STRUMENTA

FEDERICO TOMASSETTI

BUILDING LANGUAGES USING KOTLIN

WHAT WE ARE GOING TO SEE IN THIS TALK

- The process to create practically usable languages
- Evaluate the effort needed for each step
- Discuss how Kotlin makes the life of language designers easier

WHAT WE ARE NOT GOING TO SEE IN THIS TALK

- We are not going to discuss in detail each step
  - ***but*** I will give you the code
- We are not going to discuss language design
  - ***but*** I will share resources about that

WHY BUILDING LANGUAGES?

- General Purpose Languages:
  - New paradigms (Haskell)
  - Pragmatic languages (Kotlin)
- For learning how languages work
- Because you can brag about that

WHY BUILDING LANGUAGES?

- General Purpose Languages:
  - New paradigms (Haskell)
  - Pragmatic languages (Kotlin)
  - For learning how languages work
  - Because you can brag about that
- ***Or Domain Specific Languages...***

DOMAIN SPECIFIC LANGUAGES

Real DSLs, not internal DSLs!

```
<html>
  <head>
    <title>My beautiful page</title>
  </head>
<body>
  <div id="main">
    <h1>Really interesting title!</h1>
    <p>And the content is even better!</p>
  </div>
</body>
</html>
```

HTML

```
DFF : process(RST, CLK) is
begin
  if RST = '1' then
    Q <= '0';
  elsif rising_edge(CLK) then
    Q <= D;
  end if;
end process DFF;
```

VHDL

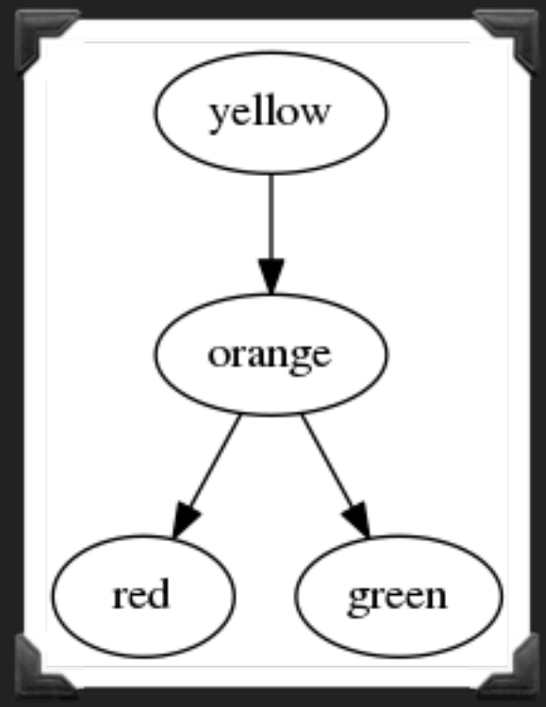
```
SELECT MAX(value) FROM TEMPERATURES WHERE city="Atlanta" AND month="August";
```

SQL

DOMAIN SPECIFIC LANGUAGES

Real DSLs, not internal DSLs!

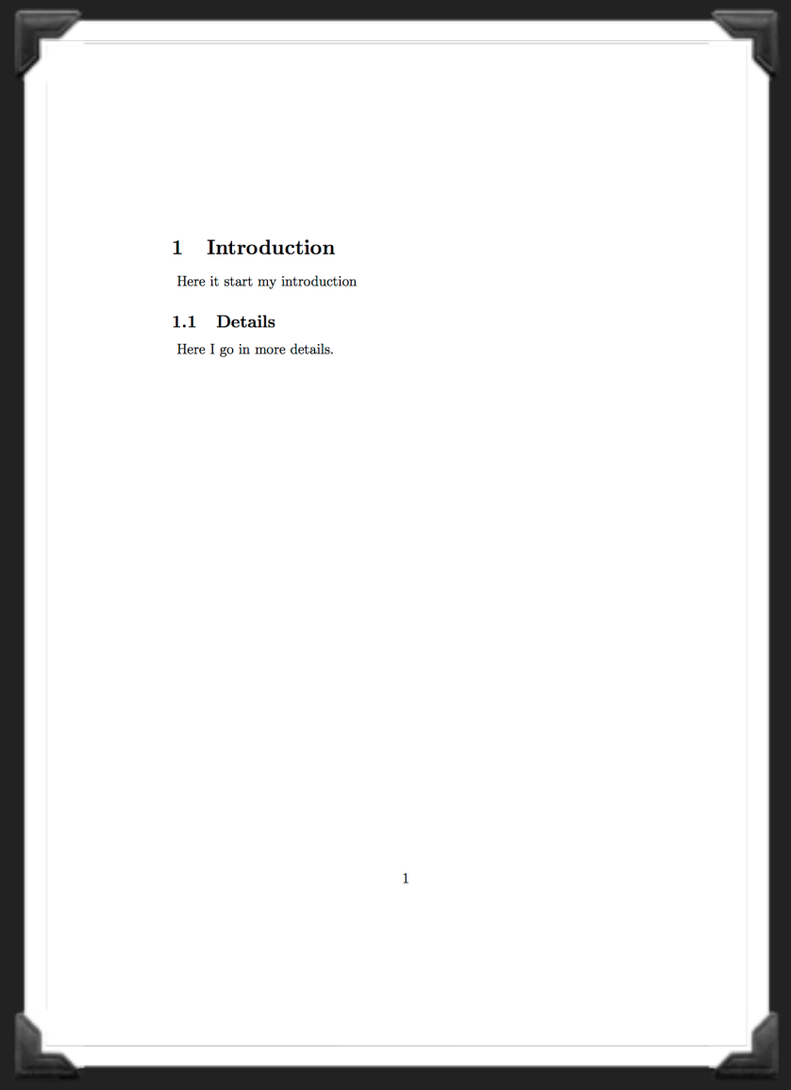
```
digraph graphname {

 yellow -> orange -> red;

 orange -> green;

}
```



DOMAIN SPECIFIC LANGUAGES

Real DSLs, not internal DSLs!

```
\documentclass[12pt]{article}
\usepackage{lingmacros}
\usepackage{tree-dvips}
\begin{document}
\section{Introduction}
Here it start my introduction
\subsection{Details}
Here I go in more details.
\end{document}
```



1 Introduction

Here it start my introduction

1.1 Details

Here I go in more details.

