow identifier")

}

if (!self.check\_tag("(")){

return new SyntaxError(self.token, "Expected '(' followed by arguments")

}

let parameters = self.parse\_arguments\_or\_parameters(self, ')') // checking parameters

if (parameters instanceof Error){

return parameters

}

for (let parameter of parameters){

if (!(parameter instanceof Identifier)){

return new SyntaxError(parameter, "Expected identifier as parameter")

}

}

subroutineToken.parameters = parameters

if (self.token != null){

return new SyntaxError(self.token, "Expected no tokens following subroutine definition")

}

self.advance\_line()

while (self.currentTokens != null){ // adding contents

self.continue()

if (self.check\_tag("end" + subroutineToken.tag)){ // closing clause

this.continue()

self.allowReturn = false

self.allowSubroutines = true

return subroutineToken

}

let result = self.parse()

if (result instanceof Error){

return result

}

subroutineToken.contents.push(result)

self.advance\_line()

}

return new SyntaxError(subroutineToken, `Expected 'end${subroutineToken.tag}' to close ${subroutineToken.tag}`)

}

return(self){

let returnToken = self.token

self.continue()

if (self.token == null){

return returnToken

}

let result = self.statement\_chain(self)

if (result instanceof Error){

return result

}

returnToken.child = result

return returnToken

}

// First parameter = References the instance of the parser

// Second parameter = Method to call that is beneath the current one

// Thid parameter = Array of tokens which are to be checked for

parse\_binary\_operator(self, nextFunction, tokens){

let result = nextFunction(self)

if (result instanceof Error){

return result

}

while (self.token != null && self.check\_instance(...tokens)){

self.token.left = result

result = self.token

self.continue()

if (self.token == null){

return new SyntaxError(result, "Incomplete Input")

}

result.right = nextFunction(self)

if (result.right instanceof Error){

return result.right

}

}

return result

}

parse\_brackets(self, end, nextFunction){

let bracket = self.token

self.continue()

if (self.check\_tag(end)){

return new SyntaxError(bracket, "'()' was empty")

}

let result = nextFunction(self)

if (result instanceof Error){

return result

}

if (self.check\_tag(end)){

self.continue()

return result

}

return new SyntaxError(bracket, "'(' was never closed")

}

}

////////////////

// EXECUTION

////////////////

class Evaluator {

static global

static currentScope

static files = new FileStorage()

async evaluate\_loop(loop){

let iterationsSinceWait = 0

let condition = await loop.evaluate\_condition()

if (condition instanceof Error){

return condition

}

while (condition){

let contents = await loop.evaluate()

let result = await this.evaluate\_many\_asts(contents)

if (result != null){

return result

}

if (!Evaluator.active){

return new Abort()

}

iterationsSinceWait++

if (iterationsSinceWait >= 128){

await wait()

iterationsSinceWait = 0

}

condition = await loop.evaluate\_condition()

if (condition instanceof Error){

condition

}

}

loop.reset()

return null

}

async evaluate\_single\_ast(ast){

if (!Evaluator.active){

return new Abort()

}

if (ast instanceof Return){

return ast

}

if (ast instanceof IfStatement || ast instanceof SwitchStatement){

let result = await ast.evaluate()

if (result instanceof Error){

return result

}

return await this.evaluate\_many\_asts(result)

}

if (ast instanceof Loop){

return await this.evaluate\_loop(ast)

}

if (ast instanceof Error){

return ast

}

if (ast == null){

return

}

let evaluated = await ast.evaluate()

if (evaluated instanceof Error){

return evaluated

}

// if (evaluated instanceof DataType){ // Old code to print everything

// console.log(evaluated.display())

// }

return null

}

async evaluate\_many\_asts(asts){

if (asts == null){

return

}

for (let ast of asts){

let result = await this.evaluate\_single\_ast(ast)

if (result != null){

return result

}

}

return null

}

}

class Interpreter extends Evaluator{

static currentText = []

static active = false

constructor(){

super()

this.plaintext = null

this.tokens = []

this.have\_tokens = false

}

build\_global(){

Evaluator.global = new SymbolTable()

