Programming Paradigms Final Project: Building a Compiler in Haskell for the Sprockell

Group 26
Martijn Verkleij & Tim Kerkhoven

University of Twente
m.f.verkleij@student.utwente.nl, t.kerkhoven@student.utwente.nl
s1466895 s1375253

June 26, 2017

Contents

Chapter 1

Introduction

In the Programming Paradigms course, the final project is a combination of compiler construction, functional programming, and concurrent programming. It required the participants to build a compiler for a self-defined language to SprIL. The language had to support features, including basic concurrency.

The language group 26 designed for this project is called Simple Haskell Language (SHL), with file extension *.shl*. It supports all the required features, as well as some additional features. These additional features include: procedures, and call-by-reference. The entire compiler runs on uses Haskell, both the code-generation and front end. The Sprockell was slightly extended as well.

Chapter 2

Summary

This chapter will give a summary of the features of SHL.

Data types SHL supports two types: integers and booleans.

Simple expressions and variables SHL supports denotations for primitive values of types as well as operations for (in)equality for values of types. SHL is strongly typed and all variables are initialised upon declaration. It also supports scoping with variable shadowing. The following expressions are supported:

- Parentheses
- Assignment
- Operation (with ==, !=, <>, &&, ||, <=, >=, <, >, +, -, *)
- Unary operation (with !, -)
- Variable
- Integer value
- Boolean value

Basic statements SHL supports the following statements:

- Block
- Declaration
- If
- While
- Call
- Fork
- Join
- Print
- Expression

Concurrency SHL supports global variables, fork and join statements to implement concurrency.

Procedures SHL supports basic procedures with call-by-reference.

Chapter 3

Problems & Solutions

This chapter is a short discussion of some of the problems encountered during the project and their solutions.

Changing the Types

During the process of building the type checker and code generation, which was done in parallel, at certain times a change to the AST or Grammar was required. This resulted in a lot of trying to find every instance which had to be changed, and finding unused code during the final day of the project.

Time

If only the schedule made at the beginning of every project works out and no unexpected problems arise, this would not be a problem. Reality, unfortunately, has not been that kind. Many unscheduled delays and problems arose to delay the schedule more than expected. The solution for this is working evenings and parts of the night on the final days.

Concurrency

The problem with concurrency was mostly that the direction it was being taken changed several times. The addition of explicit locks was discussed and partly executed, then eventually reverted. The way global variables were handled was changed multiple times, as well as some other things. Until time was taken to really think it through, and determine what it should and should not do, and what it could and could not do.

Call-by-reference

Call-by-reference became a bit of a problem because of some design flaws. Because of the way our ARs work and the visibility of certain aspects of variables when passed as arguments, changes had to be made to the old design to what it currently is (see 5.3.3 (Call-by-reference)). These changes caused quite some work during their implementation.

Code Generation

During code generation, two big types of problems arose, of which the first one was offset management. SprIL does not know labels as ILOC does, and since our code was generated with a tree, many things as procedure addresses, retrun addresses and the like were not known on the first pass. Furthermore, fixing bugs in logic often messes up offsets in loops, creating their own set of incomprehensible issues.

The second issue was memory management. Not knowing beforehand what you need in an AR does not help when debugging issues with values being read as addresses and the other way around. But by far the most annoying issue I've had was a bug in Sprockell where writes to out-of-bounds addresses to global memory were not protected. Instead, they extended the list, resulting in all variables moving one step. Good luck debugging when one cannot trust the integrity of memory.

Chapter 4

Sprockell Extensions

Shared Memory Bug

A bug in shared memory has been resolved. Whenever a value was written to shared memory to an out of bounds index, shared memory would be extended with that value. The list would be longer by one with the new value at either the first or last position in the list, depending on whether the index was too low or too high.

The extension causes an error to be thrown instead. This allows for easier debugging of code generation.

PrintOut Instruction

A print instruction, `PrintOut reg`, was added to print registers to stdout. The instruction uses `trace` from the package `Debug.Trace` to do this.

ComputeI Instruction

A compute with immediate value, `ComputeI operator reg value reg`. This instruction takes a register and a value to do a computation with the operator, which is then written to the second register. It is mostly used to calculate ARP offsets.

This instruction was inspired by the *iloc* instructions using an immediate value (eg. `AddI`, `SubI`.

Incr4 & Decr4

Two `Operators` were added to the alu operators: Incr4 and Decr4. They were originally added to more easily do the steps of four to calculate offsets, before `ComputeI` was added. They are currently unused.

Chapter 5

Detailed Language Description

This chapter will describe every feature of SHL in detail: providing a basic description; information on the syntax with at least one example; usage information along with restrictions; a description of its effects and execution; and some general information on the generated code.

Program

Syntax

```
[GLOBAL]...[PROCEDURE]...[STATEMENTS]...
```

GLOBAL Global variable declarations as defined in 5.0.2 (Global)
PROCEDURE Procedures as defined in 5.0.3 (Procedure)
STATEMENTS Statements as defined in 5.1 (Statements)

Example

```
global int number = 5;

procedure eq(int num1, int num2, bool out) {
    if ((num1 == num2)) {
        out = true;
    } else {
        out = false;
    }
}

int otherNumber = 6;
bool out;
eq(number, otherNumber, out);
print(out);
```

Usage

All files must follow the program syntax, and may only contain a single program.

Semantics

A program is a collection of code which can be used to create an executable set of instructions. It is the root node of the Abstract Syntax Tree.

