Programming Paradigms Integrating Project

Technical Computer Science Module 8A

Authors:
Iris Borgonjen s3192393
Théo Mougnibas s3581497
– Group 29

08-07-2025

Contents

Summary of language features

To begin with, “:D¡Queso!”, which is the name of our newly designed language, supports both single line and multi line comments, indicated by the standard ‘//’ and ‘/* … */’ respectively.
The language also supports the basic variable types: integers and booleans. It also supports the declaration of arrays of integers and arrays of booleans, but code generation for this feature is still a work in progress and therefore not working. Then, for integers it is possible to do simple expressions like addition, subtraction, multiplication and the different kinds of equalities and inequalities.
Variable declaration happens by giving a type and a variable name. The default value of a variable is 0. This means for integers that it is the number 0 and for booleans this means the value of False.
To make sure the language is strongly typed, we use a type declaration when declaring a variable. This type stays fixed throughout the scope. However, it is possible to declare a variable with the same name that might be of a different type on a deeper scope level.
For concurrency we decided to use the fork-join model on threads. For our implementation this means that we create new threads from a main thread. After this, the new threads can create other threads as well, giving the support to a nested thread structure. To create a thread, the user should give the scope of the thread directly and this will immediately start the thread. Joining the threads is implemented a little bit like a barrier. This means that all running threads can be joined by the main thread at some point.
With the given memory spaces, by design, there is space for a limited amount of threads, global variables and locks. This will be elaborated on later.

Problems encountered

We mainly encountered problems during the code generation phase as the scanning/parsing was already covered in haskell during lab exercises and type checking concepts were not very challenging to port from java lab exercises to an haskell version, as the most complex data structures used are a stack to take account of local variables in nested bodies and symbol tables to fill the stack.

- Team-mate dropping out

We had to reassign the work that we thought would have been made by our missing team-mate. It not only increased our workload but also might have made the design/problem solving process longer, as three brains instead of two may be sometimes helpful.

- Lack of sprockell documentation

- negating booleans

The SPRIL language is lacking a native "NOT" unary operator, so we had to build our own negating code block to use whenever we encounter a boolean negation. To do it, we simply used a basic if-else construct that is branching over the value of the boolean. If the boolean is 0, we don't branch and fall to the part that is swapping the value to a 1. Otherwise we branch to a block where we swap in-memory the value to a 0. We mainly chose this straightforward approach because it was logically simple and did not require us to think about it for long.

- Generating if/else

The logic in itself to implement an if/else block as SPRIL code is based on a branch instruction and by so, not very hard nor new. But the branch instruction behavior pushed us to use a little trick. As "branch condition jump_address" jumps whenever the condition register is non-zero and fall throught to next instruction if register contains 0, we actually are generating the body of the else "before" the body of the if in the instruction sequence. Giving us a non-complex but rather exotic pattern :

```
BRANCH condition if_body_code
– ELSE BODY CODE
JUMP after_if_body_code
– IF BODY CODE
```

But jumping over the body of the if necessitates to know its length. Thus, we generate all bodies and build the else jump instruction right before concatenating all instructions together.

The management of threads has been for us the hardest part to implement of the overall project, as we first needed to understand how threads are managed on sprockels, that is, not managed at all. And how to accommodate our code to that specificity where no dynamic forking is supported. To do so, we deeply studied the "DemoMultipleSprockells.hs" file, kindly provided on the sprockell repository. It introduced us to the technique used by the developers of sprockell to make use of non dynamic forking : a header that branches based on the sprockell id and lets only the 0 one put jumping addresses at the memory address corresponding to each used sprockell id. Thus, a second part of that header is reserved to all non-zero sprockells, where they wait for that value to be received (as all sprockells are starting to run the program at the same time) and then jumps to that specific address. We found that part quite interesting, as it avoids any nested if-else blocks to make threads jump to their respective body.