Evaluator.currentScope = Evaluator.global

Evaluator.global.push\_native\_subroutine("print", Print)

Evaluator.global.push\_native\_subroutine("input", Input)

Evaluator.global.push\_native\_subroutine("random", Random)

Evaluator.global.push\_native\_subroutine("str", TypeCast, StringType)

Evaluator.global.push\_native\_subroutine("int", TypeCast, IntegerType)

Evaluator.global.push\_native\_subroutine("float", TypeCast, FloatType)

Evaluator.global.push\_native\_subroutine("real", TypeCast, FloatType)

Evaluator.global.push\_native\_subroutine("bool", TypeCast, BooleanType)

Evaluator.global.push\_native\_subroutine("ASC", Asc)

Evaluator.global.push\_native\_subroutine("CHR", Chr)

Evaluator.global.push\_native\_subroutine("open", Open)

Evaluator.global.push\_native\_subroutine("newFile", NewFile)

}

get\_plaintext\_from\_file(fileName){

try {

this.plaintext = fs.readFileSync(fileName, 'utf8').split("\n")

Interpreter.currentText = this.plaintext

} catch (err) {

console.error(err)

}

this.have\_tokens = false

}

set\_plaintext\_manually(string){

this.plaintext = string.split("\n")

Interpreter.currentText = this.plaintext

this.have\_tokens = false

}

make\_tokens(){

let result = new Lexer(this.plaintext).make\_tokens()

if (result instanceof Error){

return result

}

this.tokens = result

this.have\_tokens = true

}

async run(){

this.build\_global()

Evaluator.files.storeAll()

Input.lastKey = null

if (!this.have\_tokens){

let result = this.make\_tokens()

if (result instanceof Error){

outputPrint(`${result.display()}\n\n ❌ Exited with failure`)

return 1

}

}

let parser = new Parser(this.tokens)

let ast = parser.parse\_next()

while (ast != null){

let result = await this.evaluate\_single\_ast(ast)

if (result instanceof Error){

outputPrint(($("#output").val().length == 0 ? '' : '\n') + `${result.display()}\n\n ❌ Exited with failure`)

return 1

}

ast = parser.parse\_next()

}

outputPrint($("#output").val().length == 0 ? " ✅ Exited successfully" : "\n ✅ Exited successfully")

return 0

}

shell(){

this.set\_plaintext\_manually(prompt(" ERL ==> "))

while (this.plaintext != "QUIT()"){

this.run()

this.set\_plaintext\_manually(prompt(" ERL ==> "))

}

}

run\_file(fileName){

this.get\_plaintext\_from\_file(fileName)

let code = this.run()

console.log(`\nExited with code ${code}`)

}

}

////////////////

// USER INTERFACE

////////////////

var savedOutput = "" // the contents of the output for switching

var currentFile = "output" // name of current file that is opened

$( () => {

////////////////

// HIGHLIGHTING

////////////////

CodeMirror.defineSimpleMode("erlcode", {

start: [

{regex: /"(?:[^\\]|\\.)\*?(?:"|$)/, token: "string"},

{regex: /'(?:[^\\]|\\.)\*?(?:'|$)/, token: "string"},

{regex: /endswitc/, token:"keyword", dedent: true, next: "endswitch"},

{regex: /(?:for|while|do|if|switch|case|procedure|function)\b/, token: "keyword", indent: true},

{regex: /(?:next|endwhile|until|endif|endswitch|endprocedure|endfunction)\b/, token: "keyword", dedent: true},

{regex: /(?:const|global|to|next|step|then|elseif|else|default|array|return)\b/ ,token: "keyword"},

{regex: /(?:input|print|str|int|float|real|bool|ASC|CHR|open|newFile|random)\b/, token: "native"},

{regex: /True|False/, token: "boolean"},

{regex: /0x[a-f\d]+|[-+]?(?:\.\d+|\d+\.?\d\*)(?:e[-+]?\d+)?/i, token: "number"},

{regex: /\/\/.\*/, token: "comment"},

{regex: /[-+\/\^\*=<>]|!=|MOD|DIV|AND|OR|NOT/, token: "operator"},

{regex: /([a-zA-Z\_]\w\*)(\.)((?:length|substring|left|right|upper|lower|close|readLine|writeLine|endOfFile))\b/, token: ["variable", null, "property"]},