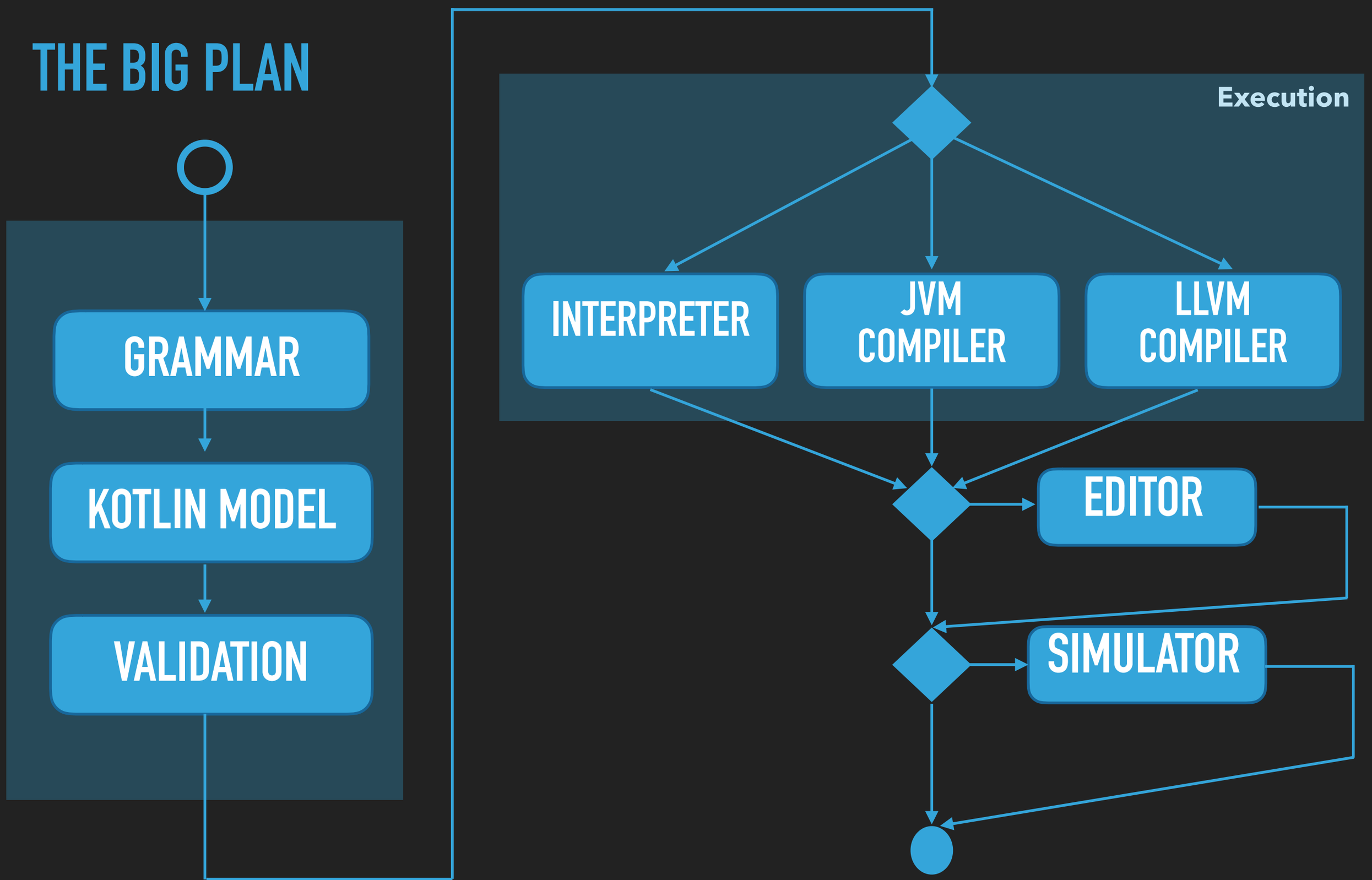
1

DOMAIN SPECIFIC LANGUAGES

Real DSLs, not internal DSLs!

- Productivity improvements
  - Up to 20 times more productive…
- Involving domain experts
  - A language that can be understood by developers *and* medical doctors, accountants or mechanical engineers

