Interpretation for lazy and curious

Oleg Taykalo

October 8, 2022

Contents

1 Interpretation. part 1

The core functionality of interpreters might be distilled to two steps.

1. Parse source code to some internal representation
2. Evaluate the internal representation, so it will be reduced to the result we need.

Here I will do my best to skip as much as possible of the first one. You dont' need to know quantum physics to be able to swim.

1.1 Can you do it?

When talking about interpretation most of the people start thinking about complex things. register allocation, memory management, grammars and DFAs. I will try to keep it simple. As simple as possible without loosing core of the essence. Let's assume you have a yaml file

```
bold
- some text
- and more text
```

How hard would that be to transform this yaml to the html?

```
<b>
  <p>some text</p>
  <p> and more text></p>
</b>
```

My guess would be that majority of the developers won't have troubles with doing so. We're just transforming on tree structure to another tree structure.

And this is the essense of interpretation. We're taking one tree structure and converting it to another tree structure. If you can convert yaml to html, you're good to go.

2 Parsing and other friends

Most of the compilers and interpreters books start with parsing. You neeed to investigate what DFA is, how it is different from NFA, and why regular expressions are not always the best choice. Then you will follow on grammars. Context free grammars, left recursion elimination and so on. After that you will be presented with tools that will make your life easier. Yacc, Bison, monadic parsers. This usually happens 200-300 pages into the book. And while this is very important part of the story, I would like to argue this can be postponed to the point where you actually know what you are doing. Therefore approach in this book will take us as fast as possible to the actual interpretation, skipping all unnecessary steps if possible. To begin with we will start with arithmetic expressions.

2.1 The simplest expressions you can think of

Implementing anything on JVM starts with interfaces. let's us to the same and implement your first Expression

```
interface Expression
```

It's not much, but we will improve it really fast. Just stay tuned.

2.1.1 Integer values

It is hard to do any computations when you don't have numbers.

```
data class EInt(val v:Int):Expression
```

For now integers are more than enough for us to have. Adding boolean and float values will follow.

2.1.2 Addition

Adding additional addition to the mix would not be a problem

```
data class EAdd(val a:EInt, val b:EInt): Expression
```

2.1.3 Multiplication

The same goes for the multiplication

```
data class EMul(val a:EInt, val b:EInt): Expression
```

2.2 What else can we add to the expressions

2.3 Representing expressions in a code

Now we have all the required components to actually start calulating. How would expression

```
2+2
```

would look in our data class representation?

```
EAdd(2,2)
```

Multiplication would not be that different

```
5*5
```

```
EMul(5, 5)
```

And if we want to express something more complex, we will just nest them

```
(2+2)*5*5
```

```
EMul(5, EMul(5, EAdd(2 , 2)))
```

3 Evaluating expressions for fun and profit

3.1 Evaluating integers. As simple as it gets

Having expresisons is nice, but why do we need them if we can't calculate them on the fly? as TDD teaches us, you whould start from the test first. So be it. You never can be too slow when doing unit tests. Good coverage for the basics will let you know details of your code workings. As well will save you hours of debugging later on.

```
@Test
fun testSimpleInteger(){
    assertEquals(EInt(5), eval(EInt(5)))
    assertEquals(EInt(-5), eval(EInt(-5)))
    assertEquals(EInt(0), eval(EInt(0)))
}
```

What would be the easiest implementation of **eval** to make it run? I came up with this one.

```
fun eval( e:Expression): Expression{
    return e;
}
```

Probably would not require a lot of effort to understand what is happening here. and it also makes our tests green.

3.2 Evaluating addition

Now we are facing our first real challenge. How can we evaluate addition?

As usual, let us start with the unit tests, so we don't have a suprises later on.

```
 @Test
    fun testSimpleAddition(){
        assertEquals(EInt(5), eval(EAdd(EInt(5), EInt(0))))
        assertEquals(EInt(5), eval(EAdd(EInt(0), EInt(5))))
        assertEquals(EInt(10), eval(EAdd(EInt(7), EInt(3))))
        assertEquals(EInt(7), eval(EAdd(EInt(-3), EInt(10))))
        assertEquals(EInt(6), eval(EAdd(EInt(10), EInt(-4))))
    }
```

obviously if we try to run them now, we would have an error

```
Expected :EInt(v=5)
Actual   :EAdd(a=EInt(v=5), b=EInt(v=0))
```

By the way - this is one of the reasons Kotlin is a nice language for these kind of exercises. You don't need to write a lot of boilerplates for the **data class**, string representation looks reasonalbe, and saves some mental space for more important things.

We have a red test. What is the simplest way to make it green?

```
fun eval( e:Expression): Expression{
    if(e is EAdd){
        return EInt( e.a.v + e.b.v)
    }
    return e
}
```

would this be enough?

Oh wow, tests are green now. But i am not big fan of 'e.a.v' looks very obscure Green test allows us to do refactoring bit.

```
data class EInt(val value:Int):Expression

fun eval( expression:Expression): Expression{
    if(expression is EAdd){
        return EInt(expression.a.value + expression.b.value)
    }
    return expression
}
```

Are we done with the addition?

What about more complex example?

```
1 + 2 + 3 => 6
assertEquals(EInt(6), eval( EAdd(EInt(1), EAdd( EInt(2),EInt(3)) )))
```

And our compiler complains.

```
Type mismatch: inferred type is EAdd but EInt was expected
```

Why is that? Oh yes. in our definition we are expecitng that **EAdd** expects integers as a parameters.

```
data class EAdd(val a:EInt, val b:EInt): Expression
```

But now we are trying to pass **EAdd** as second paramter. Of course we should have an error here.

As usual with TDD approach we fixe it with the simplest way possible

```
data class EAdd(val a:Expression, val b:Expression): Expression
```

Hopefully it will solve our issue.

