Project Documentation- Programming Paradigms

University of Twente

Takáts Bálint - s2724448
Tim Koree - s2341182

Contents

Introduction

Tim Koree

MISSION

To start, we both found that this project was a great opportunity to learn something new. We want to find out what we really like inside a programming language, starting from the way simple expressions are structured all the way to memory models. Our goal with this project is to not only learn as much as we can about structuring our own language, but to also find a way to think of some concepts we would love to see in other languages as well.

EXPECTATIONS

The language we are thinking of putting together would be a very simple version of popular languages such as Java and Python but implementing some of our own twists and ideas. To come up with the syntax, we mainly looked at already established languages such as Rust, C, Java and Python. The main goal being to take the elements of the language that we like and structure our \`perfect\` language. Of course, we will have to cut a lot of features we would've liked in the language, such as objects, classes and additional types. With this project we merely want to build a simple first version and base, that could potentially later be expanded upon if we see the potential in this new language.

Summary

Takáts Bálint

SHARED AND LOCAL VARIABLES

Our language makes use of `Shared` and `Local` variables. Local variables can be accessed by the thread that creates them, as it is stored in the local memory within a thread. Shared variables on the other hand are stored in the shared memory, thus can be accessed by all threads. This introduces data races and race conditions. To offer our language means of synchronization, we implemented a simple locking system.

LOCKS

The language implements simple locks that allow for simple synchronization. It uses a pre-defined lock, that can be acquired and released by threads.

ARRAYS

In our language the programmer can define Integer and Boolean arrays. They can also access the array using the famous notation from the C language: “array[index]”, including setting only one index of the array to a new value.

WHILE AND IF

The language allows for the creation of while loops and if statements, following the example of languages such as Java and C.

IO

Our language only allows for limited output using the “printf(expr);” statement.

THREADING

Our language makes use of Threaded blocks. A master thread creates NUM additional threads to run a block of code, created by typing a “@Threaded(NUM) { }” statement.

EXCEPTION HANDLING

Within the code generation, two custom exceptions have been defined and are caught when triggered. A MemoryOutOfBoundsException and a TooManyThreadsException.

Problems and solutions

Takáts Bálint

LANGUAGE DEFINITION

During the language definition our progress went smooth. We had a clear idea of how we wanted to define our language and started out defining our syntax within the first few days. It wasn't until later that we had to change some of our definitions to create a more realistic scope of our project.

ELABORATION

The elaboration phase went smooth and didn't spur too many problems. Due to our experience using ANTLR and practice with the lab exercises, we had enough experience and were able to tackle the tasks quite efficiently.

CODE GENERATION

We had a few problems in this phase. They were somewhat related to some ambiguous documentation within the project description, but also to us underestimating the difficulty of some of the parts. To start, we wrongly assumed that the size of integers are 4 bytes long and we implemented our memory model and language based on that assumption. We later had to change our implementation and managed to do so in under 30 minutes. Additionally, we had issues getting the Sprockell processor to run. It "Failed to load interface for \`Network.Socket\`", taking some time and help from a TA to get it to work. Another issue that we encountered was implementing negation. In retrospect, this was quite dumb because after we had implemented a lot of the other features, the negation seemed like such an easy feature to implement only requiring us to add a single line to the generated code. Lastly, the code generation for the Threading ended up being a way greater task than we had initially anticipated. Coming up with a solution and creating it proved to take up a lot of time but ended up working in the end. All in all, we did not encounter an astronomical number of difficult challenges since we worked on all the lab exercises, but there were still things that took more time than we had initially anticipated.

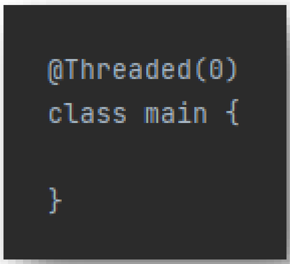
Detailed language description

Tim Koree

SYNTAX, USAGE AND SEMANTICS

CLASS DEC

The basic setup of running a program within our language starts with a class definition. To run the program, you create a class definition as indicated on figure 2.1, which in the current implementation is restricted to only using the name "main". The "@Threaded(NUM)" line indicates with how many additional threads the class is run. To run a program with only the main thread, NUM must be set to 0. For every additional thread that is used by means of a Threaded block (explained in chapter 5.1.2), NUM needs to be incremented once.

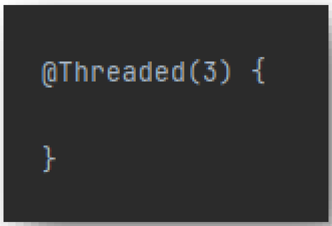


```
@Threaded(0)
class main {

}
```

Figure 5.1

THREADED BLOCK

Threading is implemented using "@Threaded(NUM) { }" blocks. The NUM represents the number of newly created threads. In the execution this means that a master thread, calls NUM threads to run the block of code within the threaded block. The master thread keeps track of all the other threads and waits until all the threads have finished their block. This means that the end of the Threaded block behaves like a join/barrier that makes sure that all the threads that are called by the master thread have returned. Nested threading is also possible. To do this you simply define a Threaded block within the already defined block. The thread that is running the outer block now assumes the role of master thread and does the same thing as the main thread does otherwise. It calls NUM new threads and makes sure they've all returned before continuing. As mentioned earlier, the top of the class also contains a @Threaded declaration. It is the programmer's responsibility to not exceed the number of threads defined at the top of the class.