Code Generation

Any program is generally built up as follows:

- **Thread control code**
  - **Thread Control Loop**
    All extra threads loop here, waiting to accept fork calls.
  - **PreCall forked procedures**
    Once a sprockell accepts a fork call, it reads the AR from global memory, and starts execution.
  - **PostCall forked procedures**
    Once an auxilary sprockell finishes the procedure, this code handles cleanup.
- **Procedures 5.0.3**
- **Global declarations 5.0.2**
- **Main code**
- **Post-program code**
  Stops auxilary threads.

Global

Syntax

`global <TYPE> <ID> [= <EXPRESSION>] ;`

TYPE A type as defined in 5.3.1 (Types)
ID A string as defined by 5.2.3 (Variable)
EXPRESSION An expression as defined in 5.2 (Expressions)

Examples

```
global bool flag = true;
global int number;
```

Usage

Used to declare global variables and an optional assignment. The type of the expression must match the type of the global variable.

All global variables MUST be defined at the top of the program, even before procedures.

The id of a global variable is unique in the whole program. No other variable of procedure may use the same id. They can therefore not be shadowed.

Semantics

The global variable declaration reserves a space in shared memory and writes a value to it. All global variables are initialized to the default value (see 5.2.4 (Integer) and 5.2.5 (Boolean)) if no value is explicitly assigned.

Global variables are reachable from anywhere below its declaration in the code, in any thread. Beware, however, that the variable can be written to from any thread as well, and using the shared memory is significantly slower than using the local memory.

Globals that are passed as arguments to a procedure have their own intricacies, see **??** for more information.

Code Generation

All globals are saved in global memory. Writes and reads to globals are done atomically, but assignments are not. This avoids data races, but to ensure atomicity on the whole assignment, one must implement his own mutual exclusion.

Procedure

Syntax

```
procedure <ID> ( [<TYPE> <VAR>] [, <TYPE> <VAR>]...)  <STATEMENT>...
```

ID A string as defined in 5.2.3 (Variable)
TYPE A type as defined in 5.3.1 (Types)
VAR A variable as defined in 5.2.3 (Variable)
STATEMENT A statement as defined in 5.1 (Statements)

Examples

```
procedure empty() print(0);
procedure other(int num, bool flip) {
    while ((num > 0)) {
        num = --num;
        flip = !flip;
    }
    print(num, flip);
}
```

Usage

Used to declare a procedure. Because call-by-reference (see 5.3.3 (Call-by-reference)) is used, a variable passed as an argument can be used to write resulting values. One could also use the global variables (see 5.0.2 (Global)), as they are accessible from everywhere.

The id of a procedure is unique in the whole program. No other variable of procedure may use the same id.

Semantics

A procedure is section of code that can be executed from anywhere, using a call or fork statement (see 5.1.6 (Call) and 5.1.4 (Fork)) and passing the appropriate number of arguments to it.

Code Generation

Procedure code has the following structure:

- **PostCall code**
  Copies all arguments into the local data area for use.

- **Procedure's statements**
- **PreReturn code**
  The final result of all arguments is read, and if they are global or local variables, saved to the appropriate location.

Statements

Declaration

Syntax

```
<TYPE> <ID> [= <EXPRESSION>] ;
```

TYPE A type as defined in 5.3.1 (Types)
ID A string as defined in 5.2.3 (Variable)
EXPRESSION An expression as defined in 5.2 (Expressions)

Examples

```
int number = (1+1);
bool flag;
```

Usage

Used to declare local variables and an optional assignment. The type of the expression must match the type of the variable.

The id of a variable is unique in the scope it is defined in. No other variable in that scope may use the same id.

Semantics

The variable declaration writes the value of the variable to the Local Data Area of where it was (most recently) defined. All variables are initialized to the default value (see 5.2.4 (Integer) and 5.2.5 (Boolean)) if no value is explicitly assigned.

Code Generation

A declaration evaluates the expression behind it before assigning it to the variable. The variable's value is saved in the appropriate Local Data Area, by following the scopes upward until the right scope is reached, after which it is saved with an offset.

If

Syntax

```
if ( <EXPRESSION> ) <STATEMENT> [else <STATEMENT>]
```

EXPRESSION An expression as defined in 5.2 (Expressions)
STATEMENT A statement as defined in 5.1 (Statements)

Examples

```
if (flag) {
    // do something
}
if (flag) print(flag); else {
    // do something
}
```

Usage

Execute a section of code based on an expression. The type of this expression must be a boolean.

Semantics

If the expression evaluates to *true*, execute the first statement. If it evaluates to *false*, execute the code after the first statement, which can be either the second statement or the code that comes after the if statement.

Code Generation

First the expression will be evaluated, after which a branch jumps to the else block upon a false result, or to the next expression if no else block is present. After the if-block, a jump to after the else block is placed when an else block is present.

While

Syntax

```
while ( <EXPRESSION> ) <STATEMENT>
```

EXPRESSION An expression as defined in 5.2 (Expressions)
STATEMENT A statement as defined in 5.1 (Statements)

Examples

```
while (flag) {
    // do something
}
```

Usage

Execute a section of code while the expression is *true*. The type of this expression must be a boolean.

Semantics

If the expression evaluates to *true*, execute the statement. Repeat this for as long as the expression keeps evaluating to *true*.

Code Generation

As long as the expression is true, the code in the following block will be executed. The expression is re-evaluated after each execution of the block.

Fork

Syntax

```
fork <ID> ( [<EXPRESSION> [, <EXPRESSION>]...]  )  ;
```

ID A string as defined by 5.2.3 (Variable)
EXPRESSION An expression as defined by 5.2 (Expressions)

Examples

```
fork proc0();
fork proc1(flag);
fork proc2(5, flag = true);
```

Usage

Run a procedure, which must have been declared somewhere, on a separate thread. The expression types must match the types defined during the procedure declaration (see 5.0.3 (Procedure)).