But apparently it would not.

We get yet another nasty error in our **eval** function

```
fun eval( expression:Expression): Expression{
    if(expression is EAdd){
        return EInt(expression.a.value + expression.b.value)
        //                          ^
        //                          |
```

```
        // unresolved reference: value
    }
    return expression
}
```

Any ideas? Sure. **value** is a field of and integer, and now EAdd works with generic expressions, which might not have a value. So we need to come up with a way of converting addition result to values.

```
fun eval( expression:Expression): Expression{
    if(expression is EAdd){
        val left: EInt = eval(expression.a) as EInt
        val right: EInt  = eval(expression.b) as EInt
        return EInt(left.value + right.value)
    }
    return expression
}
```

This expression is worth to take a closer look. To execute and addition, we need to be sure that both our parameters are integers. At the same time when we call **eval** we would get the **Expression** as a result. Therefore we need to convert it to the class we want. In our case that would be **EInt**.

Of course this little piece of code might rise some questions. What happens if during **eval** will return something different from **EInt** would not that break everything? What will happen if during **eval** some error will be raised? These are very good questions. and we will get to them later. For now let us keep our focus on general picture.

3.3 Evaluating multiplication

After spending so much time on evaluating addition, expanding our expressions to support multiplication is straighforward. But first - tests.

```
    @Test
    fun testSimpleMultiplication()
    {
        assertEquals(EInt(0), eval(EMul(EInt(0), EInt(42))))
        assertEquals(EInt(0), eval(EMul(EInt(42), EInt(0))))
        assertEquals(EInt(42), eval(EMul(EInt(1), EInt(42))))
        assertEquals(EInt(42), eval(EMul(EInt(42), EInt(1))))
        assertEquals(EInt(4), eval(EMul(EInt(2), EInt(2))))
    }
```

and implementation

```
fun eval( expression:Expression): Expression{
    if(expression is EAdd){
        val left: EInt = eval(expression.a) as EInt
        val right: EInt  = eval(expression.b) as EInt
        return EInt(left.value + right.value)
    }
    if(expression is EMul){
        val left: EInt = eval(expression.a) as EInt
        val right: EInt  = eval(expression.b) as EInt
        return EInt(left.value * right.value)
    }
    return expression
}
```

As you can see, the implementation is almost identical. The only difference we have s the actual operation.

Let's add few more tests to make sure it works if we mix addition and multiplication

```
    @Test
    fun testComplicatedMultiplication() {
        assertEquals(EInt(10), eval(EMul(EMul(EInt(1), EInt(5)), EInt(2))))
        assertEquals(EInt(12), eval(EMul(EAdd(EInt(1), EInt(5)), EInt(2))))
    }
```

All the tests are green. Nice. In less than 20 lines of code we have working caclulator that can add and multiply arbitrary numbers.

```
interface Expression
data class EInt(val value:Int):Expression
data class EMul(val a:Expression, val b:Expression): Expression
data class EAdd(val a:Expression, val b:Expression): Expression

fun eval( expression:Expression): Expression{
    if(expression is EAdd){
        val left: EInt = eval(expression.a) as EInt
        val right: EInt  = eval(expression.b) as EInt
        return EInt(left.value + right.value)
    }
```

```
    if(expression is EMul){
        val left: EInt = eval(expression.a) as EInt
        val right: EInt  = eval(expression.b) as EInt
        return EInt(left.value * right.value)
    }
    return expression
}
```

3.4 Sidenotes

3.4.1 Moving evaluation into each expression

At this point all our evaluation is sitting within one **eval** function. If we gonna add more expressions to the interpreter, amount of **if** statements will grow. This can be avoided by extracting specific parts of the code, and moving them to expressions. The result would look like this:

```
interface Expression{
    fun eval():Expression
}
data class EInt(val value:Int):Expression {
    override fun eval(): Expression {
        return this
    }
}

data class EMul(val a:Expression, val b:Expression): Expression {
    override fun eval(): Expression {
        val left = a.eval() as EInt
        val right = b.eval() as EInt
        return EInt(left.value * right.value)
    }
}

data class EAdd(val a:Expression, val b:Expression): Expression {
    override fun eval(): Expression {
        val left = a.eval() as EInt
        val right = b.eval() as EInt
        return EInt(left.value + right.value)
```

```
    }
}
```

It would require to change the test implementation as well.

```
assertEquals(EInt(12), (EMul(EAdd(EInt(1), EInt(5)), EInt(2))).eval())
```

3.4.2 Simple optimisations

You might agree that multiplying anything by zero would not help a lot. We can include it in our evaluation of **EMul**

```
data class EMul(val a:Expression, val b:Expression): Expression {
    override fun eval(): Expression {
        val left = a.eval() as EInt
        if (left.value == 0){
            return EInt(0)
        }
        val right = b.eval() as EInt
        return EInt(left.value * right.value)
    }
}
```

And if you agree that integers are immutable values, we can avoid creating new copy of the EInt every time during the evaluation

```
val zeroInt = EInt(0)

data class EMul(val a:Expression, val b:Expression): Expression {
    override fun eval(): Expression {
        val left = a.eval() as EInt
        if (left.value == 0){
            return zeroInt
        }
        val right = b.eval() as EInt
        return EInt(left.value * right.value)
    }
}
```

4 The simplest parser you can think of

To make the whole parsing easy, let us reiterate what problem are we trying to solve. We need to take a string, and convert it to the expression. It can be expressed in a test.

```
@Test()
fun testSimpleInt(){
    assertEquals(EInt(42), parse("42"))
}
```

As you might remember - we are lazy. So we will do something that would allow us to simplify things. To use a library and a format a lot of people using now. The most popular programming language of the century. YAML.