{regex: /[a-zA-Z\_]\w\*/, token: "variable"},

],

endswitch: [ // for double dedent

{regex: /h/, token:"keyword", dedent: true, next: "start"},

{regex: /(.|\n)\*?/, next:"start"}

],

meta: {

lineComment: "//",

}

});

var inputArea = CodeMirror.fromTextArea($("#input")[0], {

lineNumbers : true,

theme: "monokai",

mode: "erlcode",

autofocus: true,

indentUnit: 4,

indentWithTabs: true

})

////////////////

// RUNNING

////////////////

let interpreter = new Interpreter

$("#run").click( async () => {

if (Evaluator.active) { // checks if already running

Evaluator.active = false

$("#output").focus()

} else {

loadOutput("output")

$("#output").val('') // reset output

$("#run").html("Stop 🛑") // change button text

$("#run ,#fileDropdown ,#manageFiles").addClass("active") // keep button pushed down

$("#abort").show()

Evaluator.active = true // make active

interpreter.set\_plaintext\_manually(inputArea.getValue())

await interpreter.run() // wait for completion

}

$("#run ,#fileDropdown ,#manageFiles").removeClass("active") // reset

$("#run").html("Run ➤") // to

Evaluator.active = false // original

$("#abort").hide()

})

$("#output").on( "keydown", (event) => {

if (event.keyCode == 13 || event.which == 13){

Input.enterPressed = true

}

})

////////////////

// UPLOAD / DOWNLOAD

////////////////

$("#upload").click( () => {

$("#uploadFile").click()

})

$("#uploadFile").change(() => {

let reader = new FileReader()

let files = $("#uploadFile")[0].files

reader.onload = () => {

inputArea.setValue(reader.result)

$("#uploadFile").val('')

}

reader.readAsText(files[0])

})

$("#download").click( () => {

$("#downloadFile").attr("href","data:text/text;utf8," + inputArea.getValue()) // set file contents to input

if (inputArea.getValue().startsWith('//')){ // if starts with comment, custom name

$("#downloadFile").attr("download", inputArea.getValue().split('\n')[0].substring(3) + ".erlcode")

} else { // otherwise default name

$("#downloadFile").attr("download", "myCode.erlcode")

} // click the anchor to trigger the download

$("#downloadFile")[0].click()

})

////////////////

// FILE MODES

////////////////

$("#add").click( () => {

let name = Evaluator.files.forceAddNew("newFile.txt")

$(`<div id="${(name)}">📎 ${name}</div>`).insertBefore( "#add" )

loadOutput(name)

})

$("#console").click( () => {

loadOutput("output")

})

$("#customUpload").click( () => {

$("#customUploadFile").click()

})

$('#fileDropdown div').click( (event) => {

let clicked = event.target.id

if (clicked == "add" || clicked == "console" || clicked == "customUpload"){

return

}

loadOutput((clicked))

});

function loadOutput(fileName) {

if (fileName == currentFile){

return

} // saving old

if (currentFile == "output"){

savedOutput = $("#output").val()

} else {

Evaluator.files.get(currentFile).rewrite($("#output").val())

} // if switching to output

if (fileName == "output"){

$("#output").val(savedOutput)

$("#output").attr("readonly", true)

$("#output").attr("placeholder",

`Welcome to the OCR ERL Interpreter!

Please enter your code into the editor on the right,

The output of code will be displayed in this window.

Upload and download ERL files with the editor buttons

Navigate the file system with the file icon above`)

$("#fileName").html("Console Output")

$('.fileOption').hide()

} else { // switching to file

$("#output").val(Evaluator.files.get(fileName).contents)

$("#output").attr("readonly", false)

$("#output").attr("placeholder",

`This is the built in file editor!

