

University of Twente

Programming Paradigms – Final Project



*A report on the:
Compiler for the programming language Jafar*

By Team 22:

*Vo Nhat Minh - s2503042,
Tran Duc Duc - s2482266,
Andrea Onofrei - s2491257*

July 8, 2022

Table of Contents

<u>1. SUMMARY OF THE MAIN FEATURES OF THE PROGRAMMING LANGUAGE:</u>	<u>3</u>
<u>2. PROBLEMS AND SOLUTIONS:</u>	<u>4</u>
2.1 ARRAY ACCESSING IN STACK:	4
2.2 LIMITED REGISTERS:	4
2.3 LABEL AND JUMPING:	4
2.4 LIMITED SHARED MEMORY:	5
2.5 VARIABLES DECLARED IN FUNCTION SCOPE:	5
2.6 RUNTIME ERRORS:	5
<u>3.DETAILED LANGUAGE DESCRIPTION:</u>	<u>6</u>
3.1 BASIC TYPES	6
3.2 ARRAY TYPES:	7
3.3 SIMPLE EXPRESSIONS AND VARIABLES:	8
3.4 NESTED SCOPES:	11
3.5 BASIC STATEMENT(ASSIGNMENT/IF/WHILE/PRINT):	12
3.6 CONCURRENCY AND SHARED VARIABLES:	18
3.7 DIVISION/MODULO:	20
3.8 PROCEDURE/FUNCTION:	22
<u>4. DESCRIPTION OF THE SOFTWARE:</u>	<u>25</u>
4.1 COMPILATION:	25
4.2 EXCEPTION:	26
4.3 MODEL:	26
<u>5. TEST PLAN AND RESULT:</u>	<u>27</u>
5.1 SYNTAX TEST:	27
5.2 CONTEXTUAL TEST:	28
5.2 SEMANTIC TEST:	31
5.2 CONCURRENCY TEST:	34
<u>6. CONCLUSION:</u>	<u>36</u>
<u>APPENDICES:</u>	<u>37</u>
7.1 GRAMMAR SPECIFICATION:	37
7.2 EXTENDED TEST:	40
7.2.1 JAFAR PROGRAM:	40
7.2.2 GENERATED SPROCKELL CODE:	41
7.2.3 RUN RESULTS AND EXPLANATION:	55

1. Summary of the main features of the programming language:

The syntax of our programming language - Jafar is reminiscent of languages Java, Pascal and Python. As the language from which it was inspired, it is a strongly typed language – we enforce firm restrictions on mixing different data types and values.

On the features that we have implemented: we have all the mandatory elements: comments, basic types (integers and booleans), simple expressions and variables, basic statements, and concurrency. Moreover, we have implemented optional features: arrays, functions/procedures, and soft division.

Now, shortly about each feature and the structure of a program in Jafar: After writing the program name, the programmer can optionally declare the shared and simple variables before starting the mandatory block. Unlike regular variables, shared variables can be accessed by every thread, regardless of current thread number. Blocks are basically scopes in which a programmer can optionally declare other variables and will have to mandatory write a statement. Jafar, similarly to it's parent, Pascal, supports nested scopes and the scope syntax is similar to Java with opening { and closing }.

The programmer can declare a new variable only at the top of the block, by first indicating it's type and then its name e.g `int x,y; bool z;` Arrays are either `bool` or `int` and can have multiple dimensions e.g `int [1][2][3] a;` A difference from its parents is that in Jafar the programmer cannot assign values when declaring the variable, they will be assigned a default value (0 for `int` and `False` for `bool`). For expression in Jafar we have supported all basic expressions for integer and boolean including prefix, soft division, modulo operation and array comparison. `If`, `While` and `Print` statements are all supported by Jafar with similar syntax like Python. Jafar also supports single line comments that have the Java-style, mainly `//comment`. Functions in Jafar are declared using the “`func`” keyword and can only be declared on top of the program right below variable declaration. The procedure declaration is similar to declaration of a function, but its return type is `void` and no expression follows the `return` keyword.

For the concurrency, Jafar supports `parbegin`, `parend` model and locking. In between `parbegin` and `parend`, programmers can declare multiple blocks which then can be run concurrently. Jafar also supports locking inside a thread. A thread will grab the lock on its parent thread, preventing other sibling threads from executing and then releasing parent thread lock on `unlock`. Jafar's threads however, can only use shared variables of primitive types (boolean, integer) and non custom function (`if`, `while`, `print`), declarations and usages of non-shared variables. An upper bound of six threads, including the main thread can be created. Creation of more than six threads is prohibited and will create an error during checker phase. This part is mostly written by Andrea, Minh and Duc only write the function and concurrency respectively.

2. Problems and Solutions:

2.1 Array accessing in stack:

We have 2 kinds of expression for arrays. One for identifiers which are stored like normal variables and one without identifier which is only stored temporarily in the stack. While accessing with identifier can be easily handled by just getting its offset, accessing in stack is a little bit trickier. Since the array values are stored in reverse order, to access the next value we need to pop the first value from stack first. However, in certain cases we need to retain all the array values, for example: when comparing the array. For this part, Duc has come up with a way to handle that. Since we already know that `regSP` contains the value of the current stack pointer, we can easily get the last element of the first array. We also noticed that the `regSP + size` of array should point to the last element of the second array, so we can get the value stored in this location and do operations involving both elements. As we finish with the last element, we can pop it out of the memory stack and continue with the next element and so on. Note that we have to pop the elements of the second array out of stack after we traverse all the first array elements. This part is shared by Minh and Duc.

2.2 Limited registers:

The next problem that we have encountered is that there are only 6 registers in Sprockell that can be used freely to store values, and we also need some dedicated register to store certain values like ARP for functions/procedures. Our solution is that for the functions we use a dedicated register F only to store ARP value and we never use the register F in any operation not related to the ARP calculation. For most of computation we try to make use of the memory's Stack mechanism and reuse register as much as possible. Everytime we finish one operation and we know that we do not need its result soon, we will push its value to the stack. When we need the value again we just need to pop it from the stack with a small note that stack is LIFO so we need to pop all other values on top of the required value. For this part contribution is shared by all the team members.

2.3 Label and jumping:

For the labeling system, Sprockell does not support label jump, so we decided to maintain a Target model which represents Target similar to ILOC and an instruction counter. Our Target model also supports jump to an absolute Label, which is modeled by a String and line number Integer. Everytime we emit a new SPRIL instruction we will update the counter and when we need a jumping target we just put the counter in HashMap with a given label name. After finishing all the code generation, we simply replace all the Target labels with their corresponding line numbers and update jump instruction with the Abs [line number] which are supported by Sprockell Jump and Branch. This part is written and implemented by Duc.

2.4 Limited shared memory:

For our threading and locking algorithm, each thread required two shared memory locations, one for children threads waiting for start, and one for main thread waiting for its children threads to finish. Furthermore, each lock on a parent thread required one shared memory location. Since we allow a maximum of six threads, 18 shared locations should be reserved for threading and locking. Based on this observation, we decided to extend the shared memory to 48 slots, of which 18 slots are for threading and locking. The last 30 slots are available for shared variables, supporting reasonable array sizes and/ or multiple primitive variables. This part is written and implemented by Duc.

2.5 Variables declared in function scope:

When declaring a variable in normal scope we can easily give and store its memory offset and type in the symbol table because its position is fixed, the variables in function scope, however, can be declared as many time as the function is called and their offset are based on its function ARP. Therefore, in the elaboration phase, when put function's parameter and variable, instead of giving it a memory offset, we only store its offset in the function scope, and when we need to declare a function we know which place to put the variable in memory offset based on its function scope offset and ARP. This part is written and implemented by Minh.

2.6 Runtime errors:

When running instructions in Sprockell, some runtime errors can make the program run into an infinite loop or access the wrong memory location if we do not handle it properly. In our program, we have found and dealt with the most 2 prominent runtime errors namely index out of bound and division by zero. To deal with them we have to generate explicit check conditions for all division operations and index accessing cases whether the divisor equals zero and the index value is not inbound. If the check has failed, then a jump to the end of the program is triggered to terminate the execution. This part is shared by all team members.

3.Detailed language description:

3.1 Basic Types

Basic Types of our language are integers and booleans.

Syntax

The syntax of basic types in Jafar is similar to Java and Python. As our language is strongly typed, when declaring a variable, one will have to indicate its type before being able to use it.

```
Program basicTypes;  
  
int i, j;  
bool result,temp,temp2;  
  
{  
    i := 0;  
    j := i + 4;  
    temp := true;  
    result := temp or temp2;  
}
```

Usage

Basic types can be used for declaring the types of variables. In the type checking phase, the compiler will check if the types and operations we are doing with the variables are consistent. If for example, it was declared that one variable is integer, but then it is assigned a boolean value – it will result in an error.

Semantics

As stated before, basic types can be used for declaring the type of variables, as shown in the example above. Moreover, primitive types can be used as the return type of functions or combined into a compound type like arrays. When declaring a variable with a basic type the default size of 1 is saved in memory for that variable.

Code generation

When we declare a basic type the Symbol Table will add its information, increase the memory offset by 1 (which is the default memory storage for Integer and Boolean) and give it a memory offset. And by default, all declared variables with basic type have an initial value of 0 (for Boolean it means False). This part is written and implemented by Andrea.

3.2 Array types:

Syntax

- type [dimension1][dimension2][...] identifier
- [elem1, elem2, elem3, ...] Note elem are of primitive type

```
1 Program arr;  
2 int[2][3][2] x;  
3 {  
4     int y,i,j,z;  
5     int[2][3][2] z;  
6     x:= [[1,3],[5,7],[8,9]],[10,11],[12,13],[15,17]];  
7     y:= x[1][1][0];  
8     z:= x;  
9     y:= x[i][j][k];  
10    x[0][2][1] := 100;  
11 }
```

Usage

In Jafar, we support multi-dimensional array types with similar syntax to Python. Accessing an array will yield a list of its elements of the same type. There are 2 ways to access an array by an identifier (with optional index) of a declared array (Syntax line 7,8) or by just giving an array value (Syntax line 6). There are restrictions that all the elements need to be the same types, the index needs to be of type integer and the size of a declared array is fixed and cannot add new elements. Furthermore, all indexes need to be in the bounded size from 0 to declared size -1 otherwise the program will generate an index out of bound check and terminate immediately.

Semantics

An array is a collection of similar types of data. In Jafar, all array elements are stored linearly one by one in the memory even though it's a multidimensional array. Each memory location is associated with the value of 1 primitive type element. When accessing an array by index, even a multi-dimensional index, it will be converted to a 1D index to access the memory location of the corresponding element. If the element is a sub-array then the index will point to the first element of that child array.