THE BIG PLAN


GRAMMAR
KOTLIN MODEL
VALIDATION
Execution
INTERPRETER
JVM COMPILER
LLVM COMPILER
EDITOR
SIMULATOR

OUR EXAMPLE

A simple language to define functions

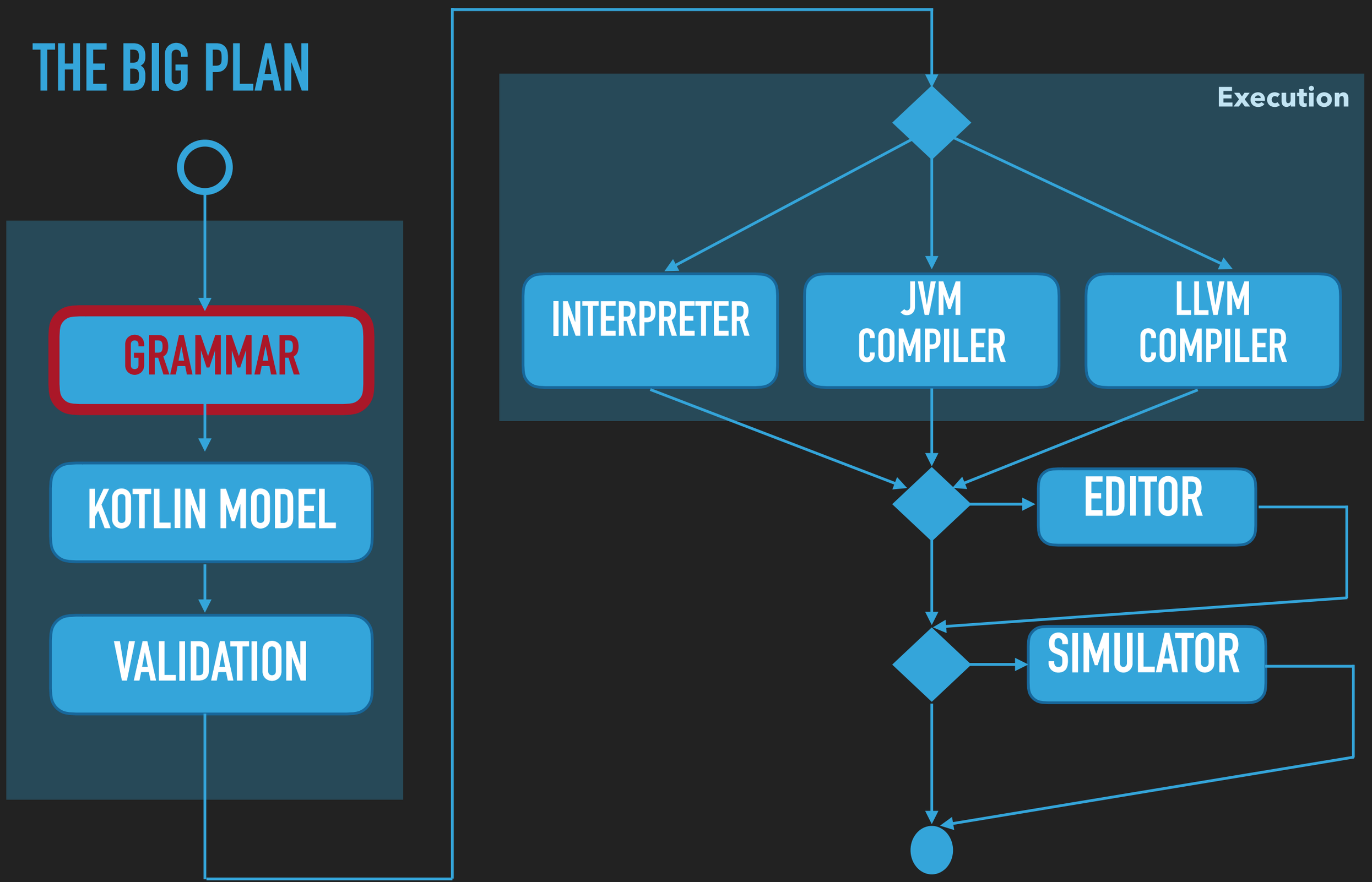
Kanvas

File

My Functions

```
1  fun double(Int p) Int {
2    2 * p
3  }
4
5  fun sumOne(Int v) Int {
6    v + 1
7  }
8
9  var a = 7
10 print(sumOne(double(10 - a))
```

THE BIG PLAN


GRAMMAR
KOTLIN MODEL
VALIDATION
Execution
INTERPRETER
JVM COMPILER
LLVM COMPILER
EDITOR
SIMULATOR

DEFINING THE GRAMMAR

THE CODE IN MY LANGUAGE

```
fun double(Int p) Int {
  2 * p
}

fun sumOne(Int v) Int {
  v + 1
}

var a = 7
print(sumOne(double(10 - a))
```

THE SINGLE TOKENS

fun	FUN
double	ID
(	LPAREN
Int	INT
p	ID
)	RPAREN
Int	INT
{	LBRACE
2	INT_LIT
*	MUL

THE CONSTRUCTS INSTANCES

INT_LIT, MUL, ID
2 * p
multiplication

INT, ID
Int p
parameter

VAR, ID, EQUAL, INT_LIT
var a = 7
varDeclaration

DEFINING THE GRAMMAR

THE CODE IN MY LANGUAGE

```
fun double(Int p) Int {
  2 * p
}

fun sumOne(Int v) Int {
  v + 1
}

var a = 7
print(sumOne(double(10 - a))
```

LEXER

THE SINGLE TOKENS

fun	FUN
double	ID
(	LPAREN
Int	INT
p	ID
)	RPAREN
Int	INT
{	LBRACE
2	INT_LIT
*	MUL

PARSER

THE CONSTRUCTS INSTANCES

INT_LIT, MUL, ID

2 * p

multiplication

ID

parameter

VAR, ID, EQUAL, INT_LIT

var a = 7

varDeclaration

DEFINING THE GRAMMAR: PHASE I THE LEXER


```
lexer grammar MiniCalcLexer;

channels { WHITESPACE }

// Whitespace
NEWLINE                 : '\r\n' | '\r' | '\n' ;
WS                      : [\t ]+ -> channel(WHITESPACE) ;

// Keywords
INPUT                   : 'input' ;
VAR                     : 'var' ;
PRINT                   : 'print';
AS                      : 'as';
INT                     : 'Int';
DECIMAL                 : 'Decimal';
STRING                  : 'String';
FUN                     : 'fun';

// Identifiers
ID                      : [_]*[a-z][A-Za-z0-9_]* ;
```