4.1 Parsing integers with yaml

To make it work we would need to add dependency to the great **snakeyaml** library

```
dependencies {
    // https://mvnrepository.com/artifact/org.yaml/snakeyaml
    implementation("org.yaml:snakeyaml:1.32")

    testImplementation(kotlin("test"))
    testImplementation("org.junit.jupiter:junit-jupiter:5.9.0")
}
```

having such a useful tool makes our lives easy.

```
import org.yaml.snakeyaml.Yaml

fun parse(input:String):Expression{
    val yaml = Yaml()
    val script = yaml.load<Any>(input)
    return EInt(script as Int)
}
```

We cannot parse anything but integers now, but at least our tests are green. Isn't that great?

4.2 Expanding yaml expressions to support addition

If you remember, our goal is to convert from strings to expressions as easy as possible. To make it happen, the language we would use should be very similar to the expressions we have.

```
EAdd(EInt(3), EInt(5))
```

might be represented in yaml as

```
add
  - 3
  - 5
```

And more complex expresison

```
EMul(EInt(10), EAdd(EInt(1), EInt(2)))
```

Will have a form

```
mul
  - 10
  - add
    - 1
    - 2
```

Obviously this is good enough if we want to use it as part of yaml files. But when you adding such an expression to the unit test, you would suffer. Suffering should be optional, and to removing it we will utilise little known feature of Yaml, called `https://yaml.org/spec/1.2.2/#chapter-7-flow-style-productions`

```
EAdd(EInt(3), EInt(5))
```

becomes

```
[add, 3, 5]
```

and

```
EMul(EInt(10), EAdd(EInt(1), EInt(2)))
```

is represented as

```
[mul, 10, [add, 1, 2]]
```

I kinda like it. It is shorter, and represents our internal structure pretty much

4.3 parsing yaml with predefined expressions

Test goes first

```
    @Test()
     fun testSimpleAdd(){
         assertEquals(EAdd(EInt(3), EInt(5)), parse("[add, 3, 5]"))
     }
```

When **snakeyaml** parses flow-style expressions, it returns ArrayList. So all we need to do is convert ArrayList to expression we want

```
fun parse(input:String):Expression{
    val yaml = Yaml()
    val script = yaml.load<Any>(input)
    return convert(script)
}
fun convert(obj:Any):Expression{
   if(obj is Int){// if yaml loaded an integer -convert it to EInt
       return EInt(obj)
   }
    if(obj is ArrayList<*>){ // if we have arrayList
        val operation= obj[0] // we assume initial element is the name of operation
        if(operation == "add"){
            val left = convert(obj[1]) // converting first parameter of the operation
            val right = convert(obj[2]) // converting second parameter of the operatio
            return EAdd(left, right)
        }
    }
    return EInt(-42) // since we don't have errors now, return something strange
}
```

Essentially what we are doing here is very similar to the eval fucntion we defined previously. Eval takes Expression as input and returns Integer values. parse takes a string as an input and returns Expression. Even conversion of addition look very similar. Pay attention to the recursive call in convert.

Of course if our source code would not have the structure we want, we will get very nasty errors. But as was said previousy - let us focus on successful flow first, taking care of corner cases later.

Surprisingly enough, this implementation makes our test green. Let's see if we try to parse more complicated expression.

```
@Test
fun testComplicatedAdd(){
    assertEquals(
        EAdd(EAdd(EInt(3), EInt(5)), EInt(42)),
        parse("[add, [add, 3,5], 42]"))
}
```

Green as well. Nice work!

4.4 Adding multiplication to the mix

Now we know what to expect, so we will add both tests from the beginning

```
@Test
fun testSimpleMultiplication(){
    assertEquals(EMul(EInt(11), EInt(33)), parse("[mul, 11, 33]"))
}
@Test
fun testComplicatedMultiplication(){
    assertEquals(
        EMul(EMul(EInt(3), EInt(5)), EInt(42)),
        parse("[mul, [mul, 3,5], 42]"))
}
```

Implementation of parser is really simple. Nothing new under the sun.

```
fun convert(obj:Any):Expression{
    ...
    if(obj is ArrayList<*>){
        ...
        if(operation =="mul"){
            val left = convert(obj[1])
            val right = convert(obj[2])
            return EMul(left, right)
        }
```

We have a parser for addition, and we have a parser for multiplication. Let's see how well they are working together.

```
@Test
fun testMulAndAdd(){
```

```
        assertEquals(
           EMul(EAdd(EInt(22), EInt(11)), EInt(44)),
           parse("[mul, [add, 22, 11], 44]"))
    }
```

Tests are green. Feels great, isn't it?

4.5 Evaluating parsed expressions

Now we have both parth of the story. We can take a string and convert it to an expression. We can take an expression and evaluate it. Let us add some more tests would confirm this fact.

```
    @Test
    fun testComplicatedExpressionWithParsing() {
        assertEquals(EInt(10), (EMul(EMul(EInt(1), EInt(5)), EInt(2))).eval())
        assertEquals(EInt(10), parse("[mul, [mul, 1, 5], 2]").eval())
        assertEquals(EInt(12), (EMul(EAdd(EInt(1), EInt(5)), EInt(2))).eval())
        assertEquals(EInt(12), parse("[mul, [add, 1, 5], 2]").eval())
    }
```

I specifically put original version here, so it is easy to compare complexity of expressions and amount of brackets.

4.6 One more touch

If you look closely on the previous test, you might wondered, why did we use **EInt(10)** instead of just **10**? How hard would that be to make it happen?