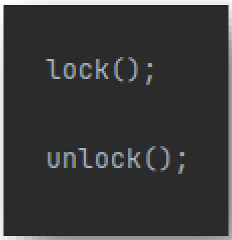


```
@Threaded(3) {

}
```

Figure 5.2

LOCK

The language uses a simple lock implementation to allow for synchronization. The programmer can make a running thread acquire a lock using the "lock();" statement and can have the thread release the lock using the "unlock();" statement. This gives the programmer some basic means of preventing concurrency issues. To continue, the lock is not re-entrant, if the same thread tries to lock again, the program will deadlock. When multiple threads try to acquire a lock only one thread succeeds, making the other threads loop until they can acquire the lock.



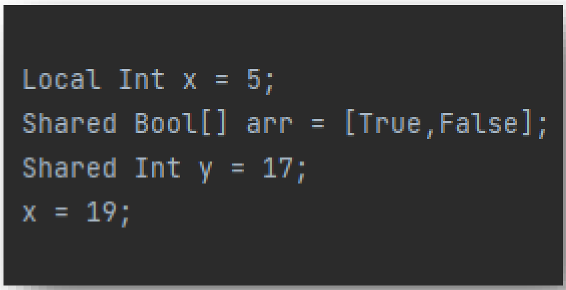
```
lock();

unlock();
```

Figure 5.3

VARDEC AND COPYOVER

The variable declaration is quite simple. You declare a variable by stating the memory location, type, variable name and value. This is also the case for arrays. The only limitation is that the amount of shared or local variables shouldn't be more than the memory allows. Copying over follows a somewhat similar format. It copies over a value, with the restriction that it has the same type. Furthermore, all types and memory specification start with a capital letter. All currently supported types include:

-	Int	-> Integer
-	Int[]	-> Integer array
-	Bool	-> Boolean
-	Bool[]	-> Boolean array

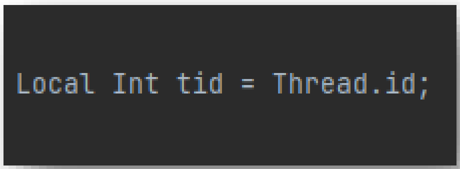


```
Local Int x = 5;
Shared Bool[] arr = [True,False];
Shared Int y = 17;
x = 19;
```

Figure 5.4

THREADID

It is also possible to get the tid of the current thread. For example, with as purpose to divide the execution within a threaded block. A restriction is that Thread.id can only be written to an Int, since it is an integer.

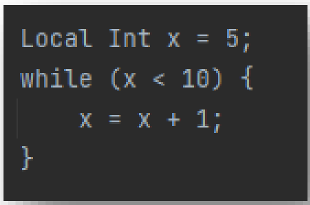


```
Local Int tid = Thread.id;
```

Figure 5.5

WHILE LOOP

The syntax allows for the creation of `while loops`. While loops can be defined using a "while(expr){}" statement, creating a block and new scope for variables. Its behaviour is identical to famous languages such as Java and C, this means that `expr` is required to be an expression that can be traced back to a Boolean type.

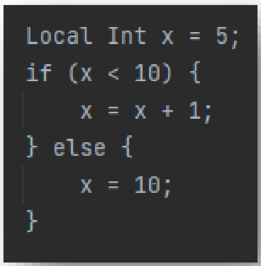


```
Local Int x = 5;
while (x < 10) {
    x = x + 1;
}
```

Figure 5.6

IF STATEMENT

The `if statements` in our language is very similar to our while loops, using a "if (expr) { }" statement, following the same consistencies.

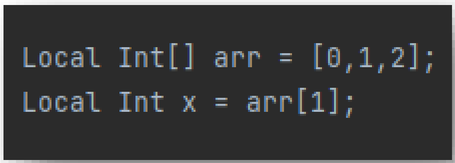


```
Local Int x = 5;
if (x < 10) {
    x = x + 1;
} else {
    x = 10;
}
```

Figure 5.7

GETINDEX

This statement is very similar to `copyOver`. It essentially uses copyOver to access an index of an array and copy it over to a variable. Restrictions include that the index must be within the size of the array, the index must be an integer and that the indexed variable must be an array.

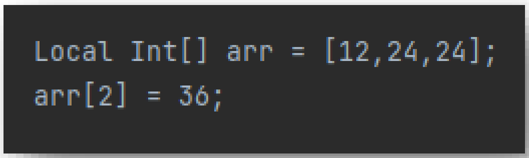


```
Local Int[] arr = [0,1,2];
Local Int x = arr[1];
```

Figure 5.8

SETINDEX

This statement is again similar to the `copyOver` and `getIndex`, only now writing into the array. Restriction again that the index must be within the size of the array and the type of the value that is being put into the array must be of the same type.

