



Module 8 Programming Paradigms

# Final Project Report

Can Olmezoglu and Humaid Mollah

Student Number : s2269236, s2347636

April, 2022

# Contents

<b>1</b>	<b>Summary of the features</b>	<b>1</b>
<b>2</b>	<b>Problems and solutions</b>	<b>2</b>
<b>3</b>	<b>Detailed language description</b>	<b>3</b>
3.1	Basic types : integers and booleans . . . . .	3
3.1.1	Syntax . . . . .	3
3.1.2	Usage . . . . .	3
3.1.3	Semantics . . . . .	3
3.1.4	Code Generation . . . . .	3
3.2	Simple expressions . . . . .	5
3.2.1	Syntax . . . . .	5
3.2.2	Usage . . . . .	5
3.2.3	Semantics . . . . .	6
3.2.4	Code Generation . . . . .	6
3.3	Typed variables and scopes . . . . .	7
3.3.1	Syntax . . . . .	7
3.3.2	Usage . . . . .	7
3.3.3	Semantics . . . . .	8
3.3.4	Code Generation . . . . .	8
3.4	Concurrency . . . . .	9
3.4.1	Syntax . . . . .	9
3.4.2	Usage . . . . .	9
3.4.3	Semantics . . . . .	10
3.4.4	Code Generation . . . . .	11
3.5	If-Else . . . . .	14
3.5.1	Syntax . . . . .	14
3.5.2	Usage . . . . .	14
3.5.3	Semantics . . . . .	14
3.5.4	Code Generation . . . . .	14
3.6	While Loop . . . . .	15
3.6.1	Syntax . . . . .	15
3.6.2	Usage . . . . .	15
3.6.3	Semantics . . . . .	15
3.6.4	Code Generation . . . . .	15
3.7	Pointers . . . . .	16
3.7.1	Syntax . . . . .	16
3.7.2	Usage . . . . .	16
3.7.3	Semantics . . . . .	17
3.7.4	Code Generation . . . . .	17
3.8	Arrays . . . . .	18
3.8.1	Syntax . . . . .	18
3.8.2	Usage . . . . .	18
3.8.3	Semantics . . . . .	19

3.8.4	Code Generation . . . . .	19
3.9	Enumerated Types . . . . .	20
3.9.1	Syntax . . . . .	20
3.9.2	Usage . . . . .	20
3.9.3	Semantics . . . . .	20
3.9.4	Code Generation . . . . .	20
3.10	Functions . . . . .	21
3.10.1	Syntax . . . . .	21
3.10.2	Usage . . . . .	21
3.10.3	Semantics . . . . .	22
3.10.4	Code Generation . . . . .	23
3.11	Print . . . . .	24
3.11.1	Syntax . . . . .	24
3.11.2	Usage . . . . .	24
3.11.3	Semantics . . . . .	24
3.11.4	Code Generation . . . . .	24
3.12	Soft Division . . . . .	25
3.12.1	Syntax . . . . .	25
3.12.2	Usage . . . . .	25
3.12.3	Semantics . . . . .	25
3.12.4	Code Generation . . . . .	25
<b>4</b>	<b>Description of the software</b>	<b>26</b>
4.1	Symbol table management . . . . .	26
4.2	Type Checking . . . . .	27
4.3	Code Generation . . . . .	28
4.4	Error Handling . . . . .	29
<b>5</b>	<b>Test Plan and Results</b>	<b>30</b>
5.1	Syntax Error Testing . . . . .	30
5.2	Contextual Error Testing . . . . .	31
5.3	Semantic Error Testing . . . . .	32
5.4	How tests can be ran . . . . .	33
<b>6</b>	<b>Conclusions</b>	<b>34</b>
<b>7</b>	<b>Appendix</b>	<b>36</b>
7.1	Grammar Specification . . . . .	36
7.2	Extended Test program . . . . .	41
7.2.1	Banking with Maximum Threads . . . . .	41
7.2.2	Generated Sprockell Code . . . . .	42
7.2.3	Output . . . . .	49

# 1 Summary of the features

- Integer and boolean as basic types
- Declaration of variables, changing of variables
- Global variables
- Simple expressions with integer and boolean
- If/else and while statements
- 1D and 2D arrays with basic types inside
- Shared arrays with dynamic access
- Parallel blocks where multiple threads run concurrently
- Locks that can be locked and unlocked
- Enumerated types
- Functions that can be recursive or take references of values
- Pointers
- Printing of integers

## 2 Problems and solutions

One of the main problems we encountered were with race conditions that occurred with the threads. Initially, we used a Spinlock to lock a thread from starting, and then the main thread and the spawned thread would try to obtain the lock, and somehow the main thread would obtain the lock before the spawned one. This left the spawned one to work there forever, and the program to never terminate. This issue was hard to catch, and the debugger component of Sprockell was tremendously helpful in that case. In order to solve it we brainstormed concurrency and made the main thread wait for the spawned thread obtaining the lock.

Another problem we had was with setting up Sprockell in a continuous integration environment, where the updated version of Haskell did not match with the dependencies of Sprockell. In order to solve it we ran the integration environment on a windows device.

When implementing functions, we had a problem when operations were done after the callee returned. In order to fix them, the book was read again and the activation record was written by hand for some implementations to compare with debug. In the end, it turned out to be an issue with the return ARP being incorrectly stored.

## 3 Detailed language description

### 3.1 Basic types : integers and booleans

#### 3.1.1 *Syntax*

```
1 factor : primitive                                #primitiveFactor
  primitive : NUM
3         | booleanVal
          ;
5
  booleanVal : (TRUE|FALSE);
7 NUM: DIGIT+;
  fragment DIGIT: [0-9];
```

LISTING 1: A compressed example of the syntax for basic (primitive) types.

#### 3.1.2 *Usage*

```
  print( 5 + 2);
2 print (true or false);
  print(not true);
4 print(-5 * 3 + 4);
  print(-2147483647);
6 print(2147483647);
```

LISTING 2: An example of basic types being defined. Print statements are used so that the lexer will recognize the code.

This feature is used to define basic integer and boolean values in the programming language. The integers are allowed to be between  $-2^{31} + 1$  and  $2^{31} - 1$ . Despite Haskell allowing for use up to larger numbers, the checker was made to not allow them as some inconsistencies were found when numbers were used in the stack of Sprockell. If the values are dynamically assigned, it can exceed the given limit as there are no checks added in code generation. In addition, in order to represent boolean values, "true" and "false" keywords were used, so they cannot be used as an identifier for variables.

#### 3.1.3 *Semantics*

This feature checks if the defined integers are up to requirements and converts boolean keywords into integer values so that it could be represented in the compiler.

#### 3.1.4 *Code Generation*

Boolean values are converted to 1 or 0, 1 for true and 0 false, and then are loaded into a register. Afterwards, the register containing the value is pushed to the stack. Integer values are loaded into a register as an "ImmValue" and then pushed into the

stack in a similar manner. When these values are used, they are obtained from the stack by getting popped.

## 3.2 Simple expressions

### 3.2.1 Syntax

```

1  expr: superiorExpr
    | superiorExpr comp superiorExpr
    ;
4  superiorExpr: term
    | superiorExpr addOp term
6    | superiorExpr OR term
    ;
8  term: factor
    | term mult factor
10   | term AND factor
    ;
12 factor : prefixOp factor           #prefixFactor
    | LPAR expr RPAR                 #parFactor
14   | primitive                     #primitiveFactor
    ;
16 addOp: PLUS | MINUS;
    comp: LE | LT | GE | GT | EQ | NE;
18 mult: STAR | DIV;
    AND: 'and';
20 OR: 'or' ;
```

LISTING 3: A compressed example of the syntax for simple expressions. Different terminals are used for some operations to introduce precedence among them.

