PP Project Module 8

University of Twente - TCS

Bryan Sanchez Haro (s2105454) and Fátima González-Novo López (s2356546)

For the final project of the Programming Paradigms module of the University of Twente, we needed to implement a compiler for a source language. To do this, we have decided to use Java for the front-end and code generation process. In the rest of this document, we will explain to you a summary of our main important features, a detailed description of the software, problems we encountered, and a conclusion.

1. Language main features

Our main inspiration to design our programming language was Java. Therefore, we tried to implement a source language similar to it. First of all, our program starts with a class definition. For example, you can define: class Main(int term1, int term2){ … }. Inside a class, you can have different functions, conditions, or variables. You can also declare conditions nested inside other conditions, and functions with nested conditions inside.

The **data types** that we have managed to implement are the basics types; integers and boolean and the parsing and scanning process of strings and arrays. Our language supports the basic operations of integers and booleans. For integers, you can do addition, subtraction, multiplication, and division. Moreover, for booleans, our support operations are '&&' and '||' and for strings only concatenation ('+'). In addition, you can infinitely combine the operators as long as the restrictions types for the declared variables are followed. So for integer, for example, you can declare: int var1 = 1 + var2 + var3 - 4 * var4 -var7…

In addition, all of our data types can be checked for comparisons for equality and inequality. Additionally, for integers, you can also check '<', '<=', '>' and '>='. As was the case with operators, you can also use them infinitely as long as their type matches. So it would be possible to declare: boolean var1 = (var2 > 4) && (var2 => 5) && True… . Another feature that our language has is the possibility of adding comments, both, inline comments as well as block comments. To do so, you can simply type either '--' or '-- * text here *--'. We have also managed to distinguish in the parsing process between **global and local variables for our languages.**

Basic statements and conditions are something we also implemented. The conditions that we have are the if-conditions; with 'else if' and 'else', while loops and for loops. In a conditional statement, you can also make an infinitely large comparison using the operators '||' and '&&', just like in java. For the for loops, they are written down in the form of (declaration initializer, comparison, declaration). Additionally, if conditions can have many 'else if' parts. Both, declaring if (variable), in the case of boolean variables, and if (variable COMPARATIVE variable) are valid options as well. We also check that the values inside the condition have been declared and initialized before.

Another feature of our language is that the initial values of declarations need to be assigned explicitly. All variables are checked for **initialization** in our scanning phase, to make sure that the program doesn't have any context constraints errors. Additionally, we have implemented the print statement feature, to

print values if desired.

We have kept track of the **local and nested scopes** with the help of “Mysymboltable”. In this class, variables are saved per scope. Once the current scope terminates, all stored variables are deleted. This is useful for conditions, or when using different scopes. If a variable has been declared in another scope higher than the one you are calling it from, then a specific error message gets printed.

Our language is strongly typed and therefore, also supports **typed variables.** We have managed to do this with the help of a HashMap dictionary. Every time a variable is declared, it gets added to the dictionary. If it’s called again, we check whether the variable has been declared or not and if so, update the value. In this process, we pay extreme attention to the scopes as stated above.

The last process that we focussed on is **concurrency**. Firstly, we added to our grammar global variables as described above. Then, we added the possibility of having locks, as well as concurrent blocks.

As for the extra features, we have managed to implement the following: scanning process of strings and arrays, also type checking for these ones; soft integer division; for conditions; parsing of functions and nested procedures (fully implemented in Sprockell, you can have nested for loops, variables...).

2. Problems and solutions

During this project, we encountered lots of different problems. Some of which are the following:

1. When creating Exception handling, we were getting some errors. That is why we had to modify the different classes generated by ANTLR in order to accept the exceptions that we were creating.

2. Initially, our grammar was only able to infinitely large expressions, and simple expressions like the one below. However, we noticed that we were checking for the types of the variables and seeing if they matched the declared type, in this case, integer; but we were not checking the operators in between.

```
int var1 = 5 + 5 - 9;
```

That is why, if we continued writing and extending the declaration with let’s say ‘&& 5’, our grammar would return true, as 5 is an integer. However, the result should be false, increasing the number of mistakes plus one because ‘&&’ is not a valid operator for integer operations. To fix this error, we had to take operators into account and create a loop that went through all the operators to check if they were indeed correct. The error was also present for boolean and string types, so we had to go through all of our code to fix the mistakes caused by this.