Semantics

Writes the argument to shared memory and tells the thread pool to start parallel execution of the procedure.

Beware, if more procedures are given to the thread pool than there are threads, fork may have to wait for a thread to finish its work before continuing execution. In the case where programs with forks are executed on only one sprockell, it will deadlock.

Code Generation

A fork call must start a procedure in another thread. To accomplish this, the first 30 addresses in global memory are reserved for this purpose. A further few addresses, namely as much as there atre threads, are reserved for an occupation bit. In practise this means that procedures with more than 7 arguments cannot be forked without memory corrupting.

The addresses are reserved as follows:

- **End flag**
  Set when the auxilary threads may cease execution. Each auxilary sprockell checks this record before attempting to read an AR from shared memory
- **Wr flag**
  Used to set the AR space as occupied by an AR that may be executed. An auxilary threads sets it to 0 when it has read the AR.
- **Rd flag**
  Read-protects the AR as it is being written, and to exclude other sprockells from handling the request.
- **Jump index**
  line number of the procedure to execute.

- **Argument count**
  Number of arguments that follow.
- **Arguments...**
  - **Value**
  - **Local address**
    The address to write the result back to, if the argument has one. For example, an expression has none.
  - **Global address**
    Idem.

Join

Syntax

```
join ;
```

Example

```
join;
```

Usage

Ensures all auxilary threads have finished execution before continuing.

Semantics

Blocks execution of the main thread until all other threads have finished their work. May only be called in code that is not executed in auxilary threads, for example by calling a procedure with fork that contains a join statement.

Code Generation

A join statement iterates over the occupation bits of all auxilary threads. Only if they are all zero, this statement may end. It will therefore throw an error when called from an auxilary thread, as can happen when used in a procedure.

Call

Syntax

```
<ID> ( [<EXPRESSION> [, <EXPRESSION>]...] ) ;
```

ID A string as defined by 5.2.3 (Variable)
EXPRESSION An expression as defined by 5.2 (Expressions)

Examples

```
proc0();
proc1(flag);
proc2(5, flag = true);
```

Usage

Execute the called procedure sequentially, which must have been declared. The expressions must have the same types as the procedure's as defined in its declaration (see 5.0.3 (Procedure))

Semantics

Go to the procedure code and execute the procedure with the expressions, then return to the call.

Code Generation

Calls a procedure sequentially. Any separate variables that are given as an argument are handled call-by-reference. Sets up the AR before jumping to the procedure.

Expression

Syntax

```
<EXPRESSION> ;
```

EXPRESSION An expression as defined in 5.2 (Expressions)

Examples

```
a = (5 + (--6));
true;
-++++++++++a;
```

Usage

Allows expressions to be executed as statements, mostly for the purpose of enabling an assignment (see 5.2.2 (Assignment)) as a statement, since an assignment is an expression.

Semantics

Execute the expression, this generally has no effects, except for an assignment.

Code Generation

As expressions have a result on stack, one must pop this value if the expression is defined as a statement. Otherwise the stack would fill up.

Block

Syntax

```
{ [STATEMENT]...}
```

STATEMENT A statement as defined in 5.1 (Statements)

Example

```
{
    int i = 0;
    {
        i = ++i;
        {
            int i = 5;
        }
        print((i == 1));
    }
    print((i == 1));
}
```

Usage

A block is a single statement that contains zero or more statements. It is mostly used within procedures and statements to executes more than one statement.

Semantics

A block opens a new scope, then executes the code within. When exiting a block, the scope is closed.

Code Generation

As any block opens a new scope, a new mini-AR is created for each scope. It only contains a pointer to it's parent's AR and all variables declared in this scope in a Local Data Area.

Print

Syntax

```
print ( <EXPRESSION> [, <EXPRESSION>]...)  ;
```

EXPRESSION An expression as defined in 5.2 (Expressions)

Examples

```
print(a);
print(true, 5, 1983);
print(a = ++a, ((11 - 2) * a));
```

Usage

Prints values of evaluated expressions to the console.

Semantics

Evaluates the expressions and prints the values as they appear in memory, meaning a boolean is represented as either a zero (*false*) or a one (*true*).

Code Generation

As SprIL does not have a print instruction, a new one was created and added to the Sprockell source code. See also chapter 4. This statement simply evaluates all expressions in its arguments and prints them one per line.

Expressions

Parentheses

Syntax

`( <EXPRESSION> )`

EXPRESSION An expression as defined in 5.2 (Expressions)

Example

```
a = -(-(----a); // the same as: a = (-1) * (-1) * (a - 2);
```

Usage

Parentheses are used to enforce which operator is used (see the example above). It can also be used to enforce the order in which an expression is evaluated, but since this already explicitly happens (see 5.2.6 (Operation)) it should not be necessary to use a parentheses expression for it.

Semantics

Everything between the parentheses is evaluated and the value is returned as the result of this expression.

Code Generation

Required around any binary expression and optional around unary expressions to, as in the example, differentiate between the negation and decrement operator. The result of the inner expression is pushed to stack.

Assignment

Syntax

`<ID> = <EXPRESSION>`

ID A string as defined in 5.2.3 (Variable)

Examples

```
a = 5;
b = (c <> (d && e));
```

Usage

Used to assign a value, in the form of an expression, to a variable. The variable must have been declared beforehand, and may be either global or local.

The type of the expression must match the type of the variable.

Semantics

Assignment evaluates the expression and writes it to the address of the variable.

Code Generation

First, the expression will be evaluated. The result is assigned to the variable, may it be in local or global memory. These are resolved by AR traversal and lookup in a static table respectively. The result is pushed to stack.