- Generating multithreading jumps/initialisation as program header

To generate that thread-dispatching header, we first needed to discover and store data about every thread existing in the given ":D¡Queso!" program. Most importantly, we needed two informations for each thread :

- a unique identifier for that thread that could be used as memory address
- the address at which the first instruction of the said thread body can be found, so the dispatcher can load it in memory, and recover it to make a specific sprockell jump to it.
- the actual thread's body

We decided to use a table that we fill with that information to help gather that information (the "tt" parameter in the code generator file refers to it).
The first one was just a matter of recursively going through all statements of the program and giving a number to each thread when we encounter a ThreadCreate instruction.

The second one led to a much more interesting problem, that is, generating thread bodies and correctly inferring at which address their first instruction will be.
As threads can be nested, we recursively look to generate potential new bodies, making the current address grow each time we generate a new body, so the next thread can know its start address.
Once this is done and the table is filled, we have the generated thread start addresses that are correct but they do not take account of the header that will be added to the top of the program, whose size depends on the number of threads. As at this point we have this information, we can accurately estimate the future dispatcher code size and add this offset to each thread start address before actually generating the dispatcher's code that uses these addresses.

- Joining threads

As described in the subject, we do not need to make specific thread joining instructions, so we implemented a join instruction that waits for all other threads to finish. We reserved address 0 to a global variable (so, in shared memory) that is incremented/decremented when we start a thread or a thread finishes. We used a lock to make it thread-safe. Thus, the code generated by a join instruction itself is busy waiting in a loop while that shared counter has a non-zero value. This implementation has the advantage that we can join after every thread creation in the main thread.
The drawbacks of this implementation is that it keeps the processor busy for no real computation work.

- Implementing locks

As suggested by the documentation, we based our lock implementation on the SPRIL "TestAndSet" **atomic** instruction that tests if the content at a memory address is zero and in that case sets it to one before sending back the value 1. If the memory address pointed to a non-zero value, this instruction will just send back 0. Knowing that, we implement a lock with this instruction and a simple integer in shared memory. This lets us create locks in code generation first pass, at the same time as classic global variables. Hence, getting a lock is just busy waiting for TestAndSet to return 1 for the lock's address as the instruction will

atomically claim “ownership” of the lock as soon as it is freed by setting it to one. Then, freeing the lock is just setting the memory at the lock’s address to 0.
This lock is very feature-stripped, and thus does not provide any fairness mechanism that would help handle starvation cases.

Detailed language description

1. Variable Declarations and Assignments

Syntax

- Variable Declarations:
  - "`entero x :)`" declares an integer variable named "x".
  - "`booleana yes :)`" declares a boolean variable named "yes".

- Assignments:
  - "`x = 5 :)`" assigns the integer value 5 to variable "x".
  - "`x = verdad :)`" assigns the boolean value "verdad" (true) to variable "x".

- Array assignments:
  - "`x = [1, 2, 3] :)`" assigns an integer array to variable "x".
  - "`x = [verdad, mentira, verdad] :)`" assigns a boolean array to variable "x".

Usage

- Variables must be declared before they are used.
- Types must match in assignments (e.g., you can't assign a boolean to an integer variable).
- Array assignments must have elements of the same type as declared.

Semantics

- Declarations allocate memory for variables.
- Assignments store values into the allocated memory locations.

Code Generation

- For variable declarations, initialize memory to 0.
- For assignments, generate code to evaluate the expression and store the result in the variable's memory address.

2. Array Declarations and Access

Syntax

- Array declarations: "`array entero a :)`" declares an integer array named "a".
- Array access: "`a[1]`" to access the element at index 1.

Usage

- Arrays must be declared with a type (e.g., integer or boolean).
- Array indices must be within bounds.

Semantics

- Arrays are stored in contiguous memory locations.
- Array access calculates the address of the element based on the base address and the index.

Code Generation

- Sadly, we did not manage to finish code generation for arrays in time.

3. Arithmetic and Logical Operations

Syntax

- Arithmetic operations: "`x = 1 + 5 :)`", "`x = 1 - 5 :)`", "`x = 1 * 5 :)`"
- Comparison operations: "`x == 5 :)`", "`x != 5 :)`", "`x > 5 :)`", "`x >= 5 :)`", "`x < 5 :)`", "`x <= 5 :)`"
- Logical operations: "`x Y y :)`" for AND, "`x O y :)`" for OR, "`~:( x :)`" for NOT.