3. A third problem that we encountered was when parsing strings: This process took us longer than we were expecting as we had created a method called isString(), that checked if an identifier was a string. The problem was that in our grammar we had defined variable as:

```
variable : ““ identifier ““
```

and we weren’t able to identify strings properly because the ““ symbol was not getting through. We kept creating other isString() methods and making modifications in our code until we understood that our error was related to our grammar. Then, we modified our grammar to:

```
variable : QUOTES identifier QUOTES
```

Afterwards, we were able to identify strings with the help of the token quotes, which can be done very easily by using ctx.

4. A fourth problem that we encounter was related to the elaboration phase of the if-condition. To create if conditions in our grammar, we needed a switch that turned False values into True and vice versa. At first, we didn't realize that this was necessary, so all of our conditions kept running into an infinite loop. To fix the error we created our own XOR condition. We needed this XOR because of the BRANCH condition of Sprockell.

5. A fifth problem that we encounter is related to the parsing procedure of arrays. At first, we kept getting a null pointer exception, and we were not sure what we were doing wrong. Then, we noticed that the mistake was in the way we were identifying arrays. In the beginning, we were eliminating the ',' and ']' and identifying the type of the first element of the array. Afterward, we checked that all of the elements that followed were of the same type. However, this did not work, as we were not able to search for the type of a variable in a dictionary if this one was not declared previously in it. The solution we applied was simple, we create a for-loop that iterated through the array and detected if it had variables that were inside a dictionary or not. If not, we were able to simply identify the values with isInteger() or isString().

```
for (int j = 0; j < variables.length; j++) {
    if (dictionary.containsKey(variables[j])) {
        containsDicKey = true;
    }
}
```

6. The six error that we encounter was with the types of variables. From the start, we didn't distinguish between global or local variables, that is why, when implementing the concurrency part of our program we had to go back to the start and modify all of our grammar rules defined before.

```
globalInt num = 9;
globalBoolean condition = True;
```

3. Detailed language description.

In this section of the report, we will provide you with a more detailed explanation of the features described in Section 1.

We will start by explaining how classes work, as to start a program in our language, declaring a class is mandatory. To do so, you just need to provide the keyword 'class', the name of your function, variable declaration inside brackets if desired, and finally '{' '}. An example of how to use them is shown below, as well as our grammar declaration:

```
1. class BankingSystem () { }
2. classes: type identifier arguments body
    ;
```

Inside a class, you can declare many different arguments. We will start by explaining declarations. As mentioned previously, our language is strongly typed. Therefore, all variables need to be initialized. Below you will see three examples of how to declare integers, booleans, and strings.

```
int var1 = 5;

string str1 = "test";

boolean b1 = False;
```

As mentioned previously, our language also supports global types, for integers, booleans, strings, and arrays. To define them you type global + variable type. Like this:

```
globalInt temp = 20;

globalBoolean condition = True;
```

Moreover, you can also declare a variable using the different operators that satisfy its type. For integers, that is '+', '-', '*', '/', with an integer or another variable of type integer, previously defined. For booleans, you can use '&&' and '||', with 'True', 'False', or another variable of type boolean previously defined. For strings, you can use '+' on strings or another predefined variable of that type. Below you can see an example:

```
int var2 = 5 + 1 * 3 - var1;

string str2 = "test" + "test1" + str1;

boolean b2 = False && True || b1;
```

Like in all programming languages, we have also defined a print statement. In it, it checks that the type is the one of a variable initialized before in the program, an integer, a boolean, or an array. If none of these is the case, an error message is thrown. An example of its use can be seen here:

```
print(var1);

print(1);

print("Test");
```

Additionally, you can type block comments by using '--*' and '*--'.

```
--*
    Example_Test
*--
```

And single inline comments with '--'.

```
-- This is an inline comment
```

Conditions are another feature of our language. We have implemented three types of conditions, as described above; if-conditions, while loops, and for loops. To use an if condition, you need to provide it with a condition. Conditions can be just a boolean, variables comparing other variables or multiple conditions tied together with 'and' or 'or'. Additionally, you can use 'else if' and 'else' statements. The functionality of 'else if' is the same as the one of 'if'. Below you can see an example:

```
if ( (var1 <= var2) and (True)) {
   int var3 = var1 + 1 + var1 * var2;
}
else if (var1 == 5) {
```

```
  int var1 = var1 + 1;
}
else {
  string result = str1 + "testing" + str1;
};
```

For comparing variables, there are different operators available, as already used in the example above. Integers support '>', '<', '>=', '<=', '==' and '!='. Booleans support '&&', '||', '==' and '!='. Finally, strings support '==' and '!='.