Variable

Syntax

`<ID>`

ID A string, starting with a letter, which may use any alphanumerical character in addition to the following characters: ~‘’”@#$\.?:_

Examples

```
a
a@__b"42"\#1337’
```

Usage

A variable must be declared (see 5.1.1 (Declaration)) before use. It has a type which is determined upon declaration.

Semantics

Evaluation of a variable returns its value.

Code Generation

The variable’s value is looked up in the AR stack or loaded from global memory and pushed to stack.

Integer

Syntax

`<INTEGER>`

INTEGER An integer string

Examples

```
42
1337
0000004201337
```

Usage

Takes the value of the integer, removes leading zeros.

Semantics

Upon evaluation it returns its integer value.

Code Generation

Its value is pushed to stack.

Boolean

Syntax

`<BOOLEAN>`

BOOLEAN Where a boolean is either "`true`" or "`false`"

Examples

```
true
false
```

Usage

Takes the value of the boolean (either one or zero) and returns it.

Semantics

Upon evaluation, return the corresponding binary representation of the boolean, where *false* equals zero and *true* equals one.

Code Generation

Its value is pushed to stack.

Operation

Syntax

`( <EXPRESSION> <OPERATOR> <EXPRESSION> )`

EXPRESSION An expression as defined in 5.2 (Expressions)

OPERATOR One of the following operators: `==`, `!=`, `&&`, `||`, `<>`, `<=`, `>=`, `<`, `>`, `+`, `-`, `*` (see 5.3.2 (Operators))

Examples

```
(true <> b)
((a + b) == (c + d))
```

Usage

Apply operator on two expressions. Both expressions must be of the same type, which must also match one of the types supported by the operator.

Semantics

After both expressions have been evaluated, the operation is evaluated and its result will be returned.

Code Generation

It's left- and right hand side are evaluated, and its results are used as arguments to the operation. The result of the operation is pushed to stack.

Unary Operation

Syntax

```
<OPERATOR> <EXPRESSION>
```

OPERATOR One of the following operators: `-`, `++`, `-`, `!` (see 5.3.2 (Operators))

Examples

```
!b
-(--a)
---a // is the same as: --(-a)
```

Usage

Apply operator on the expression. The expression type must match one of the types supported by the operator.

Semantics

After the expression has been evaluated, the operation is evaluated and its result will be returned.

Code Generation

It's right hand side is evaluated, fed to the operator and its result is pushed to stack.

Other Features

Types

Syntax

<TYPE>

TYPE Either `int` or `bool`

Operators

Syntax

<OPERATOR>

OPERATOR One of the following: `==`, `!=`, `&&`, `||`, `<>`, `<=`, `>=`, `<`, `>`, `+`, `-`, `*`, `-`, `++`, `!`

Usage

OPERATOR Operation: `supported types` → `return type`
== equals: `int, bool` → `bool`
!= not equals: `int, bool` → `bool`
&& and: `bool` → `bool`
|| or: `bool` → `bool`
<> xor: `bool` → `bool`
<= lesser than or equals: `int` → `bool`
>= greater than or equals: `int` → `bool`
< lesser than: `int` → `bool`
> greater than: `int` → `bool`
+ add: `int` → `int`
- subtract: `int` → `int`
***** multiply: `int` → `int`
– decrement: `int` → `int`
++ increment: `int` → `int`
! not: `bool` → `bool`

Beware that using decrement and increment on a variable does not assign the new value to the variable as some other languages might do.

Call-by-reference

SHL uses call-by-reference on calls to procedures. This is almost essential to make procedures useful, as the only other option to communicate values between a procedure and its caller would be through global memory.

It is implemented by passing an optional return address in shared and local memory to write the result back to after the procedure is complete. In theory an argument could be written to both at the same time, as a memory slot is used for each (and makes them distinguishable). These are used to write the results back to their memory locations before returning to the caller.

To make use of it, pass any variable as a naked argument to a call to any procedure in the program. Expressions like `(i+3)` or `i = 12` do NOT work, as the argument will only be seen as an expression and will therefore not have a memory location associated with it. Keep in mind that any global variables passed as an argument will be written to global memory upon completion of the procedure. This does not influence any assignment inside the procedure, these are still done as they are evaluated.

Error Handling

The SHL compiler does not support proper exception handling, but does throw errors of varying usefulness. During the tokenization phase, the only error thrown is an illegal character error.

During the parsing phase, the only error which might be thrown is a token list not fully parsed error, indicating the grammar cannot parse the token list.

The checker phase thrown different kinds of errors, all related to context constraints, they generally indicate the function which throws the error as well as printing some of the responsible data.

The code generation and runtime phases thrown the following kinds of errors:

Upon code generation, an error is thrown when a user attempts to compile code with a fork statement with the intention of using only one thread.

During runtime, when a join statement is executed by a thread other than the main thread, it prints an error code and ceases operation. This may turn out unhelpful, as the other threads are not notified and useful operation ceases. Since the occupation bit cannot be unset at this stage, any subsequent join statement will deadlock. Any values it would have returned are lost.

Chapter 6

Description of the Software

The compiler consists of a number of haskell files, and some additional files. This chapter will go over the functions of each of those files.

ASTBuilder.hs

The purpose of the ASTBuilder is to build an Abstract Syntax Tree using a parsetree. The ASTBuilder also contains the functions to convert an AST to a RoseTree with or without debug information.

BasicFunctions.hs

Part of the Sprockell. Any changes in the Sprockell code have been annotated with `PP26:....`

Checker.hs

The checker does type checking on an AST and adds information about scopes (symboltable) to it. It works in two passes, first collecting information about global variables and procedures, then checking for all context constraints.