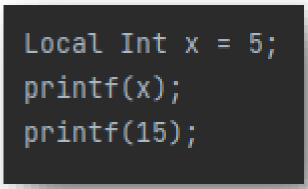


```
Local Int[] arr = [12,24,24];
arr[2] = 36;
```

Figure 5.9

PRINTF

The print statement allows for the printing of integers using the "printf(expr);" function. The contents of expr can however only be an integer. So, this can either be an actual integer or a variable containing the integer.



```
Local Int x = 5;
printf(x);
printf(15);
```

Figure 5.10

CODE GENERATION

CLASS DEC

The generated code in `visitClassDec` generates code that is later used by the Locks and Threaded blocks. It essentially reserves spots in shared memory for the Lock and the defined Threads. This means that if 3 helper threads are defined, the first 4 spots in shared memory will be reserved. The memory is reserved by first loading the index of the reserved spot into regA, like: "Load (ImmValue sharedMemSpot) regA". Followed by writing regA to the shared memory of that index using the WriteInstr command. It then continues by setting up a loop. It creates lines that only let the main thread through. The helper threads will loop until they are called by the main thread. This is done by checking whether regSprID equals reg0 which results in any helper threads jumping back when the condition they are different. In this loop, the helper threads also check whether they have been assigned a task. A task is given by a master thread, who calls a helper thread to run a block of code by setting the helper threads reserved memory spot to the location is needs to jump to. The `visitClassDec` continues by getting all the other generated code and ends with the main thread waiting until all threads have finished their task. When this is done, it tells the helper threads to end their program, followed by the main thread also ending.

THREADED BLOCK

Typing out a Threaded block results in a master thread calling helper threads as mentioned in chapter 5.1.2. To do this, the master thread first gets the program counter and sends it to the shared memory spot reserved for a free helper thread. This means, the helper thread jumps to the same location as the master thread. The master thread then jumps over the code inside the threaded block, it does this by using the `Branch` instruction and checking whether regSprID equals the master-threads ID. The helper thread runs the block of code, followed by setting its preserved register back to its base value and jumping back to the loop that is defined by `visitClassDec`. Lastly, the master thread waits on all the threads within the block it oversees, and only continues when all threads have finished running their piece of code.

NOTEXPR
The negation uses a very simple implementation. Is just gets the value of the Boolean and puts it in regA. Then simply uses a “Compute regA reg0 regA” line to flip it to the other.

PUTLOCK
To put the lock, a running thread gets the address of the predefined lock and is only able to lock if the contents of the address are 0. It uses a TestAndSet to get the contents of the lock address and attempts to set it to 1. If the address already contains a 1, a branch condition makes it jump back, essentially making it loop until the write is successful.

PUTUNLOCK
The code here is quite simple, it simply sets the contents of the lock address back to 0, using a load and WriteInstr.

THREAD ID
This part essentially just writes regSprID to a different register which is then read from in the next lines to be used in a statement.

GETINDEX
The address of the array is stored in our offsets HashMap, having a different address for every spot in the array. What it will do is first load the original address, add it to the index given, and look it up in memory. So, it starts with a load with the address, then a Compute to add it to the index, and then a ReadInstr or a Load depending on where in memory the array is stored.

SETINDEX
This is quite like the set index. It gets the location in memory by using a load with the address, calculates the correct address by adding the offset using the Compute and finally uses a WriteInstr or a Store to write the new value in memory.

IF STATEMENT
The base structure of the if statement is quite simple. It essentially checks the condition and jumps over the “if-body” if it is false. If it’s true, it’ll run the if and jump over the else body if it is there. To do this, we first check whether the Boolean is a True or False. We use a `Compute` to check whether it’s equal to reg0, if true we make the jump, if false we jump over the if body. After the conditional branch, the if body is inserted followed by a jump over the else body. Since the else is already the end of the if statement, we don’t need a jump there and can continue with the rest.

WHILE LOOP
The while loop is somewhat like the if-statement. It uses a conditional branch and jumps over the while-body if the negation of the condition is a 1. If the condition is false, it runs the body and jumps back to the condition to check again.

VARDEC
Variable declaration is quite simple on the generated code part. Using out Scope class it figures out the offset of where the variable can be stored in Shared or Local memory. It then saves the offset and maps it to the ID of the variable. Finally, the generated code consists of either a Store, or a WriteInstr to put it in save it in its correct memory location.

COPYOVER
Copy over is quite like the variable declaration. The main difference is that it doesn't add new variables to the Scope class, but simply replaces looks up the address and overwrites what is written in memory, again using Store or WriteInstr depending on where the variable is stored in memory.

IDEXPR
First gets the address belonging to the ID using the Scope class, then follows by writing the variable stored in the Shared or Local memory to regA.

ARRCONTENTS
Stores the actual contents of an array into memory. This is done by giving every index in the array its own memory location starting with the first element and using offsets of 1 for every index after. This means they are also stored in different addresses within the HashMap. Ultimately, it results in the function generating a Store or WriteInstr for every index depending on the memory location. This means it's very similar to VarDec, but instead of only storing a single variable it stores multiple variables into multiple addresses.

ADD-, MULT-, OR- AND COMPEXPR
These rules are structed almost identically. Some of them have multiple operators, if this is the case, the operator is checked, and the correct instruction is derived. From there on they are the same. They first get the entire left-hand-side and push the contents onto the stack, saving them into a different register to make sure both expr's don't save to the same register. Then, all the generated code from the right-hand-side is added and stored, followed by the actual compute using the correct operator. So, a "+" will use an Add, "<" uses a Lt etc., all ending in saving the output into a register.