Well, let's add one more operation, and call it **unparse**. It will take an expression and convert it to our yaml format. in such a way we will have the whole circle of life

Let us extend our expression interface with **unparse** method

```
interface Expression{
    fun eval():Expression
    fun unparse():String
}
```

And expand our test cases for the parsing. Make sure that unparsing parsed expression should give the same result

```
  @Test
```

```
fun testSimpleIntUnparse(){
    assertEquals("42",EInt(42).unparse())
    assertEquals("42", parse("42").unparse())
}

@Test
fun testSimpleAddUnparse(){
    val yaml = "[add, 3, 5]"
    assertEquals(yaml, EAdd(EInt(3), EInt(5)).unparse())
    assertEquals(yaml, parse(yaml).unparse())
}
@Test
fun testComplicatedAddUnparse(){
    val yaml = "[add, [add, 3, 5], 42]"
    assertEquals(
        yaml,
        EAdd(EAdd(EInt(3), EInt(5)), EInt(42)).unparse())
    assertEquals(yaml, parse(yaml).unparse())
}
@Test
fun testSimpleMultiplicationUnparse(){
    val yaml = "[mul, 11, 33]"

    assertEquals(yaml, EMul(EInt(11), EInt(33)).unparse())
    assertEquals(yaml, parse(yaml).unparse())
}
@Test
fun testComplicatedMultiplicationUnparse(){
    val yaml = "[mul, [mul, 3, 5], 42]"
    assertEquals(
        yaml,
        EMul(EMul(EInt(3), EInt(5)), EInt(42)).unparse())
    assertEquals(yaml, parse(yaml).unparse())
}
@Test
fun testMulAndAddUnparse(){
    val yaml = "[mul, [add, 22, 11], 44]"
    assertEquals(
        yaml,
        EMul(EAdd(EInt(22), EInt(11)), EInt(44)).unparse())
```

```
        assertEquals(yaml, parse(yaml).unparse())
    }
```

Remember, the more tests you have, the less corner cases you should worry about.

implementation is straightforward.

```
data class EInt(val value:Int):Expression {
    ...
    override fun unparse(): String {
        return value.toString()
    }
}

data class EMul(val a:Expression, val b:Expression): Expression {
    ...
    override fun unparse(): String {
        return "[mul, ${a.unparse()}, ${b.unparse()}]".format()
    }
}

data class EAdd(val a:Expression, val b:Expression): Expression {
    ...
    override fun unparse(): String {
        return "[add, ${a.unparse()}, ${b.unparse()}]"
    }
}
```

As expected, tests are passing.

4.7 Evaluating parsed expressions

We took some detour to make the basics work. Now we know how to read yaml, interpret the contents of the yaml using intermediate representation, and save results back to the yaml. Which in turn can be read, interpreted and saved to yaml.

But what is missing is some tests that confirm that our assumptions are correct.

```
 @Test
    fun testSimpleIntegerWithParsing() {
```

```
    assertEquals(EInt(5), parse("5").eval())
    assertEquals(EInt(-5),parse("-5").eval())
    assertEquals(EInt(0), parse("0").eval())
}

@Test
fun testSimpleAdditionWithParsing() {
    assertEquals(EInt(5), parse("[add, 5, 0]").eval())
    assertEquals(EInt(5), parse("[add, 0, 5]").eval())
    assertEquals(EInt(10),parse("[add, 7, 3]").eval())
    assertEquals(EInt(7), parse("[add, -3, 10]").eval())
    assertEquals(EInt(6), parse("[add, 10, -4]").eval())
}

@Test
fun testComplexAdditionWithParsing() {
    assertEquals(EInt(6),  parse("[add, 1, [add, 2,3]]").eval())
}

@Test
fun testSimpleMultiplicationWithParsing() {
    assertEquals(EInt(0), parse("[mul, 0, 42]").eval())
    assertEquals(EInt(0), parse("[mul, 42, 0]").eval())
    assertEquals(EInt(42),parse("[mul, 1, 42]").eval())
    assertEquals(EInt(42), parse("[mul, 42, 1]").eval())
    assertEquals(EInt(4), parse("[mul, 2, 2]").eval())
}

@Test
fun testComplicatedMultiplicationWithParsing() {
    assertEquals(EInt(10), parse("[mul, 1, [mul, 5, 2]]").eval())
    assertEquals(EInt(12), parse("[mul, 2, [add, 5, 1]]").eval())

    assertEquals("10", parse("[mul, 1, [mul, 5, 2]]").eval().unparse())
    assertEquals("12", parse("[mul, 2, [add, 5, 1]]").eval().unparse())
}
```

In the last test you can see all the parts working together. And as you might have guessed, our tests are passing.

4.8 Food for thought

Parsing plays important role in the whole evaluation process. While we simplified it to the minimum, we still can improve it.

For example - we have duplication while parsing addition and multiplication operations. One way to make it better would be to add support to binary arithmetical operations. But that would come at the cost of complicating our abstract syntax tree. As usual in the computer science - you cannot remove the complexity. You can just move it in some other place.

Here's another puzzle for you. What should this expression be parsed into?

```
[mul, 0, [add, 11, 42]]
```

One might argue it is obvious - result is

```
EMul(EInt(0), EAdd(EInt(11), EInt(42)))
```

But equally good option is

```
EInt(0)
```

Since we know that multiplication by zero would give us zero in the end, why bother creating complex expression? And as in previous example - we are simplifiying evaluation by complicating parsing. You cannot remove the complexity. You can just push it somewhere else.

4.9 Reflecting and Wrapping up

It took some time and turns to get to this point. A lot of text, and a lot of explanations. But try to look at it from the different perspective. our parser is 26 lines long, and our interpreter is just 42 lines long. In less than 100 lines we have fully functional interpreter, that can deal with complex math expressions. And load and save expressions to yaml.