### 3.2.2 Usage

```

int a = 2 * (3 + 4) ;
2 int b = 2 * 3 + 4;
int c = (2 * 3) + 4;
4 bool d = a == b;
bool e = b == c;
6 bool f = a < b;
bool g = d and d or e
```

LISTING 4: An example of simple expressions with different order to test precedence.

The feature is used to calculate simple expressions such as addition, multiplication and adding prefixes for basic types, and comparing for basic types and enums.

The precedence for operators is separated into four groups. The first group with the most precedence consists of prefix operators, parenthesis operators, identifiers, function calls, and primitive values. The second group with the second highest precedence is composed of the boolean and operation with the multiplication operation. The third group with the third highest precedence is addition, subtraction and the boolean or operator. The least precedence is the compare operator.



When two operations have the same precedence, the operation in the left is evaluated first, making the evaluation of simple expressions left associative.

### 3.2.3 *Semantics*

This feature allows for the evaluation of simple expression. The evaluation follows the aforementioned precedence and left associativity. When an operation is conducted between two variables, the result will have to be stored at a variable, else it is not accesible.

### 3.2.4 *Code Generation*

Initially, the operands are evaluated into a basic type and pushed into the stack. The operands are then obtained by popping the stack twice, and based on the order of evaluation, the calculations are done with the Sprockell instructions whenever possible.

In the case where operands are not available as direct Sprockell calculations, small codeblocks that do the same as given operation is done seen in Table 1 below.

<b>Operand</b>	<b>Method</b>
Prefix Not	XOR input with True
Prefix Negative	Substitute value from 0

TABLE 1: A table with a small explanation for self created operators

### 3.3 Typed variables and scopes

#### 3.3.1 Syntax

```
1 declaration: access? type ID ASS expr END;  
  changeAss: ID ASS expr END  ;  
3 factor : ID                                     #idFactor ;  
  ID: LETTER (LETTER | DIGIT | ARR_INDEX | COMMA | DOT | POINT )  
    *;  
5 ASS: '=';
```

LISTING 5: The syntax for declaring and changing variables

#### 3.3.2 Usage

```
1 int outerScope = 10;  
  if (outerScope > 0) {  
3     bool outerScope = outerScope == 10;  
    if (outerScope){  
5        int outerScope = 75;  
        print(outerScope);  
7    }  
    print(outerScope);  
9 }  
  print(outerScope);  
11 print(outerScope);
```

LISTING 6: An example code, prints 75 and then prints 1 for true, lastly prints 10 to show local, nested scopes.

The typed variable part of this feature is used declare variables to dynamically change values and store them. The types assigned to variable names cannot be changed in terms of type during the same scope as they are strongly typed. For example, without getting inside a block by creating an if condition as seen in Listing 6, the type of *outerScope* cannot be changed.

In other fully prevent variables getting used without being assigned a value, declaring a variable without assigning some expression to it is not a part of the grammar. Thereby, if a variable is expected to be used in the future, it has to be assigned a value in the declaration, even though that value is never used. Furthermore, the compiler does not implicitly assign default values to types, so calling uninitialized variables will lead to unexpected behaviour.

Variables inside a scope that was previously exited cannot be accessed, and depending on the current state of the program, their memory location might have been deallocated, thereby making pointers to previous inner scopes unreliable.

As seen from Listing 6, it is possible to reuse the same variable name in an inner scope, however that comes at the cost of losing access to the variable declared in the closest outer scope. Variables with the same name from different scopes cannot be used together in an inner scope.

### 3.3.3 *Semantics*

The variable names are saved in a symboltable along with some data about their type and memory location.

### 3.3.4 *Code Generation*

The value of the *expr* from Listing 5 for the variables are is popped. Following that, it is written to memory locations based on the pointer assigned to them in the front-end, and depending on the access type of the variable, it is either written to the shared memory or the normal memory of the process. For the former a *WriteInstr* is used, for the latter a *Store* is used.

In order to access variables from memory, a *ReadInstr* or *Load* is used, and then the value is pushed to the stack.

## 3.4 Concurrency

### 3.4.1 Syntax

```
1 instruction: parallelConstruct      #parallelInst
      | lockConstruct                #lockInst
3      ;
parallelConstruct: PARALLEL LBRACE threadConstruct+ RBRACE;
5 threadConstruct : THREAD block;
lockConstruct : LOCK block UNLOCK;
```

LISTING 7: A compressed example of the syntax for concurrency

### 3.4.2 Usage

```
shared int money = 0;
2 parallel {
    thread {
4        int wait = 100; while (wait > 0){
            wait = wait - 1;
6            lock {
                money = money + 1;
8            }
            unlock
10        }
        parallel {
12            thread {
                    lock {
14                        money = money - 32;
                    }
                    unlock
16                }
            thread {
18                lock
20                {
                    money = money + 45;
22                }
                    unlock
24            }
        }
26    }
    thread {
28        int wait = 200; while (wait > 0){
            wait = wait - 1;
30            lock
                {
32                money = money - 2;
```

```

34         }
        unlock
36     }
}
38 print (money);

```

LISTING 8: An example of nested concurrency with threads, lock and unlock.

If a thread is called inside a loop, i.e instantiating multiple threads using only one *threadConstruct* from Listing 7 in the program, it will lead to undefined behaviour.

At most 7 memory locations worth of concurrent elements can be used. If more is used, the system will crash as it will use more memory than what physically exists. A table with the memory costs can be seen below in Table 2.

Concurrent Element	Memory Usage
One element of a shared array	1 memory location
One shared basic type	1 memory location
Thread	1 memory location

TABLE 2: A table with the explanations for concurrency memory usage. For example, a 4 element shared array would cost 4 memory locations.

There is only one lock, thereby there is no construct in the language that allows to lock on two different variables and then access those two variables concurrently.

Calling an unlock statement back to back will not make sense to the lexer. Unlock by itself does not deal with the memory. After the code block is finished the system will release the lock automatically.

If a lock is called inside a lock/unlock statement, it will cause the threads the deadlock. Thereby calling lock inside a lock will lead the program to be stuck forever and show unexpected behaviour. A lock should be unlocked before calling lock again.

If a thread is created outside a parallel block, it will lead to undefined behaviour as it will never get triggered and is not intended to be accepted by the lexer.

Local memory declared in a thread that spawns another thread will not be transferred to the spawned thread.

In a situation where multiple threads are initiated inside a parallel block, the thread which was written first will start slightly earlier than the threads written later. These threads will interleave after all has started, but the they will start sequentially.

### 3.4.3 Semantics

The concurrency follows the *parbegin/parend* construct. Every block inside *threadConstruct* from Listing 7 will be executed in parallel with the other threads inside the *parallelConstruct* from Listing 7. When entering in a *parallelConstruct*, the thread which initiates the *parallelConstruct* will stop and wait for the *parallelConstruct* to terminate.

A *lockConstruct* can be called inside a thread. When the thread obtains the lock, it will execute the instructions inside *block*, while other threads are waiting to obtain the lock and the thread to unlock. Thereby, if a variable is to be changed atomically between threads, it could be done inside a *block* in *lockConstruct* as seen in Listing 7. Following the execution of all the instructions, the lock will be released automatically. After unlocking is done, the following statements will then be executed concurrently.

#### 3.4.4 Code Generation

On a program layout level, when a *parallelConstruct* is encountered, a global boolean **threaded** is made to be true. After that, a trigger which will be explained in the following paragraphs is added. Then all the threads are visited and the code they produce is put into a hashtable with the keys as the thread number and the value as the list of instructions the threads contain inside their blocks. This is followed by the rest of the instructions for the main thread to be generated. After the generation is complete, on the top of the instructions, jump instructions make different processes jump to their respective code blocks is added on the top of the instructions. Lastly, the thread instructions are added to the bottom of the program. An example of the program layout can be found in Table 3 below.

