MiniC

Leonardo Focardi, Jan 2023

Project description

- MiniC is a interpreter for a little set of C language
- It is implemented by ANTLR
- The interpreter is written in Kotlin
- The application have a GUI

Architecture

- ANTLRFiles: contains the grammar files
- Model: all the file to interpretate code (Lexer, Parser, Evaluator)
- View: contains the file of the GUI
- Controller: contains the controller that wraps and manipulate the class in the model
- Test: contains tests for evaluator

Implementation choices

- ANTLR (ANother Tool for Language Recognition): permits to specify the language's grammar, and it generates parser and lexer in a target language, generates, also, the abstract class for evaluator
- Kotlin: is a general purpose, multi-paradigm and open source programming language, it runs on JVM
- Kotlins coroutines (experimental): for some concurrent behaviour of controller
- TornadoFX: for the GUI

Language overview

- Type of data: int, float (bool is cast to int)
- Standard operators of C: arithmetics, booleans, relational, assignment
- Statements: if, while
- Standard I/O functions: printf, scanf
- Block scope
- MCLexer.g4 contains all the token of the language
- MiniC.g4 contains all the rules of the grammar

Token and grammar

Tokens

```
BOOL : 'true' | 'false';
IF : 'if';
ELSE : 'else';
WHILE : 'while';
TYPE : 'int' | 'double';
WS  : [ \t\r\n]+ -> skip;
WORD_END : ' ';
COMMENT : '/*' .*? '*/' -> skip;
LINE_COMMENT: '//' ~[\r\n]* -> skip;
PRINTF : 'printf';
SCANF : 'scanf';
STRING_CHAR : '"' ~[\r\n"]* PLACEHOLDER* '"';
PLACEHOLDER : '%d' | '%f';
ID : (LETTER | UNDERSCORE)+ (UNDERSCORE | DIGIT)*;
```

Tokens

```
TIMES : '*' ;
DIVIDED : '/' ;
MODULE : '%';
PLUS : '+';
MINUS : '-';
MINOR : '<';
MINOREQUAL : '<=';
MAJOR : '>';
MAJOREQUAL : '>=';
ISEQUAL : '==';
ISNOTEQUAL : '!=';
OR : '||';
AMPERSAND : '&';
AND : '&&' ;
```

Tokens

```
NOT : '!';
EQUAL : '=';
ENDOFINSTRUCTION : ';';
NUMBER : DIGIT+ ('.' DIGIT+)?;
LETTER : [a-zA-Z];
DIGIT : [0-9];
COMMA : ',';
UNDERSCORE : '_';
RBRACKETOPEN : '(';
RBRACKETCLOSE : ')';
CBRACKETOPEN : '{';
CBRACKETCLOSE : '}';
```

Grammar Rules (1)

```
program := {statement}
declaration := TYPE ID [‘=‘ (assign | expression | printfStatement | scanfStatement)] ‘;’
assign := ID ‘=‘ (assign | expression | printfStatement | scanfStatement)
statement := ‘;’|( assign | expression | printfStatement | scanfStatement  ) ‘;’ |
blockStatement | ifStatement | whileStatement | declaration
prinfStatement := ‘printf’ ’(‘ STRING_CHAR {‘,’ expression} ‘)’
scanfStatement := ‘scanf’ ‘(‘ STRING_CHAR ‘,’ ‘&’ID {‘,’ ‘&’ID} ‘)’
blockStatement := ‘{‘ {statement} ‘}’
ifStatement := ‘if’ ‘(‘ (assign | expression | printfStatement | scanfStatement) ‘)’
statement [‘else’ statement]
whileStatement := ‘while’ ‘(‘ (assign | expression | printfStatement | scanfStatement)
‘)’ statement
```

*the tokens are explicit to simplify the reading

Grammar Rules (2)

The expressions are divided in 7 levels, to eliminate the left recursion rules, and also to specify the hierarchy between operators

```
expression := e1 {‘||’ e1}

e1 := e2 {‘&&’ e2}

e2 := e3 {(‘<' | '<=' | '>' | '>=' | '==' | ‘!=‘) e3}

e3 := e4 {(‘+’ | ‘-‘) e4}

e4 := e5 {('*' | '/' | ‘%') e5}

e5 := e6 | ‘!’ e5

e6 := ‘true’ | ‘false’ | NUMBER | ID | ‘(‘ expression ‘)’
```

Model

Generation of Parser and Lexer

After the definition of the grammar, it’s possible to generate Parser and Lexer with the command:

```
antlr4 -Dlanguage=Kotlin -o <output_directory> —visitor
-no-listener -lib <antlr_files_directory> <grammar_file>
```

After that, the tool creates the files MiniCLexer.kt and MiniCParser.kt.