CodeGen.hs

CodeGen takes a checked AST and generates a program of SprIL instructions, runnable on Sprockells.

Constants.hs

Constants stores constant values used in the code generation. Simply aliases for memory addresses and offsets.

FP_ParserGen.hs

Parser generator supplied by the course.

Grammar.hs

Grammar contains the grammar used in the compiler.

HardwareTypes.hs

Part of the Sprockell. Any changes in the Sprockell code have been annotated with `PP26:`....

Main.hs

Main file, used for compilation and execution of SHL programs. Read the README.md for information on how to use it.

README.md

Constains some information about the project in general (eg. the Trello board) and instructions on how to use the compiler.

Simulation.hs

Part of the Sprockell. Any changes in the Sprockell code have been annotated with `PP26:`....

Sprockell.hs

Part of the Sprockell. Any changes in the Sprockell code have been annotated with `PP26:`....

System.hs

Part of the Sprockell. Any changes in the Sprockell code have been annotated with `PP26:`....

Test.hs

Used for internal testing, contains functions to print and write debug information, show ASTs with and without debug information, show the parse tree, and show the token list.

Tokenizer.hs

Tokenizer tokenises a string into a list of tokens.

Types.hs

Contains all the Haskell types used during compilation, including Alphabet, AST, and checking/scope types.

Chapter 7

Test Plan & Results

Implemented Tests

Following is a list of all the test files that have been used to test the compiler, and a short description of their purpose.

syntax1 Tests incorrect program syntax
syntax2 Tests incorrect procedure syntax
syntax3 Tests incorrect variable syntax
syntax4 Tests incorrect if syntax
syntax5 Tests incorrect expression syntax
wrong_type Tests whether a wrong type is detected
not_declared Tests whether a variable which has not been declared is detected
cyclic_recursion Tests for correct cyclic recursion
deep_expression Tests for correct evaluation of nested expressions
fib Tests for correct evaluation of a Fibonacci procedure
if Tests a correct simple if statement
ifelse Tests a correct simple if-else statement with some additional scoping
infinite_busy_loop Tests behaviour in an empty infinite loop
infinite_loop Tests behaviour in an infinite loop with some operation in it. Also tests integer overflows, which are not detected.
nested_procedures Tests for correct evaluation of nested procedures
recursion Tests for correct recursion
while Tests a simple correct while statement
call_by_reference Tests for correct multi-threaded call-by-reference
blocks Tests for correct handling of scopes
simple_proc Tests a simple correct procedure
banking Tests a concurrent banking application
peterson Tests for correct evaluation of Peterson's algorithm
simple_concurrency Tests a simple correct concurrent program
multiple_globals Tests behaviour of concurrent printing of global variables
join_test Tests whether join behaviour is correct

The source code, generated code and results of all tests have been documented in `testreport.pdf`.

Test Plan

The testing has been roughly divided into three cases: syntax, context constraints and semantics. For the first two phases most of the testing of correct code occurs during the semantic testing and as informal testing during the building of those parts of the compiler. Some additional tests have been written to more formally test the incorrect code.

The shape of the parse tree and Abstract Syntax Tree have been extensively observed and checked during the building of the checking part of the compiler. This has mostly been done by slightly tweaking a program a multitude of times, to produce all intended shapes of the tree and attempting to produce unintended shapes, and building the trees. This part of the testing, as well as the previous part, have not been documented very well, and might therefore appear somewhat lacking compared to the semantic testing.

The semantics, or run-time, testing has been given the most time and effort, and checks for correctness of code generation and intended behaviour. Since very little run-time error are thrown (see 5.3.4 (Error Handling)), there are only a few tests of incorrect code, or code producing unintended effects.

How To Run a Test

To run a test, simply follow the `README.md`, using the following path: `test/<fileName>`, where `fileName` is one of the tests described above. Remember that for a concurrent program, which is any program that uses at least one fork statement, multiple Sprockells have to be used.

Chapter 8

Personal Evaluation

Martijn

The project is actually quite fun to do once you have a better understanding of the time needed to deliver a working product. It helps too that stress factors like an FP-resit or falling behind on excercises are just not there. Even though I had to take the CP resit, most other grades were excellent and help in achieving a good average grade. The module as a whole is a lot of work, but is a lot more doable as a second-timer. Grades are better, workload is better.

I am a great fan of the new functional programming excercises in week 4 and 5, they were a change in pace for us, but gave us the confidence to do this project fully in Haskell, which turned out to be a great choice in terms of motivation and having the overview. With ANTLR I feel like talking to a black box (even though they do roughly the same). The new Logic programming excercises and project are a great addition too, as they show more of the power of logic programming than last year's did.

That's all. As far as I can see we did our best. Let's hope it is enough.

Tim

I personally am quite satisfied with the language and compiler we built. It's a lot better than last year's, and think it should be quite usable this time. We decided to focus mostly on a simple set of things we wanted, and I believe we mostly delivered on those things. I know call-by-reference is somewhat iffy, it uses an internal value and only writes it back to the references variable at the end of the procedure, but that should be fine for sequential programs. For parallel programs, just stick with Peterson's algorithm if you need something to be mutually exclusive, which isn't just a single write.

I quite like the Programming Paradigms module, with the exception of concurrent programming. I find building a compiler quite interesting, and I really like function programming (we actually scored a 10 for the project), yet somehow concurrent programming just isn't really my thing. It just doesn't really capture my attention, and that shows in the results. The first test I just left early because it wasn't going to work out anyway, and I had to miss the second one because I got ill the day before, so I'm currently trying to get a third attempt.

I do have to say the workload of this module is still quite high, but mostly doable. It mostly just requires some good time management and work ethics, in which I occasionally lack a bit. The way the module is structured, mostly the CC exercises, means that occasionally you have to wait quite a bit for a student assistant to be available, but I found this much less of an issue than last year.