CONSTEXPR
It checks for the constant and stores a number into a register to represent a Boolean or Integer.

PRINTF
First thing it does is that it gets the contents of the expr. Since the final value can only be an integer, it is stored in regA and immediately printed using "WriteInstr regA numberIO".

Description of the software

Takáts Bálint

ANTLR

GRAMMAR.G4

Contains the definition of our language. Additional files supporting the language definition are automatically generated using ANTLR4.

PARSER

PARSE

This is an empty class, implementing the generated GrammarBaseListener. It is used to test the parsing of our grammar.

PARSERTEST

A class using the parse class to parse a number of files located in the `src/Parser/Tests/*` folder.

SRC/PARSER/TESTS

Folder that contains a number of text files used for testing the syntax using the generated parser.

ELABORATION

SYMBOLTABLECLASS

This is a symbol table implementation. We had previously implemented it for the lab exercises, we just had to modify it a little bit to better suit our needs.

TYPECHECKTEST

This file contains a list of tests whether our type checking class performs as expected and filters out bad code and adds the correct error messages.

TYPECHECK

This class extends GrammarBaseListener. It is responsible for type checking the parsed grammar. In it, exit and enter rules are defined. If it encounters an error while visiting the parse tree it adds an error to the errorList and then continues the walk. The error contains information about the line number and position of the error plus information about the type of the error. For example type mismatch or scope errors. The class contains a private tree, which is a ParseTreeProperty, responsible for saving the type of expressions.

PAIR
This file contains an implementation of Pair as it is needed by SymbolTableClass. We had to resort to defining one locally (using StackOverFlow for reference) because the standard Pair implementations didn't seem to work for the team.

TYPE
This contains an enum class with all the possible types of our language.

GENERATION

SCOPE
This file contains an implementation of the scope. We use this to store all the variables in the code generation phase. It is implemented using a HashMap and it contains the offsets of the variables in each scope. Moreover, it is also responsible for storing data about whether the variable is a local or a global one.

ASSEMBLER
This file contains a wrapper for our code generator class. It is responsible for compiling the code for Sprockell. It tokenizes, parses the input and then using Generator, defined in Generator.java, it translates our language to the target language. As the last step, it adds a few lines of Haskell syntax to make it easier to run and then saves it as specified.

GENERATOR
This file is the heath of the compiler. It is responsible for the Sprockell code generation. We implemented it using BaseVisitors. We choose this so that we can return the code and visit the children in the order we need want to. In this file there are also two ParseTreeProperties defined. One is responsible for saving the last used register by an expression and the other one is responsible for keeping track of the scope. They are named regs and scope respectively.

MEMORYOUTOFBOUNDSEXCEPTION
This file contains an exception that is thrown when the compiler tries to allocate more memory than what is available on the Sprockell processor.

TOOMANYTHREADSEXCEPTION
This file contains an exception that is thrown when the compiler tries to allocate more threads than what is declared at the top of the file.

SAMPLE

SRC/SAMPLE/COMPLETEPROGRAM
Folder containing the source file specified in `Appendix C – Extended Test Program`.

SRC/SAMPLE/REQUIRED
Folder containing the required implenentations using our language. These include the Peterson's algorithm and the elementary banking system.

SRC/SAMPLE/TESTS
Folder containing a number of tests used by both the contextual testing and the semantic testing.

SPROCKELL-MASTER
Folder containing the Sprockell code. Is kept in to allow for immediate testing of the generated haskell files.

SRC/SPROCKELL-MASTER/SRC
Contains 3 important folder:

- **CompleteProgram**, contains the generated code belonging to the source file specified in `Appendix C – Extended Test Program`.
- **Required**, contains the generated code belonging to the elementary banking system and Peterson's algorithm.
- **Tests**, contains all the tests that are generated for the semantic error testing.

Test plan and results

Tim Koree

GENERAL

During our progression within the project, we tried to be as thorough with testing as possible. The main idea was to hand-test newly implemented code and use automated testing to verify whether our previously written code still performed as it should. To do this we made use of JUnit test classes to easily test our implementations.

SYNTAX ERRORS

To check for syntax errors, ANTLR automatically parses the input and matches it up to the grammar. If a part of the input can't be parsed using the grammar, it will return an error message showing that the syntax is faulty. We tested the parsing by creating a `ParserTest` class. It uses JUnit tests to read from multiple files and tries to parse them. We created several correct and incorrect programs that can be run. If the parse fails ANTLR automatically writes to the terminal, meaning we can systematically test whether our grammar still parses the correct files and filters out the incorrect ones.

CONTEXTUAL ERRORS

The type-checking is tested using a `ParseTreeWalker`, generated by ANTLR. We created functions that check if the types are correct when a certain rule is visited within the tree. This way every node is checked, passing a custom error message to an error list if the types, scopes or other contextual errors are found. To test the type checking, the `TypeCheckTest` class contains several JUnit tests that check files within our `src/Sample/Tests` folder. It creates an instance of our walker and of our `TypeChecker`. The method "goodCode()" runs through the tests that have an expected amount of 0 errors, whilst the "wrongCode()" has any number above 0. Every individual test takes a Boolean, which must be set to true to display the error messages that the test outputs (if any). It also takes an integer representing the number of expected errors within the test, making it easy to see how many errors it should return. The advantage of using JUnit is that we can re-run all the examples very easy and measure it up against the expected number of errors. This means that if we change something in our grammar, we can easily check whether all the implemented type-checking features still work correctly.