Type here to edit the file

To rename the file, click the pencil

To save the file, click the save icon

Reset the file to its original value with the button

To delete the file, click the bin`)

$("#fileName").html(fileName)

$('.fileOption').show()

} // final updates

currentFile = fileName

$("#output").focus()

}

$("#customUploadFile").change( () => {

let reader = new FileReader()

let files = $("#customUploadFile")[0].files

reader.onload = () => {

let name = Evaluator.files.forceAddNew(files[0].name, reader.result)

$(`<div id="${name}">📎 ${name}</div>`).insertBefore( "#add" )

$("#customUploadFile").val('')

loadOutput(name)

}

reader.readAsText(files[0])

})

////////////////

// FILE OPTIONS

////////////////

$("#renameFile").click( () => {

let inputName = prompt(`Enter the new name of '${currentFile}', with the file extention:`)

while (!validateFileName(inputName)){

inputName = prompt("Enter the new name in a valid format:\nIt must contain a single '.', and be a '.txt' or '.csv' file")

}

let name = Evaluator.files.rename(currentFile, inputName)

$($('#' + currentFile.replace('.', "\\."))[0]).text(`📎 ${name}`)

$($('#' + currentFile.replace('.', "\\."))[0]).attr("id", name)

currentFile = name

$("#fileName").html(name)

})

$("#customDownload").click( () => {

$("#downloadFile").attr("href","data:text/text;utf8," + $("#output").val())

$("#downloadFile").attr("download", currentFile)

$("#downloadFile")[0].click()

})

$("#resetFile").click( () => {

Evaluator.files.get(currentFile).writeStored() // reset the file

$("#output").val(Evaluator.files.get(currentFile).contents) // display

})

$("#deleteFile").click( () => {

let fileToDelete = currentFile

loadOutput("output")

Evaluator.files.delete(fileToDelete)

$($('#' + fileToDelete.replace('.', "\\."))[0]).remove()

})

})

////////////////

// GLOBAL FUNCTIONS

////////////////

function outputPrint(message, newLine=true){

$("#output").val($("#output").val() + message + (newLine ? '\n' : ''))

$("#output").scrollTop($("#output")[0].scrollHeight)

}

function wait(){

return new Promise((resolve) => {

setTimeout(() => {

resolve()

}, 4 // modify to change rate of input checking

)}) // in milliseconds

}

function validateFileName(name){

name = name.split('.')

if (name.length != 2){

return false

}

return ["txt", "csv"].includes(name[name.length - 1])

}

# webpage html (index.html)

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>OCR ERL Interpreter</title>

<link rel="stylesheet" href="main.css">

<link rel="stylesheet" href="./syntax-highlighting/codemirror.css">

<link rel="stylesheet" href="./syntax-highlighting/monokai.css">

<script src="./syntax-highlighting/codemirror.js"></script>

<script src="./syntax-highlighting/simpleMode.js"></script>

<script src="https://code.jquery.com/jquery-3.7.1.js" integrity="sha256-eKhayi8LEQwp4NKxN+CfCh+3qOVUtJn3QNZ0TciWLP4=" crossorigin="anonymous"></script>

</head>

<body>

<div id = "frame">

<div id="title">OCR Exam Reference Language Interpeter</div>

<div id="version">Beta v1.4<br>by Patrick Williams</div>

<!-- Links -->

<div id="pages">

<a class="page" href="./syntax.html" target="\_blank">The ERL Syntax Guide</a>

<a class="page" href="https://www.ocr.org.uk/Images/558027-specification-gcse-computer-science-j277.pdf#page=27" target="\_blank">OCR Documentation</a>

<a class="page" id="report" style href="https://forms.office.com/e/LQ54CkNLJE" target="\_blank">Give Feedback</a>

<a class="page" id="report" href="https://forms.office.com/e/HSrVJjfCHR" target="\_blank">Report Bug</a>

</div>

<!-- Execution -->

<textarea id="input"></textarea>

<textarea id="output" placeholder=

"Welcome to the OCR ERL Interpreter!