I cannot comment on the lectures this year, as I didn't go to most of them, seeing as I had been there last year. In retrospect it would have been a good idea to go to the CP lectures, but I can't do much about that now.

I finish this up, I enjoyed this module, with the exception of CP. There were a few problems along the way, but they quickly dealt with and weren't really an issue for me in the end. Every module can improve in some ways, but I would have to say I'm quite content with this one.

Appendices

Appendix A

Grammar Specification

```haskell
grammar :: Grammar
grammar nt = case nt of

    -- Program
    Program ->  [[ (*:) [Global], (*:) [Proc], (*:) [Stat] ]]

    -- Globals
    Global  ->  [[ global, Type, Var, (?:) [ass, Expr], eol ]]

    -- Procedures
    Proc    ->  [[ procedure, Pid, lPar, (?:) [Type, Var, (*:) [comma, Type, Var]], rPar, Stat ]]

    -- Statements
    Stat    ->  [[ Type, Var, (?:) [ass, Expr], eol ]                                   -- declaration
                ,[ ifStr, lPar, Expr, rPar, Stat, (?:) [elseStr, Stat] ]                -- if
                ,[ while, lPar, Expr, rPar, Stat ]                                      -- while
                ,[ fork, Pid, lPar, (?:) [Expr, (*:) [comma, Expr]], rPar, eol ]        -- fork
                ,[ join, eol ]                                                          -- join
                ,[ Pid, lPar, (?:) [Expr, (*:) [comma, Expr]], rPar, eol ]              -- call
                ,[ Expr, eol ]                                                          -- expression
                ,[ lBrace, (*:) [Stat], rBrace ]                                        -- block
                ,[ printStr, lPar, Expr, (*:) [comma, Expr], rPar, eol ]]               -- print

    -- Expressions
    Expr    ->  [[ lPar, Expr, rPar ]                -- parentheses
                ,[ Var, ass, Expr ]                  -- assignment
                ,[ Var ]                             -- variable
                ,[ IntType ]                         -- integer
                ,[ BoolType ]                        -- boolean
                ,[ lPar, Expr, Op, Expr, rPar ]      -- operation
                ,[ Unary, Expr ]]                    -- unary operation

    -- Other
    Type    ->  [[ typeStr ]]   -- type
```

```
    Var     ->  [[ var ]]       -- variable

    Pid     ->  [[ Var ]]       -- procedure identifier

    IntType ->  [[ intType ]]   -- number

    BoolType->  [[ boolType ]]  -- boolean

    Op      ->  [[ op ]]        -- operator

    Unary   ->  [[ Op ]]        -- unary operator


-- shorthand names can be handy, such as:
lPar        = Symbol "("            -- Terminals WILL be shown in the parse tree
rPar        = Symbol ")"            -- Symbols WILL NOT be shown in the parse tree
lBrace      = Terminal "{"
rBrace      = Symbol "}"
procedure   = Symbol "procedure"
ifStr       = Terminal "if"
elseStr     = Terminal "else"
while       = Terminal "while"
ass         = Terminal "="
fork        = Terminal "fork"
join        = Terminal "join"
global      = Symbol "global"
printStr    = Terminal "print"


eol         = Symbol ";"
comma       = Symbol ","

var         = SyntCat Var
```

```
intType     = SyntCat IntType
boolType    = SyntCat BoolType
op          = SyntCat Op
unary       = SyntCat Unary
typeStr     = SyntCat Type
```

Appendix B

Extended Test Program

The extended test program shown here is Peterson's algorithm. It shows how, using the available methods for concurrency, two thread using the same variable have mutually exclusive access to it.

Peterson's Algorithm Test

```
global bool flag_0 = false;
global bool flag_1 = false;
global int turn = 0;
global int i = 0;

procedure p_0() {
    flag_0 = true;
    turn = 1;
    while ((flag_1 && (turn == 1))) {
        // wait
    }
    // begin critical section
    int j = 5;
    while ((j > 0)) {
        i = ++i;
        j = --j;
    }
    // end critical section
    flag_0 = false;
}

procedure p_1() {
    flag_1 = true;
    turn = 0;
    while ((flag_0 && (turn == 0))) {
        // wait
    }
    // begin critical section
```

```
    int j = 5;
    while ((j > 0)) {
        i = --i;
        j = --j;
    }
    // end critical section
    flag_1 = false;
}

procedure test1(int j) {
    while ((j > 0)) {
        fork p_0();
        fork p_1();
        join;
        print(i);

        fork p_1();
        fork p_0();
        join;
        print(i);

        j = --j;
    }
}

test1(10);
```