Let's look at our code in its full glory.

```
// Domain.kt
interface Expression{
    fun eval():Expression
    fun unparse():String
}
data class EInt(val value:Int):Expression {
    override fun eval(): Expression {
```

```
        return this
    }

    override fun unparse(): String {
        return value.toString()
    }
}

val zeroInt = EInt(0)
data class EMul(val a:Expression, val b:Expression): Expression {
    override fun eval(): Expression {
        val left = a.eval() as EInt
        if (left.value == 0){
            return zeroInt
        }
        val right = b.eval() as EInt
        return EInt(left.value * right.value)
    }

    override fun unparse(): String {
        return "[mul, ${a.unparse()}, ${b.unparse()}]".format()
    }
}

data class EAdd(val a:Expression, val b:Expression): Expression {
    override fun eval(): Expression {
        val left = a.eval() as EInt
        val right = b.eval() as EInt
        return EInt(left.value + right.value)

    }

    override fun unparse(): String {
        return "[add, ${a.unparse()}, ${b.unparse()}]"
    }
}
// Parser.kt
import org.yaml.snakeyaml.Yaml

fun parse(input:String):Expression{
```

```
    val yaml = Yaml()
    val script = yaml.load<Any>(input)
    return convert(script)
}
fun convert(obj:Any):Expression{
   if(obj is Int){
       return EInt(obj)
   }
    if(obj is ArrayList<*>){
        val operation= obj[0]
        if(operation == "add"){
            val left = convert(obj[1])
            val right = convert(obj[2])
            return EAdd(left, right)
        }
        if(operation =="mul"){
            val left = convert(obj[1])
            val right = convert(obj[2])
            return EMul(left, right)
        }
    }
    return EInt(-42)
}
```

5 Core language, host language and a lot of sugar

When implementing and interpreter you want to have ability to define new type of capabilites. At the same time you don't want to spend too much time refactoring base implementation. Changing Ast and tests for it, implementations, is very time consuming process.

5.1 core, host, sugared

This can be avoided by introducing new level of abstration. As you know, you can solve every problem in IT by adding an additional level of abstraction. Let's call our arimetic language we have the core language. It is core of our interpreter, and we don't want to expose its internal for the customers. We would like to keep it as small as possible, so all refatoring and optimisations would have smaller blast radius. As for ease of use we can create new

expressions, which we would call sugared expressions. They will be used as our interface, and should be converted to the core language to be interpreted

5.2 Adding sugar

Let's say we want to add new operation to the mix **sub**, which stands for the substraction. how would it fit to our idea of sugared language?

```
interface SugarExpression{
    fun desugar():Expression
}

data class SugarInt(val v:Int):SugarExpression{
    override fun desugar(): Expression {
        return EInt(v)
    }
}

data class SugarAdd(val left:SugarExpression, val right:SugarExpression) : SugarExpress
    override fun desugar(): Expression {
       return EAdd(left.desugar(), right.desugar())
    }

}

data class SugarMul(val left:SugarExpression, val right:SugarExpression): SugarExpressi
    override fun desugar(): Expression {
       return EMul(left.desugar(), right.desugar())
    }
}
```

Now it is pretty much replicating what with had with the usual expression. Main difference might be that we don't evaluate anything here. We're just converting Sugared expression to our core expressions.

5.2.1 Defining substraction

Now we can extend our sugared languge with substraction function.

```
// Take a closer look at this method
data class SugarSub(val left:SugarExpression, val right: SugarExpression):SugarExpressi
```

```
    override fun desugar(): Expression {
       return EAdd(left.desugar(),  EMul(EInt(-1), right.desugar() ))
    }
}
```

Just have a look. We don't have substraction at our core language. But our customers might never know about it. For what they care, they can call **sub** function, and it would return the expected result to them. Isnt' that nice?

But sharp-eyed reader might notice, that this implemantation would not work. Our parser returns Expressions, not SugarExpressions. We need to change that. And add few more unit tests to the mix.

our parser looks like this now

```
import org.yaml.snakeyaml.Yaml

fun parse(input:String):SugarExpression{
    val yaml = Yaml()
    val script = yaml.load<Any>(input)
    return convert(script)
}
fun convert(obj:Any):SugarExpression{
   if(obj is Int){
       return SugarInt(obj)
   }
    if(obj is ArrayList<*>){
        val operation= obj[0]
        if(operation == "add"){
            val left = convert(obj[1])
            val right = convert(obj[2])
            return SugarAdd(left, right)
        }
        if(operation =="mul"){
            val left = convert(obj[1])
            val right = convert(obj[2])
            return SugarMul(left, right)
        }
        if(operation =="sub"){
            val left = convert(obj[1])
            val right = convert(obj[2])
            return SugarSub(left, right)
```

```
        }
    }
    return SugarInt(-42)
}
```

our tests should be changed in quite a few places. For example, parsing should be updated

```
@Test
    fun testComplicatedMultiplicationUnparse(){
        val yaml = "[mul, [mul, 3, 5], 42]"
        assertEquals(
            yaml,
            SugarMul(SugarMul(SugarInt(3), SugarInt(5)), SugarInt(42)).desugar().unpars
        assertEquals(yaml, parse(yaml).desugar().unparse())
    }
    @Test
    fun testMulAndAddUnparse(){
        val yaml = "[mul, [add, 22, 11], 44]"
        assertEquals(
            yaml,
            SugarMul(SugarAdd(SugarInt(22), SugarInt(11)), SugarInt(44)).desugar().unpa
        assertEquals(yaml, parse(yaml).desugar().unparse())
    }
```