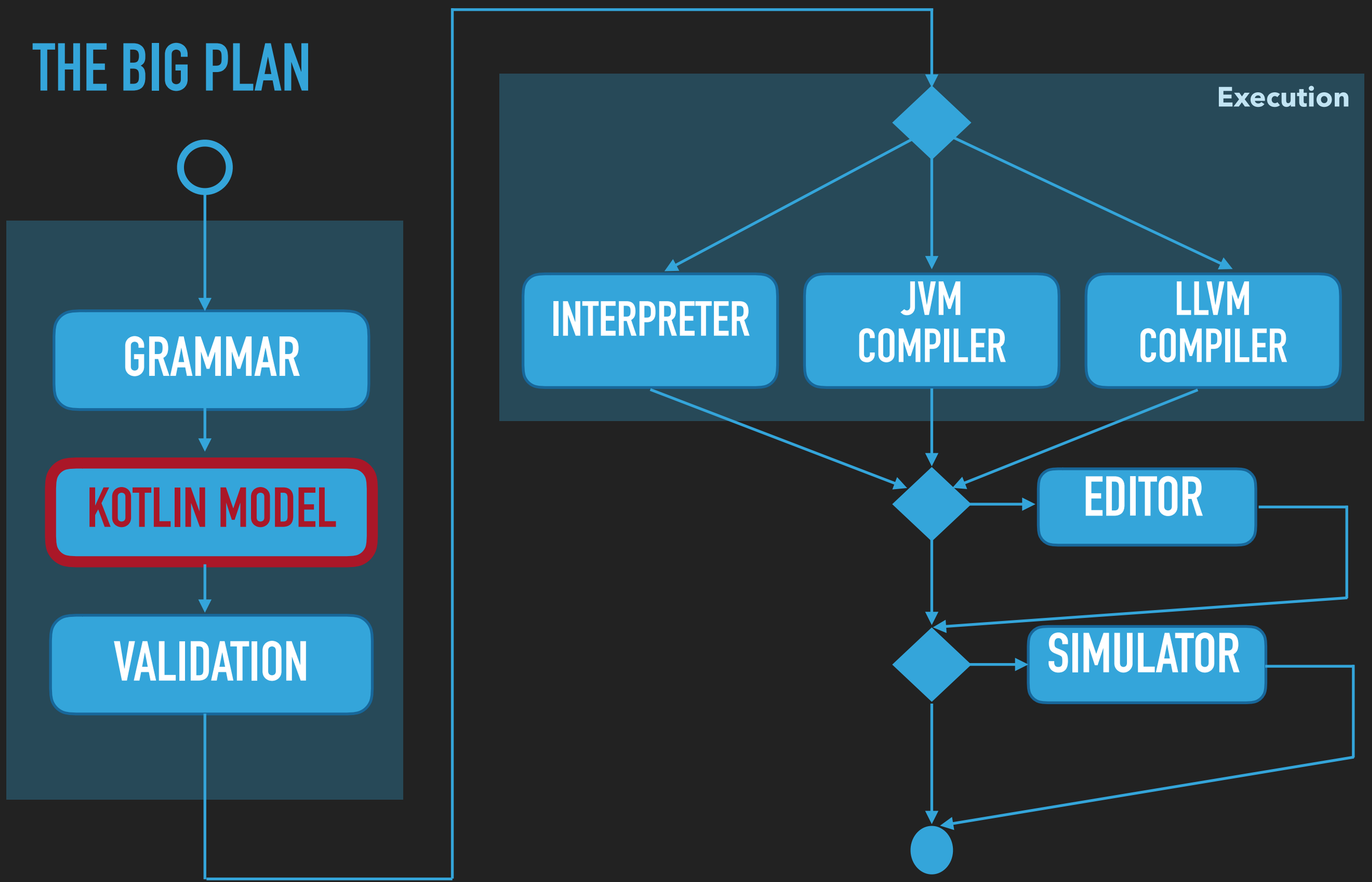
50 – 80 LINES

DEFINING THE GRAMMAR: PHASE II THE PARSER


```
parser grammar MiniCalcParser;

options { tokenVocab=MiniCalcLexer; }

miniCalcFile : lines=line+ ;

line       : statement (NEWLINE | EOF) ;

statement : inputDeclaration # inputDeclarationStatement
          | varDeclaration   # varDeclarationStatement
          | assignment       # assignmentStatement
          | print            # printStatement
          | expression       # expressionStatement
          | function         # functionStatement;

function : FUN name=ID LPAREN
           (params+=param (COMMA params+=param)*)?
           RPAREN type LBRACE NEWLINE
           (statements+=statement NEWLINE)* RBRACE;

param : type name=ID ;

print : PRINT LPAREN expression RPAREN ;

inputDeclaration : INPUT type name=ID ;

varDeclaration : VAR assignment ;

assignment : ID ASSIGN expression ;
```

40 – 50 LINES

THE BIG PLAN


GRAMMAR
KOTLIN MODEL
VALIDATION
Execution
INTERPRETER
JVM COMPILER
LLVM COMPILER
EDITOR
SIMULATOR

DEFINING THE MODEL

```
data class VarDeclaration(override val name: String?,
                          val value: Expression?,
                          override val position: Position? = null)
    : Statement, ValueDeclaration

data class InputDeclaration(override val name: String?,
                            val type: Type?,
                            override val position: Position? = null)
    : Statement, ValueDeclaration

data class Assignment(val varDecl: ReferenceByName<ValueDeclaration>,
                      val value: Expression?,
                      override val position: Position? = null)
    : Statement
```

120 - 130 LINES

DEFINING THE MODEL

```
data class MiniCalcFile(override val statements : List<Statement>,
                        override val position: Position? = null)
    : Node, ScopeDelimiter, StatementsContainer {

    @Derived val inputs
        get() = statements.filterIsInstance(InputDeclaration::class.java)

    @Derived val topLevelFunctions
        get() = statements.filterIsInstance(FunctionDeclaration::class.java)

    @Derived val topLevelVariables
        get() = statements.filterIsInstance(VarDeclaration::class.java)
}
```

120 – 130 LINES

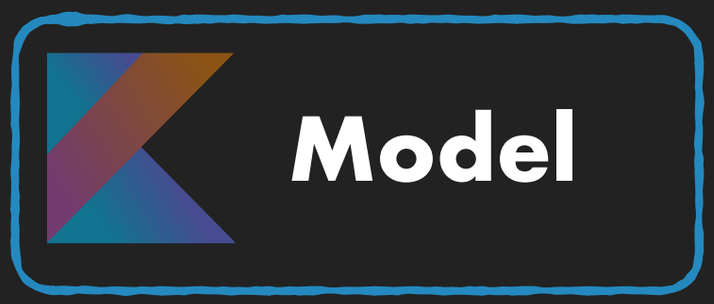
MAPPING ANTLR OUTPUT TO THE MODEL

```
fun BinaryOperationContext.toAst(
        considerPosition: Boolean = false) : Expression =
    when (operator.text) {
        "+" -> SumExpression(left.toAst(considerPosition),
                right.toAst(considerPosition), toPosition(considerPosition))
        "-" -> SubtractionExpression(left.toAst(considerPosition),
                right.toAst(considerPosition), toPosition(considerPosition))
        "*" -> MultiplicationExpression(left.toAst(considerPosition),
                right.toAst(considerPosition), toPosition(considerPosition))
        "/" -> DivisionExpression(left.toAst(considerPosition),
                right.toAst(considerPosition), toPosition(considerPosition))
    else -> throw UnsupportedOperationException(this.javaClass.canonicalName)
}
```