Generated Code

```
Compute Equal 1 0 6
Branch 6 (Rel 2)
Jump (Rel 7)
TestAndSet (DirAddr 2)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
Load (ImmValue 0) 7
Jump (Rel 630)
ReadInstr (DirAddr 0)
Receive 3
Compute Equal 3 0 6
Branch 6 (Rel 2)
EndProg
TestAndSet (DirAddr 2)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-8))
ComputeI Add 1 30 3
```

```
TestAndSet (IndAddr 3)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
ReadInstr (DirAddr 3)
Receive 3
Push 3
ComputeI Add 7 1 4
ReadInstr (DirAddr 4)
Receive 5
Load (ImmValue 5) 2
Compute Equal 5 0 6
Branch 6 (Rel 18)
ReadInstr (IndAddr 2)
Receive 3
Store 3 (IndAddr 4)
Compute Incr 2 0 2
Compute Incr 4 0 4
ReadInstr (IndAddr 2)
Receive 3
Store 3 (IndAddr 4)
Compute Incr 2 0 2
Compute Incr 4 0 4
ReadInstr (IndAddr 2)
Receive 3
Store 3 (IndAddr 4)
Compute Incr 2 0 2
Compute Incr 4 0 4
Compute Decr 5 0 5
Jump (Rel (-18))
Load (ImmValue 57) 5
Store 5 (IndAddr 4)
Compute Incr 4 0 4
Store 7 (IndAddr 4)
Compute Add 4 0 7
Pop 2
WriteInstr 0 (DirAddr 1)
Jump (Ind 2)
ComputeI Add 1 30 3
WriteInstr 0 (IndAddr 3)
Jump (Abs 9)
Load (ImmValue 1) 2
Compute Sub 7 2 2
Load (ImmValue 1) 5
ComputeI Gt 5 0 6
Branch 6 (Rel 7)
Load (IndAddr 2) 3
Compute Add 7 5 6
```

```
Store 3 (IndAddr 6)
Compute Incr 5 0 5
ComputeI Add 2 3 2
Jump (Rel (-7))
Compute Add 7 0 4
ComputeI Add 4 1 4
Store 7 (IndAddr 4)
Compute Add 4 0 7
Load (ImmValue 1) 6
Push 6
Load (ImmValue 33) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 34) 4
Pop 6
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Pop 0
Load (ImmValue 1) 6
Push 6
Load (ImmValue 39) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 40) 4
Pop 6
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Pop 0
Load (ImmValue 35) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 36) 4
ReadInstr (IndAddr 4)
Receive 5
Push 5
WriteInstr 0 (IndAddr 2)
Load (ImmValue 39) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 40) 4
```

```
ReadInstr (IndAddr 4)
Receive 5
Push 5
WriteInstr 0 (IndAddr 2)
Load (ImmValue 1) 6
Push 6
Pop 3
Pop 2
Compute Equal 2 3 4
Push 4
Pop 3
Pop 2
Compute And 2 3 4
Push 4
Pop 6
ComputeI Xor 6 1 6
Branch 6 (Rel 7)
Compute Add 7 0 4
ComputeI Add 4 1 4
Store 7 (IndAddr 4)
Compute Add 4 0 7
Load (IndAddr 7) 7
Jump (Rel (-38))
Load (ImmValue 5) 6
Push 6
Compute Add 7 0 6
ComputeI Add 6 1 6
Pop 5
Store 5 (IndAddr 6)
Compute Add 7 0 6
ComputeI Add 6 1 6
Load (IndAddr 6) 5
Push 5
Load (ImmValue 0) 6
Push 6
Pop 3
Pop 2
Compute Gt 2 3 4
Push 4
Pop 6
ComputeI Xor 6 1 6
Branch 6 (Rel 45)
Compute Add 7 0 4
ComputeI Add 4 2 4
Store 7 (IndAddr 4)
Compute Add 4 0 7
Load (ImmValue 37) 2
TestAndSet (IndAddr 2)
```

```
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 38) 4
ReadInstr (IndAddr 4)
Receive 5
Push 5
WriteInstr 0 (IndAddr 2)
Pop 2
Compute Incr 2 0 4
Push 4
Load (ImmValue 37) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 38) 4
Pop 6
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Pop 0
Compute Add 7 0 6
Load (IndAddr 6) 6
ComputeI Add 6 1 6
Load (IndAddr 6) 5
Push 5
Pop 2
Compute Decr 2 0 4
Push 4
Compute Add 7 0 6
Load (IndAddr 6) 6
ComputeI Add 6 1 6
Pop 2
Store 2 (IndAddr 6)
Push 2
Pop 0
Load (IndAddr 7) 7
Jump (Rel (-56))
Load (ImmValue 0) 6
Push 6
Load (ImmValue 33) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 34) 4
Pop 6
WriteInstr 6 (IndAddr 4)
```

```
WriteInstr 0 (IndAddr 2)
Pop 0
Load (IndAddr 7) 7
Load (ImmValue 0) 2
Compute Sub 7 2 2
ComputeI Add 0 1 5
ComputeI Gt 5 0 6
Branch 6 (Rel 23)
Compute Add 7 5 6
Load (IndAddr 6) 4
Load (IndAddr 2) 3
Compute Lt 3 0 6
Branch 6 (Rel 2)
Store 4 (IndAddr 3)
Compute Incr 2 0 2
Load (IndAddr 2) 3
Compute Lt 3 0 6
Branch 6 (Rel 10)
Compute Add 3 0 6
TestAndSet (IndAddr 6)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-4))
ComputeI Add 3 1 3
WriteInstr 4 (IndAddr 3)
ComputeI Sub 3 1 3
WriteInstr 0 (IndAddr 3)
Compute Incr 5 0 5
ComputeI Add 2 2 2
Jump (Rel (-23))
Compute Decr 7 0 2
Load (IndAddr 2) 6
Load (IndAddr 7) 7
Jump (Ind 6)
Load (ImmValue 1) 2
Compute Sub 7 2 2
Load (ImmValue 1) 5
ComputeI Gt 5 0 6
Branch 6 (Rel 7)
Load (IndAddr 2) 3
Compute Add 7 5 6
Store 3 (IndAddr 6)
Compute Incr 5 0 5