Do you see this beatuful pattern of parse -> desugar -> unparse

Of course, evaluation tests should be changed as well

```
@Test
    fun testComplicatedMultiplicationWithParsing() {
        assertEquals(EInt(10), parse("[mul, 1, [mul, 5, 2]]").desugar().eval())
        assertEquals(EInt(12), parse("[mul, 2, [add, 5, 1]]").desugar().eval())

        assertEquals("10", parse("[mul, 1, [mul, 5, 2]]").desugar().eval().unparse())
        assertEquals("12", parse("[mul, 2, [add, 5, 1]]").desugar().eval().unparse())
    }
```

they follow the same pattern more or less. parse -> desugar -> eval -> unparse

Now that our little refactoring is done, we are in a good shape to add test for our substraction.

```
@Test
fun testMulAndSubUnparse(){

    val yaml = "[mul, [sub, 22, 11], 44]"
    val desugared = "[mul, [add, [mul, -1, 22], 11], 44]"
    assertEquals(
        desugared,
        SugarMul(SugarSub(SugarInt(22), SugarInt(11)), SugarInt(44)).desugar().unpa
    assertEquals(desugared, parse(yaml).desugar().unparse())
}
```

It is a good idea to test that evaluation of substraciton works as well.

```
@Test
fun testMulAndSubEvaluationWithParsing(){
    val yaml = "[mul, [sub, 22, 11], 44]"
    assertEquals("484", parse(yaml).desugar().eval().unparse())
}
```

We've just expanded our language with one more built-in function, wihout touching the core interpretation logic. And whast is even more interesting - the result of evaluation is correct.

5.3 Adding negation operation

Now we can expand our primitive list by adding operation **neg** which should change the sight of the operation.

Since we're working with the sugared operations, we can approach to this having this definition

```
data class SugarNeg(val value:SugarExpression):SugarExpression{
    override fun desugar(): Expression {
       return EMul(EInt(-1), value.desugar())
    }

}
```

Or we might go other way around.

```
data class SugarNeg2(val value:SugarExpression):SugarExpression{
    override fun desugar(): Expression {
       return SugarSub(SugarSub(value, value), value).desugar()
```

```
    }

}
```

Second one would not be the most efficient implementation, but would reduce our dependency on the core language. As you might see there's no core language structures besides **Expression** as return value.

This is a great example of flexibility sugaring and desugaring brings to the table.

If we implemented negation first, we might change our implementation of **sub** as well.

```
data class SugarSub2(val left:SugarExpression, val right:SugarExpression):SugarExpressi
    override fun desugar(): Expression {
       return EAdd(SugarNeg(left).desugar(), right.desugar())
    }
}
```

As you can see, sugaring gives a lot of flexibility towards how our functions are implemented.

Let's finish our negation by adding some tests and parsing bits.

```
        if(operation =="neg"){
            val value = convert(obj[1])
            return SugarNeg(value)
        }
```

and test cases for the parsing

```
    @Test
    fun testNegUnparse()
    {
        val yaml = "[neg, [mul, 2, 10]]"
        val desugared = "[mul, -1, [mul, 2, 10]]"
        assertEquals(desugared, SugarNeg(SugarMul(SugarInt(2), SugarInt(10))).desugar()
        assertEquals(desugared, parse(yaml).desugar().unparse())
    }
```

and for the evaluation as well

```
    @Test
    fun testNegationEvaluation(){
        val yaml = "[neg, [mul, [sub, 22, 11], 44]]"
```

```
        assertEquals("-484", parse(yaml).desugar().eval().unparse())
    }
```

Remember, there's no such thing as too many tests, if they all make sense.

We will get back to the sugaring concept later. It is very powerful idea worth revisiting.

5.4 Food for thought

- You want to add **uminus** operation, which is operates exactly like negation, but has different representation in yaml syntax.

  ```
  [uminus, [mul, [sub, 22, 11]], 44]
  ```

  Can you think of a way to make this evaluation possible touching only parsing?

6 Adding primitive functions to the language

Now that we know how to add new functions as part of our sugaring process, we might ask ourselves : "Is it possible to define new functions without touching the host language"? Now every new defined function should have some correspondence in Kotlin. It would be nice to make it happen in yaml directly.

Let's come up with the syntax for the function definition. It might have a lot of different flavors, but all of them should take care of few elements

1. keyword for function definition
2. function name
3. function arguments
4. body of the function

We can start with the double function

```
fun   # keyword for the function definition
 - [double, x]  # function name and arguments
 - [add, x, x]  # body of the function
```

Of course short version would be preferrable for unit tests.

```
[fun, [double, x], [add, x, x]]
```

After defining **double** function, it is really tempting to introduce **quad** function

```
[fun, [quad, x], [double, [double, x]]]
```

functions should be able to call one another, otherwise what is the point in defining them?

Functions without parameters should also be possible.

```
[fun, [const42], 42]
```

We will work with functions having single argument for now, and will deal with multiple arguments representation little bit later.

6.1 Dealing with function and argument names

Now we have a good understanding how our function definitions would look like in yaml, let's think about how we gonna represent them in Kotlin.

Before we start implementing anyting, we need to decide, how we can represent function and variables names.

```
[fun, [double, x], [add, x, x]]
```

When looking into this expression from yaml perspective ‘x‘ and ‘double‘ will be represented as strings. But semantically they should be different. They should represent some identifier, or a symbol if you wish.

Therefore we need to expand our parser to support symbols.

```
data class ESymbol(val value:String):Expression{
    override fun eval(): Expression {
        return this
    }

    override fun unparse(): String {
        return value
    }
}
```

Here how our representation would look like

```
data class EFunDef(val name:ESymbol, val argument:ESymbol, val body:Expression):Expres
    override fun eval(): Expression {
        return this
    }

    override fun unparse(): String {
        return "[fun, [${name.unparse()}, ${argument.unparse()}], ${body.unparse()}]"
    }
}
```

Function definition is evaluated to itself. It is just a delcaration of the function, it doesn't do anything at all.

To make a function call, we need to have different structure. In some places it is called **function application**, i would use **function call**, since it is less academic and more commonly used

```
data class EFunCall(val name:ESymbol, val argument:Expression):Expression{
    override fun eval(): Expression {
        TODO("Not yet implemented")
    }

    override fun unparse(): String {
        return "[${name.unparse()}, ${argument.unparse()}]"
    }

}
```

We dont' know exactly how evaluation will work, but we can alreade expand our parser to support both function definitions and function calls.