70 - 90 LINES

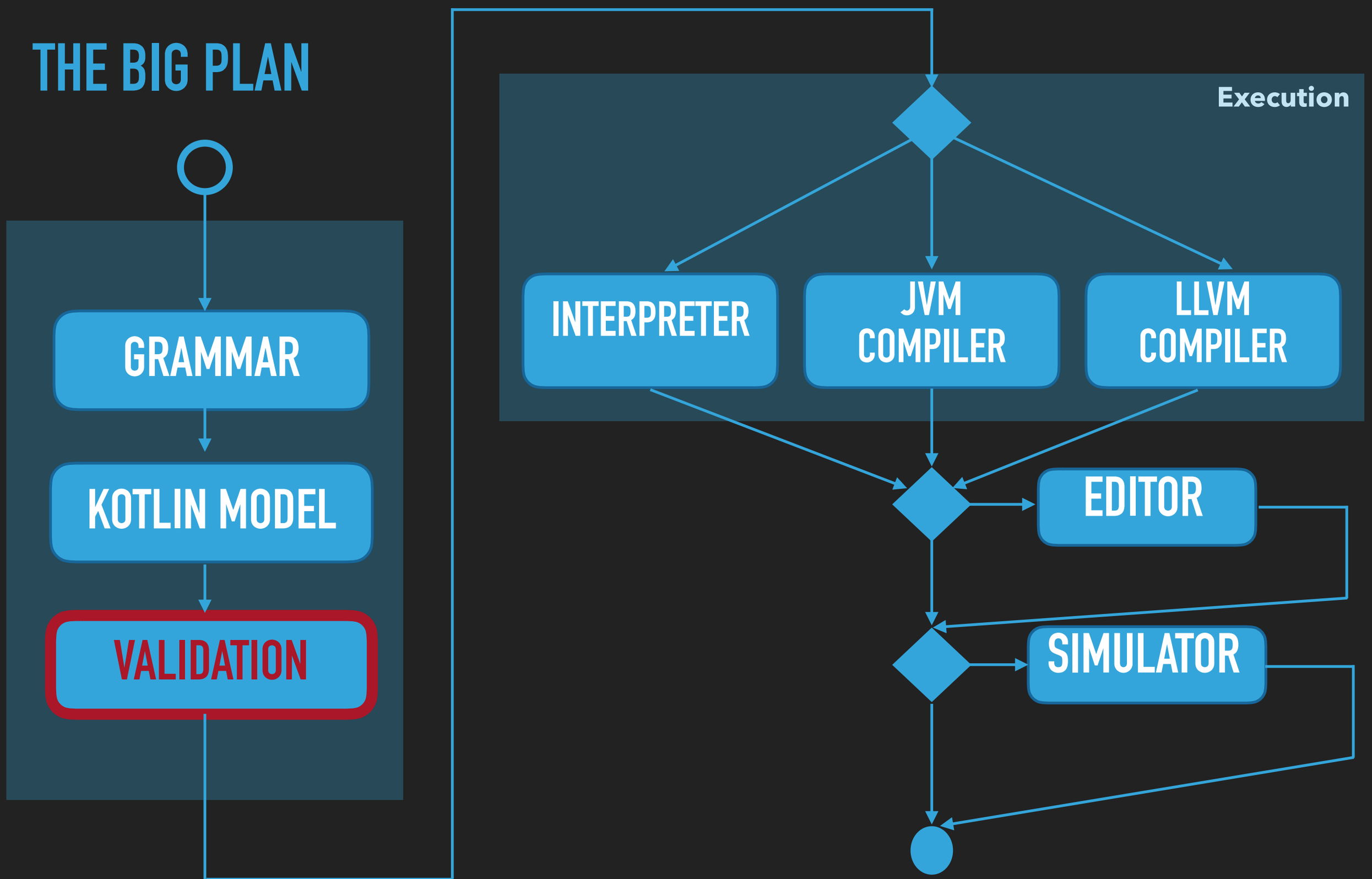
MAPPING ANTLR OUTPUT TO THE MODEL


```
fun BinaryOperationContext.toAst(
        considerPosition: Boolean = false) : Expression =
    when (operator.text) {
        "+" -> SumExpression(left.toAst(considerPosition),
                right.toAst(considerPosition), toPosition(considerPosition))
        "-" -> SubtractionExpression(left.toAst(considerPosition),
                right.toAst(considerPosition), toPosition(considerPosition))
        "*" -> MultiplicationExpression(left.toAst(considerPosition),
                right.toAst(considerPosition), toPosition(considerPosition))
        "/" -> DivisionExpression(left.toAst(considerPosition),
                right.toAst(considerPosition), toPosition(considerPosition))
    else -> throw UnsupportedOperationException(this.javaClass.canonicalName)
}
```



70 - 90 LINES

THE BIG PLAN


GRAMMAR
KOTLIN MODEL
VALIDATION
Execution
INTERPRETER
JVM COMPILER
LLVM COMPILER
EDITOR
SIMULATOR

VALIDATION

- Now all information is expressed in my model
- The model is composed by instances of data classes
- The model is a tree: each element is a Node. The root node represent the whole file, contains top-level elements (e.g., functions), which contain smaller and smaller elements until we reach the leaves
- To do validation we check properties on the tree

VALIDATION – GENERIC FUNCTIONS

```
fun Node.process(operation: (Node) -> Unit) {
    operation( p1: this)
    this.javaClass.kotlin.memberProperties
            .filter { it.findAnnotation<Derived>() == null }
            .forEach { p ->
        val v = p.get(this)
        when (v) {
            is Node -> v.process(operation)
            is Collection<*> -> v.forEach { (it as? Node)?.process(operation) }
        }
    }
}

fun <T: Node> Node.specificProcess(klass: Class<T>, operation: (T) -> Unit) {
    process { if (klass.isInstance(it)) {
        operation(it as T) }
    }
}
```

VALIDATION

```
// check a variable is not duplicated
val varsByName = HashMap<String, VarDeclaration>()
this.specificProcess(VarDeclaration::class.java) {
    if (varsByName.containsKey(it.name!!)) {
        errors.add(Error( message: "A variable '${it.name}' has been already declared",
                it.position!!))
    } else {
        varsByName[it.name] = it
    }
}
```

60 - 100 LINES

VALIDATION

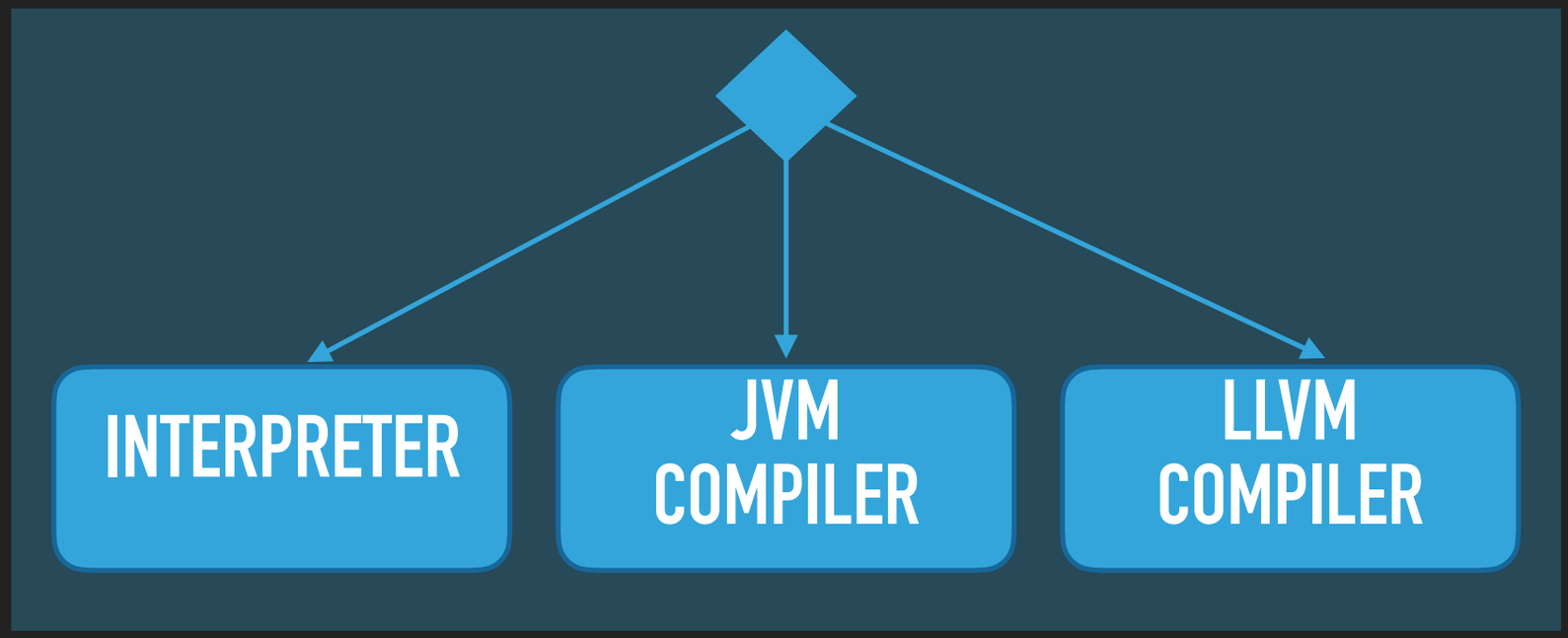
```
// check assignments use compatible types
this.specificProcess(Assignment::class.java) {
    if (it.varDecl.resolved) {
        val actualType = it.value!!.type()
        val formalType = it.varDecl.referred!!.type()
        if (!formalType!!.isAssignableBy(actualType)) {
            errors.add(Error( message: "Cannot assign $actualType to $formalType",
                    it.position!!))
        }
    }
}
```