SEMANTIC ERRORS

The Assembler class contains 3 tests all with their own purpose. The first test "compile()" takes regular text files from the `src/Sample/Tests/*` folder, containing code in our programming language and generates properly converted code into a ready-to-run Haskell file located at `src/sprockell-master/src/Tests/*`. It uses a list of different programs, including the two mandatory files at the top. All these files use print statements that the Haskell file outputs to confirm whether the behaviour is as intended. This does mean that testing requires added files within the Sample/Tests folder, adding them to the Assembler class, running the compile test, followed by running the individual Haskell files. Every delivered file should output as indicated in the table at `Appendix B – Semantic Errors Table`. The two other tests are checking whether the defined exceptions are still caught when they should be. They start with getting the contents of a file, containing code in our language and attempting to convert them to Haskell files. Both tests should fail as the code is faulty and print a `MemoryOutOfBoundsException` and `TooManyThreadsException` into the terminal respectively.

Conclusions

Tim Koree

GENERAL

Now that we are roughly finished with our project, we can with all certainty say that we did not expect this level of attention to detail within a language. The way the code generation requires an almost assembly-like approach, really made us question how bad things would've gotten if we hadn't put the work into making the ILOC exercises. It also made us realize that programming languages are structured insanely thorough, following a very structured set of steps to build one block on top of another. This project truly combined much of the knowledge we had been taught and felt like a very satisfying application of our skills.

WHAT WE LEARNED

In retrospect, this project ended up being a great challenge. We managed to see how much we have learned about programming languages and how our syntax preferences translate to our own language. We learned a lot about the time and effort that goes into making a programming language and learned how much freedom it gives you as a programmer. The concept of being able to build whatever you desire and to tackle problems in a completely new way is a very exciting and interesting idea.

FINAL CONCLUSIONS

Even though our language feels very incomplete and rough, for the scope of this project that can't be helped. This does however give us a way greater appreciation for the work that goes into implementing self-defined languages. Seeing that our tiny language already took us over 2 weeks to construct, really speaks about the scale of popular languages.

Appendix A – Grammar Specification

```
grammar Grammar;

program : def EOF                                #beginDec
        ;

def     : thread CLASS 'main' stat               #classDec
        ;

thread  : THREADED '(' NUM ')'                   #threadedDec
        ;

stat    : '{' stat* '}'                          #blockStat
        | mem type ID '=' expr ';'               #varDec
        | IF '(' expr ')' stat (ELSE stat)?      #ifStatement
        | WHILE '(' expr ')' stat                #whileLoop
        | THREADED '(' NUM ')' tstat             #threadedBlock
        | 'lock' '(' ')' ';'                     #putLock
        | 'unlock' '(' ')' ';'                   #putUnlock
        | ID '=' expr ';'                        #copyOver
        | ID '[' expr ']' '=' expr ';'           #setIndex
        | OUT '(' expr ')' ';'                   #output
        ;

tstat   : '{' stat* '}'                          #threadedBlockStat
        ;

expr:   '!' expr                                 #notExpr
        | expr (PLUS | MINUS) expr               #addExpr
        | expr MULT expr                         #multExpr
        | expr '&&' expr                         #andExpr
        | expr '||'  expr                        #orExpr
        | expr (GT | LT | EQ | NEQ) expr         #compExpr
        | '(' expr ')'                           #parExpr
        | (NUM | TRUE | FALSE)                   #constExpr
        | '[' arr ']'                            #arrayExpr
        | 'Thread.id'                            #getThreadId
        | ID '[' expr ']'                        #getIndex
        | ID                                     #idExpr
        ;

arr     : expr (',' expr)*                       #arrContents
        |                                        #emptyArr
        ;

type    : 'Int[]'                                #intArray
        | 'Bool[]'                               #boolArray
        | 'Int'                                  #int
        | 'Bool'                                 #bool
        ;

mem     : 'Local'                                #isLocal
        | 'Shared'                               #isShared
        ;

CLASS: 'class';
```

```
OUT : 'printf';
IF: 'if';
ELSE: 'else';
WHILE: 'while';
TRUE: 'True';
FALSE: 'False';
THREADED: '@Threaded';
LOCK: 'Lock';
PLUS: '+';
MINUS: '-';
MULT: '*';
GT: '>';
LT: '<';
EQ: '==';
NEQ: '!=';


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

NUM: DIGIT+;
ID: LETTER (LETTER | DIGIT)*;
WS: [ \t\r\n]+ -> skip;
```