Usage

- Operands must be of compatible types (e.g., integers for arithmetic operations, booleans for logical operations).
- Results must be assigned to variables.

Semantics

- Arithmetic operations perform standard integer arithmetic.
- Comparison operations return boolean values (1 for true, 0 for false).
- Logical operations perform standard boolean algebra.

Code Generation

- Generate code to evaluate the operands, perform the operation, and store the result on top of the stack.

4. Control Structures

Syntax

- If-else statements:
```
 si verdad { imprimir ¡5! :) } sino { imprimir ¡10! :) }
```

- While loops:
```
  durante verdad { x = 3 :) }
```

Usage

- Conditions must evaluate to boolean values.
- The bodies of if-else and while statements are enclosed in braces “{}” or in "iniciamos"-->"cerramos".

Semantics

- If the condition is true, execute the true branch; otherwise, execute the false branch (if present).
- While loops execute the body repeatedly as long as the condition is true.

Code Generation

- For if-else, generate conditional jumps to skip the false branch if the condition is true, or to skip the true branch if the condition is false.
- For while loops, generate code to evaluate the condition, and jump to the start of the loop if the condition is true.

5. Input/Output

Syntax

- Print statements: "`imprimir ¡x! :)`"

Usage

- The expression inside "¡ !" is evaluated and printed.
- Only defined for booleans and integers.

Semantics

- The value of the expression is output to the console.

Code Generation

- Generate code to evaluate the expression and call the wrintelnstr instruction with numberIO target address.

6. Scoping and Blocks

Syntax

- blocks: "`{ entero x :) }`"
- Nested scope blocks: "`{ entero x :) { booleana a :) } }`"
- As for if/else and while constructs, braces can be replaced with "iniciamos"-->"cerramos".

Usage

- Variables declared within a block are local to that block.
- Nested blocks can access variables from outer blocks unless shadowed by a variable using the same name declared in the scope.

Semantics

- Variables are allocated and deallocated as blocks are entered and exited.
- Deallocation is thought as just “forgetting” the address of a given variable when we leave a scope.
- Inner blocks can shadow outer block variables.

Code Generation

- For each block, a stack storing memory addresses given to each variable is used to replace references by loading the content of the memory address to register A.

7. Threads

Syntax

- Thread creation: "`hilo { entero x :) x = 5 + 1 :) }`"
- Thread joining: "`esperamos :)`"

Usage

- Threads are used for concurrent execution.
- Threads can be created and joined to synchronize execution.

Semantics

- Thread creation makes the parent thread notify another Sprockell core to run the embodied code when it reaches this part, making multiple computations run in parallel. Before that, the core waiting for notification is just busy waiting.
- Thread joining waits for all running threads to complete.

Code Generation

- For thread creation, generate code to create a new thread and start its execution when the parent thread reaches the place it was declared.
- For thread joining, generate code to wait for all threads to complete.

8. Locks

Syntax

- Lock creation: `esclusa lock :)`
- Lock acquisition: `obtener lock :)`
- Lock release: `liberar lock :)`

Usage

- Locks are used for synchronization between threads.
- A thread must acquire a lock before entering a critical section and release it afterward.

Semantics

- Lock creation allocates a new lock.
- Lock acquisition blocks until the lock is available.
- Lock release makes the lock available for other threads.

Code Generation

- For lock creation, get a free memory for the lock. It does not generate any SPRILL instruction.
- For lock acquisition, generate code to wait until the lock is available (i.e value at the lock address is 0).
- For lock release, generate code to make the lock available.

A detailed description of the code generated is present in the “problems encountered” section.

9. Global Variables

Syntax

- Global variable declaration: `global entero x :)`

Usage

- Global variables are accessible from all threads and scopes.
- They must be declared with a type.

Semantics

- Global variables are stored in a shared memory space accessible by all threads.

Code Generation

- Allocate memory in a shared space for global variables.
- Generate code to read/write from/to these shared memory locations.

10. Comments

Syntax

- Multi-line comments: `/* hello \n */`
- Single-line comments: `// hello`

Usage

- Comments are ignored by the compiler and are used for documentation.

Semantics

- Comments are removed during parsing and do not affect the execution of the program.