60 - 100 LINES

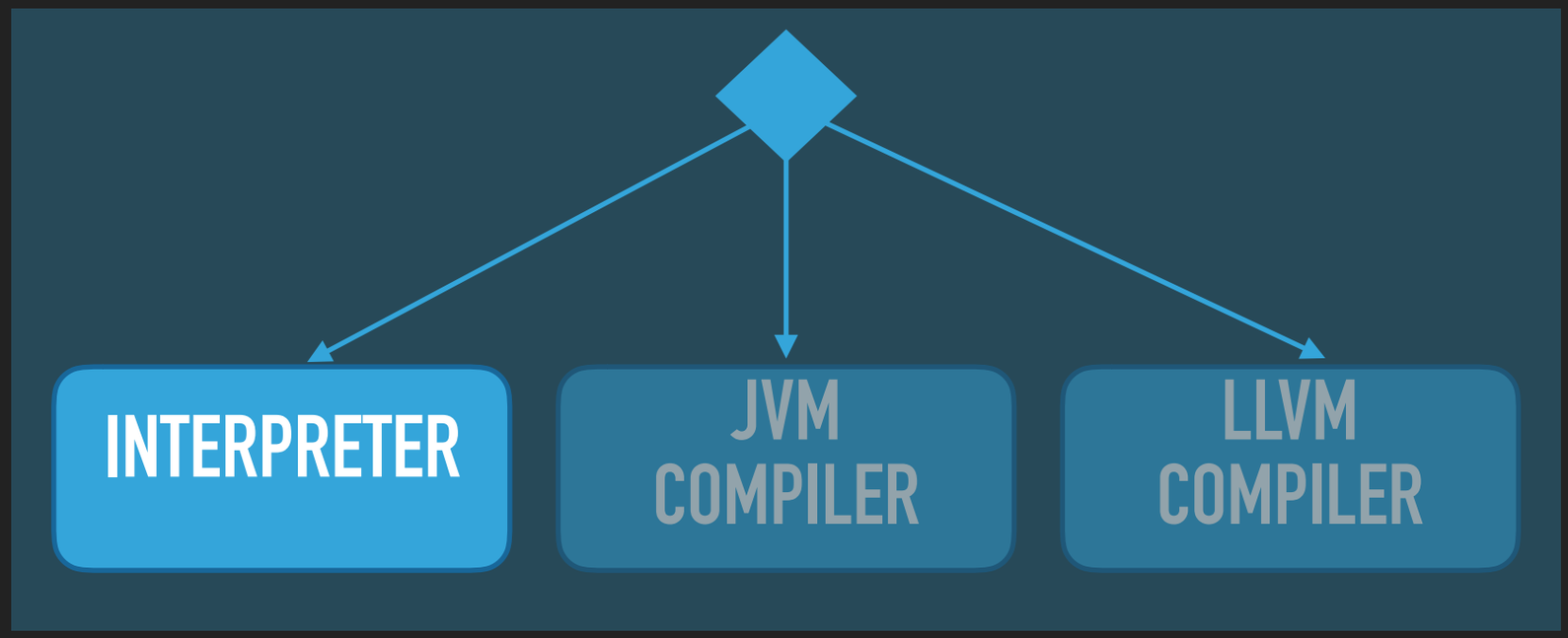
KOTLIN SO FAR

- Data classes and custom properties in the model
- Method extensions and when operator in the mapping
- Navigating a tree is easier with lambdas
- Because of extensions methods I can organize my code by aspects (e.g., all extension methods regarding validation in one file)

COMPILER OR INTERPRETER?