Code generation

The code generation for array has 2 different cases:

- Array of value only(without identifier): This case means we need to visit all the element expressions of an array one by one and push them all to the stack. The only notable thing is that all array elements are pushed in reversed order and when getting the value to assignment or comparison we need to start with the last index of the array.
- Array with Identifier/Index: The accessing or assigning an array by the index is done by calculating the first element's offset. To access an array with given indexes we first need to get the identifier's start offset and its type. Then for each index dimension, we need to generate a check for index out of bound which will check if the index is inbound $0 \leq \text{index} \leq \text{array size} - 1$ and branching to jump to the end of the program. For each valid index, we will next add the given index multiplied by the corresponding dimension element's size to the final index and go to the next dimension. The final offset is the array start offset plus the final index. After that with the given size of the variable and start offset we can linearly go through all elements to get its value. The reason why we just calculate the offset and push it to the stack is because of the limited local memory with only 32 and if we do push all values into the stack the memory can be corrupted

This part is written by Minh and implemented by Minh and Andrea

3.3 Simple expressions and variables:

In Jafar we have supported all basic expressions for integer and boolean including prefix (-1 and !true), soft division, modulo operation and array comparison.

Syntax

An expression can be a constant, variable identifier, expression in parenthesis, boolean value, something of the form expression arithmetic-operation expression, expression relational-operation expression, unary-operation expression, as well as arrays, and functions/procedures. Variables both shared and normal ones can be declared at the top of the program, as well as in the blocks. About shared values we would talk later in the concurrency part.


```
Program expressions;
```

```
int i, j, res;  
bool result,temp,temp1;  
int[1] x;  
  
{  
    i := 4;  
    j := i / 4;  
    x:=[4];  
    temp := (i == j);  
    temp1 := x[0] > j;  
    res := 4 mod i;  
    result := temp and temp1;  
}
```

```
Program expressions2;
```

```
shared int m;  
int i, j;  
bool res,fres;  
  
{  
    i := -4;  
    res := !fres;  
    m := j+i;  
}
```

Usage

Similarly to other programming languages, in Jafar, expressions are used to perform operations on the declared types or work as a condition for the if and while statement, a value assigned to variable or output of print statement. There are many expressions and they can come with different operators. The most basic expressions are atom expressions which can be a number for Int or true/false for Bool or an identifier of a variable. Then the programmer can compare that basic expression using (>,<,<=,>=,<=,!<=) operators, which are useful for the if, while statements and the functions and procedures. Moreover, he can perform operations on integers(+ , - , *).However, if these operators are used on the bool types or elements of arrays that contain bool elements, then it will result in an error during the type-checking phase. To perform operations on the boolean type we use (or, and, negation !). They cannot be used on integers and arrays with integer values, as the compiler will signal an error. Moreover, negation ! and – can be used as unary operators for booleans and integers. The (==) operators can be used on arrays as well.

Semantics

As mentioned above, in Jafar all the expressions will start with atom expressions which are Numbers or True/False values. For the more complex expressions which come with an operator, it will first calculate the value of all sub-expressions and then apply the operator to those values. In Jafar, all expression values can only be stored temporarily in the stack. This means every time moving out of the sub expression, its value will be on top of the stack.

Code generation

The code generation for expression can be divided into 4 parts:

- Atom expression: If it's a value like a number or true/false we just need to parse its value into an integer (which means true/false will be converted into 1/0) and then load them into a register and push the register value into the stack. For the identifier, we will first need to get its memory offset through CheckerResult and then load the value at that memory location into a register and push it to stack
- Operation expression: All operations contain 2 sub expressions. We first need to visit sub-expressions to generate code for them. Then the value of 2 sub expressions will be on top of the stack. Next, we will pop the value of 2 expressions and store them in 2 registers. The key remark here is that due to the stack mechanism the later visited expression will be on top of the former visited expression and it's really important in some operations like subtraction, less than, ... where the order of operands will affect the correctness of the operation.
- Parenthesized expression: The value of parenthesize expression is just the value of its sub-expression. Therefore, when visiting parenthesized expression we just need to generate code for its sub-expression.
- Array equality operation expression: It is a special case of expression that need to mention. Because there are 2 different expressions to access an array by value or by an identifier with index, we have in total 3 cases of comparison for arrays. And before handling those cases we first need to generate the code to check if 2 compared arrays are of the same size otherwise it will skip the part below.

Case 1: Array of values compared with array identifier: For this case, we first need to visit the array value expression and identifier later because when visit the array value all of its values will be stored in the stack(in reverse order). Then the offset of the array identifier will be placed on top of the array and we just need to pop it. To compare, we need to set the register to point to the last elements of the identifier array which can be done with the given size of the array and the popped offset. Now because the offset and the top of the stack are all the last elements of these arrays. We just need to generate code to compare each element one by one and use "and" operation with 1 "true value initialized" register. After each comparison, we will decrease the offset pointer of the identifier array by 1 and pop the top value array from the stack.

Case2: Identifier with identifier: For this case, we just need to visit both sub-expressions and only their start offset are placed on top of the stack. Because the equal comparison is assumed that the 2 arrays of the same size we just need to loop for all elements of the array, get their value with load and generate a comparison code for them(similar to the comparison part in case 1).

Case 3: Array value and array value: This is the most difficult case since we need to use the stack pointer to locate the value of the first array and second array element(detail of this mentioned in the Problem and Solution section). The comparison is similar to the 2 cases above.

The syntax/usage/semantic are written by Andrea, code generation is written by Minh, the implementation is shared by all team members.

3.4 Nested scopes:

Syntax:

In Jafar we have 3 types of scopes. Scopes that define the variables that are used everywhere, in all the blocks of the program – the global scope. Moreover, we have the block scope – variables declared inside a block cannot be accessed from out of the block. However, inside the block, in another block created in the block, they can be accessed. And lastly the function scope. With function I mean both the functions and procedures created by the programmer and the predefined functions while and if. (perbegin creates a new scope as well, it will be discussed in the concurrency part)

```
program nestedscope;

int x;
{
    {
        int y;
        int x;
        {
            int z;
            x := 1;
            {
                x := y;
                z := x;
            }
        }
    }
}
```

```
Program fib;

int result;

func fib(int n): int {
    int res1;
    if (n <= 1): {
        res1 := 1;
    } else {
        res1 := fib(n-2) + fib(n-1);
    }
    return res1;
}

{
    result := fib(7);
    {
        int result;
        print(result); // will print 0
    }
}
```

Usage:

The programmer can create a new scope using the brackets { }. And the scope is very useful for the following reasons: 1. Every scope creates an isolated name space, thus variables declared in a scope cannot be accessed outside of it. 2. In nested scopes the programmer can re-declare the same name as in outer scope, thus effectively hiding the outer declaration, which gives the programmer control over access to different variables from outer scope. Moreover, as seen in the example it is possible to define in another scope variables with the same name as already defined. However, in case we have nested blocks for example {bl 1 {bl 2 { bl3}}} and we already defined the variable with a specific type in bl1 or bl2, then declaration bl3 will overwrite the identifier in parent scope. In cases of parallel scope like {bl1 {bl2} {bl3}} in case we have defined a variable in bl2, in bl3, we don't have the

declaration restriction but also cannot access the variable of parallel scope, we can define it of any type defined in the language and finally for child scope like `{{b11}b12}`, scope 2 cannot access variables of child scope. In functions/procedures, the variables are considered as local variables in the function scope. A function itself can also be considered as a variable of the global scope which means that we cannot declare 2 functions with the same identifier.

Semantics:

The nested scopes in Jafar are defined using the symbol table with the stack and mapping mechanism to keep track of the declaration of variables within a scope. The storing strategy is only to put a new variable on the stack top. Everytime a scope is started with opening scope syntax `{`, it will create a new mapping on top of the stack. To put a new variable into the scope, the symbol table only allows one to put a new identifier into the deepest scope if it checks that there is no duplicated identifier. Everytime it needs to access an identifier, the symbol will start to find the variable from the top mapping and then go down to the bottom until it successfully finds the identifier. When closing the symbol table with syntax `}` the mapping in the deepest scope will be discarded.

Code generation

The nested scopes in Jafar are implemented using Stack and LinkedHashMap. A typical declaration of a symbol table is `Stack<LinkedHashMap<String, X>>` where the String is the identifier and X is any type that we want to map with an identifier which can be an offset or its type. The reason why we use LinkedHashMap is that LinkedHashMap will retain the order in which the key was inserted, which is really important to set up the offset for the variable. The use of the symbol table is mostly in the type-checking phase of the Checker to map the offset and type to the newly declared variable. Everytime we enter a block statement we need to consider it as opening a scope and stack for max offset in scope is initialized with 1 and pushing a new hashmap, and initialized on top of the stack. When putting a new variable into the stack, it will loop at the stack top to verify that the variable is not declared within scope and then assign the current max offset within scope (function case) or use max offset stack to calculate memory offset (sum off all max offset in all opening scopes) and finally putting it into scope. The hashmap and max offset will be popped out of the stack everytime we exit the block statement. The syntax/usage/semantic are written by Andrea, code generation is written by Minh and the implementation is done by Minh.

3.5 Basic statement(assignment/if/while/print):

3.3.1 Assignment

In Jafar, assignment is possible only inside a block, and not everywhere in the program. So, we cannot assign a variable a value when declaring it, it initially has a predefined value – which is False for variables of bool type and 0 for int.

Syntax

An assignment in Jafar has the form `var_name := value;`

```
Program assignment;

shared int m;
int i, j, res;
bool fres;

{
    i := -4;
    j := 1;
    res := ((i * j - (i mod j)) / (j + 33)) mod m;
    m := res + i ;
    fres := !fres;
    res := m mod res;
}
```

Usage

For an assignment to work the programmer needs to indicate the name of the variable that he wants to assign the value to and then on the right-hand side he will have to write the expression that will represent the value of the variable. As mentioned in the previous parts, the value that we are assigning to a variable should be the same type as the type of the variable, otherwise compiler will signal a type error.

Semantics

Assignment is used to give values to a variable. The value is an expression, so can be a constant, other variable or operations on other variables, as well as an array or function. The value is then stored in the corresponding memory offset of the target variable.

Code generation

Code generation for assignment required 2 sub steps. First is to visit the target and generate code to get its offset (from CheckResult or ARP in case of function) and push the offset to the stack and then assign an expression to get its value into the stack as well. To assign variables we first pop the first value from the stack which is the assign value and then the second popped value is the target offset. At this step we need to check if the target variable

is declared shared or not. If it's declared we need different instructions for the assignment(in Concurrency section below) otherwise we just need to use store instruction to save the value of expression to the indirect address in the register where we pop target offset to. There are 2 special sub cases for the assignment when the assigning value is an array.