Example Programs

1. Simple Assignment

```
entero x :)
x = 5 :)
imprimir ¡x! :)
```

This program declares an integer variable "x", assigns it the value 5, and prints it.

2. If-Else Statement

```
booleana condition :)
condition = verdad :)
si condition { imprimir ¡1! :) } sino { imprimir ¡0! :) }
```

This program declares a boolean variable "condition", sets it to true ("verdad"), and prints 1 if the condition is true, otherwise prints 0.

3. While Loop

```
entero x :)
x = 0 :)
durante x < 5 { x = x + 1 :) }
imprimir ¡x! :)
```

This program declares an integer variable "x", initializes it to 0, increments it by 1 in a loop until it reaches 5, and then prints the final value of "x".

4. Thread Creation

```
entero x :)
hilo { entero y :) y = 10 :) imprimir ¡y! :) }
```

```
esperamos :)
```

This program declares an integer variable "x", creates a new thread that declares a local variable "y", assigns it the value 10, prints it, and then waits for all threads to complete.

5. Lock Usage

```
esclusa lock :)
obtener lock :)
imprimir ¡1! :)
liberar lock :)
```

This program creates a lock, acquires it, prints 1, and then releases the lock.

6. Global Variable

```
global entero sharedVar :)
sharedVar = 20 :)
imprimir ¡sharedVar! :)
```

This program declares a global integer variable "sharedVar", assigns it the value 20, and prints it.

7. Error Handling

- Type Errors: The compiler checks that types match in assignments and operations.
- Undeclared Variables: The compiler checks that all variables are declared before use.
- Division by Zero: We did not implement division operation in our language.

More code generation details are present in the “problems encountered” section.

Description of the software

MyParser.hs

- Symbol Tables

In "MyParser.hs", symbol tables are used primarily for type checking. The symbol table is a list of tuples where each tuple consists of a variable name and its type. This helps in keeping track of variables and their types within different scopes.

- SymbolTable : Defined as "type SymbolTable = [(String, Type)]", it maps variable names to their types.
- STStack : Defined as "type STStack = [SymbolTable]", it is a stack of symbol tables used to handle nested scopes. Each new scope pushes a new symbol table onto the stack, and exiting a scope pops the symbol table off the stack.

- Grammar Rules and Parsing

The grammar rules are defined using the Parsec library, which allows for the creation of parsers in a declarative manner. The file defines parsers for different language constructs : types, expressions, and statements.

- Type Parsing: The "typeIdentifier" parser handles the parsing of type identifiers like "entero", "booleana", and "array".
- Expression Parsing: Various parsers like "primaryExpr", "unaryExpr", "multExpr", "addExpr", etc., handle different levels of expressions, from primary expressions to complex binary operations.
- Statement Parsing: Parsers like "declaration", "assignment", "ifStatement", "whileStatement", etc., handle different types of statements in the language.

- Inherited and Synthesized Attributes

In the context of parsing and type checking, inherited and synthesized attributes are used implicitly:

- Synthesized Attributes : The type of an expression is synthesized from the types of its sub-expressions.
- Inherited Attributes : The symbol table is an inherited attribute that is passed down to child nodes ttype checking. The type of an expression in a condition (if/while) can be thought as Inherited as we want the childs to be of type boolean, even if we actually infer it from synthesis to check that it is, in fact, a boolean.

MyCodeGen.hs

- Symbol Tables

In "MyCodeGen.hs", symbol tables are used for code generation and memory management. The symbol tables here are slightly different from those in "MyParser.hs" as they map variable names to memory addresses.

- GlobalSymbolTable : Defined as "type GlobalSymbolTable = [(String, MemAddr)]", it maps global variable names to their memory addresses.
- LocalVarStack : Defined as "type LocalVarStack = [GlobalSymbolTable]", it is a stack of symbol tables used to handle local variables in nested scopes.

● Code Generation

The code generation process involves generating Sprockell instructions from the parsed AST. This process is divided into two passes:

- First Pass : Fills the global symbol table with locks and global variables. It allocates memory addresses for global variables and locks.
- Second Pass : Generates the actual code, handling threads, locks, and other constructs. It uses the symbol tables to look up memory addresses for variables/locks and generates the appropriate instructions.