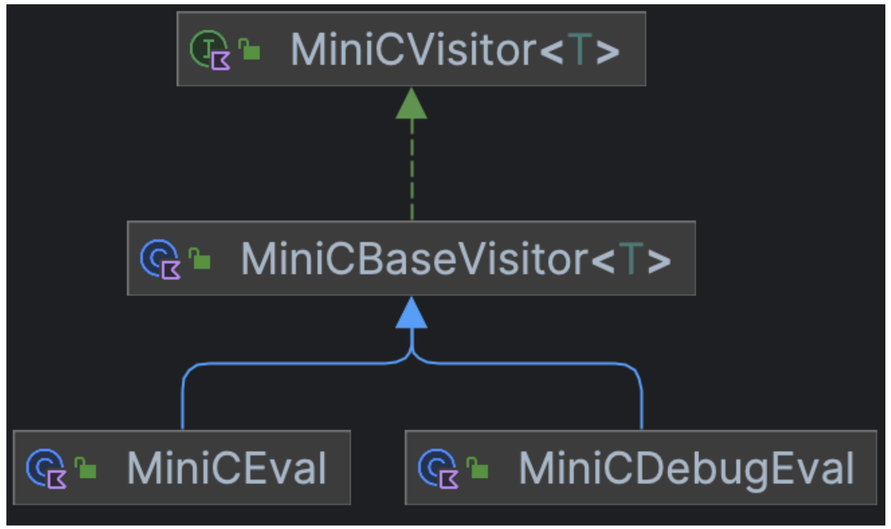
The tool also generates the necessary files for the visitor pattern, which provide the abstract classes to be implemented for the evaluator.

The visitor pattern is the best choice because it permits to control the visit of parse tree

Implementation of evaluator

To create the evaluator, it’s necessary to implement the abstract class MiniCBaseVisitor.kt, all the methods must be implemented, to evaluate all the grammar rules.

This way we can control the visit of the parse tree in the best way.




MiniCEval
visitScanfStatement(ScanfStatementContext) Number
insideMemory(String) Boolean
visitE2(E2Context) Double
visitSimpleDeclaration(SimpleDeclarationContext) Unit
insideUndefinedVar(String) Boolean
visitE6(E6Context) Double
visitAssign(AssignContext) Number
visitE4(E4Context) Double
insertInUndefinedAtNextLevel(String, String) Unit
getFromMemory(String) Pair<Number, Int>
insertInMemoryAtNextLevel(String, Number) Unit
insertInUndefinedAtCurrentLevel(String, String) Unit
purgeAllMemory() Unit
visitWhileStatement(WhileStatementContext) Unit
visitE1(E1Context) Double
visitIfStatement(IfStatementContext) Unit
removeFromUndefinedVar(String) String
visitE5(E5Context) Double
getFromUndefinedVar(String) Pair<String, Int>
visitE3(E3Context) Double
visitAssignDeclaration(AssignDeclarationContext) Unit
visitExpression(ExpressionContext) Double
visitStatement(StatementContext) Unit
visitBlockStatement(BlockStatementContext) Unit
visitPrintfStatement(PrintfStatementContext) Number
insertInMemoryAtCurrentLevel(String, Number) Unit
memoryBlock Vector<HashMap<String, Number>>
undefinedVarBlock Vector<HashMap<String, String>>
memoryToString String

Memory(1)

The memory is implemented by two Vector of HashMap:

`undefinedVarBlock: Vector<HashMap<String,String>>`

`memoryBlock: Vector<HashMap<String,Number>>`

The undefinedVarBlock contains all the undefined var, for each block of code; by the couple <id,type>

The memoryBlock contains all the defined variable for each block, by the couple <id,value>

Memory(2)

The level of blocks is represented by an Integer (blockLevel); that is initialised to the value 0.

Every time the evaluator visit a block ({}), the variable will be incremented, at the beginning of the block, and will be decremented at the end of block.

All the variable that are declare inside the block are stored in the memory, and it will be lost at the end of the block.

Memory(3)

Everytime that a variable is looked up, the evaluator searches at the current level, if is not present, the evaluator will search in the higher level block.

e.g.

```
int a=9; //level 0: {a=9}
{
   int b=a; //level 0: {a=9} level 1: {b=9}
}
//level 0: {a=9}
```

Memory(4)

It’s impossibile to declare a variable more than once, also if the variable is inside the block.

e.g.

```
int a=9;
{
   int a=8; //error by the valuator
}
```

visitAssign

This method gets the id and the value of the variable; if the variable is declared but not defined, it will be remove from undefinedVarBlock and it will be moved inside memoryBlock, else if it is defined the value of the variable it will be overwritten.