Jump Instructions for thread 1 and 2
Main thread instructions
Trigger for thread 1 and 2
Main thread instructions <i>continued</i>
Thread 1 instructions
Thread 2 instructions

TABLE 3: A table showing the layout of generated instructions for a program consisting of one main thread and a *parallelConstruct* spawning two threads in the middle of code for the main thread.

Initially, the shared memory is expected to consist of only 0's as seen in Table 4. When a process goes over the instructions list, it will first write the number 1 to the shared memory address which is it's process ID obtained from *regSprID*, an example for Listing 8 seen in 5. It will then jump to a location based on their process ID. If the process is the main process and thereby has the ID 0, it will not pass any of the jump conditions and will therefore go into the main thread. The other processes will jump to the beginning of their code block.

<b>Address</b>	0	1	2	3	4	5	6	7
<b>Value</b>	0	0	0	0	0	0	0	0

TABLE 4: A table showing the memory values inside the shared memory when no code has been executed.

Address	0	1	2	3	4	5	6	7
Value	0	1	1	1	1	0	0	0

TABLE 5: A table showing the memory values when threads start executing the first instructions and changing the shared memory.

After a thread finishes writing 1 to it's address, it will then jump to the beginning of it's code block, as seen below the "Main thread instructions *continued*" component in Table 3. In the beginning of a threads instructions, a spinlock like implementation is made to keep the thread from executing it's instructions before being triggered. The thread will try to *TestAndSet* the memory location of their process ID, and as it was initially set to 1 it will fail and loop.

When a process wants to spawn a parallel block, it will write 0 to the shared memory address corresponding to memory addresses for the threads inside the parallel block. For example, when entering in line 2, the main thread will need to start threads 1 and 2, so it will write 0 to shared memory locations 1 and 2, creating the memory seen in Table 6.

Address	0	1	2	3	4	5	6	7
Value	0	0	0	1	1	0	0	0

TABLE 6: A table showing the memory values when threads 1 and 2 are spawned, thereby memory addresses 1 and 2 are zero.

After the memory location is changed, the thread gets out of the loop on top of it, as finally it can test that the memory address is 0 and set it 1. After exiting the loop, the thread writes 2 to it's shared memory address as seen in Table 7.

Address	0	1	2	3	4	5	6	7
Value	0	2	2	1	1	0	0	0

TABLE 7: A table showing the memory values when threads 1 and 2 obtain access, thereby changing the memory values to 2.

The main thread, after setting the memory location of the spawned thread to 0, waits for the memory location of the spawned thread to be 2. After it's 2, the main thread waits on a test and set loop on the memory address of the spawned thread.

After the spawned thread is finished executing, it will write 0 to it's shared memory address, thereby letting the main thread get out of the loop and start executing.

Address	0	1	2	3	4	5	6	7
Value	0	0	0	1	1	0	0	0

TABLE 8: A table showing the memory values after the spawned threads are finished, but the main thread has not resumed operations yet, thereby changing the memory values to 2.

<b>Address</b>	0	1	2	3	4	5	6	7
<b>Value</b>	0	1	1	1	1	0	0	0

TABLE 9: A table showing the memory values after the spawned threads are finished and the main thread resumes operations as the test and set operation was successful.

The reason for the spawned thread changing the memory address to 2 after getting out of the loop is avoiding race conditions. To explain further, if the main thread directly goes on to testing and setting the memory address for the spawned thread, then sometimes it would set it to 1 before the actual thread that was meant to be spawned. In that case, the thread would never be able to exit the loop. As the main thread now waits until the spawned thread gets access before trying test and set, the race condition is avoided.

The lock works on a spinlock logic using the instruction *TestAndSet* and shared memory address 0. When a thread enters a lockConstruct, it will enter in a loop trying to test and set memory address 0. It will only exit the loop if it was able to test and set the memory address 0 to hold the value 1. As it is 1, the other threads will be stuck in the loop, thereby the thread which has entered the lock will have sole access. As the testandset is said to be atomic, only one thread can enter inside their code block at a time. After a thread finishes executing, it will let the other threads start by setting the memory address 0 to be 0.

<b>Address</b>	0	1	2	3	4	5	6	7
<b>Value</b>	0	2	2	1	1	0	0	0

TABLE 10: A table showing the memory state before either thread 1 or 2 obtains the lock.

<b>Address</b>	0	1	2	3	4	5	6	7
<b>Value</b>	1	2	2	1	1	0	0	0

TABLE 11: A table showing the memory state when either thread 1 or 2 obtains the lock.

<b>Address</b>	0	1	2	3	4	5	6	7
<b>Value</b>	0	2	2	1	1	0	0	0

TABLE 12: A table showing the memory state when the thread which obtained the lock releases the lock.



## 3.5 If-Else

### 3.5.1 Syntax

```
1 ifConstruct : IF LPAR expr RPAR block (ELSE block)? ;
2
3 block: LBRACE instruction* RBRACE;
```

LISTING 9: A compressed example of the syntax for the if-else construct

### 3.5.2 Usage

```
1 function bool
2     belowSpeedLimit ( int speedLimit , int currentSpeed ) {
3         if ( currentSpeed >= speedLimit ) {
4             return false;
5         }
6         else {
7             return true;
8         }
9     }
10
11 int fine = 500;
12
13 if ( not belowSpeedLimit(100,101) ) { print(fine);}
14 else {print(0);}
15 if ( not belowSpeedLimit(100,99) ) { print(fine);}
16 else {print(0);}
```

LISTING 10: An example of and if else block being used within and outside a function definition

The compiler supports a basic if else command. The if construct can also be defined with just an if block as can be seen in the example above. The expression within the LPAR and RPAR should be a Boolean value for this construct to behave normally, otherwise an error is raised as an exception.

### 3.5.3 Semantics

This feature checks whether the "expr" within the LPAR and RPAR is a boolean value. It also checks for correct syntax which is by the use of curly braces to specify the if and else block.

### 3.5.4 Code Generation

The code for the if else block is generated as follows: 1. Visit the "expr" that controls the main flow. 2. Visit the blocks but do not add them. 3. Create the jump based on the sizes of the blocks. 4. Finally add all instructions.

## 3.6 While Loop

### 3.6.1 Syntax

```
whileConstruct : WHILE LPAR expr RPAR block ;  
2  
block: LBRACE instruction* RBRACE;
```

LISTING 11: A compressed example of the syntax for the while construct

### 3.6.2 Usage

```
1 int a = 100;  
  while (a > 0){  
3     a = a - 1;  
    print(a);  
5 }  
print(d);
```

LISTING 12: An example of of a while loop

The compiler supports a basic while loop whose usage can be seen in the example above. The condition for the while loop should be a boolean value, otherwise the compiler raises an exception.

### 3.6.3 Semantics

This feature checks if expression within the LPAR and RPAR is really a boolean value, otherwise the Checker throws an error. It also checks for correct syntax usage of the while loop where the instructions can be listed inside the LCURLY and RCURLY block.

### 3.6.4 Code Generation

The code for the while loop first checks whether the condition ("expr") is true. If it is indeed true, the compiler goes on to generate the code inside the block which can contain 0 or more instructions. Otherwise it simply terminates the loop.