6.2 Expanding parser for function definitions and calls

When we are starting new implementation, and not totally sure what expected result is, we write tests. Parsing function definition looks like a good point.

7 Pros and cons of sugaring

7.1 performance hit

7.2 simplified runtime

7.3 Haskell core language

7.4 targeting degugaring

7.5 WebAssembly desugar

7.6 python desugar

8 Errors and strange things

8.1 adding division

8.2 what should happen if we divide by zero?

8.3 It's all about environment

9 Defining variables for fun and profit

While working with the expressions, it might be useful to have some variables. for example, ability to compute something like this:

```
[mul, x, 3]
```

To make it happen, we need to do some preparations. First of all, we need to understand what 'x' means during parsing. Secondly, we should take the value attached to 'x' from somewhere.

If we try to use default yaml parser, it will say that 'x' is represented as a string. And while we are not working with strings in our language at the moment, we definitely will be in the future. So during parsing we shall convert this string to some new type of the data. Something that can be treated as identifier or variable name. Let's call it a symbol

```
data class ESymbol(val value:String):Expression{
    override fun eval(): Expression {
        return this
    }

    override fun unparse(): String {
        return value
```

```
    }
}

data class SugarSymbol(val name:String):SugarExpression{
    override fun desugar(): Expression {
       return ESymbol(name)
    }

}
```

Here's an example how it should work

```
    @Test
    fun testOperationWithSymbol(){
        assertEquals(SugarAdd(SugarSymbol("x"), SugarInt(5)),
                     parse("[add, x, 5]"))

        assertEquals(SugarAdd(SugarSymbol("x"), SugarSymbol("y")),
                     parse("[add, x, y]"))

        assertEquals(SugarAdd(
                         SugarAdd(SugarSymbol("x"), SugarSymbol("y")),
                        SugarSymbol("z")),
            parse("[add, [add, x, y], z]"))

    }
```

Now we have a way to introduce variables to the expressions, but how are we planning to evaluate them? Which brings us to the next question

9.1 Where do our variables live?

Variables should be defined somewhere. There should be a binding between symbol, and the value that symbol represents.

The easiest way to make this mapping would be a dictionary. Or a hashmap if you like.

```
data class Environment(val bindings:HashMap<ESymbol, Expression>){
    fun addBinding( name:ESymbol,  value:Expression){
        bindings[name] = value
    }
```

```
    fun isDefined(name:ESymbol): Boolean{
        return bindings.containsKey(name)
    }
    fun get(name:ESymbol):Expression{
        return bindings[name]!!
    }
}
```

As you can see we are assuming that we use core language as part of bindings. Since our evaluation process works on the core language, not the sugared one, this is very logical thing to do.

Test cases are following:

```
    @Test
    fun isDefined() {
        val env = Environment(hashMapOf())
        assertFalse(env.isDefined(ESymbol("x")))
        val env2 = Environment(hashMapOf( Pair(ESymbol("x"), EInt(5))))
        assertTrue(env2.isDefined(ESymbol("x")))
    }
    @Test
    fun addBinding() {
        val env = Environment(hashMapOf())
        env.addBinding(ESymbol("x"), EInt(5))
        assertTrue(env.isDefined(ESymbol("x")))
        assertEquals(EInt(5), env.bindings[ESymbol("x")])
        assertEquals(EInt(5), env.get(ESymbol("x")))

    }
```

What would happen if you try to get the variable that is not defined in the environment? You will get a runtime error of course. Error handling will be taken care of later. I promise :) Let us focus on well-behaving programs, where nothing strange happens.

9.2 Using environment in the evaluation

We know how to define variable, and bind values to them. But it wont' be useful until we can attach this environment to our evaluation.

```
## somewhere out here we should have an environmet with defined 'x'
[add, x, 5]
```

The only place where we can plug it in is our **eval** function. Time for the improvement

Remeber our **Expression** interface from very long time ago?

```
interface Expression{
    fun eval():Expression
    fun unparse():String
}
```

We will tweak it a little bit

```
interface Expression{
    fun eval(env:Environment):Expression
    fun unparse():String
}
```

We're claiming here that evaluation cannot happen in vacuum. We need to take into account an environment we're in. Sometimes it might be an empty environment, somethimes it might be full of variable definitions.

This is a small step for our interface, but a huge impact for our interpreter. It breaks all our evaluation implementations. It breaks all our evaluation tests. Good news is that we have tests, so we know what is broken. What else can be done rather than fixing it?

9.3 Fixing evaluation with environment

Our first case is really simple one.

```
data class EInt(val value:Int):Expression {
    override fun eval(): Expression {
        return this
    }
    ...
}
```

How shall we interpret integer values if they live in some environment? Easy. Integers are always integers, so we can simply ignore the environment.

```
data class EInt(val value:Int):Expression {
    override fun eval(env:Environment): Expression {
        return this
    }

    override fun unparse(): String {
        return value.toString()
    }
}
```

Done.

With addition and multiplication it would be not as straighforward.

Let's stop and think for a moment. What we would like to happen when we have an expression defined within environment?

```
## env: {x = 5}
[add, x, 10]
```

I would say it should be converted to

```
[add, 5, 10]
```

We don't care about the variable name. We care only about value attached to it. Its binding.

Having variable name, variable definition and expression where this variable exists, we want to substitute variable with its value.

```
fun substitute(variableName:ESymbol), env:Environment, expession:Expression): Expressio
```

If you're thinking this function should be part of the **Expression** interface, in tend to agree with you.