Please enter your code into the editor on the right

The output of code will be displayed in this window

Upload and download ERL files with the editor buttons

Navigate the file system with the file icon above"

readonly></textarea>

<button id="run" title="Execute ERL code">Run ➤</button>

<!-- Uploading ERL -->

<button class="rect" id="upload" title="Upload ERL or .txt file from computer">📂</button>

<input type="file" id="uploadFile" style="display:none" accept=".txt,.erlcode">

<!-- Downloading ERL -->

<button class="rect" id="download" title="Download current code as ERL file to save">💾</button>

<a download="myCode.erlcode" href="data:text/text;utf8," id="downloadFile" style="display:none"></a>

<!-- File Navigation -->

<div class="fileNavigation" id="fileName">Console Output</div>

<div id="fileDropdown">

<button class="rect fileNavigation" id="manageFiles" title="Save the current contents of this file">🗂️</button>

<div id="dropdownContents">

<div id="console">🖥️ Console Output</div>

<div id="add">🛠️ Create New File</div>

<div id="customUpload">📂 Upload a File</div>

</div>

</div>

<!-- Uploading File -->

<input type="file" id="customUploadFile" style="display:none" accept="text/plain">

<!-- File Options -->

<button class="rect fileOption" id="customDownload" title="Save the current contents of this file">💾</button>

<button class="rect fileOption" id="renameFile" title="Rename the file">✏️</button>

<button class="rect fileOption" id="resetFile" title="Reset this file to its original value">🔄</button>

<button class="rect fileOption" id="deleteFile" title="Remove this file from the Interpreter">🗑️</button>

<div id="files"></div>

<script src="./src.js"></script>

</div>

</body>

</html>

# styling (main.css)

body {

background-color: #0F0F0F;

font-family: monospace;

}

#frame {

display: grid;

grid-template-columns: repeat(30, 1fr);

grid-template-rows: repeat(20, 1fr);

height: 98vh;

grid-gap: 0.3vw;

}

#title {

grid-area: 1 / 1 / 1 / 31;

color: whitesmoke;

font-size: 40px;

text-align: center;

}

#pages {

grid-area: 2 / 1 / 3 / 18;

display: flex;

}

.page {

padding-left: 1vw;

font-size: 18px;

color: whitesmoke;

cursor: pointer;

text-decoration: underline;

flex-grow: 1;

align-self: center;

text-decoration-thickness: 1px;

}

#output {

background-color: #161617;

border-radius: 10px;

color: whitesmoke;

outline: 0;

font-size: 18px;

resize: none;

border: 3px solid #005B41;

grid-area: 4 / 18 / 21 / 31;

tab-size: 1;

}

#output:focus {

outline: 1px solid #3a772e;

font-size: 18px;

}

button {

background-color: #3a772e;

border: none;

box-shadow: 0 5px #23481c;

cursor: pointer;

text-align: center;

text-decoration: none;

outline: none;

color: #fff;

}

.rect {

border-radius: 10px;

width: 50px;

height: 50px;

justify-self: right;

font-size: 30px;

}

#upload {

grid-area: 3 / 17 / 4 / 18;

z-index: 2;

margin: 6px 6px 3px;

}

#download {

grid-area: 4 / 17 / 5 / 18;

z-index: 3;

margin: 0px 6px;

}

#run {

grid-area: 2 / 27 / 4 / 31;

height: 70px;

font-size: 30px;

border-radius: 35px;

margin-top: 45px;

}

#run.active, #deleteFile {

background-color: rgb(108, 10, 10);

box-shadow: 0 5px rgb(54, 5, 5);

}

#run.active:hover, #deleteFile:hover {

background-color: rgb(142, 0, 0);

}

button:active {

transform: translateY(2px);

}

button:hover:not(.active) {

background-color: #4e9f3d

}

.fileOption {

display: none;

}

#version {

grid-area: 1 / 1 / 2 / 4;

color: whitesmoke;

}

#renameFile {

grid-area: 4 / 29 / 5 / 31;

z-index: 2;

margin: 6px 6px 3px;

}

#customDownload {

grid-area: 5 / 29 / 6 / 31;

z-index: 3;

margin: 0px 6px 3px;

}

#resetFile {

grid-area: 6 / 29 / 7 / 31;

z-index: 3;

margin: 0px 6px 3px;

}

#deleteFile {

grid-area: 7 / 29 / 8 / 31;

z-index: 3;

margin: 0px 6px;