## 3.7 Pointers

### 3.7.1 Syntax

```
1 declarePointer: access? POINTER ID ASS factor END;  
  
3 usePointer : ID &  
  
5 POINTER: "pointer";
```

LISTING 13: A compressed example of the syntax for pointers

### 3.7.2 Usage

```
1 int a=100;  
   pointer b=a;  
3 int d = 200;  
   pointer c = d;  
5 if (b<c){  
    a = a+500;  
    d = d+500;  
7 }  
9 print (b&);  
   print (c&);  
11 c& = 100;  
   print (d);
```

LISTING 14: An example of pointers being defined and used for both referencing and updating a variable

```
1 function void swap (int x&,int y&){  
    int temp = x;  
3    x = y;  
    y= temp;  
5 }  
   int a = 5;  
7 int b = 77;  
   pointer c = a;  
9 pointer d = b;  
   run(swap(c,d));  
11 print(a);  
   print(b);
```

LISTING 15: An example of pass by reference with the use of pointers

Pointers can be assigned to a variable only. The type of the pointer is set to follow that of the variable that it is being assigned to. This is so that an invalid type is not assigned to the variable by referencing the pointer. Pointers can be used to point to variables and update the values of the variables. You can have more than one

pointer pointing to the same variable. Pass by reference has been depicted through integration testing which can be found in the test files.

### *3.7.3 Semantics*

This feature checks if the variable that the pointer is assigned to really exists in that scope. It also checks whether the type value is correct when its is being used to update the variable by trying to reference the pointer.

### *3.7.4 Code Generation*

The pointer code is generated in the back-end by using 2 Hash maps. The first stores the address of the variable in a hash-map along with the names of the variables. The second stores the pointer-variable pair. Therefore, when it is invoked, the address can be read and the the variable whose address is stored is loaded instead.

## 3.8 Arrays

### 3.8.1 Syntax

```
1 declareArray: access? type ID LSQR NUM RSQR ASS darray END;  
3 declare2dArray: access? type ID LSQR NUM RSQR LSQR NUM RSQR  
  ASS LBRACE darray (COMMA darray)* RBRACE END;  
5 darray: LBRACE expr (COMMA (expr))* RBRACE;  
7 1darrayAccess: ID % NUM  
9 2darrayAccess: ID % NUM % NUM
```

LISTING 16: A compressed example of the syntax for arrays

### 3.8.2 Usage

```
1 int a[5] = {100,250,30,47,55};  
  a%1 = a%2+a%0;  
3 print(a%1);  
  bool arr[3] = {true,false,true};  
5 print(arr%1==arr%2);  
  
7 int arr[2][3] = { {1,2,3} , {4,5,6} };  
  print(arr%1%2);  
9 int b[1][4] = { {7,8,9,10} };  
  print(b%0%3);  
11 print(arr%1%2 + 4 == b%0%3);
```

LISTING 17: An example of arrays being defined and used.

```
1 int i =0;  
  int a[5] = {100,250,30,47,55};  
3 while(i<5){  
  print(a%i);  
5 i = i +1;  
  }  
7  
  int j=0;  
9 int arr[2][3] = { {1,2,3} , {4,5,7} };  
  while(j<2){  
11 print(arr%j%0);  
  j=j+1;  
13 }
```

LISTING 18: An example of the use of dynamic arrays

This feature is used to define both 1 dimensional and 2 dimensional arrays and shows how they can be used. The array element can be accessed by using only numbers and variables alike. Array values can be stored in a variable and can also be compared with other data types. Array values cannot be accessed through simple expression.

### *3.8.3 Semantics*

This feature checks if the size of the array that is given is equal to the number of values that have been listed. It also checks whether the values are of the same type of the array. During usage, it checks whether a valid index is used. Also, its type is checked when the array element is used in an expression.

### *3.8.4 Code Generation*

Array values are treated like variables and are generated by storing the unique indexes in the memory. The register that stores these values are loaded as soon as that index is used in the program. If a wrong array index is used, an error is thrown.

## 3.9 Enumerated Types

### 3.9.1 Syntax

```
1 declareEnum: access? type ENUM ID enumAssign END;  
3 enumAssign: LBRACE ID (ASS (expr))? (COMMA ID (ASS (expr)))?)*  
    RBRACE;  
5 enumUsage: Enum ID . Value ID
```

LISTING 19: A compressed example of the syntax for enumerated types.

### 3.9.2 Usage

```
1 int enum cars {  
    bmw = 1,  
3    mercedes = 2,  
    ferrari =3  
5 };  
7 print(cars.bmw);  
9 int var = cars.ferrari;  
    print(var);
```

LISTING 20: An example of enumerated being defined and used. Print statements are used so that the lexer will recognize the code.

This feature is used to define enumerated types and show how they can be used. To define enums, you have to specify the type of the enum and give it an identifier. For the individual enum value, a final value (either int or bool) needs to be assigned to that value which **cannot** be changed. This value remains constant throughout the course of the program and can be used to make checks or used in expressions along with other data types.

### 3.9.3 Semantics

This feature checks if the value assigned to each enum is the same type as the type used in the enum declaration. It also checks wheather the enum value is being changed and if so, an error is raised.

### 3.9.4 Code Generation

Enumerated types can also be treated as variables of type int or bool. One again, the individual values in the enum are stored in registers and loaded when they are called along with the identifier.

## 3.10 Functions

### 3.10.1 Syntax

```
1 instruction: functionConstruct      #functionInst
          | runProcedureConstruct    #runProInst
3          | returnConstruct          #returnInst
          ;
5 returnConstruct : RETURN expr? END;
  runProcedureConstruct: RUN LPAR factor RPAR END;
7 functionConstruct: FUNCTION type ID LPAR ((type ID) (COMMA
          type ID )* )? RPAR block;

9 RETURN : 'return';
  FUNCTION: 'function';
11 RUN: 'run';

13 type: INTEGER
      | BOOLEAN
15      | VOID
      ;
17
  BOOLEAN: 'bool' ;
19 INTEGER: 'int' ;
  VOID: 'void' ;
```

LISTING 21: A compressed example of the syntax for functions.

### 3.10.2 Usage

```
function int fib( int a ){
2   if ( (a == 0) or (a == 1)){
       return a;
4   }
   return fib(a - 1) + fib (a - 2);
6 }
print( fib(6));
```

LISTING 22: An example of a recursive function defined, finding the a'th fibonacci in this case.

Functions should be declared following the syntax of a *functionConstruct* from Listing 21. If a function is called before being declared as such, the compiler will not understand what is being called.

Functions cannot take a global variable, an enumerated type or an array as a parameter. Functions can only take pointers or values of integers and booleans.

Functions have one global scope, thereby same variable names cannot be re-declared in inner scopes.

Function overloading is not allowed, thereby functions with the same name but parameters with differing types cannot be declared.



When a function is being called, another function cannot be called inside the parameters for the initial function call. An illustration can be seen in Line 13 of Listing 23, where the function `gcd` is called with function `modulo` as a parameter. This issue can easily be fixed with moving the function call to a different line, as seen in Line 13 of Listing 24.

```

1 function int modulo (int a, int b) {
    if (b > a){
3         return a;
    }
5     int c = a / b ;
    return a - c * b;
7 }

9 function int gcd(int x, int y) {
    if (y == 0) {
11         return x;
    }
13     return gcd(y, modulo( x , y ));
}

```

LISTING 23: An example of a function being broken with a function call made as a parameter in a different function call

```

function int modulo (int a, int b) {
2     if (b > a){
        return a;
4     }
    int c = a / b ;
6     return a - c * b;
}

8
function int gcd(int x, int y) {
10     if (y == 0) {
        return x;
12     }
    int mod = modulo( x , y );
14     return gcd(y, mod);
}

```

LISTING 24: An example solution to not call a function inside the parameters of another function

### 3.10.3 Semantics

This feature allows the user to make a function which creates a code block with its own memory space that can be executed on call. It also allows for pass by reference and recursive function calls.