Our poor **Expression** is getting bigger and bigger, but it is a good things.

```
interface Expression{
    fun eval(env:Environment):Expression
    fun substitute(symbol:ESymbol, env:Environment):Environment
    fun unparse():String
}
```

Now it breaks even more things in our code. But this is the price we need to pay.

```
data class EInt(val value:Int):Expression {
    override fun eval(env:Environment): Expression {
        return this
    }

    override fun substitute(symbol: ESymbol, env: Environment): Expression {
        return this
    }

    override fun unparse(): String {
        return value.toString()
    }
}
```

for integers substituion is as easy as it gets. We are just ignoring all the symbols and environment.

Let's see what we can do with **ESymbol**.

```
data class ESymbol(val name:String):Expression{
    ...
    override fun substitute(symbol: ESymbol, env: Environment): Expression {
        if(symbol.name == name){
            return env.get(symbol)
        }
        return this
    }
   ...
}
```

if we are trying to substitue symbol, and it is definted in the environment, we will return record from the variable.

It might sound little bit confusing so here are few examples.

```
# env: {x = 5}
x
```

should be transformed to

```
5
```

And if variable is not bound in the environment

```
# env: {x = 5}
y
```

should remain untouched

```
y
```

Now that we've done with substitution for the symbols, addition and multiplication should be simple enough.

```
data class EAdd(val a:Expression, val b:Expression): Expression {
    ...
    override fun substitute(symbol: ESymbol, env: Environment): Expression {
        return EAdd(a.substitute(symbol, env), b.substitute(symbol,env))
    }
    ...
}

data class EMul(val a:Expression, val b:Expression): Expression {
    ...
    override fun substitute(symbol: ESymbol, env: Environment): Expression {
        return EMul(a.substitute(symbol, env), b.substitute(symbol, env))
    }
    ...
}
```

As usual, our tests would show us an example what we expect to happen, and how the variable substitution should look like.

```
@Test
    fun testSubstitution(){
        val env1 = Environment(hashMapOf(Pair(ESymbol("x"), EInt(5))))
        assertEquals(EInt(5), ESymbol("x").substitute(ESymbol("x"), env1))
        assertEquals(ESymbol("y"), ESymbol("y").substitute(ESymbol("x"), env1))

        assertEquals("[add, 5, 5]",
            parse("[add, x, x]").desugar()
                .substitute(ESymbol("x"), env1).unparse())

        assertEquals("[mul, 5, 5]",
```

```
        parse("[mul, x, x]").desugar()
            .substitute(ESymbol("x"), env1).unparse())

    assertEquals("[add, [mul, 5, 5], 5]",
        parse("[add, [mul, x, x], x]").desugar()
            .substitute(ESymbol("x"), env1).unparse())


    val env2 = Environment(hashMapOf(Pair(ESymbol("x"), EInt(5)),
        Pair(ESymbol("y"), EInt(7))))
        parse("[add, [mul, x, y], x]").desugar()
            .substitute(ESymbol("x"), env2).unparse())


    assertEquals("[add, [mul, 5, 7], 5]",
        parse("[add, [mul, x, y], x]").desugar()
            .substitute(ESymbol("x"), env2)
            .substitute(ESymbol("y"), env2)
            .unparse())

    val complexEnv = Environment(hashMapOf(Pair(ESymbol("x"),
        parse("[mul, 11, [add, [add, 3, 4], 11]]").desugar())))

    assertEquals("[add, [mul, 11, [add, [add, 3, 4], 11]], [mul, 11, [add, [add, 3
        parse("[add, x, x]").desugar()
            .substitute(ESymbol("x"), complexEnv)
            .unparse()
        )
}
```

As you can see, substitutions can be as complex as we want them to be. Now it is good time to do something about evaluation.

9.4 improving taking into account the variables.

At this point we know how to substitute variables, and we also know how to evaluate expressions. So mixing these two together we will get the evaluation process.

```
    assertEquals("10",
        parse("[add, x, x]").desugar()
```

```
            .substitute(ESymbol("x"), env1)
            .eval(env1).unparse())

    assertEquals("25",
        parse("[mul, x, x]").desugar()
            .substitute(ESymbol("x"), env1)
            .eval(env1)
            .unparse())

    assertEquals("30",
        parse("[add, [mul, x, x], x]").desugar()
            .substitute(ESymbol("x"), env1)
            .eval(env1)
            .unparse())
```

these examples would work, evaluation is simple and straghtforward

```
    assertEquals("[add, [mul, 5, y], 5]",
        parse("[add, [mul, x, y], x]").desugar()
            .substitute(ESymbol("x"), env2)
            .eval(env1)
            .unparse())
```

and this will error out. Because we cannot evaluate the symbol ' y'

As you can see for the evaluation it is really important to substitute all the variables before we do the evaluation. Otherwise we would get as what is know as 'variable not found' exception.

this is easily fixable. We just need to substitute 'y' as well

```
    assertEquals("[add, [mul, 5, y], 5]",
        parse("[add, [mul, x, y], x]").desugar()
            .substitute(ESymbol("x"), env2)
            .substitute(ESymbol("y"), env2)
            .eval(env1)
            .unparse())
```

Now it is green.

In you looked carefully, you might see that substitution uses env2 for both variables, while evaluation uses 'env1'. This shows us flexibility the

interpreter shows. But at the same time it reminds us how careful we should be about passing the correct environment.

Some might ask - why do we need to pass environment during the substitugion phase? Can't it be taken from the environment during the evaluation phase? Very good question. It could be done that way, but you well need to do foreach replacement for every variable in the environment. While the expression itself might use only oe or two of them. Or manybe none at all.

9.5 What if our variables are more complex than simple constants?

Prevously our values for 'x' and 'y' were simple constants. But what if we want them to be expressions? What needs to be changed, and how much effort would it require to make evaluation aware of more complicated expressions?

Let us start with the simple one

```
# env : {x = [add, 40,2]}
[add, x, x]
```

Actually we have several options