If the variable is undefined the evaluator gives error.

The method returns the value of the assignment.

e.g.

```
a=8;
```

visitSimpleDeclaration and visitAssignDeclation

visitSimpleDeclaration:

The variable,with its type, will be put inside undefinedVarBlock.

e.g.

```
int a;
```

visitAssigndeclaration:

The variable, with its value, will be put inside memoryBlock.

e.g.

```
int a=8;
```

visitPrintfStatement

Take all characters of the given string, and print them.

If it finds any format specifications, it evaluates the expression after the command, and prints it in the specified format.

Returns the numbers of printed characters

e.g

```
printf(“hello\n”);
printf(“%f\n”,a); //a is a inizialized variable
printf(“%d”,4*9);
```

visitScanfStatement

Takes the input of the user from the standard input, then controls if it is in the correct format, if is not it gives back an error, then it checks if the input is in the correct datatype (consistent with the specified format), in the affermative case it puts the value given in the memory, else it gives back an error.

Returns the number of assigned variables.

e.g.

```
scanf(“%f”,&a); //assign a to the given value
scanf(“aaa%d”,&a); //the value has to be “aaa<intNumber>”
```

visitBlockStatement

Increase the blockLevel value, and add new HashMap to the vectors undefinedVarBlock and memoryBlock; after that it evaluates all the statements inside the block.

At the end of evaluation of all statements, all the variables inside the memory will be delete, and the blockLevel value will be decreased

See above for example.

visitIfStatement

This method evaluates the expression inside the round brackets; if the expression is true (== 0), the statements inside the if will be evaluated; in negative case, if there is an else, the statement inside the else will be evaluated.

e.g.

```
if(true){
}
if(printf(“aaa”)){ //is false because printf(“aaa”) returns 3
}
else{
}
```

visitWhileStatement

This method evaluates the expression inside the round brackets; while the expression is true (== 0), the statements will be evaluated.

e.g.

```
while(a<10){

}

while(0){} while(true){} //infinite cycles
```

visitExpression and visitE[1..6]

Those methods evaluates all the level of the expression by their precedence.

Notable cases:

- If a value is divided by zero the evaluator it gives back an error
- If an id is not defined it gives back an error
- If a variable is not initialised it gives back an error

Private methods of MiniCEval.kt

- getMemoryToString():it returns the status of the memory in a string format
- insideUndefinedVar(key: String):it returns true if it is inside the undefined var memory
- insideMemory(key: String):it returns true if it is inside the memory
- getFromUndefinedVar(key: String):it returns the type of the variable (String)
- getFromMemory(key: String):it returns the value of the variable (Number)
- removeFromUndefinedVar(key: String):it removes the variable from undefined var memory

Test methods of MiniCEval.kt

- insertInMemoryAtCurrentLevel(key: String, value: Number): it inserts the given variable with its value inside the memory
- insertInUndefinedAtCurrentLevel(key: String, type: String): it inserts the given variable with its type inside the memory
- insertInMemoryAtNextLevel(key: String, value: Number): it inserts the given variabile with its value inside the memory at next level (blockLevel + 1)
- insertInMemoryAtNextLevel(key: String, type: String): it inserts the given variabile with its type inside the memory at next level (blockLevel + 1)

MiniCDebugEval.kt

This class is a copy of MiniCEval.kt but in this case, every time that a variable, is assigned, or is declared, it writes the status of the memory in the standard output, in this format:

```
after evaluation of: <statement that changes the memory>
undefined var:
level n: {<undefined variable at this level>}
level n+1: {<…>}
……
memory :
level n: {<inizialized variable at this level>}
level n+1: {<…>}
……
```

MiniCException.kt

It contains all the custom exception that are used by evaluator:

- DoubleDeclarationException: a variable is declared more than once
- UndefinedVariableException: an id is not defined
- NotInizializedVariableException: when a variable is declared but not initialized
- MismatchedTypeException: when the format in a scanf is different from the value type given
- BadFormatException: when the format given to a scanf is different from the format string
- ArithmeticException: when a value is divided by zero

ErrorListener.kt and SyntaxErr.kt

These two classes are used to handle errors generated by parser and lexer.

ErrorListener extends BaseErrorListener, and implements the syntaxError method; it puts all the received errors in a MutableList of SyntaxErr It contains also the methods to clear the list.