#### 3.10.4 Code Generation

Functions are created based on a modified version of an activation record stored in memory. When a function passes through the checker, the amount of variables it has with the amount of local data it uses is stored. An example of the activation record can be seen in Table 13.

Local data
Caller ARP
ARP
Return address
Parameters

TABLE 13: A table showing the layout of the activation record for called functions in the language.

In the code generation phase, when a function is called from outside a function block, the code will create an activation record for the function call. In order to calculate the activation record pointer of the called function, it will take the sum of the current memory used, the local data size for the called function, and 2 (1 for the Caller ARP, 1 for ARP). As the caller is not a function and therefore the activation record pointer would not need to be reset when the callee is done, the caller ARP is left empty. The parameters will then be loaded into the memory locations starting from the calculated  $\text{ARP} + 2$ . The ARP value will be loaded into register F, which is reserved for the ARP. The PC counter will be loaded in  $\text{ARP} + 1$  so the callee can return. Lastly, an instruction called *FakeInst* will be added, which is converted into jump instructions for the function.

If a function is called from inside a function, one major difference from being called outside a function is how the ARP of the callee will be set up. Instead of calculating how much memory is currently being used, as the memory is now allocated dynamically, it will calculate the ARP obtained from register F, plus the sum of the local data size of callee, the parameters size for the caller and two for the return address of the caller, and caller ARP of the callee.

After all of the instructions are visited in the program, the compiler will add the code block of the functions below the program. Following that, it will find the *FakeInst* and change them into jump instructions for the functions.

If the parameter is a pointer, indicated by whether it contains " or not, the scanner will mark the variable as such using the model *VariableData*. When changing passed by reference variable inside a function, the memory location is obtained from the pointer, while calling the ID's to the variable is made to obtain the value that the pointer points to. These are done to through the *IndAddr* calls.

## 3.11 Print

### 3.11.1 Syntax

```
1 printConstruct : PRINT LPAR expr RPAR END;
```

LISTING 25: A compressed example of the syntax for print

### 3.11.2 Usage

```
print (5);
```

LISTING 26: An example of print being used

The print does not support letters, or prints the string true or false. It also does not print enum values or whole arrays.

### 3.11.3 Semantics

This feature prints the value its given through I/O channels.

### 3.11.4 Code Generation

The expr inside the print construct is visited. The stack is popped to obtain the value inside the expr. Then a *WriteInstr* is used to write the value to numberIO for printing.

## 3.12 Soft Division

### 3.12.1 Syntax

```
1 term: factor
    | term mult factor;
3 mult: STAR | DIV;
DIV: '/';
```

LISTING 27: A compressed example of the syntax for soft division

### 3.12.2 Usage

```
1 print (80/10);
```

LISTING 28: An example of soft division being used

The soft division does not terminate in division by 0.

### 3.12.3 Semantics

This feature allows for soft division being done with integers.

### 3.12.4 Code Generation

A basic algorithm where the divisor is subtracted from the dividend is written in Sprockell Instructions. The amount of subtractions is the resulting value. If one of the operands is negative, they will make a register corresponding to them store whether they are larger than zero. Then both registers from two operands are XOR'ed together to find out whether the resulting number will negative or not.

## 4 Description of the software

### 4.1 Symbol table management

This feature allows there to be conventions on how memory is allocated and is accessed. In order to create the necessary abstractions to execute this convention, a structure similar to *sheaf of tables* described in Chapter 5.5 of Engineering a Compiler is used [1].

This feature is written inside the front-end of the compiler. With a tree listener, every time a *declaration* from Listing 5 is exited, a new variable is added to the current scope.

When a scope is left, the memory allocated to the variables inside the scope is freed up. Thereby, values inside inner scopes that was exited is not accessible even with pointers. This is done in order to preserve memory space.

When a *changeAss* from Listing 5 a variable is checked for the current scope, if it does not exist the rest of the scopes will be visited from the most inner to outer. This is done in order to have nested scopes and being able to use variables that was declared before.

Every variable has a `VariableData` object to represent it, this object stores the necessary elements to generate the code, such as whether the variable is a parameter, the offset of the variable and whether the variable is shared or not. These values are obtained by the scanners by using the name of the variable, where the variable of the closest scope is returned. The scanner puts those values in a `ParseTreeProperty` which then uses them to generate code.

## 4.2 Type Checking

The type checking is done inside a file called *Scanner.java* . The type checking uses the scope table and uses a tree listening structure. When exiting variables and expressions, it assigns them a type, based on previously assigned types. If any of the types do not match, it will add them to an errors list which is added to an object called Result.

In addition, type checking uses a lot of Parse Tree Properties, where the number of a thread or the data for functions are stored inside, so that code generation can use them. For example, when a function goes through the elaboration phase, it will get a FunctionData object created for it where the size of the local data and parameters will be stored. This is then used in the code generation to create the activation register pointer.

### 4.3 Code Generation

Code generation is a tree visitor which upon visiting a parse tree returns a list of Sprockell instructions. It was chosen to be that way in order to account for while loops and threads, where information is not added based on how its parsed. To explain further, the branching would have to be written after the length of the block was obtained for if, thereby having the control over when the instructions will be written allows to control code generation better.

In another example, the code for functions or threads are added last to have control over the jumping instructions, and it is only allowed by list of Sprockell instructions. In the previous compilers made, the code would get added directly to a global variable with the instructions. This made jumping less intuitive and with this the instruction can be stored until the whole instruction list for the program is generated, and then the other code like threads and functions will be added.

Some parse tree properties were used through the result scanner returns. For example, to see whether a variable is global or not, the code generation will search for the context of the variable in a parse tree property called globals stored in the result object.

## 4.4 Error Handling

As the scanner goes through input, it will store all the errors it finds out inside a list object of strings. Those strings contain descriptions of the errors and the lines. After the elaboration is done, if the the list has an element the scanner will throw an exception.



## 5 Test Plan and Results

### 5.1 Syntax Error Testing

To test syntax errors, all the features were major features like if else, while, concurrency etc were tested to raise an exception thrown by the lexer.

The syntax test for each of these features consists of 2 parts. The first one tests whether the compiler indeed throws an exception with a Syntax Error message and following that, it tests the usage of correct syntax which should throw no errors. This is done by using a try catch block which ensures that if the parser runs into an exception, the test fails.

The syntax has been tested against the sample code that can be found inside the sample directory. The tests can be found inside *ut/pp/tests/syntax\_test* .



## 5.2 Contextual Error Testing

Contextual errors for the mandatory part of the project are scope errors and type checking errors. In addition, for extra parts, there could be type errors with functions not being able to take or return certain types. In order to test for error handling of the compiler, some programs were written with mistakes and how the compiler reacted to them was checked. In order for the tests to not be too time consuming, we did not execute the code generation part for these, only the scanner and the lexer. The tests for contextual files can be found under *ut/pp/tests/contextual*.

In order to handle the magnitude of tests and features, all of the tests were written in a class with their feature name. This was done in order to make sure that all of the features were at least tested to some extent. For example, when scopes for array names were tested, it was located at arrays, as the scopes were initially added for basic types. In addition, some features required slightly more than type checking, for example functions where the names had to be checked as well. The tests being located in functions made sure that the separation was clean.

In order to make sure the scanner worked as expected, all of the testable parts of the scanner were attempted to be tested. To quantify whether the scanner was tested to a high extent, code coverage was used. This allowed to make sure the scanner behaved as expected and there not any mental mistakes. The coverage report for all tests can be seen in Figure 1.

FIGURE 1: The test coverage for Scanner and ScopeTable

 Scanner	100% (1/1)	97% (41/42)	90% (345/3...
 ScopeTable	100% (1/1)	75% (9/12)	84% (72/85)

For all of the features, a potential situation where the program should fail was tested, and then the same situation was fixed and it was tested whether the program would accept it. This was chosen as a method to increase the readability of the tests, as the reader is currently able to test expected behaviour. In order to increase readability, the all of the tests were given Javadoc comments explaining what behaviour was expected.