Case 1: If the assigning array is an array of values, we need first generate code to set up the register to point to the last offset of the array because all pop assigning array values are in reverse order.

Case2: If the assigning array is an array with an identifier, we have to pay attention that the popped value now is not a value but the offset of the assigning array. We then need to loop through the all index of each array and assign them.

This part is mostly written by Andrea with the code generation being written by Minh. The implementation is shared by all team members.

3.3.2 *If*

Syntax

If expression: block1 (else block2)?

The expression here can be a boolean expression or a conditional statement.

Example:

```
1  Program arr;  
2  {  
3      int x,y,res;  
4      if (x >= y): {  
5          res := x-y;  
6      } else {  
7          res := y-x;  
8      }  
9  }
```

Usage

Jafar if statement does exactly the same thing with other programming languages if statement. It can be used with or without brackets on the condition statement: $(x > y)$ for example. If statements also support multiple conditional statements, for example: *if* $(x > y$ and $x == 1)$ { bla bla }. However, Jafer programmers need to make sure the conditional

statement results in a boolean expression. Jafar' if statement also supports branching represented by ***“else”*** which indicates the branch that the program will execute if the condition is false.

Semantics

If statements check for the condition, if the condition holds or by boolean expression equals true then the code inside *block1* will be executed. If ***“else”*** is added and the condition does not hold, code inside *block2* will be executed. If ***“else”*** is not used and the condition does not hold, nothing will be executed and the program jumps to the next line of code after the ***“if”*** statement.

Code generation

Code generator will first emit code to get the condition boolean expression on the top of the memory stack, note that we visited the condition expression before and push the result to top of the stack. Then we create a label named ***“end label”***, which represents the end of the whole if statement. Hope that you can still remember the label system we mentioned before.

Since we can choose to have ***“else”*** or not, we have two different ways to generate code. If ***“else”*** is not included, we simply check the condition, branch to ***“end label”*** if the condition is false, otherwise execute the code inside the *“block1”* statement. We need to do some extra work if ***“else”*** presents. The Jafar program will generate a *“else label”* label, which represents the line where *“block2”* starts. The program will then branch on the condition, if the condition does not hold, we jump to ***“else label”***, execute *“block2”* code and finish by visiting the ***“end label”***. Otherwise, the program executes *“block1”* and jumps to the ***“end label”***.

- To simplify, the generated linear code looks like: ***“condition”*** code -> Branch ***“condition”*** if false jump to ***“else label”*** -> *“block1”* code -> Jump to ***“end label”*** -> ***“else label”*** -> *“block2”* code -> ***“end label”***

This part is written and implemented by Duc.


3.3.3 While

Syntax

while *expr*: *block*

- Expr here represents a conditional statement or a boolean expression.
- Example:


```

3      int x, y, ans;
4
5      {
6          x:= 60;
7          y:= 36;
8          while x != y :
9              {
10              x := y
11             }
12         ans:=x;
13     }

```

Usage

Jafar while loop does exactly the same as other programming languages while loop.

Semantics

Jafar while loop continuously checks for a condition and executes the body code (“*block*”) until the condition does not hold. The Jafar Program will then continue executing the remaining code outside of the while loop.

Code generation

Code generator first jumps to the conditional statement check, marked by the “*cond*” label. Here, the Jafar program will generate Sprockell code for a branch on the conditional statement. If the conditional statement holds, jump to “*body*” and execute “*block*” code and go back to the branch on condition. Otherwise, if the conditional statement does not hold, the program will exit the while loop and start executing the remaining code.

- Simplified Linear Code: Jump “**cond**” -> “**body**” : Nop -> “*block*” code -> “**cond**” : Nop -> “*condition*” calculation code -> Branch on “*condition*” jump to “**body**” if true -> end of while.

This part is written and implemented by Duc.

3.3.4 *Print*

Syntax

```
1 Program print;  
2 {  
3     int x;  
4     print(3+5);  
5     print(x);  
6     print(True);  
7     print(2==1);  
8 }
```

Usage

Jafar has print statements with similar syntax to Python. Print statement can be used to print any local variable or expression which will then be a numeric number (which means True will be printed as 1 and False is 0).

Semantics

The print statement in Jafar makes use of WriteInstr numberIO in Sprockell to print the result into the terminal.

Code generation

For each print statement we will first generate the code to calculate and push the output expression into the stack and then pop the value out of the stack to print out. There is 1 exception when we have to print out the array. If the array is accessed by index and identifier, the popped value is the first array element offset, we then need a for loop to get all element offset and value to print out. If the array is accessed by value, we need to use the stack pointer to get the first element which is not at the top of the stack due to reverse order push. Then we need the 2nd for loop to pop all elements and discard them.

This part is written and implemented by Minh.

3.6 Concurrency and shared variables:

Syntax

- Parbegin-parend:
 `parbegin:`
 `block+`
 `parend;`
- Lock/ unlock: `lock()/ unlock();`
- Shared variables: `shared type id;`

```
1  Program nestedThreads;
2
3  shared int a;
4  shared int b;
5  shared int c;
6  {
7      a := 67;
8      b := 89;
9      parbegin:
10     {
11         parbegin:
12         {
13             lock();
14             a := a + 1;
15             b := b + 1;
16             print(a);
17             unlock();
18         }
19
20         {
21             lock();
22             a := a + 12;
23             b := b + 16;
24             print(b);
25             unlock();
26         }
27     }
28     parend;
29 }
30 {
31     c := 19;
32     print(c);
33 }
34 parend;
35 }
```

Usage

For the concurrency, Jafar supports parbegin, parend model and locking. In between parbegin and parend, programmers can declare *multiple blocks* which then can be run concurrently. Jafar program will wait until all threads finish to continue executing the main thread. Jafar also supports locking inside a thread, a thread will grab the lock on its parent thread, prevent other sibling threads from executing and then release parent thread lock on unlock. Jafar's threads however, can only use **shared variables**, declarations and usages of non-shared variables are prevented. Though a limitation of JAFAR is that an upper bound of six threads, including the main thread can be created, nested parbegin-parend threads are supported by Jafar.

Shared variables are also introduced to work in a multi-threaded environment, they can be used by any thread, at any scope level.

- Shared variables have to have unique names among other shared and non shared variables.
- Shared variables have to be declared before the body part of the program. E.g: before opening the first scope.

Semantics

Parbegin-parend is used to create multiple threads, which then be executed concurrently. The parent thread has to wait for all child threads to finish before continuing executing. Lock and unlock is introduced to prevent two or more sibling threads from executing concurrently. In order to support six threads and six locks, we extended our shared memory to the size of 48:

ThreadBeginWait	ThreadEndWait	LockZone	Shared variables
0 -> 5	6 -> 11	12 -> 17	18+

ThreadBeginWait and ThreadEndWait locations are used for parbegin-parend to enforce parent thread to wait for its children. LockZone meanwhile is used for lock/ unlock strategy. Shared variables are stored in Sprockell shared memory. Any thread can read and/ or write to a shared variable. Shared memory offset starting from 18, increases by size of the last variable whenever a variable is declared, up to maximum offset of 48.

- Because of Sprockell read and write to shared memory behavior, we need to use ReadInstr + Receive and WriteInstr to interact with the shared memory.
- Shared variables all together can occupy up to 30 cells of shared memory, which is a reasonable size for a medium sized program.

Code generation

Checker phase: On checker phase, checker will store a mapping from a thread to its responsible parse tree. Any statements inside a parbegin-parend thread block will inherit the same thread number unless it creates a new thread. The counter of threads is used for checking if max threads reached.

Generator phase: Since we already have a mapping from a thread, which is represented by Sprockell Program, to its parse tree, we can easily emit code to the corresponding Program. Therefore, Sprockell can run the code concurrently using multiple Program instances. Jafar will make sure to generate a Sprockell Program for each of its threads. For the actual run time strategy, first the parent will write 1 to shared memory location (**ThreadBeginWait + childThreadNumber**), the program/ thread with childThreadNumber will busy wait until it receive 1 on (**ThreadBeginWait + childThreadNumber**) shared memory cell. Only then, it can start to execute its own code. Upon finishing its own code, the child thread will write a 1 to (**ThreadEndWait + childThreadNumber**) shared memory location. The parent thread keeps looking at (**ThreadEndWait + childThreadNumber**) of all its children threads. If the parent thread sees all of its children Thread End indication slots filled with 1, it can

continue to execute, otherwise the parent thread will have to go back to the condition and perform checking again.

- For locking, a children thread first performs a TestAndSet on **LockZone + its ParentThreadId**. If we can perform a test equal 0 and set to 1, which means no other threads executing and the thread can grab the lock and execute its own code. Otherwise, jump back to perform the test and set again. When unlock is called, the thread simply writes back 0 to **LockZone + its ParentThreadId** to indicate the lock is freed and other threads can grab the lock.
- All the read and write to perform Parbegin Parend and lock/ unlocking required Sprockell ReadInstr, Receive and WriteInstr to actually exchange information between a Thread/ Sprockell program to Sprockell shared memory.

For shared variables, we need to distinguish between shared and non-shared variables in every call of an identifier to decide whether we need ReadInstr + Receive and WriteInstr or Load and Store to actually load it to a register. Moreover, the shared offset and size of a shared variable also need to be received somehow. Luckily, checker result provides a convenient and simple way to check whether a variable is shared or non-shared, what is the variable offset by looking at the sharedOffset map which maps a shared variable with its offset on shared memory. The size of the shared variable can also be hooked from the same check result with the sharedType map.

This part is written and implemented by Duc.

3.7 Division/Modulo:

Syntax

- Division: /
- Modulo Operation: *mod*
- Example:

```
1  Program divmod;  
2  int x,y,ans;  
3  {  
4      x:=5;  
5      y:=2;  
6      ans:=x/y;  
7      ans:= 12/5;  
8      ans:=3 mod y;  
9      ans:= x mod y;  
10 }
```

Usage

- `/`: Basic operation to calculate rounded down division between two integers.
- `mod`: Basic operation to calculate modulo of two integers.
- Note that by nature, division and modulo by 0 is not allowed and will be forced to terminate by Jafar programming language.

Semantics

Since Sprockell does not support division and modulo between two integers, we have to come up with our own mechanism for calculating division and modulo. We decided to create a “soft” division, which will round down the result of a division between two integers because of the lack of support for float numbers by Sprockell. The idea is simple: a soft division of “ a / b ” can be easily calculated by having a *counter* which increases by 1 everytime a subtraction “ $a - b$ ” can be made given that “ $a > b$ ”. When “ $a > b$ ” is not satisfied, the *counter* will represent the result of soft division “ a / b ”. The same idea goes to the modulo of “ $a \bmod b$ ” but instead of a counter representing the result of this operator, “ a ” will be the expected result.