Test

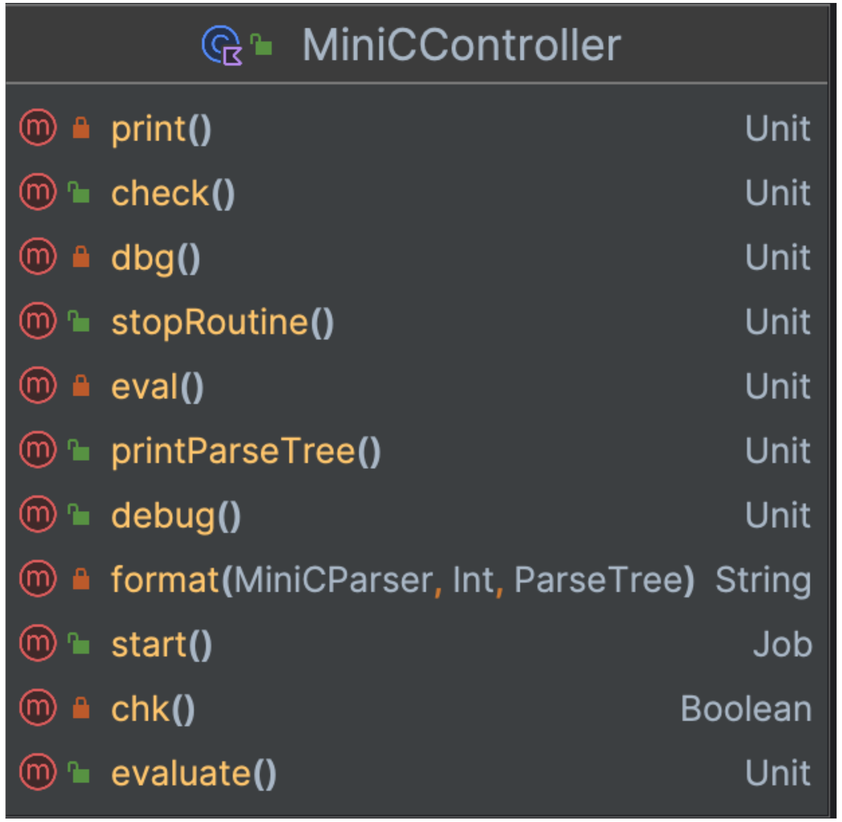
MiniCEvalTest.kt

This class implements the unit tests for each method except for the visitScanfStatement; this test was not implemented because interaction with user it’s needed

the methods inside the MiniCEval.kt are used by all the tests to manipulate the memory.

Controller

MiniCController.kt



MiniCController.kt

This class groups all parser, lexer and evaluator functionality.

It has two public fields:

- inputText (SimpleStringProprierty): it is the field where the code is written
- isRunning (Boolean): if it’s true, the program written is running

And three private fields:

- parserListener and lexerListener: ErrorListeners for parser and lexer
- channel: more info under

MiniCController.kt

It contains the following private methods:

- chk(): it creates the instance of parser and lexer and controls if there are errors in the written code; it returns true if the code is correct, otherwise false
- eval(): after a chk(),if it’s true it evaluates the code.
- dbg(): after a chk(),if it’s true it evaluates the code via MiniCDebugEval.
- print(): after a chk(),if it’s true it prints the parse tree.
- format(…): it formats the parse tree

Concurrent behaviour(1)

The controller behaves like a server, that waits for some request from the user of this class.

This behaviour is implemented by kotlin coroutines.

The coroutines are lighter than threads, and communicate with each other with exchange of message.

The channel field is for the communication between the coroutines, and implements, synchronous communication, for Int typed messages

For more info about coroutines: https://kotlinlang.org/docs/coroutines-overview.html

Concurrent behaviour(2)

The start method, contains a infinite loop; inside this loop the coroutine waits for some message inside the channel; when a request is received, it launches the corresponding private method in another coroutine and assign that to the variable job.

There’s not active waiting because the receive() of the channel is suspensive.

When another request is received, It controls if at least one job is running, in the affermative case, it writes an error to standard output.

It also implements the command to stops all the running jobs.

The start method must be called every time that the controller is used.

Public methods

The following methods act as a wrapper to send messages over the channel, thus hiding the underlying implementation to the user:

- evaluate()
- check()
- printParseTree()
- debug()
- stopRoutine()

View

MiniCView.kt

It would be possible to use the application without a GUI, implementing a main that uses the MiniCController.kt class; but to provide a better experience to the user the gui has been implemented.