Appendix B – Semantic Errors Table

File	Expected output	Note
bankingsystem.hs	Sprockell 1 says 32 Sprockell 5 says 5 Sprockell 2 says 37 Sprockell 3 says 69 Sprockell 6 says 42 Sprockell 4 says 15 Sprockell 0 says 15	This is a threaded program simulating a simple banking system. Thus, the order can differ. Only requirement is that the bank must stay above or equal to 0. (It prints every intermediate amount on the bank after a transaction has been made).
petersons.hs	Sprockell 1 says 127 Sprockell 1 says 82 Sprockell 2 says 109 Sprockell 2 says 64 Sprockell 0 says 64	This algorithm works as a lock, it allows 2 threads to run the critical section without interference.
negation.hs	Sprockell 0 says 200	Checks whether simple negation of a Boolean works.
arrays.hs	Sprockell 0 says 999	Checks whether simple creation and reading of an array works.
if.hs	Sprockell 0 says 0 Sprockell 0 says 2 Sprockell 0 says 0 Sprockell 0 says 2	Tests several if-statements and prints the values belonging to the values in the conditions.
ifScope.hs	Sprockell 0 says 3 Sprockell 0 says 5 Sprockell 0 says 12	Checks whether scoping is implemented correctly for if-statements.
locks.hs	Sprockell 1 says 41 Sprockell 2 says 41 Sprockell 0 says 41	Locks on a shared resource whilst multiple threads try to read/write, testing whether there are no data races.
checkSorted.hs	Sprockell 0 says 1 Sprockell 0 says 0	Checks whether an array is sorted.
threaded.hs	Sprockell 1 says 1 Sprockell 2 says 2 Sprockell 3 says 3 Sprockell 4 says 4 Sprockell 5 says 5 Sprockell 6 says 6 Sprockell 1 says 0 Sprockell 2 says 0 Sprockell 3 says 3 Sprockell 4 says 4 Sprockell 1 says 1 Sprockell 3 says 3	Prints the thread id’s of called threads within their respective (sometimes nested) block.

while.hs	Sprockell 0 says 1 Sprockell 0 says 2 Sprockell 0 says 3 Sprockell 0 says 4	Prints the iteration of the while loop.
whileScope.hs	Sprockell 0 says 1 Sprockell 0 says 1 Sprockell 0 says 1 Sprockell 0 says 1 Sprockell 0 says 1 Sprockell 0 says 20	Checks whether scoping is implemented correctly for while-loops.
mult.hs	Sprockell 0 says 40	Calculates and prints the multiplication of two integers.
and.hs	Sprockell 0 says 1 Sprockell 0 says 2	Checks whether the AND/OR operators are implemented correctly.
eqArrays.hs	Sprockell 0 says 1 Sprockell 0 says 4	Tests whether equality checks for arrays is implemented correctly.
setIndex.hs	Sprockell 0 says 45 Sprockell 0 says 200	Tests whether setting an index of an array to a new value is possible.
sortArr.hs	Sprockell 0 says 1 Sprockell 0 says 2 Sprockell 0 says 2 Sprockell 0 says 2 Sprockell 0 says 3 Sprockell 0 says 4 Sprockell 0 says 4 Sprockell 0 says 5 Sprockell 0 says 6 Sprockell 0 says 6 Sprockell 0 says 7 Sprockell 0 says 23 Sprockell 0 says 34 Sprockell 0 says 56 Sprockell 0 says 87 Sprockell 0 says 654	Sorts and prints an array.
infLoop.hs	-	Expected to cause an infinite loop.
isPrime.hs	Sprockell 0 says 1	Checks whether variable x is a prime.

Appendix C – Extended Test Program

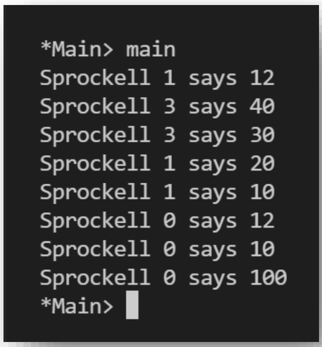
Tim Koree

CODE

```
@Threaded(4)
class main {
    Shared Int[] arr = [12,53,17];

    @Threaded(2) {

        if (Thread.id == 1) {
            lock();
            printf(arr[0]);
            unlock();
        } else {
            lock();
            arr[1] = 50;
            arr[2] = 100;
            unlock();
        }

        @Threaded(2) {
            lock();
            while ((Thread.id * arr[0]) < arr[1] ) {
                arr[1] = arr[1] - 10;
                printf(arr[1]);
            }
            unlock();
        }
    }

    Local Int it = 0;
    while (it < 3) {
        printf(arr[it]);
        it = it + 1;
    }
}
```

EXPECTED OUTPUT

The expected output is as in figure 8.1. Do take note that the program is threaded, and the order of the prints thus can differ.



```
*Main> main
Sprockell 1 says 12
Sprockell 3 says 40
Sprockell 3 says 30
Sprockell 1 says 20
Sprockell 1 says 10
Sprockell 0 says 12
Sprockell 0 says 10
Sprockell 0 says 100
*Main>
```