Most of the tests for contextual testing was written to be trivial as complex programs were tested with Semantic error testing, and the tests for the contextual errors were expected to work on specific failures, not the combination of multiple features failing.

### 5.3 Semantic Error Testing

To test semantic errors, an automatic runner was used. The automatic runner takes a .txt file consisting a program, executes it and returns the I/O output. This I/O output was then tested against the expected values.

In order to find better test programs to make sure the compiler could represent the combination of features, Rosetta code was used. Rosetta code is a website where solutions to certain tasks are presented, where users are expected to write solutions to them in different programming languages to see if those languages have the necessary tools to execute the solutions. The solutions that were deemed to fit with the language features in this language were tested. Some solutions picked from Rosetta Code were greatest common divisor, insertion sort, checking the existence of an element in a 2d array and a while loop with division.

In order to test concurrency better, the banking application was tested without locks in order to see whether the result would be the same or not, thereby testing the lock made an actual change, In addition, tests were made to see whether threads took turn printing certain numbers when they were made to run concurrently, in order to see whether they actually ran concurrently or not. In order to test concurrency more, a vector sum was made where two threads read different parts a global array, and the sum of their calculations were returned.

In order to test functions extensively, for greatest common divisor and isPrime were tested against a Java function with the same logic for multiple differing inputs. Thereby it was tested whether the control flow of the programs ran expected against multiple inputs.

To quantify the quality of tests the whether code generation part behaved as expected or not, code coverage was used. The idea was that if every method was covered by the tests, their behaviour would be tested to some degree and there would not be any method with totally unexpected behaviour. The final coverage numbers can be seen in Figure 2 and 3. The tests can be found inside *ut/pp/tests/semantic\_tests*.

FIGURE 2: The test coverage for CodeGen, the main generation visitor class



FIGURE 3: The test coverage for ArraySp, a method which is also used for code generation of dynamic array access



## 5.4 How tests can be ran

The tests can be run through the maven command "mvn test", as specified in the readme document. The tests are automated so all of them can be ran with that command.

## 6 Conclusions

One of the main difficulties we had was with re-learning everything, and in order to solve that we spent a lot of time researching, which made it harder for us to start the project. After starting the project, we felt like we made a lot of progress and implemented a lot of the features we wanted. However one regret we had was not knowing how difficult it was to cross test different features. For example, we really struggled with testing functions with arrays, and combining them, eventually not being able to do it.

Another struggle we had was doing the project coupled with another module, which led time management to be a nightmare at times. However we overcame that by working on weekends as well, which was not a massive problem as the project was really enjoyable to make. The project stressed us a bit too much at times, because failing it would mean a direct extension of our studies, and therefore we had troubles focusing efficiently sometimes.

We really enjoyed the module when it was done 10 months ago, but were very disappointed to fail and this pushed us to do a project we wanted to be proud of. This led to some over creating of features, but we were happy with the overall product as we feel it shows a lot of effort.

In conclusion, we are happy with our language. We wished to have implemented a better integration of features and error catching. However, it is nice to have found so many small issues with the features as it means we have tested as carefully as we could. We hope our effort and extra time working is reflected in our product.

## References

- [1] Keith D Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

## 7 Appendix

### 7.1 Grammar Specification

```
1 grammar MyLang;
3
4 /** Outermost nonterminal */
5 program: instruction*;
6
7 /** Program instructions */
8 instruction:
9     statement                                #statementInst
10         // used for declarations and changing
11     | ifConstruct                            #ifInst
12         // standard if statement
13     | whileConstruct                        #whileInst
14         // standard while statement
15     | parallelConstruct                    #parallelInst
16         // creates parallel block
17     | printConstruct                      #printInst
18         // print basic types
19     | lockConstruct                       #lockInst
20         // locks and unlock
21     | returnConstruct                    #returnInst
22         // returns from functions
23     | functionConstruct                  #functionInst
24         // defines functions
25     | runProcedureConstruct              #runProInst
26         // runs void functions
27     ;
28
29 /** Statement instructions */
30 statement: declaration                    #declStat
31         // declares a variable of basic type
32     | changeAss                          #changeStat
33         // changes a variable of basic type / array
34         value
35     | declareArray                      #declArray
36         // declares a one dimensional array
37     | declare2dArray                  #decl2dArray
38         // declares a two dimensional array
39     | declareEnum                      #declEnum
40         // declares an enum
41     | declarePointer                  #declPointer
42         // declares a pointer
43     ;
```

```

    /** If statement */
31 ifConstruct : IF LPAR expr RPAR block (ELSE block)? ;

33 /** While statement */
    whileConstruct : WHILE LPAR expr RPAR block ;
35
    /** Parallel block */
37 parallelConstruct: PARALLEL LBRACE threadConstruct+ RBRACE;

39 /** Prints values */
    printConstruct : PRINT LPAR expr RPAR END;
41
    /** locking and unlocking */
43 lockConstruct : LOCK block UNLOCK;

45 /** Returning from functions */
    returnConstruct : RETURN expr? END;
47
    /** Defining functions */
49 functionConstruct: FUNCTION type ID LPAR ((type ID) (COMMA
    type ID )* )? RPAR block;

51 /** Void function caller */
    runProcedureConstruct: RUN LPAR factor RPAR END;
53
    /** Thread definition */
55 threadConstruct : THREAD block;

57 /** Block of instructions */
    block: LBRACE instruction* RBRACE;
59
    /** Declaring a pointer */
61 declarePointer: POINTER ID ASS factor END;

63 /** Declaring an enum */
    declareEnum: access? type ENUM ID enumAssign END;
65
    /** Assigning values inside an enum */
67 enumAssign: LBRACE ID (ASS (expr))? (COMMA ID (ASS (expr)))?)*
    RBRACE;

69 /** Declaring a 2D array */
    declare2dArray: access? type ID LSQR NUM RSQR LSQR NUM RSQR
    ASS LBRACE darray (COMMA darray)* RBRACE END;
71
    /** Declaring a 1D array */
73 declareArray: access? type ID LSQR NUM RSQR ASS darray END;

75 /** The expression on the right hand side of an array

```



```

    declaration    **/
darray: LBRACE expr (COMMA (expr))* RBRACE;
77

79 /** Expressions for comparing - lowest precedence **/
expr: superiorExpr
81     | superiorExpr comp superiorExpr
    ;
83

/** Expressions for addition, subtraction, OR operation -
    second lowest precedence **/
85 superiorExpr: term
    | superiorExpr addOp term
87     | superiorExpr OR term
    ;
89

/** Expressions for multiplication, division, AND operation -
    second highest precedence **/
91 term: factor
    | term mult factor
93     | term AND factor
    ;
95

/** Expressions for prefixing, parenthesis , variable call,
    primitive declaration
    and function calls - maximum precedence **/
97 factor : prefixOp factor           #prefixFactor
    | LPAR expr RPAR                 #parFactor
    | ID                             #idFactor
101    | primitive                    #primitiveFactor
    | ID LPAR (expr (COMMA expr)* )? RPAR #funcCall
103    ;
105

/** Declare basic types **/
107 declaration: access? type ID ASS expr END;

109 /** Used to change variables of basic types and array values
    **/
changeAss: ID ASS expr END ;
111

/** Used to write primitive types**/
113 primitive : NUM
    | booleanVal
115    ;

117 /** Used to write boolean values**/
booleanVal : (TRUE|FALSE);
119