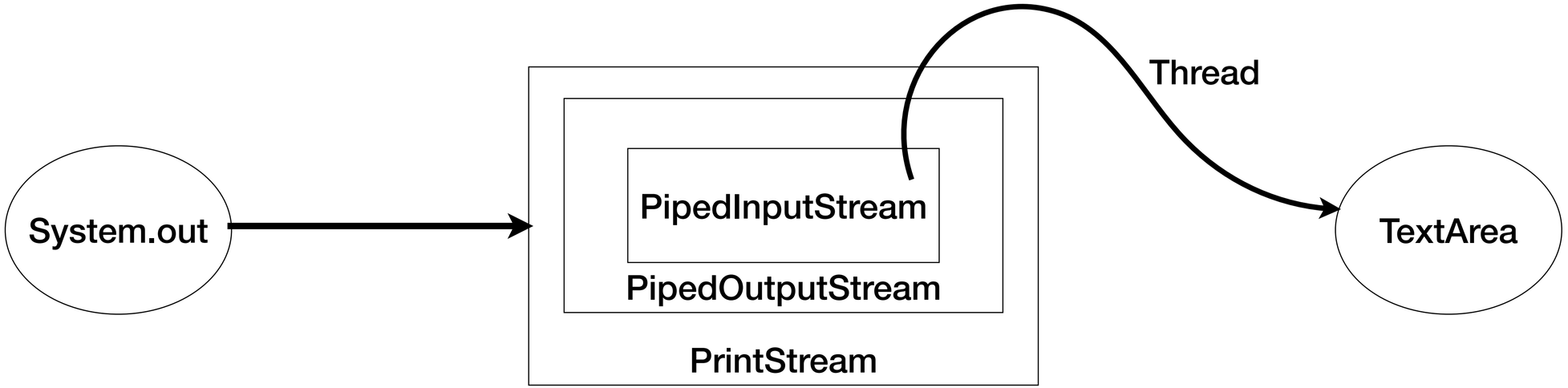
This class is implemented by TornadoFX library, that is a lightweight JavaFX implementation in Kotlin

For more info: https://tornadofx.io/

The task of this class is to redirect the input and output from the console to the interface, it also implements all the functionality to dialogue with the controller via the GUI

Redirection of output

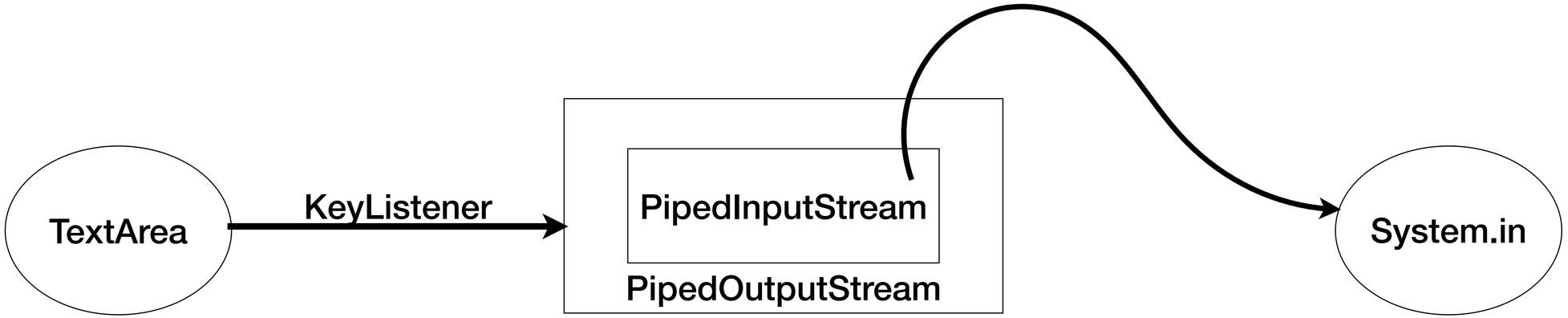
The redirection of the output works via a PrintStream, a PipedOutputStream and a PipedInputStream, and follows this schema:



System.out is redirected to PrintStream, that is linked to PipedOutputStream and PipedInputStream; the TextArea read from PipedInputStream by a thread that wait for some output to print

Redirection of input

The redirection of the input works via PipedInputStream and a PipedOutputStream, and follows this schema:



System.in is redirected to PipedInputStream.

When enter is pressed the TextArea places the inserted text inside the PipedOutputStream.

Other stuffs

Other stuffs

- Resources directory: it contains the image for the logo
- Images directory: it contains the images for the examples
- Lib directory: it contains all the required libraries

For the user guide refer to readme.md