Figure 8.1

GENERATED SPROCKELL CODE

```
import Sprockell
prog :: [Instruction]
prog = [
    Load (ImmValue 0) regA,
    WriteInstr regA (DirAddr 0),
    Load (ImmValue 1) regA,
    WriteInstr regA (DirAddr 1),
    Load (ImmValue 2) regA,
    WriteInstr regA (DirAddr 2),
    Load (ImmValue 3) regA,
    WriteInstr regA (DirAddr 3),
    Load (ImmValue 4) regA,
    WriteInstr regA (DirAddr 4),
    ReadInstr (IndAddr regSprID),
    Receive regA,
    Compute Equal regSprID reg0 regB,
    Branch regB (Rel 4),
    Compute NEq regA regSprID regB,
    Branch regB (Ind regA),
    Jump (Rel (-6)),
    Load (ImmValue 12) regA,
    WriteInstr regA (DirAddr 5),
    Load (ImmValue 53) regA,
    WriteInstr regA (DirAddr 6),
    Load (ImmValue 17) regA,
    WriteInstr regA (DirAddr 7),
    ReadInstr (DirAddr 1),
    Receive regA,
    Load (ImmValue 1) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-4)),
    WriteInstr regPC (DirAddr 1),
    Load (ImmValue 0) regA,
    Compute Equal regSprID regA regB,
    Branch regB (Rel 165),
    Compute Add regSprID reg0 regA,
    Push regA,
    Load (ImmValue 1) regA,
    Pop regB,
    Compute Equal regB regA regA,
    Compute Equal regA reg0 regA,
    Branch regA (Rel 15),
    TestAndSet (DirAddr 0),
    Receive regD,
    Load (ImmValue 1) regC,
    Compute NEq regD regC regD,
    Branch regD (Rel (-4)),
    Load (ImmValue 0) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    WriteInstr regA numberIO,
    Load (ImmValue 0) regA,
    WriteInstr regA (DirAddr 0),
    Jump (Rel 18),
    TestAndSet (DirAddr 0),
    Receive regD,
    Load (ImmValue 1) regC,
    Compute NEq regD regC regD,
    Branch regD (Rel (-4)),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regE,
    Compute Add regA regE regE,
    Load (ImmValue 50) regA,
    WriteInstr regA (IndAddr regE),
    Load (ImmValue 2) regA,
    Load (ImmValue 5) regE,
    Compute Add regA regE regE,
    Load (ImmValue 100) regA,
    WriteInstr regA (IndAddr regE),
    Load (ImmValue 0) regA,
    WriteInstr regA (DirAddr 0),
    ReadInstr (DirAddr 3),
    Receive regA,
    Load (ImmValue 3) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-4)),
    WriteInstr regPC (DirAddr 3),
    Load (ImmValue 1) regA,
    Compute Equal regSprID regA regB,
    Branch regB (Rel 49),
    TestAndSet (DirAddr 0),
    Receive regD,
    Load (ImmValue 1) regC,
    Compute NEq regD regC regD,
    Branch regD (Rel (-4)),
    Compute Add regSprID reg0 regA,
    Push regA,
    Load (ImmValue 0) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Pop regB,
    Compute Mul regB regA regA,
    Push regA,
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Pop regB,
    Compute Lt regB regA regA,
    Compute Equal regA reg0 regA,
    Branch regA (Rel 21),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regE,
    Compute Add regA regE regE,
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Push regA,
    Load (ImmValue 10) regA,
    Pop regB,
    Compute Sub regB regA regA,
    WriteInstr regA (IndAddr regE),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    WriteInstr regA numberIO,
    Jump (Rel  (-38)),
    Load (ImmValue 0) regA,
    WriteInstr regA (DirAddr 0),
    WriteInstr regSprID (DirAddr 3),
    Jump (Abs 10),
    ReadInstr (DirAddr 4),
    Receive regA,
    Load (ImmValue 4) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-4)),
    WriteInstr regPC (DirAddr 4),
    Load (ImmValue 1) regA,
    Compute Equal regSprID regA regB,
    Branch regB (Rel 49),
    TestAndSet (DirAddr 0),
    Receive regD,
    Load (ImmValue 1) regC,
    Compute NEq regD regC regD,
    Branch regD (Rel (-4)),
    Compute Add regSprID reg0 regA,
    Push regA,
    Load (ImmValue 0) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Pop regB,
    Compute Mul regB regA regA,
    Push regA,
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Pop regB,
    Compute Lt regB regA regA,
    Compute Equal regA reg0 regA,
    Branch regA (Rel 21),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regE,
    Compute Add regA regE regE,
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Push regA,
    Load (ImmValue 10) regA,
    Pop regB,
    Compute Sub regB regA regA,
    WriteInstr regA (IndAddr regE),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    WriteInstr regA numberIO,
    Jump (Rel  (-38)),
    Load (ImmValue 0) regA,
    WriteInstr regA (DirAddr 0),
    WriteInstr regSprID (DirAddr 4),
    Jump (Abs 10),
    ReadInstr (DirAddr 3),
    Receive regA,
    Load (ImmValue 3) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-4)),
    ReadInstr (DirAddr 4),
    Receive regA,
    Load (ImmValue 4) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-9)),
    WriteInstr regSprID (DirAddr 1),
    Jump (Abs 10),
    ReadInstr (DirAddr 2),
    Receive regA,
    Load (ImmValue 2) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-4)),
    WriteInstr regPC (DirAddr 2),
    Load (ImmValue 0) regA,