```

```

    /** Used to declare prefix operations**/
121 prefixOp: MINUS | NOT;

123 /** Multiplication operators **/
    mult: STAR | DIV;
125
    /** Addition operators **/
127 addOp: PLUS | MINUS;

129 /** Shared access **/
    access: SHARED ;

131
    /** Comparation operators **/
133 comp: LE | LT | GE | GT | EQ | NE;

135 /** Variable types **/
    type: INTEGER
137         | BOOLEAN
        | VOID
139         ;

141 /** The keywords **/
    DIV: '/';
143 AND: 'and';
    BOOLEAN: 'bool' ;
145 INTEGER: 'int' ;
    ELSE: 'else' ;
147 END: ';' ;
    FALSE: 'false';
149 SHARED: 'shared';
    IF: 'if' ;
151 THREAD: 'thread' ;
    NOT: 'not' ;
153 OR: 'or' ;
    TRUE: 'true' ;
155 WHILE: 'while';
    PRINT: 'print';
157 PARALLEL: 'parallel';
    LOCK : 'lock';
159 UNLOCK : 'unlock';
    RETURN : 'return';
161 FUNCTION: 'function';
    ENUM: 'enum';
163 POINTER: 'pointer';
    VOID: 'void';
165 RUN: 'run';

167 ASS: '=';
    EQ: '==';

```

```

169 GE:      '>=';
    GT:      '>';
171 LE:      '<=';
    LBRACE:  '{';
173 LPAR:    '(';
    LSQR:    '[';
175 LT:      '<';
    MINUS:   '-';
177 NE:      '!=';
    PLUS:    '+';
179 RBRACE:  '}';
    RPAR:    ')';
181 RSQR:    ']';
    STAR:    '*';
183 COMMA:   ',';
    ARR_INDEX: '%';
185 DOT:     '.';
    POINT:   '&';
187
189 fragment LETTER: [a-zA-Z];
    fragment DIGIT: [0-9];
191
    NUM: DIGIT+;
193 ID: LETTER (LETTER | DIGIT | ARR_INDEX | DOT | POINT)*;
    WS: [ \t\r\n]+ -> skip;

```

LISTING 29: Language Description.

## 7.2 Extended Test program

### 7.2.1 Banking with Maximum Threads

```
2 shared int money = 0;

4 parallel {
    thread {    int wait = 100; while (wait > 0){
6        wait = wait - 1;
        lock {
8            money = money + 1;
        }
10       unlock
    }
12 }
    thread {
14     int wait = 100; while (wait > 0){
        wait = wait - 1;
16     lock {
        money = money - 2;
18     }
        unlock
20    }
}
22 thread {
    lock {
24     money = money + 5;
    }
26    unlock
}
28 thread {    int wait = 100; while (wait > 0){
        wait = wait - 1;
30     lock {
        money = money - 1;
32     }
        unlock
34    }
}
36 thread {
    int wait = 100; while (wait > 0){
38     wait = wait - 1;
        lock {
40         money = money + 3;
        }
42     unlock
    }
44 }
    thread {
```

```

46     lock {
        money = money + 78;
48     }
    unlock
50 }
}
52 print(money);

```

LISTING 30: Extended Test Program.

### 7.2.2 Generated Sprockell Code

```

1 import Sprockell
  prog :: [Instruction]
3 prog = [Compute Equal regSprID reg0 regA,
  Branch regA (Rel (3)),
5 Load (ImmValue 1) regA,
  WriteInstr regA (IndAddr regSprID),
7 Load (ImmValue 1) regA,
  Compute Sub regA regSprID regA,
9 Branch regA (Rel (2)),
  Jump (Rel (91)),
11 Load (ImmValue 2) regA,
  Compute Sub regA regSprID regA,
13 Branch regA (Rel (2)),
  Jump (Rel (137)),
15 Load (ImmValue 3) regA,
  Compute Sub regA regSprID regA,
17 Branch regA (Rel (2)),
  Jump (Rel (183)),
19 Load (ImmValue 4) regA,
  Compute Sub regA regSprID regA,
21 Branch regA (Rel (2)),
  Jump (Rel (203)),
23 Load (ImmValue 5) regA,
  Compute Sub regA regSprID regA,
25 Branch regA (Rel (2)),
  Jump (Rel (249)),
27 Load (ImmValue 6) regA,
  Compute Sub regA regSprID regA,
29 Branch regA (Rel (2)),
  Jump (Rel (295)),
31 Load (ImmValue 0) regA,
  Push regA,
33 Pop regA,
  WriteInstr regA (DirAddr 7),
35 WriteInstr reg0 (DirAddr 1),
  ReadInstr (DirAddr 1),
37 Receive regA,

```

```

    Load (ImmValue 2) regB,
39 Compute Sub regB regA regB,
    Branch regA (Rel (-4)),
41 WriteInstr reg0 (DirAddr 2),
    ReadInstr (DirAddr 2),
43 Receive regA,
    Load (ImmValue 2) regB,
45 Compute Sub regB regA regB,
    Branch regA (Rel (-4)),
47 WriteInstr reg0 (DirAddr 3),
    ReadInstr (DirAddr 3),
49 Receive regA,
    Load (ImmValue 2) regB,
51 Compute Sub regB regA regB,
    Branch regA (Rel (-4)),
53 WriteInstr reg0 (DirAddr 4),
    ReadInstr (DirAddr 4),
55 Receive regA,
    Load (ImmValue 2) regB,
57 Compute Sub regB regA regB,
    Branch regA (Rel (-4)),
59 WriteInstr reg0 (DirAddr 5),
    ReadInstr (DirAddr 5),
61 Receive regA,
    Load (ImmValue 2) regB,
63 Compute Sub regB regA regB,
    Branch regA (Rel (-4)),
65 WriteInstr reg0 (DirAddr 6),
    ReadInstr (DirAddr 6),
67 Receive regA,
    Load (ImmValue 2) regB,
69 Compute Sub regB regA regB,
    Branch regA (Rel (-4)),
71 TestAndSet (DirAddr 1),
    Receive regA,
73 Compute Equal regA reg0 regA,
    Branch regA (Rel (-3)),
75 TestAndSet (DirAddr 2),
    Receive regA,
77 Compute Equal regA reg0 regA,
    Branch regA (Rel (-3)),
79 TestAndSet (DirAddr 3),
    Receive regA,
81 Compute Equal regA reg0 regA,
    Branch regA (Rel (-3)),
83 TestAndSet (DirAddr 4),
    Receive regA,
85 Compute Equal regA reg0 regA,
    Branch regA (Rel (-3)),

```

```

87 TestAndSet (DirAddr 5),
   Receive regA,
89 Compute Equal regA reg0 regA,
   Branch regA (Rel (-3)),
91 TestAndSet (DirAddr 6),
   Receive regA,
93 Compute Equal regA reg0 regA,
   Branch regA (Rel (-3)),
95 ReadInstr (DirAddr 7),
   Receive regA,
97 Push regA,
   Pop regA,
99 WriteInstr regA numberIO,
   EndProg,
101 TestAndSet (DirAddr 1),
   Receive regA,
103 Compute Equal regA reg0 regA,
   Branch regA (Rel (-3)),
105 Load (ImmValue 2) regA,
   WriteInstr regA (DirAddr 1),
107 Load (ImmValue 100) regA,
   Push regA,
109 Pop regA,
   Store regA (DirAddr 1),
111 Load (DirAddr 1) regA,
   Push regA,
113 Load (ImmValue 0) regA,
   Push regA,
115 Pop regB,
   Pop regA,
117 Compute Gt regA regB regA,
   Push regA,
119 Pop regA,
   Branch regA (Rel (2)),
121 Jump (Rel (28)),
   Load (DirAddr 1) regA,
123 Push regA,
   Load (ImmValue 1) regA,
125 Push regA,
   Pop regB,
127 Pop regA,
   Compute Sub regA regB regA,
129 Push regA,
   Pop regA,
131 Store regA (DirAddr 1),
   TestAndSet (DirAddr 0),
133 Receive regA,
   Compute Equal regA reg0 regA,
135 Branch regA (Rel (-3)),