}

#fileName {

grid-area: 3 / 18 / 4 / 22;

color: white;

background-color: #23481c;

border: none;

box-shadow: 0 5px #152a11;

height: 50px;

border-radius: 10px;

text-align: center;

vertical-align: middle;

line-height: 50px;

font-size: 18px;

}

.fileNavigation {

margin-top: 6px;

}

#manageFiles {

cursor: default;

pointer-events: none;

}

#manageFiles.active {

background-color: #23481c;

box-shadow: 0 5px #152a11;

}

#fileDropdown {

position: relative;

display: inline-block;

grid-area: 3 / 22 / 4 / 23;

}

#dropdownContents {

border-radius: 6px;

display: none;

position: absolute;

background-color: #418633;

min-width: 230px;

box-shadow: 0px 8px 16px 0px rgba(0,0,0,0.2);

z-index: 3;

border: 1px solid #418633;

}

#dropdownContents div {

border-radius: 6px;

color: whitesmoke;

padding: 12px 16px;

text-decoration: none;

display: block;

cursor: pointer;

}

#dropdownContents div:hover {background-color: #356e2a;}

#fileDropdown:hover:not(.active) #dropdownContents {display: block;}

#fileDropdown:hover:not(.active) #manageFiles {background-color: #4e9f3d;}

#console, #add, #customUpload {

background-color: #295d1f;

border: 3px solid #418633;

}

# guide page (syntax.html)

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>OCR ERL Syntax</title>

<link rel="stylesheet" href="./syntax-highlighting/codemirror.css">

<link rel="stylesheet" href="./syntax-highlighting/monokai.css">

<script src="./syntax-highlighting/codemirror.js"></script>

<script src="./syntax-highlighting/simpleMode.js"></script>

<script src="https://code.jquery.com/jquery-3.7.1.js" integrity="sha256-eKhayi8LEQwp4NKxN+CfCh+3qOVUtJn3QNZ0TciWLP4=" crossorigin="anonymous"></script>

<style>

body {

color: whitesmoke;

background-color: #0F0F0F;

font-family: monospace;

}

h1 {

font-size: 40px;

text-align: center;

font-weight: 100;

}

a {

font-size: 18px;

color: whitesmoke;

cursor: pointer;

text-decoration: underline;

align-self: center;

}

</style>

</head>

<body>

<a class="page" href="./index.html">Return To Interpreter</a>

<h1>Exam Reference Language Syntax</h1>

<div style="width: 80vw; margin: auto;">

<textarea class="syntax">

// This is a layout of the syntax for the OCR Exam Reference Language

// ERL has a lot of differences to other languages, such as Python

// This document explains all of these differences so you can write ERL

// This document can also be edited, so write your own algorithms to practice

// ARITHMETIC

(2 + 3) \* 7 / 4 // Normal arithmetic works as expected

3 ^ 7 // ERL uses ^ for powers, instead of \*\*

27 MOD 5 // ERL uses MOD and DIV for modulus and remainder divison

16 DIV 7 // This is instead of % and // in python

// VARIABLES

x = 5 // variables do not need a keyword to be defined

myVariable = "OCR" // they can be of any type

const pi = 3.1415 // use the const keyword to create constants that cannot be modified

// BASIC FUNCTIONS

print("hello world") // printing is the same as python

name = input("Enter your name: ") // inputs also work similarly

// comments can always be made with two slashes

// prints with values seperated by commas will be printed with spaces between them

// BOOLEANS AND COMPARISONS

boolean = True // booleans require capitalisation

result = NOT ( True AND False ) OR True // the boolean operators are in full capitals

check = value >= 7 AND name != "Pramukhi" // the boolean operators are as expected

// the full list is >, >=, <, <=, == and !=

// IF STATEMENTS

if answer == "Yes" then // then is required after each condition

print("Correct") // this is instead of a colon, such as in python

endif // if statements must be closed with endif

if percentage >= 90 then // multiple branches can be added

print("A\*")

elseif percentage >= 75 then // elseif is the keyword for adding additional cases

print("A")

elseif percentage >= 60 then // then is also needed after each elseif condition

print("B")

else // else is used for a default case

print("C") // it does not require then

endif // endif is always used

// LOOPS

answer = input("What is 3 \* 7?")