    Compute Equal regSprID regA regB,
    Branch regB (Rel 165),
    Compute Add regSprID reg0 regA,
    Push regA,
    Load (ImmValue 1) regA,
    Pop regB,
    Compute Equal regB regA regA,
    Compute Equal regA reg0 regA,
    Branch regA (Rel 15),
    TestAndSet (DirAddr 0),
    Receive regD,
    Load (ImmValue 1) regC,
    Compute NEq regD regC regD,
    Branch regD (Rel (-4)),
    Load (ImmValue 0) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    WriteInstr regA numberIO,
    Load (ImmValue 0) regA,
    WriteInstr regA (DirAddr 0),
    Jump (Rel 18),
    TestAndSet (DirAddr 0),
    Receive regD,
    Load (ImmValue 1) regC,
    Compute NEq regD regC regD,
    Branch regD (Rel (-4)),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regE,
    Compute Add regA regE regE,
    Load (ImmValue 50) regA,
    WriteInstr regA (IndAddr regE),
    Load (ImmValue 2) regA,
    Load (ImmValue 5) regE,
    Compute Add regA regE regE,
    Load (ImmValue 100) regA,
    WriteInstr regA (IndAddr regE),
    Load (ImmValue 0) regA,
    WriteInstr regA (DirAddr 0),
    ReadInstr (DirAddr 1),
    Receive regA,
    Load (ImmValue 1) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-4)),
    WriteInstr regPC (DirAddr 1),
    Load (ImmValue 2) regA,
    Compute Equal regSprID regA regB,
    Branch regB (Rel 49),
    TestAndSet (DirAddr 0),
    Receive regD,
    Load (ImmValue 1) regC,
    Compute NEq regD regC regD,
    Branch regD (Rel (-4)),
    Compute Add regSprID reg0 regA,
    Push regA,
    Load (ImmValue 0) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Pop regB,
    Compute Mul regB regA regA,
    Push regA,
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Pop regB,
    Compute Lt regB regA regA,
    Compute Equal regA reg0 regA,
    Branch regA (Rel 21),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regE,
    Compute Add regA regE regE,
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Push regA,
    Load (ImmValue 10) regA,
    Pop regB,
    Compute Sub regB regA regA,
    WriteInstr regA (IndAddr regE),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    WriteInstr regA numberIO,
    Jump (Rel  (-38)),
    Load (ImmValue 0) regA,
    WriteInstr regA (DirAddr 0),
    WriteInstr regSprID (DirAddr 1),
    Jump (Abs 10),
    ReadInstr (DirAddr 3),
    Receive regA,
    Load (ImmValue 3) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-4)),
    WriteInstr regPC (DirAddr 3),
    Load (ImmValue 2) regA,
    Compute Equal regSprID regA regB,
    Branch regB (Rel 49),
    TestAndSet (DirAddr 0),
    Receive regD,
    Load (ImmValue 1) regC,
    Compute NEq regD regC regD,
    Branch regD (Rel (-4)),
    Compute Add regSprID reg0 regA,
    Push regA,
    Load (ImmValue 0) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Pop regB,
    Compute Mul regB regA regA,
    Push regA,
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Pop regB,
    Compute Lt regB regA regA,
    Compute Equal regA reg0 regA,
    Branch regA (Rel 21),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regE,
    Compute Add regA regE regE,
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    Push regA,
    Load (ImmValue 10) regA,
    Pop regB,
    Compute Sub regB regA regA,
    WriteInstr regA (IndAddr regE),
    Load (ImmValue 1) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    WriteInstr regA numberIO,
    Jump (Rel  (-38)),
    Load (ImmValue 0) regA,
    WriteInstr regA (DirAddr 0),
    WriteInstr regSprID (DirAddr 3),
    Jump (Abs 10),
    ReadInstr (DirAddr 1),
    Receive regA,
    Load (ImmValue 1) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-4)),
    ReadInstr (DirAddr 3),
    Receive regA,
    Load (ImmValue 3) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-9)),
    WriteInstr regSprID (DirAddr 2),
    Jump (Abs 10),
    ReadInstr (DirAddr 1),
    Receive regA,
    Load (ImmValue 1) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-4)),
    ReadInstr (DirAddr 2),
    Receive regA,
    Load (ImmValue 2) regC,
    Compute NEq regA regC regB,
    Branch regB (Rel (-9)),
    Load (ImmValue 0) regA,
    Store regA (DirAddr 0),
    Load (DirAddr 0) regA,
    Push regA,
    Load (ImmValue 3) regA,
    Pop regB,
    Compute Lt regB regA regA,
    Compute Equal regA reg0 regA,
    Branch regA (Rel 14),
    Load (DirAddr 0) regA,
    Load (ImmValue 5) regB,
    Compute Add regB regA regA,
    ReadInstr (IndAddr regA),
    Receive regA,
    WriteInstr regA numberIO,
    Load (DirAddr 0) regA,
    Push regA,
    Load (ImmValue 1) regA,
    Pop regB,
    Compute Add regB regA regA,
    Store regA (DirAddr 0),
    Jump (Rel  (-19)),
    Compute Add reg0 regPC regA,
    Compute Equal regSprID reg0 regB,
    Branch regB (Rel 2),
    EndProg,
    WriteInstr regA (DirAddr 1),
    WriteInstr regA (DirAddr 2),
    WriteInstr regA (DirAddr 3),
    WriteInstr regA (DirAddr 4),
    EndProg]
main = run [prog,prog,prog,prog,prog]
```