Spec.hs

● Testing Framework

"Spec.hs" uses the Hspec library to define tests for the parser, type checker, and code generator. It ensures that each component functions correctly and interacts properly with the others.

- Parser Tests: Tests for parsing various constructs like variable declarations, assignments, conditionals, loops, and threads.
- Type Checker Tests: Tests for valid and invalid type assignments, ensuring that the type checker correctly identifies type errors.
- Code Generation Tests: Tests for generating correct Sprockell code for various language constructs, including basic print statements, if-else statements, and thread creation.

Test plan and results

Test Plan

1. Parser Tests:
    - Objective: Ensure that the parser correctly parses valid constructs of the language and rejects invalid ones.
    - Scope: Includes tests for variable declarations, assignments, conditionals, loops, threads, and other language constructs.
    - Examples:
        - Parsing valid integer and boolean declarations.
        - Rejecting invalid declarations and assignments.
        - Parsing valid and invalid array declarations and accesses.
        - Parsing valid and invalid if-else statements, while loops, and thread creations.
2. Type Checker Tests:
    - Objective: Ensure that the type checker correctly identifies type errors and accepts valid type assignments.
    - Scope: Includes tests for valid and invalid type assignments, ensuring that the type checker correctly identifies type mismatches and undeclared variables.
    - Examples:
        - Accepting valid assignments and conditionals.
        - Rejecting invalid assignments and non-boolean conditions.
        - Ensuring that undeclared variables and non-homogeneous arrays are correctly identified as type errors.
3. Code Generation Tests:
    - Objective: Ensure that the code generator produces correct Sprockell code for various language constructs.
    - Scope: Includes tests for generating code for basic print statements, if-else statements, thread creation, and other constructs.
    - Examples:
        - Generating code for basic print programs.
        - Generating code for if-else statements and while loops.
        - Generating code for thread creation and synchronization using locks.
4. Runtime Tests :
    - Objective : Ensure that test programs are behaving as expected.
    - Scope : Simple programs including loops, prints, if/else and calculations. Also test a runtime error on hanging loops.
    - Examples :
        - Running a while loop that counts to 5.
        - Running a program that is computing the number of days in the month of february of a given year.

Test Results

The test results are documented in the Spec.hs file, where each test case is defined using the Hspec framework. The tests are designed to be self-documenting, with descriptive names and comments explaining the expected behavior.

- Parser Tests Results: The parser tests ensure that the parser correctly handles valid and invalid constructs. For example:

```
it "parses a valid integer declaration" $ do
    let input = "entero x :)"
      parseMyLang input "shouldBe" Right [Declaration Entero "x"]
```

This test checks that the parser correctly parses a valid integer declaration.

- Type Checker Tests Results: The type checker tests ensure that the type checker correctly identifies type errors. For example:

```
it "rejects an invalid assignment" $ do
    let stmts = [
            Declaration Entero "x",
            Assignment "x" (BoolLit True)  -- type error
            ]
    checkTypeInvalid stmts
```

This test checks that the type checker correctly identifies an invalid assignment where a boolean value is assigned to an integer variable.

- Code Generation Tests Results: The code generation tests ensure that the code generator produces correct Sprockell code. For example:

```
it "basic print program" $ do
    let prog = "imprimir¡5!:)"
    let ast = case parseMyLang prog of
            (Left _) -> error "Parse error"
            (Right tree) -> tree
      codeGen ast "shouldBe" [...(instructions)]
```

This test checks that the code generator produces the correct Sprockell instructions for a basic print program.

- Runtime Tests results : As we cannot capture output of the Sprockell processors, this catecorgy of programs have to be checked by the user. They still are run automatically, but the output has to be manually observed. For example :

```
  it "while loop that counts to 5" $ do
        let program = "entero x :) x = 0 :) durante x < 5 { x = x + 1 :) }
imprimir ¡x!:)"
        case parseMyLang program of
            Left err -> expectationFailure (show err)
            Right ast -> do
                let prog = codeGen ast
                putStrLn "Generated code:"
                print prog
                run [prog]
                pendingWith "Manual verification needed: should print 5"
```

All the tests are automated and can be run using the "stack test" command.