Code generation

To calculate “ a / b ” or “ $a \bmod b$ ”, first we load both “ a ” and “ b ” to a register and do a comparison of “ $a > b$ ” (1). If it is true, we continue to the next step (2), otherwise jump to the end of the calculation and push the result, which is either “*counter*” or “ a ” respectively to the memory stack. At (1), we do a calculation “ $a - b$ ” and store the result to register containing “ a ” and do a sum of “*counter*” with “1” and store the result to “*counter*” register, note that subtraction and addition is supported by Sprockell naturally. We jump back to step (1) and do the same steps over again. In the end, the result of “ a / b ” or “ $a \bmod b$ ” will be pushed and available on top of the memory stack.

- An interesting thing here is that (1) and (2) represent exactly the label used to perform jump and branch in our code generation phase.

Jafar will check for division and modulo by 0 during the generation phase, on operators of “ a / b ” or “ $a \bmod b$ ”, if “ b ” is 0, the Program will jump to EndProg and terminate without executing any further code.

This part is mostly written by Duc. The implementation is shared by all team members.

3.8 Procedure/Function:

Syntax

- func [function name](type params; type params): return type {
 //block content
 return [value/variable];
}
- Example

```
1  Program function;  
2  int sum;  
3  func add(int x,y):void {  
4      sum := x+y;  
5      return;  
6  }  
7  func fib(int n): int {  
8      int res1;  
9      if (n <= 1): {  
10         res1 := 1;  
11     } else {  
12         res1 := fib(n-2) + fib(n-1);  
13     }  
14     return res1;  
15 }  
16 {  
17     int fib7;  
18     add(2,3);  
19     print(sum);  
20     fib7:= fib(7);  
21 }
```

Usage

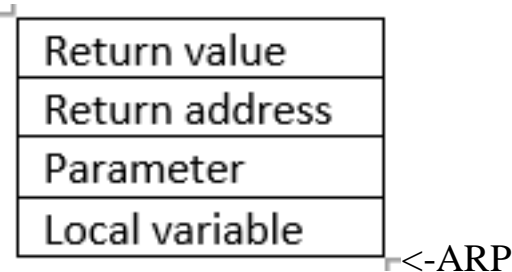
Function and procedure in Jafar are similar to Java and Python. All the function structures will start with the keyword “func” followed by a function identifier with its parameter and finally the return type(Array/Int/Bool). For the procedure, the return type is set to Void and no expression follows “return” keyword. There are some restrictions on function usage in Jafar. The first thing is that every function needs and only has 1 “return” keyword at the end

of the function because the return is not declared separately in the grammar as an expression but combined into function grammar. The reason for the design choice is that we don't want the return expression to be misused elsewhere except function declaration. The second thing is for "non-Void" function we need to return an expression and the result should be assigned to some variable (line 20 of syntax) and for "Void" procedure, it shouldn't have any expression after return (line 5) and only be used like a statement and not assigned to any variable.

Semantics

A function is a module of code that complete a specific task with the given parameters and its local variables and can be called anywhere and even called recursively. For each function, we can use it as an expression for calculation/assignment or a statement which can be reused at any time for different contexts.

For each function call, the memory needs to extend the ARP to save memory space for a function activation record. The activation record of function in Jafar will look as follow



The size of the activation record = size of return type + size of all parameters and local variable type + 1 for offset of the return address. The size will be defined at the elaboration phase when it has the information about the function when parsing function declaration and the new ARP point to the last offset of the activation record.

Code generation

The code generation of the function calls will have 3 main phases:

- Pre-call: The function will first visit all of its parameters' expr to calculate their value and push them into the stack. After popping them from the stack, we store the value of parameters in memory and because we have stored all parameter local offset in function scope

$$\text{offset} = \text{currentARP value (always in register F)} + \text{param local offset} + \text{return size} + 1 (\text{for return address}).$$
 After storing all parameters, we will extend the ARP with the formula above (in semantic). For the return address, we will first calculate the return address by first getting the current number of instructions + 5 (which will point to future NOP instruction). Then calculate the location and store the return address in memory based on ARP (it will take 4 instructions)

$$\text{return address} = \text{ARP} - \text{parameter/local variable size}$$

Next, we will get the function declaration label and generate the jump to the execution of the function (take 1 instruction). Finally, we will generate the NOP instruction which is the return address after finishing the function call

- Function declaration: The first step before all function declaration is that we need to generate a jump instruction from the start of the program to the main block so that it'll skip all the instructions of the function declaration and only go back when all parameters are properly initialized. For the function declaration, all the expressions and statements are generated as normal except the return expression. For the return expression (only available for non-Void function), we will first visit the and generate code for the expression, then upon returning we will generate code to target the return value field's offset in memory.

$$\text{return value's offset} = \text{ARP} - \text{param/variable size} - 1(\text{address})$$

 Next, we will pop the value of the expression from the stack and store it into the return value's field. Finally, the most important part of the function declaration is generating code to jump back to the return address which is taken from the activation record.
- Post call: At the post-call, we will first get the return value and then push them into the stack if it's a function expression. After that, we will destroy the activation record by reducing the ARP.

This part is written and implemented by Minh.

4. *Description of the software:*

For the software, we have in total 20 Java files (excluding Main.java) and divided into 3 main packages: compilation, exception and model.

4.1 *Compilation:*

The compilation package is the core of the project where all classes are used to convert Jafar to SPRIL instructions and finally execute them.

- **Checker:** The checker is responsible for the elaboration/type checking phase of the front end. Its job is to visit all parse tree nodes in post order to define their type and offset attribute rules in a **synthesized** way. For each node the offset will be defined based on how many variables have been allocated and the current scope, the type will be based on the type of child expression. The checker also verifies the context-sensitive rules, for example: if the type of expression is matched with the context or whether the variables are declared in current scope and it will add and throw errors and the end of the checking phase if there are any invalid expressions/statements.
- **Symbol Table:** This class acts as a supporter for the Checker. It is used to keep track of declaration scope to calculate the variable offset and not allow multiple declarations of the same id in the same scope. It also used to get the offset and type of any identifier if it exists in parents scope. There is 1 remark about the symbol table in the function case: it needs to set the offset locally for function scope instead of the memory offset.
- **CheckerResult:** This class is the result of the Checker after type checking. It will store all necessary information which is later on be used in the code generation. It use the HashMap, ParseTreeProperty data structure to map a given variable identifier or parse tree node to its value.
- **Generator:** The generator is used to generate the SPRIL instruction from the parse tree of Jafar with the help of CheckerResult. The Generator uses the TreeVisitor to visit all parse tree nodes with the freedom of order. The result of the generator is then a list of programs for each thread and each program is the list of SPRIL instructions.
- **Compiler:** It works as a bridge between Jafar and SPRIL. It will first parse the Jafar program according to Jafar grammar to get the parse tree, then it will call checker to get CheckResult and finally call Generator with CheckResult to generate the list of SRIL instructions.
- **HaskellFileGenerator:** This class is used to generate a Haskell file(.hs) from a given Jafar file(.jafar). It is also possible to run the haskell file with the support of ProcessBuilder from java.io package. In the default setting, HaskellFileGenerator will read Jafar file from **sample** package, write SPRIL

instructions generated from the compilation into **samplespril** package, and write the final result of executing SPRIL with Sprockell in **sampleoutput**

- **Type/TypeKind:** All variable types supported in Jafar (Void/Int/Bool/Array). It is used to get the default how much memory should be allocated to the given type.

This part is written by Minh. The implementation is shared by all team members

4.2 Exception:

The exception package contains all exceptions which are thrown during the compilation process and format the error message in human readable text and in an instructive way to find the errors.

- **ErrorListener:** The list of errors from Checker and is formatted to get the line of occurring errors.
- **ParseException:** The list of errors occurring from parsing Jafar programs according to the grammar.

This part is written by Minh. The implementation is based on exception from compiler construction block 6 and shared by all team members

4.3 Model:

The model package contains 10 classes which model needed data types to construct all SPRIL instructions.

- **Address/Operator/Reg/Target/OpCode** they model exactly data type **AddrImmDI/Operator/RegAddr/Target/Instruction** respectively in Sprockell Hardware Types.
- **Instr/Op:** They are abstract classes of a fully crafted SPRIL instruction
- **Program:** Each Program object models exactly one SProckell program. It consists of a list of multiple Instr/Op-s and a label-line mapping to work with labels. Program is also responsible for storing its own thread id, in order to generate multiple Sprockell programs for multithreading purposes.
- **Label:** Models label object with Label name and its corresponding line number inside a program.

This part is written by Minh. The implementation is shared by all team members

5. Test plan and result:

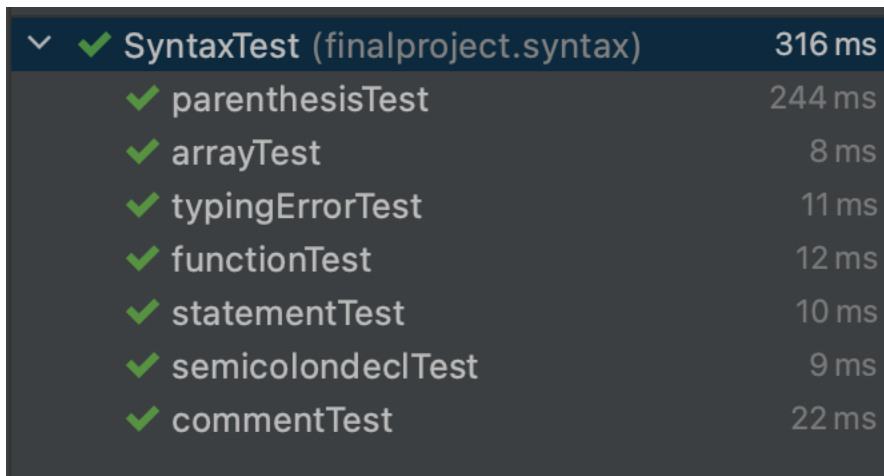
All the test files in Jafar are JUnit tests and can be run automatically. They are all located in “src/test/finalproject”. In here you can find all the packages of separated tests. For each package and test case we have different structure as explained below

5.1 Syntax test:

Result:

For the detailed result of the tests and the test cases, you can go to

src/test/java/finalproject/syntax/SyntaxTest.java. Here is the result of running all syntax junit tests:

A screenshot of a JUnit test runner interface showing the results for 'SyntaxTest (finalproject.syntax)'. The test suite is expanded, showing a list of sub-tests, each with a green checkmark icon indicating success. The sub-tests and their durations are: parenthesisTest (244 ms), arrayTest (8 ms), typingErrorTest (11 ms), functionTest (12 ms), statementTest (10 ms), semicolondeclTest (9 ms), and commentTest (22 ms). The total duration for the suite is 316 ms.