while answer != "21" // while loops can be created with the while keyword

print("Incorrect, try again") // they are followed by a condition, with no colon afterwards

answer = input("What is 3 \* 7?") // they repeat while the condition is true

endwhile // they must end with the endwhile keyword

do // do until loops are created with the do keyword

password = input("Enter password: ") // they close with until followed by a condition

until password == "Abc123" // they repeat when the condition is false, and stop when it is true

for i = 0 to 5 // counter controlled loops are created with for

print(i) // they define a variable, with an initial value and final value

next i // they are always closed with next, followed by the variable fileName

// when starting to run, for loops set the variable to the starting value

// it will increment on each iteration, up until and including the final value

// this means they are inclusive, wheras languages like python do not include the last value

// for example, the loop above would print 0, 1, 2, 3, 4, 5

for countdown = 10 to 1 step -1 // for loops can also be given a step

print(countdown) // this is a positive or negative integer following the final value

next countdown // in this case, the loop prints the numbers from 10 to 1 backwards

// FUNCTIONS AND PROCEDURES

procedure checkID(age) // procedures are subroutines that do not return a value

if age >= 18 then // they have an optional list of parameters that is defined on the first line

print("Allowed") // the code inside them can use these parameters

else // any variables created will be removed after the procedure is over

print("Denied")

endif

endprocedure // procedures are closed with endprocedure

checkID(21) // they can be called with their name, followed by brackets with the parameters

function square(number) // functions return values

return number^2 // they stop running when a return statement is run

endfunction // they are closed with endfunction

result = square(7) // the results of a function can be used by being stored in variables

global complete = False // global variables can be changed by using the global keyword

// this only works inside a function

// if not used, a local variable with the same name will be made, and the original will be unchanged

// global variables can still be acessed without the keyword, so long as a local with the same name does not exist

// TYPE CASTING

str(7.3) // to a string from any type

int("27") // to an integer(

int(4.3) // when used on a float, it will be truncated

int(True) // when used on a boolean, 1 or 0 will be returned

float("7.3") // to a float

real(4) // the real keyword can also be used

bool("True") // "True" or "False" will return their values

bool(7) // returns False if 0, otherwise returns True

// ARRAYS

array emptyArray[5] // creates an empty array with 5 items

// arrays are always 0 indexed, so this will have indexes 0 to 4

array colours = [ "red", "blue", "yellow", "green"] // they can also be made with items

// the type of every item in array must be the same

// this means that the array must be just strings, or just integers and so on

colours[2] = "pink" // values can be changed as expected

print( colours[0] ) // as well as being acessed

// there is no way to append or remove items from arrays

// this is because they are a constant size, so always stay the same length

// arrays can also have multiple dimensions

array chessBoard[8, 8] // this makes an empty 8 by 8 array

array names = [ ["Patrick", "Williams"], ["Pramukhi", "Vadrevu"] ] // this also makes a 2D array

names[0, 1] = "Frawley" // the items are acessed with commas seperating each index

// this can continue for any number of dimensions

// the length of all arrays inside an array must always be the same length

// SWITCH STATEMENTS

username = input("=> ") // switch statements are created with a value to compare from

switch username: // the switch statement value is followed by a colon

case "admin": // each case keyword is followed by a value and a colon

print("Administrator") // the code inside will only be run if they match

case "user1": // multiple cases can be created

print("User 1")

default: // default cases can be added at the end, the equivalent of else

print("Unauthorised") // these also need a colon

endswitch // a switch is closed by endswitch

// BUILT-IN FUNCTIONS

ASC("A") // returns the ascii value of a letter, in this case 65

CHR(97) // returns the character associated with an ascii value, in this case 'a'

diceRoll = random(1, 6) // will return a random integer from 1 to 6 inclusibe

value = random(-3.0, 2.0) // floats as arguments will return floats

// STRING METHODS

myString = "Hello World!" // this will be the example string

myString.length // returns the length, in this case 12

// length can also be used on arrays in this Interpreter, however it is not explicitly stated in OCR's ERL

myString.substring(3, 6) // this will return "lo Wor"