While loops can be implemented by typing a condition. For typing a condition, the same rule holds as the one in the conditional statement from if conditions. An example can be seen below:

```
while ( (5 < 6) and (8 < 12) or (8 < 12)){
  int var2 = 7;
};
```

To implement a for loop, you need to write the 'for' type followed by brackets with a variable initializer, a condition, and finally a variable assignment. We have implemented it very similarly to the ones in Java. The same rule holds for declaring a condition, like the ones that were defined for while and if conditions. An example is illustrated below:

```
for (int 1; x < 10; x = x + 1) {
  x = x + 1;
}
```

Now that we have explained the conditions, we will move on to explaining the functionality of our definitions. Definitions work very similar to classes. Inside of then, you can have if-conditions, while loops or for loops, as well as declarations and everything previously defined. An example can be seen below:

```
def withdraw (int amount, int account){
  if (account >= amount) {
      account = account - amount;
  }
};
```

For the concurrency part of our application, we have defined locks as well as concurrent locks in our grammar. Below you can see how we defined them:

```
statements: if_condition END
        | while_loop END
        | for_loop END
        | declaration END
        | return1 END
        | print END
        | concurrentBlock END
```

```
    | lockCondition END
    ;

concurrentBlock: 'parbegin' '{' statements* '}'
    ;

lockCondition : 'lock' '(' identifier ')' '{' statements* '}'
    ;
```

For normal locks, you can use lock (identifier) {} and to use concurrent locks, you need to do:

```
parbegin {
  int car2 = 54;
}
```

4. *Description of the software*:

Our project is ordered in the following manner. In the first level, we have a READ-ME file with an explanation of how to run our program. Additionally, we have three folders. The first folder is called ANTLR and there you can find our grammar. In our second folder called Sprokkell, you can find four different classes; Instances, Instructions, Operators, and Registers. In our third folder called Test-Files, we have all of our tests files. We have categorized them into syntax-error, context-errors, and semantic-errors tests. The folder also includes tests for Peterson's algorithm and for the banking application. Finally, we have six files that were automatically generated by ANTLR, when generating the ANTLR recognizer for our grammar. These files are called GrammarBaseListener, GramarBaseVisitor, GrammarLexer, GrammarListener, GrammarParser, and GrammarVisitor. Additionally, we have decided to create the following files, of which a further explanation will be given:

1. GrammarChecker: This class is used to check the syntax of different programs, in the scanning phase, and see if they are correct. If so, they can be parsed.
2. GrammarGenerator: We start by creating the file. We check if the name of the file has the same name as the one we gave it. If it is already there, you will get ask if you want the program to be overwritten. If so, the program visits all methods, checking what is necessary and what is not. It visits the non-terminal states. After visiting the whole tree, the file gets generated.
3. GrammarTest: Our different test methods can be run through this class.
4. MyCompiler: Firstly, we run the GrammarChecker, we read the file and if there are no errors, we can move into the next phase, the generator phase. We then create one Haskell file by calling GrammarGenerator, and this one generates a Haskell file with all Sprokkell instructions. Afterward, if there are no errors, we compile it with GHCI and return the result if there is a write instruction for instance.
5. MySimbolTable: This class is used to associate different values with their respective keys. In our case, the keys are the type of the variable. This class is also used for scope checking.

5. *Test plan and results*.

All of our tests are located in a folder called “Test-Files”. They have been categorized into syntax, semantic and contextual tests. We have a folder for each of these categories and inside them, another two folders with correct and incorrect tests. Additionally, to test the functionality of our concurrency feature, we have included two more tests. The first one follows Peterson’s algorithm for mutual exclusion, which explanation was found in the module guide. The second test consists of a banking system application with three basic methods, withdrawal, deposit, and transfer.

We have lot’s of syntax tests, as we have decided to test our basic functions to ensure that the scanning procedure works correctly, to avoid later mistakes in the parsing phase.