✓ SyntaxTest (finalproject.syntax)	316 ms
✓ parenthesisTest	244 ms
✓ arrayTest	8 ms
✓ typingErrorTest	11 ms
✓ functionTest	12 ms
✓ statementTest	10 ms
✓ semicolondeclTest	9 ms
✓ commentTest	22 ms

Test plan and result explanation:

In comparison with Concurrency Tests, Semantic Tests and Contextual Tests, for the Syntax Tests we do not have sample files that are showing that the syntax is correct. We took this decision as syntax errors are usually related to really small things, as spelling errors in the lexical syntax or language – construct errors. Because of this we zoom in to the level of blocks and statements, so that it would be really clear where the error appeared. Moreover, the failed tests have a syntax similar to the assertEquals, for the reader of the tests to understand what kind of syntax error there is in the block. We have used in the SyntaxTest 3 methods that help us to check the syntax. The most important method is checkSyntax(): it takes a string which represents the body of the program and creates a lexer and a parser from the stream of the string. After that we remove any previous error listeners and add the customized ErrorListener to the Jafar Lexer and Jafar Parser. After this, if there are any errors caught by the ErrorListener, then we throw a ParseException. Now, the customized methods fails and pass are created in order to be able to test the syntax and are pretty similar to assertEquals(for the fail and error that we get) and assertTrue. fails(String s,String exp) takes a string s and passes it to the checkSyntax. We know that the provided string of body s is not correctly written syntactically, so for sure a parse exception would be caught. After it

is caught, fails would assert true. The assertEquals and exp is purely symbolical, for the reader of the failed tests to understand which syntax error is obtained. The pass is really similar to assertsTrue. It passes the string to checkSyntax and if a parseError is caught, asserts false.

Here are the details of every test method and what it covers:

- **commentTest()**: the comment test is checking comments in Jafar. It checks that comments work for both a small comment and a very large one. Moreover the tests fail in case \n is missed which is the symbol for newline and in case the comment does not have 2 backslashes //.
- **semicolondeclTest()**: the semicolon test checks that one of the most common syntax error in most of the programming languages – missing the semicolon will result in a syntax error. We checked for cases in which instead of semicolon another sign is used, in case 2 semicolons are used one after another, etc. errors that appear a lot of times during programming.
- **paranthesisTest()**: checks that all functions that need parameters inside parenthesis, fail in case their parameter is not given inside parenthesis.
- **typingErrorTest()**: the typing error test shows that in Jafar typing errors are recognized and result in a syntax error. The tests contain the most popular typing errors that programmers have like opening parenthesis and not closing them, instead of assigning values using :=, uses by mistake =, mistyping keywords like funv, instead of func.
- **statementTest()**: checks syntax errors that programmer can make when writing statements in Jafar, shows that correct statements would be parsed.
- **functionTest()**: shows that functions in Jafar are able to be parsed, and in case programmer misses for example : in the definition of functions, or does not have a return type or a return keyword in function block – it will result in syntax error.
- **arrayTest()**: shows that arrays both simple and multidimensional are parsed correctly in Jafar and in case programmer misses to write for example the size of array, or is assigned something not specified by the grammar – it will result in syntax error.

5.2 Contextual Test:

Result:

For the detailed result including the source code of the program, you can go to
“src/test/finalproject/contextual/jafarprogram”

The junit test is located in “src/test/finalproject/contextual/ContextualTest.java”

Here is the result of running all contextual junit tests:

✓ ContextualTest (finalproject.contextual)	1 sec 152 ms
✓ testArrayIndexOffset	686 ms
✓ testArrayTypes	162 ms
✓ testBasicTypes	37 ms
✓ testFunctionReturnType	37 ms
✓ testIfWhileContext	17 ms
✓ testFunctionScope	32 ms
✓ testBasicOffsetsInScope	10 ms
✓ testNestedScopeOffsets	3 ms
✓ testComment	0 ms
✓ testArrayIndexContext	13 ms
✓ testNonSharedVariableInConcurrency	13 ms
✓ testOperationContext	32 ms
✓ testPrefixContext	19 ms
✓ testFunctionOffset	10 ms
✓ testSharedVariable	35 ms
✓ testNestedScope	13 ms
✓ testThreadCreationErr	5 ms
✓ testDeclarationErr	28 ms

Test plan and result explanation:

To make contextual tests for Jafar, we only need to parse the source program file and then call Checker on the result parse tree. The Checker will then do all the job of type checking and result in a CheckerResult and a list of type-checking errors. For each test case, If we want to check the incorrect context we will define our own expected error message and check if it is contained in Checker's list of errors. If we want to check the correctness of the Checker result(eg: offset, type assignment, ...) then we will get the parse tree and call CheckerResult on the parse tree node that we want to test.

Here is the detailed of every test cases and what it covers:

- **testBasicTypes:** This test covers the assignment and declaration of basic type. For the test we check if the type of declared variable is stored in CheckerResult as it was defined. We also check failed test cases where we try to assign a value different type to a variable and the checker can recognize all of them as errors correctly.
- **testArrayType:** This test covers the assignment of arrays, especially multi-dimensional arrays. We have checked if the elements of a multidimensional array are of expected type (possible children is also a multidimensional array), assignment of an array is an array with different size. The test also cover some edge cases that can be

checked at elaboration phase which are index expressions not of type integer and accessing more dimensions than declared.

- **testArrayIndexType:** This test covers the part that all accessing indexes of an array are of type integer and errors when accessing with indexes of different types.
 - **testArrayOffset:** This test to check if an array is accessing with an identifier even though with indexes, the offset of the main array's 1st element as intended.
 - **testIfWhileContext:** This test covers the check for condition of if/while are of type Bool and errors when assigning different types to the condition of if/while.
 - **testPrefixContext:** This test checks whether the unary minus and the not symbol only work with boolean and integer type as expected. The errors are about using the prefix operation with wrong types.
 - **testOperationContext:** This test covers the test for all operations in Jafar and the results of these operations are of the expected types and both of the operands have the same type. This also checks if the "==" operation works for all types. The errors are about applying the operation in the incorrect type or the left and right operands are not of the same type.
 - **testFunctionScope:** This test checks how a function can access its parameter/local variables and global variables. The error is when the function accesses a variable declared in a parallel block(main block/different function) or 2 function with the same name are declared.
 - **testFunctionReturnType:** This test are all errors when trying to return a different type of value than the declared one, forget to return a value or still return value even though it's a void function. It also checks the context where procedure and function are used. Function should always be used as an expression and assign value to a variable upon returning while procedure should be used like a statement without any assignment.
 - **testFunctionOffset:** This test is to check whether variables inside a function are assigned local offset instead of memory offset and check if the offset is assigned correctly. The special case is that array is declared and the offset for the variable declared after this array should be increased by the array size instead of 1 like normal primitive type.
 - **testBasicOffsetInScope:** This test checks if all declared variables in basic scope have correct offset. The special case is that array is declared and the offset for the variable declared after this array should be increased by the array size instead of 1 like normal primitive type.
 - **testNestedScope:** This test checks nested scope declaration are accepted also including accessing variable of parent scope or declare variables of the same name in parallel scope.
 - **testNestedScopeOffset:** This test checks variables in different nested scope are correctly assigned. An important remark is that variables in parallel scope can have the same memory offset because when we reach

the 2nd scope the 1st scope is already closed and the memory for the 1st scope is released.

- **testComment:** This test shows that the single line comment works as intended by skipping all the next instruction and parsing without errors.
- **testDeclarationError:** This test shows all edge cases of declaring or assigning a variable including: same identifiers declaration in the same scope, access an undeclared variable, access variable declared in parallel scope or closing scope.
- **testSharedVariable:** Test cases address possible cases where the declaration of shared variables is not in the correct place (E.g: inside a scope, inside threads ..). It also contains the cases where shared variables are misused.
Shared array test is included in testSharedVariable to test our correctness over shared array.
- **testThreadCreationErr:** The main program (tid 0) creates 3 threads, first child thread (tid 1) creates 3 more threads, bringing the total number of threads exceeds the maximum number of threads which is 6. Thus, the thread creation will fail and return an error message.
- **testNonSharedVariableInConcurrency:** Both test cases address the same error: usage of non shared variable in concurrent context. For nonSharedMissuseErr1, x is declared as a non-shared variable but used in threads. Meanwhile in nonSharedMissuseErr2, variable x is declared inside a thread, which is prohibited.

This part is written by Minh. The implementation is shared by all team members

5.2 Semantic Test:

Result:

For the detailed result including the **source code of the program**, **generated spril code**, **result output** you can go to **src/test/finalproject/semantic** directory where **jafarprogram**, **spril** and **output** are the expected test package results respectively.

The junit test is located in **src/test/finalproject/semantic/SemanticTest.java**

Here is the result of running all semantic junit tests:

✓	SemanticTest (finalproject.semantic)	1 min 34 sec
✓	testArrayIndexOutOfBounds	13 sec 720 ms
✓	testGCD	5 sec 907 ms
✓	testPrime	6 sec 159 ms
✓	testMatrixMultiplication	6 sec 83 ms
✓	testArraySort	5 sec 655 ms
✓	testNumDaysFebruary	7 sec 445 ms
✓	testFibonacciFunctionRecursion	5 sec 570 ms
✓	testVectorAddition	5 sec 708 ms
✓	testDivisionByZero	10 sec 681 ms
✓	test3DArrayCompare	6 sec 226 ms
✓	testDivision	5 sec 592 ms
✓	testModulo	5 sec 130 ms
✓	testModuloByZero	5 sec 414 ms
✓	testSimpleSumProcedure	5 sec 93 ms

Important note:

- Running semantic tests can take up to 50s due to a lot of test cases.
- **testInfiniteLoop** is located at the end of semantic test and is commented out because it'll return **failed as expected result**. Because we use timeout and if the test will not be terminated in 5s it will timeout and return fail

Here is the result for infinite loop test

⚠	SemanticTest (finalproject.semantic)	5 sec 97 ms
⚠	testInfiniteLoop	5 sec 97 ms

Test plan and result explanation:

To test the semantics of our program, we first put the Jafar program into “*semantic/jafarprogram*” with the format “[*filename*].jafar”. The program is then generated into SPRIL instructions by the compiler to semantic/spril with format [*filename*].hs. We then use ProcessBuilder in Java to run all generated SPRIL files and get the output result with Process.getOutputStream. The result will then be returned by buildAndRunJafar function and also be written to package output with the format [*filename*].result.txt. For every test we will compare the running result with the given expected result to check its correctness.