Conclusion

Concerning the language that we defined, we are very happy with the ideas that we came up with. We really liked the process of coming up with fun aspects of the language, such as certain symbols and keywords. In addition, it is very cool to be able to define a new language and be able to write a program in the language that does what it is supposed to do. About the module as a whole, we feel like it was very educational. We learned a lot about the insides of a programming language. This was different from subjects that we had seen before. Besides, we learned a lot about Haskell, a language that we knew very little of before starting the module. We feel that we made a lot of progress during the module and it taught us a lot of useful programming knowledge. However, sometimes it was also a bit hard. There were a lot of deadlines and other things that we needed to finish. There was quite some pressure to work very hard and complete assignments on time. This has caused a little bit of stress at times. In addition, we experienced some difficulties within the group at the end of the module, but it was impossible to find a new member. We see that also as a downside of working together with the same group for almost the whole module. However, a positive thing about working together with the same group for a long time is that we could get to know each other quite well and really experience how it is to work with each other.

Appendix A: Contributions

Language features

Part	Member
Comments	Theo Mougnibas, Iris Borgonjen
Basic types	Theo Mougnibas, Iris Borgonjen
Arrays	Theo Mougnibas
Integer expressions	Iris Borgonjen
Boolean expressions	Theo Mougnibas, Iris Borgonjen
Variable declaration	Theo Mougnibas
Scopes	Iris Borgonjen
Variable assignment	Theo Mougnibas
If statement	Iris Borgonjen
While statement	Iris Borgonjen
Print statement	Theo Mougnibas
Fork join construct threads	Theo Mougnibas, Iris Borgonjen
Locks	Theo Mougnibas

Report

Part	Member
Summary of language features	Iris Borgonjen
Problems encountered	Theo Mougnibas
Detailed language description	Theo Mougnibas, Iris Borgonjen
Description of the software	Theo Mougnibas, Iris Borgonjen
Test plan and results	Theo Mougnibas
Conclusion	Iris Borgonjen

Appendix B: Grammar specification

```haskell
lexer :: Tok.TokenParser ()
lexer = Tok.makeTokenParser haskellStyle
  { Tok.commentLine = "//"
  , Tok.commentStart = "/*"
  , Tok.commentEnd = "*/"
  , Tok.identStart = letter <|> char '_'
  , Tok.identLetter = alphaNum <|> char '_'
  , Tok.reservedNames = ["entero", "booleana", "array", "si",
"durante", "imprimir", "hilo", "empezamos", "esperamos", "global",
"esclusa", "liberar", "obtener", "iniciamos", "cerramos", "verdad",
"mentira"]
  , Tok.reservedOpNames = ["+", "-", "*", "==", "!=", "<", "<=", ">",
">=", "Y", "O", "~:(", "=", ":)", ",", "[", "]", "¡", "!", "{", "}"]
  }

-- helper parsers
reserved :: String -> Parser ()
reserved = Tok.reserved lexer

reservedOp :: String -> Parser ()
reservedOp = Tok.reservedOp lexer

identifier :: Parser String
identifier = Tok.identifier lexer

integer :: Parser Integer
integer = Tok.integer lexer

symbol :: String -> Parser String
symbol = Tok.symbol lexer

whiteSpace :: Parser ()
whiteSpace = Tok.whiteSpace lexer

parens :: Parser a -> Parser a
parens = Tok.parens lexer

braces :: Parser a -> Parser a
braces = Tok.braces lexer

brackets :: Parser a -> Parser a
brackets = Tok.brackets lexer

commaSep :: Parser a -> Parser [a]
commaSep = Tok.commaSep lexer

-- EDSL definition
data Type = Entero | Booleana | Array Type| Lock | Global Type
|CompilationError String
          deriving (Show, Eq)

data Op = Mul
    | Add
    | Sub
    | Inv
    | Not
    | Or
    | And
    | Lt
    | Leq
    | Gt
    | Geq
    | Eq
    | Neq
        deriving (Show,Eq)

data Expr =
            ArrayAccess String Expr
          | IntLit Integer
          | BoolLit Bool
          | Var String
          | ArrayLit [Expr]
          | BinOp Op Expr Expr
          | UnOp Op Expr
          | Paren Expr
          deriving (Show,Eq)

data Stmt = Declaration Type String
          | Assignment String Expr
          | If Expr [Stmt] [Stmt] -- The if contains the else body
          | While Expr [Stmt]
          | Print Expr
          | ThreadCreate [Stmt]
          | ThreadJoin
          | StartThread String
          | LockCreate String
          | LockFree String
          | LockGet String
          | ScopeBlock [Stmt]
          deriving (Show,Eq)
```