Now, I will explain some tests in detail:

1. Syntax errors Tests: We have created 14 tests in the category of syntax errors, seven of which are incorrect programs. In these tests, we check for things like spelling errors or language construct errors. For example, when declaring a variable, you need to add ‘;’ at the end of the declaration. So we have checked that those ‘;’ are not forgotten. Additionally, we are checking that the comments are used properly, as well as the types of our functions. An inline comment should therefore be ‘--’ and not ‘- -’ for example. Moreover, we have checked that comparisons are made with the correct operators. So for example, ‘while (var1 = 5)’ is not allowed.

2. Context constraints Tests: We have created eight tests in the category of context constraint, four of which are incorrect programs. Here, we check for the correct use of declarations, scope levels and type errors made. Declaring a variable without its type, that has never been declared before, will produce an error. That is because we need to identify its type to know which operations it can perform. Additionally, we check that variables inside conditions have been initialized somewhere in the program before, in a lower scope. Moreover, we are checking that variables declared on higher scopes are not used in lower ones. Lastly, we are checking the operators of the different variables, and making sure that they match the variable type. For example, a string cannot be divided.

3. Semantic errors Tests: We have created eight tests in the category of semantic errors, four of which are incorrect programs. Here, we mainly check for run-time errors. We are ensuring that the different exceptions that we created are thrown.

To run all of our tests, we have created a class called Grammar tests, which reads each text file and compares the expected results with the real outcome.

Moreover, we have developed some automated tests that are automatically generated from a correct program file, similar to what QuickCheck does.

Conclusions.

With this project, we have been able to grasps all concepts learned in the three subjects; CP, CC, and FP, of the Programming Paradigms module and put them into practice. Even though our language consists

mostly of the basic features with some extra ones that have been added (like division or error handling for example), we have been able to learn so much from its implementation. There is no doubt that we would have loved to have more time to implement all of the extra features. Yet, we found challenging as well as rewarding the work that we have submitted.

As everyone might agree, the Programming Paradigms module is very time-consuming. This last month, we have had unlucky circumstances that didn't enable us to work at our fullest potential. Nonetheless, if it hadn't been because of it, we strongly believe we would have had enough time to implement more features. Lack of time was definitely our main inconvenience during the project.

This project implements the theoretical knowledge learnt in the course in a practical way, and we have been able to recognize many of the implementations made during class tutorials in this project. With all, even though it might be one of the hardest modules we have had so far, it has been one of the most rewarding ones.

Appendices

1. **Grammar specification:** The complete listing of our grammar(s), in ANTLR .

```
grammar Grammar;

comments: program EOF
        ;

program : (classes END )*
        ;

classes : type identifier arguments body
        ;

arguments: '(' (type identifier (',' type identifier)* )? ')'
         |
         ;

body: '{' ( statements | funct )* '}'
    ;

statements: if_condition END
          | while_loop END
          | for_loop END
          | declaration END
          | return1 END
          | print END
          | concurrentBlock END
          | lockCondition END
          ;

concurrentBlock: 'parbegin' '{' statements* '}'
               ;

lockCondition : 'lock' '(' identifier ')' '{' statements* '}'
              ;
```

```
funct : 'def' identifier arguments body END
      ;


return1 : 'return' variable
        ;

print : 'print' '(' variable ')'
      ;

if_condition  :  'if'  '('  condition  ')'  body  ('else  if'  '('  condition  ')'  body  )*
('else' body)?
             ;

for_loop : 'for' '(' declaration END condition END declaration ')' body
         ;

while_loop : 'while' '(' condition ')' body ;

declaration : type identifier STARTFUNC expression
            | identifier STARTFUNC expression
            ;

expression : '(' expression ')'
           | variable (OPERATORS variable)*
           ;

condition : variable COMPARATIVES variable
          | multiple_conditions
          | identifier
          | STATE
          ;

multiple_conditions : '(' condition ')' (MCOND '(' condition ')')+
                    ;
```

```
variable : QUOTES identifier QUOTES
         | OPENBRT variable (',' variable)* CLOSEBRT
         | identifier
         | integer
         |
         | STATE
         ;

type: BOOLEAN
    | INT
    | ARRAY
    | CLASS
    | STRING
    | GLOBAL_INTEGER
    | GLOBAL_BOOLEAN
    | GLOBAL_ARRAY
    | GLOBAL_STRING
    ;

integer : INTEGER;
identifier : IDENTIFIER;

// LEXER
INT: 'int';
ARRAY: 'array';
BOOLEAN: 'boolean';
CLASS: 'class';
STATE : ('True'|'False');
STRING : 'string' ;
MCOND : ('and'| 'or');
GLOBAL_INTEGER : 'globalInt';
GLOBAL_BOOLEAN : 'globalBoolean';
GLOBAL_ARRAY : 'globalArray';
GLOBAL_STRING : 'globalString' ;
INTEGER : [0-9]+ ;
IDENTIFIER : [a-zA-Z_][a-zA-Z_0-9]* ;
```

```
WHITESPACE : [ \t\n\r]+ -> skip;

COMMENT : '--' .*? '\n' -> skip;

MULTILINE_COMMENT: '--*' .*?'*--' -> skip;

END : (';');

OPERATORS: ('+'|'-'|'*'|'/'|'||'|'&&');

COMPARATIVES: ('>'|'<'|'=='|'!='|'>='|'<='|'||'|'&&');

STARTFUNC: ('=');

QUOTES : ('"');

OPENBRT : ('[');

CLOSEBRT : (']');

GLOBAL :('global');
```