The semantic tests try to cover all the features of our program in all of their test cases including their edge case

Here is the detail of every test case and what it covers:

- **testNumDaysFebuary:** It's a basic algorithm that makes use of "mod" operation and "basic function" call with 1 input parameter as the year. We test is with input "2020,2021,2022,2023,2024" and get "29,28,28,28,29" respectively as the correct result.
- **testPrime:** It's again a basic function but it makes use of "soft division" in combination with "if/while and function". We have tested it with different input numbers and all the checks are correct. Note: The result will be 1 if the input is a prime number
- **testGCD:** This is a test that finds the greatest common divisor of 2 numbers which make use of "soft division" and "if/while" but placed in a basic block instead of a function. The result is 12 as the gcd of 60 and 36
- **testArraySort:** This is a test to check array assignment/array accessing of 1D array in combination with while loop. The test result show that array [11,3,8,7,1] is sorted into [1,3,7,8,11]
- **testMatrixMultiplication:** This test is to show the correctness of "multi-dimension" array in which we multiply a 2x3 array with 3x2 array and the result is a 2x2 array with correct values. It also shows that variable can be declared in "nested scope" and the lower scope can access variable of higher scope.
- **test3DArrayCompare:** This test covers the "equal comparison" of array feature for 1 and multi-dimension. The test also shows that arrays are compared by value not by the offset in memory
- **testArrayIndexOutOfBounds:** This is to test the edge case and "**runtime error**" of the array when we access an index larger or smaller than the array size. The result shows that no code is executed after the runtime error and the program terminates immediately.
- **testVectorAddition:** This test shows how "function" can properly receive "arrays as input" arguments and "return an array" as result. The result show that 2 arrays [9,3,5] and [2,4,15] are [11,7,20]
- **testSimpleSumProcedure:** This test shows how the return "Void" works in Jafar and shows that the function can access variables of parent scope(global in this case). The result is that the global variable sum is written as 97 which is the sum of 24 and 73
- **testFibonacciFunctionRecursion:** This test shows how "function recursion" works in Jafar. The results are 8 and 21 which are the 5th and 7th Fibonacci numbers.
- **testDivision/Modulo:** These 2 tests show how "soft division" is implemented in Jafar and check the correctness of the result. For each test we've tried 8 operations and they all return correct answers.
- **testDivision/ModuloByZero:** These 2 tests cover the **runtime error** when try to divide by 0. If it's not handled properly the program will be in an infinite loop. The result of these tests show that all test cases are

terminated and the check for division by zero works correctly which will skip all next instructions and jump directly to the end of the program.

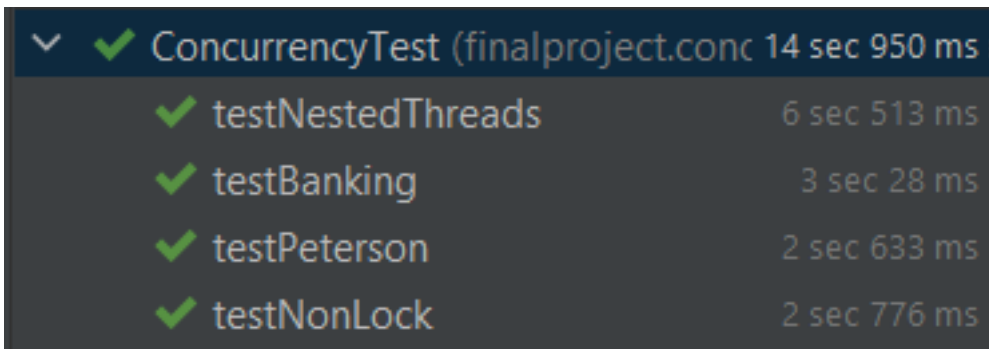
- **testInfiniteLoop:** this test is commented out by default. This test shows that when we try to make an infinite loop the program works as intended. We've set the run out time to 5s but it can be set to higher value and still works correctly.

This part is written and implemented by Minh.

5.2 Concurrency Test:

Result:

For detailed results visit "*src/test/java/finalproject/concurrency*". In here, you found "*jafarprogram*" which is the Jafar source file folder, "*spril*" folder contains Sprockell source files and also the haskell file generated from our Jafar language and finally "*output*" folder contains all the result by running the corresponding Jafar program without debugging. You can run "*ConcurrencyTest.java*" for quickly test all the jafarprogram-s given:



✓	ConcurencyTest (finalproject.conc 14 sec 950 ms :
✓	testNestedThreads 6 sec 513 ms
✓	testBanking 3 sec 28 ms
✓	testPeterson 2 sec 633 ms
✓	testNonLock 2 sec 776 ms

Test plan and result explanation:

To set up a method inside Concurrency test, which should do the following: first getting the result in form of String by using "*haskell.buildAndRunJafar(String filename)*". Noted that this method also generate a haskell file in "*spril*" and the result file in "*output*" for debugging convenience. Next, you have to create an expected result, which is a string, you can check whether Jafar's result contains the expected string, or equal, or not equal, it is up to you to decide.

For our predefined test, four cases were given:

- **testNestedThreads:** the purpose is for checking whether nested thread and lock inside nested thread working. "*a*" initialized by integer 67, "*b*" by 89 and "*c*" by 0 (default value). Since the second thread prints "*c*" running concurrently with the first thread, it is possible that "*c*" will be printed out before "*a*" and "*b*". Reassigning and printing of "*a*" and "*b*" is inside a child parbegin-parend of the first thread and is protected by a "sibling lock", "*a*" will first be printed as integer 68 and then "*b*" as integer 106.
- **testPeterson:** the purpose is to verify the correctness of a concurrent system using Peterson's algorithm. The program is expected to print a

sequence of numbers from 0 to 10, which reflect the correctness of Perterson's algorithm.

- **testBanking:** testBanking verifies the correctness of *lock()* and *unlock()*. For this test, we initialize the "*balance*" as integer 0, and create three threads concurrently modify the balance. Since we have 10 times increase balance by 10, 10 times decrease balance by 5 and 10 times increase balance by 20, the result balance should be 250 with *lock()* and *unlock()* working.
- **testNonLock:** the purpose of this test is to prove we need *lock()* to prevent data races. The same procedure with testBanking, however here we only have two threads: one increases balance 15 times, each time by 10 and one decreases balance 15 times, each time by 5. Since operations inside each thread are not guarded by a lock, there could be data races happens and the final result can be different from **75** ($15 * 10 - 15 * 5$).

This part is written and implemented by Duc.

6. Conclusion:

The project is new and interesting but requires a lot of hard work. We managed to finish all the mandatory parts but also some of the additional features. Though there are still limitations, Jafar works quite well with simple programs, which really make us proud of. We understood more about what happens when a programming language compiles and runs, especially when we run a concurrent program.

Reflection:

Duc: Every aspect I have learned in this module is fresh and interesting for me. However, for the concurrent programming part, it is not actually close to our project, we have to research a lot to actually come up with our solution, which I feel is a very interesting way to communicate between threads. Compiler construction is super nice, functional programming is interesting to have. However the logic programming part is not really useful, we just learn it for one or two days, work on the project for a week and then that is it, not even related to other components. This should be separated in my opinion. To conclude: interesting module but need a little bit more organization.

Minh: For our language, I'm proud that we can create our own programming language with all basic features to support basic calculations. If I have more time, I want to clean up our code and combine some parts to make it look better. We also want to support String and Char for our programming language if possible. For the module I feel it is quite tough and intense. However, I really like the LP and CC project, particularly the writing function with pattern matching and memory model for activation record. For the FP and CP, I found that the recording lectures from previous years are quite out of date and should be updated because some students might use them to support their studying but the lecture with out of date contents might affect that. I also suggest the teacher to host more QnA sessions.

Andrea: To be frank, I have started this module not because of my passion for compiler construction, concurrent programming or functional programming, but because this is the single elective I can take in order to go for the semester abroad and still finish in 3 years. Before starting it I knew that it is a really difficult module, all my friends were convincing not to take it, and I had quite bad expectations from it. How wrong I was, I have enjoyed this module a lot! I agree that the workload was quite intense and I am really lucky that I had really good partners that have supported me a lot, helped me grow, and when I was ill, helped me a lot. Otherwise, I think that the projects would have been a lot more difficult. I really enjoyed the cc and fp components, they were really well organized and connected to the exercises in the module guide. The concurrent programming part I believe that needs to be changed a bit, as the exercises in the module guide were really confusing and I needed the help of the TA's a lot to understand what is going on and how to solve them. Now about our language. I believe that Jafar is a really beautiful language and I would love to code in it. It has in my opinion a really nice-looking syntax, being able to take the best from its parents Java, Python and Pascal. Overall, it was really fun to work on it, and I believe that this project and generally this module is really important for us as programmers, to understand what is actually happening when we are writing code.

Appendices:

7.1 Grammar specification:

For the Jafar project, we choose ANTLR 4 to generate our grammar.