Appendix C: Extended test program

The following test includes (almost) all features that exist in the language.

```
global entero x :)
esclusa lock :) // lock for the global variable
x = 1 :)

booleana r :) // initialized to False
booleana s :)
s = verdad :)

/*
    initialize some (nested) threads
    use the lock to prevent race conditions
*/

hilo {
    imprimir¡x! :)
    hilo {
        obtener lock :)
        x = x + 1 :)
        imprimir¡x! :)
        liberar lock :)
    }
}

hilo {
    obtener lock :)
    x = x * 1 :)
    imprimir ¡x! :)
    liberar lock :)
}

esperamos :)

si ~:( r {
    imprimir ¡s! :)
    imprimir ¡s O r! :)
    imprimir ¡s Y r! :)
}

durante s iniciamos
    si ¡x <= 0! {
        s = mentira :)
    } sino {
        x = x - 1 :)
    }
```

```
cerramos

imprimir ¡x! :)
```

This results in the following SprIL code:

```
Branch 1 (Rel 2),
Jump (Rel 8),
Load (ImmValue 4) 2,
Compute Sprockell.Add 2 1 4,
ReadInstr (IndAddr 4),
Receive 6,
Compute Equal 6 0 7,
Branch 7 (Rel (-4)),
Jump (Ind 6),
Load (ImmValue 1) 2,
Push 2,
Pop 2,
WriteInstr 2 (DirAddr 3),
Load (ImmValue 0) 2,
Store 2 (DirAddr 2),
Load (ImmValue 0) 2,
Store 2 (DirAddr 1),
Load (ImmValue 1) 2,
Push 2,
Pop 2,
Store 2 (DirAddr 1),
TestAndSet (DirAddr 1),
Receive 2,
Branch 2 (Rel 2),
Jump (Rel (-3)),
ReadInstr (DirAddr 0),
Receive 2,
Compute Incr 2 2 2,
WriteInstr 2 (DirAddr 0),
WriteInstr 0 (DirAddr 1),
Load (ImmValue 33) 4,
WriteInstr 4 (DirAddr 5),
Jump (Rel 59),
ReadInstr (DirAddr 3),
Receive 2,
Push 2,
Pop 2,
WriteInstr 2 (DirAddr 65536),
TestAndSet (DirAddr 1),
Receive 2,
Branch 2 (Rel 2),
Jump (Rel (-3)),
ReadInstr (DirAddr 0),
Receive 2,
Compute Incr 2 2 2,
WriteInstr 2 (DirAddr 0),
WriteInstr 0 (DirAddr 1),
Load (ImmValue 50) 4,
WriteInstr 4 (DirAddr 6),
Jump (Rel 32),
TestAndSet (DirAddr 2),
Receive 2,
Branch 2 (Rel 2),
Jump (Rel (-3)),
ReadInstr (DirAddr 3),
Receive 2,
Push 2,
Load (ImmValue 1) 2,
Push 2,
Pop 3,
Pop 2,
Compute Sprockell.Add 2 3 4,
Push 4,
Pop 2,
WriteInstr 2 (DirAddr 3),
ReadInstr (DirAddr 3),
Receive 2,
Push 2,
Pop 2,
WriteInstr 2 (DirAddr 65536),
WriteInstr 0 (DirAddr 2),
TestAndSet (DirAddr 1),
Receive 2,
Branch 2 (Rel 2),
Jump (Rel (-3)),
ReadInstr (DirAddr 0),
Receive 2,
Compute Decr 2 2 2,
WriteInstr 2 (DirAddr 0),
WriteInstr 0 (DirAddr 1),
EndProg,