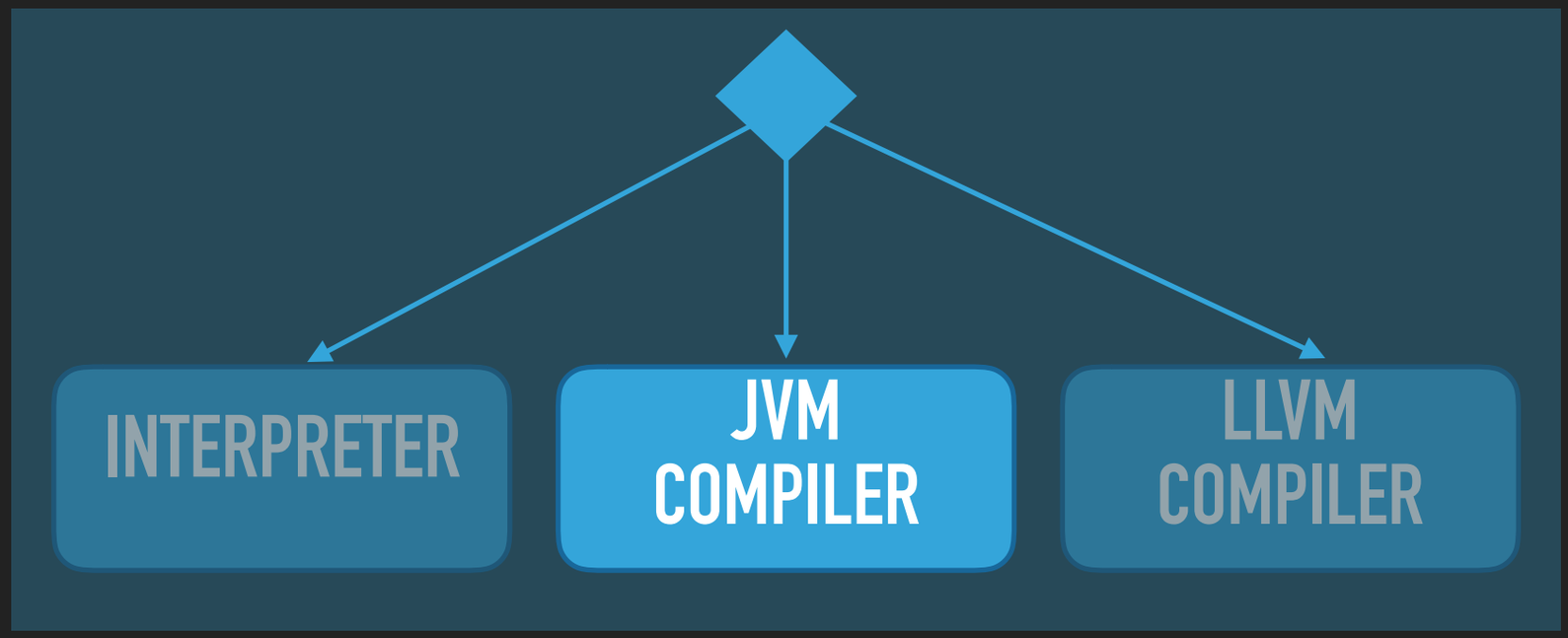


COMPILER OR INTERPRETER?



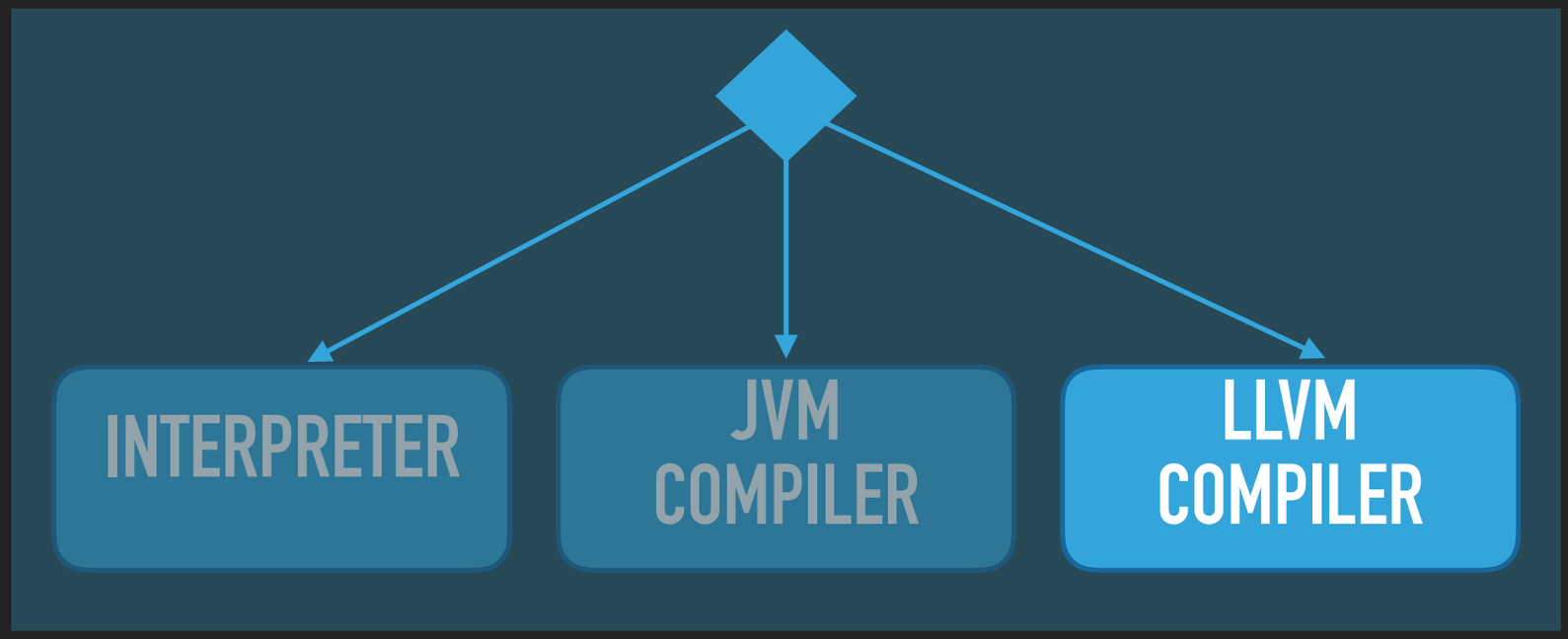
A program that navigates the model and depending on what it find does something like performing a calculation or add a value in a symbol table

COMPILER OR INTERPRETER?



A program that navigates the model and depending on what it find it produce different JVM bytecode

COMPILER OR INTERPRETER?



A program that navigates the model and depending on what it find it produce different LLVM bitcode

INTERPRETER

```
private fun executeStatement(
        statement: Statement,
        symbolTable: CombinedSymbolTable<Any, FunctionDeclaration>) : Any? =
    when (statement) {
        is ExpressionStatement -> evaluate(
                statement.expression!!, symbolTable)
        is VarDeclaration -> symbolTable.setValue(
                statement.name!!, evaluate(statement.value!!, symbolTable))
        is Print -> systemInterface.print(
                evaluate(statement.value!!, symbolTable).toString())
        is Assignment -> symbolTable.setValue(
                statement.varDecl.name, evaluate(statement.value!!, symbolTable))
        is FunctionDeclaration -> symbolTable.setFunction(
                statement.name!!, statement)
        is InputDeclaration -> symbolTable.setValue(
                statement.name!!, inputValues[statement.name]!!)
        else -> throw UnsupportedOperationException(
                statement.javaClass.canonicalName)
    }
```