Detailed Grammar:

```
grammar JAFAR;

/** JAFAR program. */
program
: PROGRAM ID SEMI body EOF // program basic; int x; { something }
;

/** Body of a program. */
body
: (sharedDecl|decl)* func* block // int x,a; share bool y; func add(int g,h):void{return;} { }
;

/** Variable declaration block. */
decl: (var SEMI)+ #varDecl //int x,y; bool check;
;

sharedDecl: SHARED type ID (COMMA ID)* SEMI //shared int x,y; shared bool[1][2] check;
;

/** Function declaration. */
func: FUNC ID LPAR (params)? RPAR COLON type BEGIN (decl)* (stat)* RETURN (expr)? SEMI END #funcDecl
// func add(int x,y):int {return (x+y);}
; // fun justPrint():void {print(x); return;}

params: var (SEMI var)* #paramsDecl
;

/** Variable declaration. */
var : type ID (COMMA ID)* //int x,y
;

/** Grouped sequence of statements. */
block
: BEGIN (decl)* (stat)+ END //{int a,x,b; a:=b; x:= x+1;}
;

threadBlock
: block
;

/** Statement. */
stat: target ASS expr SEMI #assStat // x:= 1;
| IF expr COLON block (ELSE block)? #ifStat // if (a == b): {thenpart} else {elsepart}
| WHILE expr COLON block #whileStat // while (a == b): { }
| block #blockStat //{ @block content }
| PARBEGIN COLON (threadBlock)+ PAREND SEMI #threadStat // parbegin: { @thread1block }
{ @thread2block } parend;
| LOCK LPAR RPAR SEMI #lockStat // lock();
| UNLOCK LPAR RPAR SEMI #unlockStat // unlock();
| PRINT LPAR expr RPAR SEMI #outStat // print(x);
```

```

| ID LPAR (expr (COMMA expr)*)? RPAR SEMI #funcStat // funcname(x,1,True);
;

/** Target of an assignment. */
target
: ID      #idTarget // a,x
| arrayID (LSQ expr RSQ)+ #arrayTarget // a[0][1] := x;
;

/** Expression. */
expr: prfOp expr      #prfExpr // (-1) (!x)
| expr multOp expr    #multExpr // a/b, a*b, a mod b
| expr plusOp expr    #plusExpr // a+b, a-b
| expr compOp expr    #compExpr // a<=b, a<b, a==b, a>b, a>=b, [1,2,3] == x[i]
| expr boolOp expr    #boolExpr // a and b, a or b
| LPAR expr RPAR      #parExpr // (1+2)
| ID                  #idExpr
| NUM                  #numExpr
| TRUE                 #trueExpr
| FALSE                #falseExpr
| LSQ (expr (COMMA expr)*)? RSQ #arrayExpr // [1,2,3] // [[1,2],[2,3]]
| arrayID (LSQ expr RSQ)+ #indexExpr // a[1][2][3][4] given int [5][5][5][5]
| ID LPAR (expr (COMMA expr)*)? RPAR #funcExpr // sum(12,x);
;

arrayID: ID;

/** Prefix operator. */
prfOp: MINUS | NOT;

/** Multiplicative operator. */
multOp: STAR | SLASH | MOD;

/** Additive operator. */
plusOp: PLUS | MINUS;

/** Boolean operator. */
boolOp: AND | OR;

/** Comparison operator. */
compOp: LE | LT | GE | GT | EQ | NE;

/** Data type. */
type: INTEGER (LSQ NUM RSQ)+ #integerArrayType // int [4][2]
| BOOLEAN (LSQ NUM RSQ)+ #booleanArrayType // bool [2][3]
| INTEGER #intType
| BOOLEAN #boolType
| VOID #voidType
;
// Keywords

AND: A N D;
BEGIN: L B R A C E;
BOOLEAN: B O O L;
INTEGER: I N T;
VOID: V O I D;

```

```

SHARED: S H A R E D;
ELSE:  E L S E ;
END:   R B R A C E;
EXIT:  E X I T ;
FALSE: F A L S E ;
FUNC:  F U N C;
RETURN: R E T U R N;
IF:    I F ;
INPUT: I N P U T;
NOT:   E X P L N M A R K;
OR:    O R ;
PRINT: P R I N T ;
PROC:  P R O C E D U R E ;
PROGRAM: P R O G R A M ;
TRUE:  T R U E ;
WHILE: W H I L E ;
THREAD: T H R E A D ;
PARBEGIN: P A R B E G I N;
PAREND:  P A R E N D;
LOCK: L O C K;
UNLOCK: U N L O C K;
MOD: M O D;
ASS:  ':=';
COLON: ':';
COMMA: ',';
DQUOTE: '"';
EQ:  '==';
GE:  '>=';
GT:  '>';
LE:  '<=';
LBRACE: '{';
LPAR: '(';
LT:  '<';
MINUS: '-';
NE:  '!=';
PLUS: '+';
RBRACE: '}';
RPAR: ')';
SEMI: ';';
SLASH: '/';
STAR: '*';
LSQ: '[';
RSQ: ']';
EXPLNMARK: '!';

// Content-bearing token types
ID: LETTER (LETTER | DIGIT)*;
NUM: DIGIT (DIGIT)*;
STR: DQUOTE .*? DQUOTE;

fragment LETTER: [a-zA-Z];
fragment DIGIT: [0-9];

// Skipped token types
COMMENT: (SLASH SLASH .*? '\n') -> skip;
WS: [ \t\r\n]+ -> skip;

```

```
fragment A: [aA];
fragment B: [bB];
fragment C: [cC];
fragment D: [dD];
fragment E: [eE];
fragment F: [fF];
fragment G: [gG];
fragment H: [hH];
fragment I: [iI];
fragment J: [jJ];
fragment K: [kK];
fragment L: [lL];
fragment M: [mM];
fragment N: [nN];
fragment O: [oO];
fragment P: [pP];
fragment Q: [qQ];
fragment R: [rR];
fragment S: [sS];
fragment T: [tT];
fragment U: [uU];
fragment V: [vV];
fragment W: [wW];
fragment X: [xX];
fragment Y: [yY];
fragment Z: [zZ];
```

7.2 *Extended test:*

7.2.1 Jafar program:

Here, we cover most of Jafar programming language features, with exceptionals of Parbegin-parend threads and usage of Shared variables as we believe the correctness of those two features has already been reflected in the Concurrency test.

```
Program extendedTest;

int[3][3] a;
int result;

func isPrime(int x): bool {
  int i;
  bool stop;
  i := 2;
  stop := false;
  while ((!stop) and (i*i < x)): {
    stop := (i * (x/i)) == x;
    i := i + 1;
  }
  return !stop;
}

func gcd(int x; int y): int {
  while x != y :
```

```

{
  if x > y: {
    x := x-y;
  } else {
    y := y-x;
  }
}
return x;
}

func sum(int x; int y): void {
  result := x + y;
  return;
}

{
  result := 0;
  a[0] := [5,6,7];
  {
    int i;
    a[0] := [2,3,5];
    a[1] := [5,6,7];
    print(a);
    i := 0;
    while (i < 3): {
      if (isPrime(a[0][i]) and (gcd(a[0][i], a[1][i]) == 1)): {
        // adding a[0][i] and a[1][i] to the result
        sum(result, a[0][i] * a[1][i]);
      }
      i := i + 1;
    }
    print(result);
  }
  // a[2] := [11,1,2001];
  a[2] := [132,12,321];
  print(a);
  print(a[0] == a[1]);
}

```

7.2.2 Generated Sprockell code:

```

import Sprockell

prog0 :: [Instruction]
prog0 = [
  Load (ImmValue 13) regF
  , Jump (Abs 242)
  , Nop
  , Load (ImmValue (-1)) regA
  , Compute Add regF regA regA
  , Push regA
  , Load (ImmValue 2) regA
  , Push regA
  , Pop regA
  , Pop regD

```

```

, Store regA (IndAddr regD)
, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Push regA
, Load (ImmValue 0) regA
, Push regA
, Pop regA
, Pop regD
, Store regA (IndAddr regD)
, Jump (Abs 81)
, Nop
, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Push regA
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Load (ImmValue (-2)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regB
, Pop regA
, Compute Equal regB reg0 regD
, Branch regD (Abs 839)
, Load (ImmValue 0) regC
, Jump (Abs 46)
, Nop
, Compute Sub regA regB regA
, Load (ImmValue 1) regD
, Compute Add regC regD regC
, Nop
, Compute GtE regA regB regD
, Branch regD (Abs 42)
, Push regC
, Pop regB
, Pop regA
, Compute Mul regA regB regA
, Push regA
, Load (ImmValue (-2)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regB
, Pop regA
, Compute Equal regA regB regA
, Push regA
, Pop regA
, Pop regD
, Store regA (IndAddr regD)
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Push regA
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA

```

```

, Load (ImmValue 1) regA
, Push regA
, Pop regB
, Pop regA
, Compute Add regA regB regA
, Push regA
, Pop regA
, Pop regD
, Store regA (IndAddr regD)
, Nop
, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regA
, Load (ImmValue 1) regB
, Compute Xor regB regA regA
, Push regA
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regB
, Pop regA
, Compute Mul regA regB regA
, Push regA
, Load (ImmValue (-2)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regB
, Pop regA
, Compute Lt regA regB regA
, Push regA
, Pop regB
, Pop regA
, Compute And regA regB regA
, Push regA
, Pop regA
, Load (ImmValue 1) regB
, Compute Equal regA regB regA
, Branch regA (Abs 20)
, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regA
, Load (ImmValue 1) regB
, Compute Xor regB regA regA
, Push regA
, Load (ImmValue (-4)) regC
, Compute Add regF regC regC
, Pop regA
, Store regA (IndAddr regC)
, Load (ImmValue (-3)) regB
, Compute Add regF regB regB
, Load (IndAddr regB) regB
, Jump (Ind regB)

```

```

, Nop
, Jump (Abs 191)
, Nop
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regB
, Pop regA
, Compute Gt regA regB regA
, Push regA
, Pop regA
, Compute Equal regA reg0 regA
, Branch regA (Abs 171)
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Push regA
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regB
, Pop regA
, Compute Sub regA regB regA
, Push regA
, Pop regA
, Pop regD
, Store regA (IndAddr regD)
, Jump (Abs 190)
, Nop
, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Push regA
, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regB
, Pop regA
, Compute Sub regA regB regA
, Push regA
, Pop regA
, Pop regD
, Store regA (IndAddr regD)
, Nop
, Nop
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA

```



```

, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regB
, Pop regA
, Compute NEq regA regB regA
, Push regA
, Pop regA
, Load (ImmValue 1) regB
, Compute Equal regA regB regA
, Branch regA (Abs 136)
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Load (ImmValue (-3)) regC
, Compute Add regF regC regC
, Pop regA
, Store regA (IndAddr regC)
, Load (ImmValue (-2)) regB
, Compute Add regF regB regB
, Load (IndAddr regB) regB
, Jump (Ind regB)
, Nop
, Load (ImmValue 10) regA
, Push regA
, Load (ImmValue (-1)) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Load (ImmValue 0) regA
, Compute Add regF regA regA
, Load (IndAddr regA) regA
, Push regA
, Pop regB
, Pop regA
, Compute Add regA regB regA
, Push regA
, Pop regA
, Pop regD
, Store regA (IndAddr regD)
, Load (ImmValue (-2)) regB
, Compute Add regF regB regB
, Load (IndAddr regB) regB
, Jump (Ind regB)
, Nop
, Load (ImmValue 10) regA
, Push regA
, Load (ImmValue 0) regA
, Push regA
, Pop regA
, Pop regD
, Store regA (IndAddr regD)
, Load (ImmValue 0) regE
, Push regE
, Load (ImmValue 0) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)