TestAndSet (DirAddr 1),
Receive 2,
Branch 2 (Rel 2),
Jump (Rel (-3)),
ReadInstr (DirAddr 0),
Receive 2,
Compute Decr 2 2 2,
WriteInstr 2 (DirAddr 0),
WriteInstr 0 (DirAddr 1),
EndProg,
TestAndSet (DirAddr 1),
Receive 2,
Branch 2 (Rel 2),
Jump (Rel (-3)),
ReadInstr (DirAddr 0),
Receive 2,
Compute Incr 2 2 2,
WriteInstr 2 (DirAddr 0),
WriteInstr 0 (DirAddr 1),
Load (ImmValue 103) 4,
WriteInstr 4 (DirAddr 7),
Jump (Rel 32),
TestAndSet (DirAddr 2),
Receive 2,
Branch 2 (Rel 2),
Jump (Rel (-3)),
ReadInstr (DirAddr 3),
Receive 2,
Push 2,
Load (ImmValue 1) 2,
Push 2,
Pop 3,
Pop 2,
Compute Sprockell.Mul 2 3 4,
Push 4,
Pop 2,
WriteInstr 2 (DirAddr 3),
ReadInstr (DirAddr 3),
Receive 2,
Push 2,
Pop 2,
WriteInstr 2 (DirAddr 65536),
WriteInstr 0 (DirAddr 2),
TestAndSet (DirAddr 1),
Receive 2,
Branch 2 (Rel 2),
Jump (Rel (-3)),
ReadInstr (DirAddr 0),
Receive 2,
Compute Decr 2 2 2,
WriteInstr 2 (DirAddr 0),
WriteInstr 0 (DirAddr 1),
EndProg,
ReadInstr (DirAddr 0),
Receive 2,
Branch 2 (Rel 2),
Jump (Rel 2),
Jump (Rel (-4)),
Load (DirAddr 2) 2,
Push 2,
Pop 2,
Branch 2 (Rel 4),
Load (ImmValue 1) 3,
Push 3,
Jump (Rel 2),
Push 0,
Nop,
Pop 2,
Branch 2 (Rel 2),
Jump (Rel 25),
Load (DirAddr 1) 2,
Push 2,
Pop 2,
WriteInstr 2 (DirAddr 65536),
Load (DirAddr 1) 2,
Push 2,
Load (DirAddr 2) 2,
Push 2,
Pop 3,
Pop 2,
Compute Sprockell.Or 2 3 4,
Push 4,
Pop 2,
WriteInstr 2 (DirAddr 65536),
Load (DirAddr 1) 2,
Push 2,
Load (DirAddr 2) 2,
Push 2,
Pop 3,
Pop 2,
Compute Sprockell.And 2 3 4,
Push 4,
Pop 2,
WriteInstr 2 (DirAddr 65536),
Nop,
Load (DirAddr 1) 2,
Push 2,
Pop 2,
Branch 2 (Rel 2),
Jump (Rel 30),
ReadInstr (DirAddr 3),
Receive 2,
Push 2,
Load (ImmValue 0) 2,
Push 2,
Pop 3,
Pop 2,
Compute LtE 2 3 4,
Push 4,
Pop 2,
Branch 2 (Rel 13),
ReadInstr (DirAddr 3),
Receive 2,
Push 2,
Load (ImmValue 1) 2,
Push 2,
Pop 3,
Pop 2,
Compute Sprockell.Sub 2 3 4,
Push 4,
Pop 2,
WriteInstr 2 (DirAddr 3),
Jump (Rel 5),
Load (ImmValue 0) 2,
Push 2,
Pop 2,
Store 2 (DirAddr 1),
Nop,
Jump (Rel (-33)),
Nop,
ReadInstr (DirAddr 3),
Receive 2,
Push 2,
Pop 2,
WriteInstr 2 (DirAddr 65536),
EndProg
```

Running the test file with “stack run -- test.hola” gives the following output in the terminal:

```
Sprockell 1 says 1
Sprockell 3 says 1
Sprockell 2 says 2
Sprockell 0 says 1
Sprockell 0 says 1
Sprockell 0 says 0
Sprockell 0 says 0
```