2. **Extended test program:**

-- In our language :

```
class Main(){
    int var1 = 5;
    int var2 = 10;
    boolean bol = True;

    for( int i = 0 ; i < 5 ; i = i + 1 ){
       var1 = var1 + 5;
       var2 = var2 + var1;
    };

    int temp = 5;
    while(temp > 0){
       var1 = var1 - 1;
       var2 = var2 + 1;
       temp = temp - 1;
    };
    if (var1 >= 0){
       bol = False;
    };

    print(var2);
    print(bol);
};
```

-- Sprockell output :

```
import Sprockell
prog :: [Instruction]
prog = [
                                    Load (ImmValue 5) regA ,
                                    Store regA (DirAddr 0) ,
                                    Load (ImmValue 10) regA ,
                                    Store regA (DirAddr 1) ,
```

Load (ImmValue 1) regA ,

Store regA (DirAddr 2) ,

Load (ImmValue 0) regA ,

Store regA (DirAddr 3) ,

Load (DirAddr 3) regA ,

Load (ImmValue 5) regB ,

Compute Lt regA regB regA ,

Store regA (DirAddr 5) ,

Load (ImmValue 1) regB ,

Compute Xor regA regB regA ,

Branch regA (Abs 28) ,

Load (DirAddr 0) regA ,

Load (ImmValue 5) regB ,

Compute Add regA regB regA ,

Store regA (DirAddr 0) ,

Load (DirAddr 1) regA ,

Load (DirAddr 0) regB ,

Compute Add regA regB regA ,

Store regA (DirAddr 1) ,

Load (DirAddr 3) regA ,

Load (ImmValue 1) regB ,

Compute Add regA regB regA ,

Store regA (DirAddr 3) ,

Jump (Rel (-19)) ,

Load (ImmValue 5) regA ,

Store regA (DirAddr 4) ,

Load (DirAddr 4) regA ,

Load (ImmValue 0) regB ,

Compute Gt regA regB regA ,

Compute Lt regA regB regA ,

Store regA (DirAddr 6) ,

Load (ImmValue 1) regB ,

Compute Xor regA regB regA ,

Branch regA (Abs 48) ,

```
Load (DirAddr 0) regA ,
Load (ImmValue 1) regB ,
Compute Sub regA regB regA ,
Store regA (DirAddr 0) ,
Load (DirAddr 1) regA ,
Load (ImmValue 1) regB ,
Compute Add regA regB regA ,
Store regA (DirAddr 1) ,
Load (DirAddr 4) regA ,
Load (ImmValue 1) regB ,
Compute Sub regA regB regA ,
Store regA (DirAddr 4) ,
Jump (Rel (-20)) ,
Load (DirAddr 0) regA ,
Load (ImmValue 0) regB ,
Compute Gt regA regB regA ,
Store regA (DirAddr 6) ,
Load (ImmValue 1) regB ,
Compute Xor regA regB regA ,
Branch regA (Abs 56) ,
Load (ImmValue 0) regA ,
Store regA (DirAddr 2) ,
Load (DirAddr 1) regA ,
WriteInstr regA numberIO ,
Load (DirAddr 2) regA ,
EndProg]

main = run [prog]
```