// the first argument for substring is the index of the first character in the substring

// the second argument will be the length of the substring

// this is different from many languages, which often accept the start and end index, not length

myString.substring(n, 1) // this will return the character at position n

myString.left(4) // this will give "Hell", the 4 left characters

myString.right(n) // like left(), right() will return the n last characters in a string

name = firstname + " " = surname // strings can be concatenated with +

print("The answer is:" + answer) // when used inside prints, they will not be seperated by spaces, unlike commas

myString.upper // returns "HELLO WORLD!"

myString.lower // returns "hello world!"

// note there are no brackets () after upper or lower, unlike in python

// FILE HANDLING

file = open( "myFile.txt" ) // use the open keyword to open a file

file.readLine() // returns the next line in the file

file.endOfFile() // returns a boolean based on if the whole file has been read

while NOT file.endOfFile() // these can be used together, as shown

print( file.readLine() ) // this loop will output the entire contents of a file

endwhile

file.close() // files should be closed after being used

newFile( "results.csv" ) // files can be made using newFile()

results = open( "results.csv" ) // these still must be opened after creation

results.writeLine( "55%" ) // values can be written to a file

// this will always be written at the end of the file

results.writeLine( "\n78%") // \n is required to write on a new line

// This is now all of the features of the ERL

// Return to the Interpreter to try yourself!</textarea>

</div>

<script>

$( () => {

CodeMirror.defineSimpleMode("erlcode", {

start: [

{regex: /"(?:[^\\]|\\.)\*?(?:"|$)/, token: "string"},

{regex: /'(?:[^\\]|\\.)\*?(?:'|$)/, token: "string"},

{regex: /endswitc/, token:"keyword", dedent: true, next: "endswitch"},

{regex: /(?:for|while|do|if|switch|case|procedure|function)\b/, token: "keyword", indent: true},

{regex: /(?:next|endwhile|until|endif|endswitch|endprocedure|endfunction)\b/, token: "keyword", dedent: true},

{regex: /(?:const|global|to|next|step|then|elseif|else|default|array|return)\b/ ,token: "keyword"},

{regex: /(?:input|print|str|int|float|real|bool|ASC|CHR|open|newFile|random)\b/, token: "native"},

{regex: /True|False/, token: "boolean"},

{regex: /0x[a-f\d]+|[-+]?(?:\.\d+|\d+\.?\d\*)(?:e[-+]?\d+)?/i, token: "number"},

{regex: /\/\/.\*/, token: "visible-comment"},

{regex: /[-+\/\^\*=<>]|!=|MOD|DIV|AND|OR|NOT/, token: "operator"},

{regex: /([a-zA-Z\_]\w\*)(\.)([a-zA-Z\_]\w\*)/, token: ["variable", null, "property"]},

{regex: /[a-zA-Z\_]\w\*/, token: "variable"},

],

endswitch: [ // for double dedent

{regex: /h/, token:"keyword", dedent: true, next: "start"},

{regex: /(.|\n)\*?/, next:"start"}

],

meta: {

lineComment: "//",

}

});

CodeMirror.fromTextArea($(".syntax")[0], {

lineNumbers : true,

theme: "monokai",

mode: "erlcode",

indentUnit: 4,

indentWithTabs: true

})

})

</script>

</body>

</html>

# modules (for the interface)

JQuery: <https://code.jquery.com/jquery-3.7.1.js>

CodeMirror: <https://codemirror.net/5/>

Theme for CodeMirror (I modified slightly): <https://github.com/FarhadG/code-mirror-themes>

Custom Highlighting (CodeMirror addon): <https://codemirror.net/5/addon/mode/simple.js>

### small snippets

I referenced links to the relevant sources that I used in the program throughout. This was mainly in the User Interface section, where I needed to find several solutions to the quirks of JavaScript and how it was not working fully as intended.

I also used the documentation for the various resources that I have linked above throughout development for a lot help with learning the different features. Overall though, all the files that have been pasted above have been almost completely written by me excluding these tiny fixes.