110 – 140 LINES

INTERPRETER

```
private fun evaluate(expression: Expression,
                     symbolTable: CombinedSymbolTable<Any, FunctionDeclaration>)
        : Any =
    when (expression) {
        is IntLit -> expression.value.toInt()
        is DecLit -> expression.value.toDouble()
        is StringLit -> expression.parts
                .map { it.evaluate(symbolTable) }
                .joinToString(separator = "")
        is ValueReference -> symbolTable.getValue(expression.ref!!.name)
        is SumExpression -> {
            val l = evaluate(expression.left!!, symbolTable)
            val r = evaluate(expression.right!!, symbolTable)
            when (l) {
                is Int -> l + r as Int
                is Double -> l + r as Double
                is String -> l + r.toString()
                else -> throw UnsupportedOperationException(
                        l.toString() + " from evaluating " + expression.left)
            }
        }
```

110 – 140 LINES

JVM COMPILER

```
is MultiplicationExpression -> {
    val lt = this.left!!.type()
    val rt = this.right!!.type()
    if (lt is IntType && rt is IntType) {
        this.left.pushAsInt(methodVisitor, context)
        this.right.pushAsInt(methodVisitor, context)
        methodVisitor.visitInsn(IMUL)
    } else if (lt is NumberType && rt is NumberType) {
        this.left.pushAsDouble(methodVisitor, context)
        this.right.pushAsDouble(methodVisitor, context)
        methodVisitor.visitInsn(DMUL)
    } else {
        throw UnsupportedOperationException(lt.toString()
                + " from evaluating " + this.left)
    }
}
```

350 – 450 LINES

JVM COMPILER

Functions that produce bytecode so that the result will be on top of the stack

```
is MultiplicationExpression -> {
    val lt = this.left!!.type()
    val rt = this.right!!.type()
    if (lt is IntType && rt is IntType) {
        this.left.pushAsInt(methodVisitor, context)
        this.right.pushAsInt(methodVisitor, context)
        methodVisitor.visitInsn(IMUL)
    } else if (lt is NumberType && rt is NumberType) {
        this.left.pushAsDouble(methodVisitor, context)
        this.right.pushAsDouble(methodVisitor, context)
        methodVisitor.visitInsn(DMUL)
    } else {
        throw UnsupportedOperationException(lt.toString()
                + " from evaluating " + this.left)
    }
}
```

Bytecode Instructions

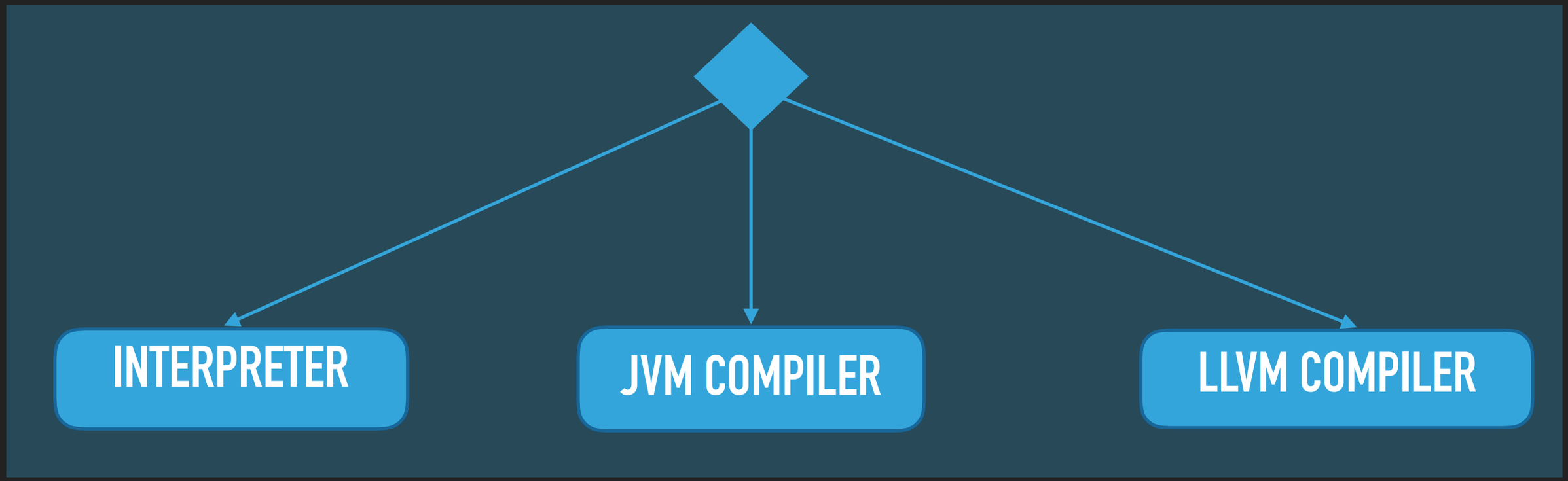
350 – 450 LINES

LLVM COMPILER

```
is MultiplicationExpression -> {
    when (this.type()) {
        is DecimalType -> {
            blockBuilder.tempValue(
                    FloatMultiplication(
                            toDecimalValue(blockBuilder,
                                    symbolTable, left!!),
                            toDecimalValue(blockBuilder,
                                    symbolTable, right!!)))
                    .reference()
        }
        is IntType -> {
            blockBuilder.tempValue(
                    IntMultiplication(
                            left!!.llvmValue(blockBuilder,
                                    symbolTable),
                            right!!.llvmValue(blockBuilder,
                                    symbolTable)))
                    .reference()
        }
        else -> throw UnsupportedOperationException(this.toString())
    }
}
```

550 - 650 LINES

COMPILER OR INTERPRETER?



No framework, simple structure based on symbol tables

ASM library to generate JVM bytecode

KLLVM library to generate LLVM bitcode

EDITOR

- Validation
  - Show syntax errors
  - Show semantic errors
- Syntax highlighting
- Auto-completion

EDITOR

- Validation
  - Show syntax errors *obtained by the parser*
  - Show semantic errors *obtained by validation*
- Syntax highlighting *obtained by the lexer*
- Auto-completion *this is a little tricky*

SIMULATOR

- Basically an interpreter wrapped by an UI
- Additional features possible
- Time manipulation
- Customizable skins
- Advanced logging/representation of internal data

WHY KOTLIN CHANGED HOW I BUILD LANGUAGES

- Without Kotlin I had to use Language Workbenches because the effort was too high
- Kotlin makes possible to express concisely every aspect of the language
- This solution gives us much more flexibility
- I can be very productive: you can build simple languages with parser, interpreter, an editor for ~ 1K LOC

DO YOU WANT TO LEARN MORE?

- All the code
- The slides
- A coupon for a book on building languages using Kotlin

https://tomassetti.me/kotlinconf