```

```

, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regE
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regE regE
, Push regE
, Load (ImmValue 1) regA
, Pop regE
, Compute Add regA regE regE
, Push regE
, Load (ImmValue 5) regA
, Push regA
, Load (ImmValue 6) regA
, Push regA
, Load (ImmValue 7) regA
, Push regA
, Load (ImmValue 3) regD
, Compute Add regD regSP regD
, Load (IndAddr regD) regD
, Pop regA
, Load (ImmValue 2) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regA
, Load (ImmValue 1) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regA
, Load (ImmValue 0) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regD
, Load (ImmValue 0) regE
, Push regE
, Load (ImmValue 0) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regE
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regE regE
, Push regE
, Load (ImmValue 1) regA
, Pop regE
, Compute Add regA regE regE
, Push regE
, Load (ImmValue 2) regA
, Push regA
, Load (ImmValue 3) regA
, Push regA
, Load (ImmValue 5) regA
, Push regA
, Load (ImmValue 3) regD
, Compute Add regD regSP regD
, Load (IndAddr regD) regD
, Pop regA

```

```

, Load (ImmValue 2) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regA
, Load (ImmValue 1) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regA
, Load (ImmValue 0) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regD
, Load (ImmValue 0) regE
, Push regE
, Load (ImmValue 1) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regE
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regE regE
, Push regE
, Load (ImmValue 1) regA
, Pop regE
, Compute Add regA regE regE
, Push regE
, Load (ImmValue 5) regA
, Push regA
, Load (ImmValue 6) regA
, Push regA
, Load (ImmValue 7) regA
, Push regA
, Load (ImmValue 3) regD
, Compute Add regD regSP regD
, Load (IndAddr regD) regD
, Pop regA
, Load (ImmValue 2) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regA
, Load (ImmValue 1) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regA
, Load (ImmValue 0) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regD
, Load (ImmValue 1) regA
, Push regA
, Pop regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO

```

```

, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (ImmValue 13) regA
, Push regA
, Load (ImmValue 0) regA
, Push regA
, Pop regA
, Pop regD
, Store regA (IndAddr regD)
, Jump (Abs 677)
, Nop
, Load (ImmValue 0) regD
, Push regD
, Load (ImmValue 0) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Load (DirAddr 13) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)

```

```

, Pop regD
, Load (ImmValue 1) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Pop regD
, Load (ImmValue 1) regA
, Compute Add regA regD regD
, Load (IndAddr regD) regD
, Push regD
, Load (ImmValue 3) regB
, Compute Add regF regB regB
, Pop regA
, Store regA (IndAddr regB)
, Load (ImmValue 5) regB
, Compute Add regF regB regF
, Load (ImmValue 465) regA
, Load (ImmValue (-3)) regB
, Compute Add regF regB regB
, Store regA (IndAddr regB)
, Jump (Abs 2)
, Nop
, Load (ImmValue (-4)) regB
, Compute Add regF regB regB
, Load (IndAddr regB) regA
, Push regA
, Load (ImmValue (-5)) regB
, Compute Add regF regB regF
, Load (ImmValue 0) regD
, Push regD
, Load (ImmValue 0) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Load (DirAddr 13) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 1) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Pop regD
, Load (ImmValue 1) regA
, Compute Add regA regD regD
, Load (IndAddr regD) regD
, Push regD
, Load (ImmValue 3) regB

```

```

, Compute Add regF regB regB
, Pop regA
, Store regA (IndAddr regB)
, Load (ImmValue 0) regD
, Push regD
, Load (ImmValue 1) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Load (DirAddr 13) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 1) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Pop regD
, Load (ImmValue 1) regA
, Compute Add regA regD regD
, Load (IndAddr regD) regD
, Push regD
, Load (ImmValue 4) regB
, Compute Add regF regB regB
, Pop regA
, Store regA (IndAddr regB)
, Load (ImmValue 4) regB
, Compute Add regF regB regF
, Load (ImmValue 553) regA
, Load (ImmValue (-2)) regB
, Compute Add regF regB regB
, Store regA (IndAddr regB)
, Jump (Abs 134)
, Nop
, Load (ImmValue (-3)) regB
, Compute Add regF regB regB
, Load (IndAddr regB) regA
, Push regA
, Load (ImmValue (-4)) regB
, Compute Add regF regB regF
, Load (ImmValue 1) regA
, Push regA
, Pop regB
, Pop regA
, Compute Equal regA regB regA
, Push regA
, Pop regB
, Pop regA

```

```

, Compute And regA regB regA
, Push regA
, Pop regA
, Compute Equal regA reg0 regA
, Branch regA (Abs 663)
, Load (DirAddr 10) regA
, Push regA
, Load (ImmValue 2) regB
, Compute Add regF regB regB
, Pop regA
, Store regA (IndAddr regB)
, Load (ImmValue 0) regD
, Push regD
, Load (ImmValue 0) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Load (DirAddr 13) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 1) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Pop regD
, Load (ImmValue 1) regA
, Compute Add regA regD regD
, Load (IndAddr regD) regD
, Push regD
, Load (ImmValue 0) regD
, Push regD
, Load (ImmValue 1) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Load (DirAddr 13) regA
, Push regA
, Pop regA

```

```

, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 1) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Pop regD
, Load (ImmValue 1) regA
, Compute Add regA regD regD
, Load (IndAddr regD) regD
, Push regD
, Pop regB
, Pop regA
, Compute Mul regA regB regA
, Push regA
, Load (ImmValue 3) regB
, Compute Add regF regB regB
, Pop regA
, Store regA (IndAddr regB)
, Load (ImmValue 3) regB
, Compute Add regF regB regF
, Load (ImmValue 660) regA
, Load (ImmValue (-2)) regB
, Compute Add regF regB regB
, Store regA (IndAddr regB)
, Jump (Abs 220)
, Nop
, Load (ImmValue (-3)) regB
, Compute Add regF regB regF
, Nop
, Load (ImmValue 13) regA
, Push regA
, Load (DirAddr 13) regA
, Push regA
, Load (ImmValue 1) regA
, Push regA
, Pop regB
, Pop regA
, Compute Add regA regB regA
, Push regA
, Pop regA
, Pop regD
, Store regA (IndAddr regD)
, Nop
, Load (DirAddr 13) regA
, Push regA
, Load (ImmValue 3) regA
, Push regA
, Pop regB
, Pop regA
, Compute Lt regA regB regA
, Push regA
, Pop regA
, Load (ImmValue 1) regB
, Compute Equal regA regB regA
, Branch regA (Abs 420)
, Load (DirAddr 10) regA
, Push regA

```



```

, Pop regA
, WriteInstr regA numberIO
, Load (ImmValue 0) regE
, Push regE
, Load (ImmValue 2) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regE
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regE regE
, Push regE
, Load (ImmValue 1) regA
, Pop regE
, Compute Add regA regE regE
, Push regE
, Load (ImmValue 132) regA
, Push regA
, Load (ImmValue 12) regA
, Push regA
, Load (ImmValue 321) regA
, Push regA
, Load (ImmValue 3) regD
, Compute Add regD regSP regD
, Load (IndAddr regD) regD
, Pop regA
, Load (ImmValue 2) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regA
, Load (ImmValue 1) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regA
, Load (ImmValue 0) regB
, Compute Add regB regD regB
, Store regA (IndAddr regB)
, Pop regD
, Load (ImmValue 1) regA
, Push regA
, Pop regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA

```

```

, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (IndAddr regA) regB
, WriteInstr regB numberIO
, Load (ImmValue 1) regB
, Compute Add regA regB regA
, Load (ImmValue 0) regD
, Push regD
, Load (ImmValue 0) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Pop regD
, Load (ImmValue 1) regA
, Compute Add regA regD regD
, Push regD
, Load (ImmValue 0) regD
, Push regD
, Load (ImmValue 1) regA
, Push regA
, Pop regA
, Load (ImmValue 2) regB
, Compute Gt regA regB regB
, Branch regB (Abs 839)
, Compute Lt regA reg0 regB
, Branch regB (Abs 839)
, Pop regD
, Load (ImmValue 3) regB
, Compute Mul regB regA regA
, Compute Add regA regD regD
, Push regD
, Pop regD
, Load (ImmValue 1) regA
, Compute Add regA regD regD
, Push regD
, Pop regB
, Pop regA
, Load (ImmValue 1) regC
, Load (IndAddr regA) regD

```

```

, Load (IndAddr regB) regE
, Compute Equal regD regE regD
, Compute And regC regD regC
, Load (ImmValue 1) regD
, Compute Add regA regD regA
, Compute Add regB regD regB
, Load (IndAddr regA) regD
, Load (IndAddr regB) regE
, Compute Equal regD regE regD
, Compute And regC regD regC
, Load (ImmValue 1) regD
, Compute Add regA regD regA
, Compute Add regB regD regB
, Load (IndAddr regA) regD
, Load (IndAddr regB) regE
, Compute Equal regD regE regD
, Compute And regC regD regC
, Load (ImmValue 1) regD
, Compute Add regA regD regA
, Compute Add regB regD regB
, Push regC
, Pop regA
, WriteInstr regA numberIO
, EndProg
]

```

```
main = run [prog0]
```

7.2.3 Run results and explanation:

```

— print two dimensional array a in innerscope.
— a[0][0] -> a[0][2], a[0] = [2,3,5] declared in innerscope
Sprockell 0 says 2
Sprockell 0 says 3
Sprockell 0 says 5
— a[1][0] -> a[1][2], a[1] = [5,6,7] declared in innerscope
Sprockell 0 says 5
Sprockell 0 says 6
Sprockell 0 says 7
— a[2][0] -> a[2][2], a[2] is not declared thus default value [0,0,0]
Sprockell 0 says 0
Sprockell 0 says 0
Sprockell 0 says 0
— print out result = 45, since result = result + a[0][i] * a[1][i] with i from 0 to — 3 and if a[0][i] is prime and gcd(a[0][i], a[1][i])
== 1. Therefore, result = 0 + — 2*5 + 5*7 = 45.
Sprockell 0 says 45
— print two dimensional array a in main scope.
— a[0][0] -> a[0][2], a[0] = [2,3,5] declared in innerscope
Sprockell 0 says 2
Sprockell 0 says 3
Sprockell 0 says 5
— a[1][0] -> a[1][2], a[1] = [5,6,7] declared in innerscope
Sprockell 0 says 5
Sprockell 0 says 6
Sprockell 0 says 7
— a[2][0] -> a[2][2], a[2] = [132, 12, 321] declared in outer scope.
— Note that there is another a[2] = [11, 1, 2001] in outer scope but it actually — has been commented thus taking no effect.
Sprockell 0 says 132

```

Sprockell 0 says 12

Sprockell 0 says 321

– print `a[0] == a[1]`. Since `a[0] = [2,3,5]` and `a[1] = [5,6,7]`, the Jafar program will — print out 0.

Sprockell 0 says 0