```

```

    ReadInstr (DirAddr 7),
137 Receive regA,
    Push regA,
139 Load (ImmValue 1) regA,
    Push regA,
141 Pop regB,
    Pop regA,
143 Compute Add regA regB regA,
    Push regA,
145 Pop regA,
    WriteInstr regA (DirAddr 7),
147 WriteInstr reg0 (DirAddr 0),
    Jump (Rel (-37)),
149 WriteInstr reg0 (DirAddr 1),
    EndProg,
151 TestAndSet (DirAddr 2),
    Receive regA,
153 Compute Equal regA reg0 regA,
    Branch regA (Rel (-3)),
155 Load (ImmValue 2) regA,
    WriteInstr regA (DirAddr 2),
157 Load (ImmValue 100) regA,
    Push regA,
159 Pop regA,
    Store regA (DirAddr 1),
161 Load (DirAddr 1) regA,
    Push regA,
163 Load (ImmValue 0) regA,
    Push regA,
165 Pop regB,
    Pop regA,
167 Compute Gt regA regB regA,
    Push regA,
169 Pop regA,
    Branch regA (Rel (2)),
171 Jump (Rel (28)),
    Load (DirAddr 1) regA,
173 Push regA,
    Load (ImmValue 1) regA,
175 Push regA,
    Pop regB,
177 Pop regA,
    Compute Sub regA regB regA,
179 Push regA,
    Pop regA,
181 Store regA (DirAddr 1),
    TestAndSet (DirAddr 0),
183 Receive regA,
    Compute Equal regA reg0 regA,

```



```

185 Branch  regA (Rel (-3)),
    ReadInstr  (DirAddr 7),
187 Receive  regA,
    Push  regA,
189 Load  (ImmValue 2) regA,
    Push  regA,
191 Pop  regB,
    Pop  regA,
193 Compute  Sub regA regB regA,
    Push  regA,
195 Pop  regA,
    WriteInstr  regA (DirAddr 7),
197 WriteInstr  reg0 (DirAddr 0),
    Jump  (Rel (-37)),
199 WriteInstr  reg0 (DirAddr 2),
    EndProg,
201 TestAndSet  (DirAddr 3),
    Receive  regA,
203 Compute  Equal regA reg0 regA,
    Branch  regA (Rel (-3)),
205 Load  (ImmValue 2) regA,
    WriteInstr  regA (DirAddr 3),
207 TestAndSet  (DirAddr 0),
    Receive  regA,
209 Compute  Equal regA reg0 regA,
    Branch  regA (Rel (-3)),
211 ReadInstr  (DirAddr 7),
    Receive  regA,
213 Push  regA,
    Load  (ImmValue 5) regA,
215 Push  regA,
    Pop  regB,
217 Pop  regA,
    Compute  Add regA regB regA,
219 Push  regA,
    Pop  regA,
221 WriteInstr  regA (DirAddr 7),
    WriteInstr  reg0 (DirAddr 0),
223 WriteInstr  reg0 (DirAddr 3),
    EndProg,
225 TestAndSet  (DirAddr 4),
    Receive  regA,
227 Compute  Equal regA reg0 regA,
    Branch  regA (Rel (-3)),
229 Load  (ImmValue 2) regA,
    WriteInstr  regA (DirAddr 4),
231 Load  (ImmValue 100) regA,
    Push  regA,
233 Pop  regA,

```

```

    Store  regA (DirAddr 1),
235 Load  (DirAddr 1) regA,
    Push  regA,
237 Load  (ImmValue 0) regA,
    Push  regA,
239 Pop   regB,
    Pop   regA,
241 Compute Gt regA regB regA,
    Push  regA,
243 Pop   regA,
    Branch regA (Rel (2)),
245 Jump  (Rel (28)),
    Load  (DirAddr 1) regA,
247 Push  regA,
    Load  (ImmValue 1) regA,
249 Push  regA,
    Pop   regB,
251 Pop   regA,
    Compute Sub regA regB regA,
253 Push  regA,
    Pop   regA,
255 Store  regA (DirAddr 1),
    TestAndSet (DirAddr 0),
257 Receive regA,
    Compute Equal regA reg0 regA,
259 Branch regA (Rel (-3)),
    ReadInstr (DirAddr 7),
261 Receive regA,
    Push  regA,
263 Load  (ImmValue 1) regA,
    Push  regA,
265 Pop   regB,
    Pop   regA,
267 Compute Sub regA regB regA,
    Push  regA,
269 Pop   regA,
    WriteInstr regA (DirAddr 7),
271 WriteInstr reg0 (DirAddr 0),
    Jump  (Rel (-37)),
273 WriteInstr reg0 (DirAddr 4),
    EndProg,
275 TestAndSet (DirAddr 5),
    Receive regA,
277 Compute Equal regA reg0 regA,
    Branch regA (Rel (-3)),
279 Load  (ImmValue 2) regA,
    WriteInstr regA (DirAddr 5),
281 Load  (ImmValue 100) regA,
    Push  regA,

```

```

283 Pop    regA,
    Store regA (DirAddr 1),
285 Load  (DirAddr 1) regA,
    Push regA,
287 Load  (ImmValue 0) regA,
    Push regA,
289 Pop    regB,
    Pop    regA,
291 Compute Gt regA regB regA,
    Push regA,
293 Pop    regA,
    Branch regA (Rel (2)),
295 Jump   (Rel (28)),
    Load  (DirAddr 1) regA,
297 Push   regA,
    Load  (ImmValue 1) regA,
299 Push   regA,
    Pop    regB,
301 Pop    regA,
    Compute Sub regA regB regA,
303 Push   regA,
    Pop    regA,
305 Store  regA (DirAddr 1),
    TestAndSet (DirAddr 0),
307 Receive regA,
    Compute Equal regA reg0 regA,
309 Branch regA (Rel (-3)),
    ReadInstr (DirAddr 7),
311 Receive regA,
    Push regA,
313 Load  (ImmValue 3) regA,
    Push regA,
315 Pop    regB,
    Pop    regA,
317 Compute Add regA regB regA,
    Push regA,
319 Pop    regA,
    WriteInstr regA (DirAddr 7),
321 WriteInstr reg0 (DirAddr 0),
    Jump (Rel (-37)),
323 WriteInstr reg0 (DirAddr 5),
    EndProg,
325 TestAndSet (DirAddr 6),
    Receive regA,
327 Compute Equal regA reg0 regA,
    Branch regA (Rel (-3)),
329 Load  (ImmValue 2) regA,
    WriteInstr regA (DirAddr 6),
331 TestAndSet (DirAddr 0),

```

```

    Receive  regA,
333 Compute  Equal regA reg0 regA,
    Branch  regA (Rel (-3)),
335 ReadInstr (DirAddr 7),
    Receive  regA,
337 Push    regA,
    Load    (ImmValue 78) regA,
339 Push    regA,
    Pop      regB,
341 Pop      regA,
    Compute  Add regA regB regA,
343 Push    regA,
    Pop      regA,
345 WriteInstr regA (DirAddr 7),
    WriteInstr reg0 (DirAddr 0),
347 WriteInstr reg0 (DirAddr 6),
    EndProg]
349
main = run[prog,prog,prog,prog,prog,prog,prog]

```

LISTING 31: Generated Sprockell Code

### 7.2.3 Output

```
[Sprockell 0 says 183]
```

LISTING 32: Output