ComputeI Add 2 3 2
Jump (Rel (-7))
Compute Add 7 0 4
ComputeI Add 4 1 4
Store 7 (IndAddr 4)
```

```
Compute Add 4 0 7
Load (ImmValue 1) 6
Push 6
Load (ImmValue 35) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 36) 4
Pop 6
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Pop 0
Load (ImmValue 0) 6
Push 6
Load (ImmValue 39) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 40) 4
Pop 6
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Pop 0
Load (ImmValue 33) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 34) 4
ReadInstr (IndAddr 4)
Receive 5
Push 5
WriteInstr 0 (IndAddr 2)
Load (ImmValue 39) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 40) 4
ReadInstr (IndAddr 4)
Receive 5
Push 5
WriteInstr 0 (IndAddr 2)
Load (ImmValue 0) 6
Push 6
Pop 3
```

```
Pop 2
Compute Equal 2 3 4
Push 4
Pop 3
Pop 2
Compute And 2 3 4
Push 4
Pop 6
ComputeI Xor 6 1 6
Branch 6 (Rel 7)
Compute Add 7 0 4
ComputeI Add 4 1 4
Store 7 (IndAddr 4)
Compute Add 4 0 7
Load (IndAddr 7) 7
Jump (Rel (-38))
Load (ImmValue 5) 6
Push 6
Compute Add 7 0 6
ComputeI Add 6 1 6
Pop 5
Store 5 (IndAddr 6)
Compute Add 7 0 6
ComputeI Add 6 1 6
Load (IndAddr 6) 5
Push 5
Load (ImmValue 0) 6
Push 6
Pop 3
Pop 2
Compute Gt 2 3 4
Push 4
Pop 6
ComputeI Xor 6 1 6
Branch 6 (Rel 45)
Compute Add 7 0 4
ComputeI Add 4 2 4
Store 7 (IndAddr 4)
Compute Add 4 0 7
Load (ImmValue 37) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 38) 4
ReadInstr (IndAddr 4)
Receive 5
Push 5
```

```
WriteInstr 0 (IndAddr 2)
Pop 2
Compute Decr 2 0 4
Push 4
Load (ImmValue 37) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 38) 4
Pop 6
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Pop 0
Compute Add 7 0 6
Load (IndAddr 6) 6
ComputeI Add 6 1 6
Load (IndAddr 6) 5
Push 5
Pop 2
Compute Decr 2 0 4
Push 4
Compute Add 7 0 6
Load (IndAddr 6) 6
ComputeI Add 6 1 6
Pop 2
Store 2 (IndAddr 6)
Push 2
Pop 0
Load (IndAddr 7) 7
Jump (Rel (-56))
Load (ImmValue 0) 6
Push 6
Load (ImmValue 35) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 36) 4
Pop 6
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Pop 0
Load (IndAddr 7) 7
Load (ImmValue 0) 2
Compute Sub 7 2 2
ComputeI Add 0 1 5
ComputeI Gt 5 0 6
```

```
Branch 6 (Rel 23)
Compute Add 7 5 6
Load (IndAddr 6) 4
Load (IndAddr 2) 3
Compute Lt 3 0 6
Branch 6 (Rel 2)
Store 4 (IndAddr 3)
Compute Incr 2 0 2
Load (IndAddr 2) 3
Compute Lt 3 0 6
Branch 6 (Rel 10)
Compute Add 3 0 6
TestAndSet (IndAddr 6)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-4))
ComputeI Add 3 1 3
WriteInstr 4 (IndAddr 3)
ComputeI Sub 3 1 3
WriteInstr 0 (IndAddr 3)
Compute Incr 5 0 5
ComputeI Add 2 2 2
Jump (Rel (-23))
Compute Decr 7 0 2
Load (IndAddr 2) 6
Load (IndAddr 7) 7
Jump (Ind 6)
Load (ImmValue 4) 2
Compute Sub 7 2 2
Load (ImmValue 1) 5
ComputeI Gt 5 1 6
Branch 6 (Rel 7)
Load (IndAddr 2) 3
Compute Add 7 5 6
Store 3 (IndAddr 6)
Compute Incr 5 0 5
ComputeI Add 2 3 2
Jump (Rel (-7))
Compute Add 7 0 4
ComputeI Add 4 2 4
Store 7 (IndAddr 4)
Compute Add 4 0 7
Compute Add 7 0 6
Load (IndAddr 6) 6
ComputeI Add 6 1 6
Load (IndAddr 6) 5
Push 5
Load (ImmValue 0) 6
```

```
Push 6
Pop 3
Pop 2
Compute Gt 2 3 4
Push 4
Pop 6
ComputeI Xor 6 1 6
Branch 6 (Rel 148)
Compute Add 7 0 4
ComputeI Add 4 1 4
Store 7 (IndAddr 4)
Compute Add 4 0 7
TestAndSet (DirAddr 1)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
Load (ImmValue 5) 4
Load (ImmValue 0) 5
WriteInstr 5 (DirAddr 4)
Load (ImmValue 60) 6
Push 6
Pop 5
WriteInstr 5 (DirAddr 3)
WriteInstr 0 (DirAddr 2)
Load (ImmValue 1) 3
ReadInstr (IndAddr 3)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
TestAndSet (DirAddr 1)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
Load (ImmValue 5) 4
Load (ImmValue 0) 5
WriteInstr 5 (DirAddr 4)
Load (ImmValue 245) 6
Push 6
Pop 5
WriteInstr 5 (DirAddr 3)
WriteInstr 0 (DirAddr 2)
Load (ImmValue 1) 3
ReadInstr (IndAddr 3)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
Compute Equal 0 1 6
Branch 6 (Rel 4)
```

```
Load (ImmValue 2) 2
PrintOut 2
EndProg
Load (ImmValue 30) 3
Load (ImmValue 0) 2
ReadInstr (IndAddr 3)
Receive 4
Compute Add 2 4 2
ComputeI NEq 3 33 6
Compute Incr 3 0 3
Branch 6 (Rel (-5))
Compute Equal 2 0 6
Branch 6 (Rel 2)
Jump (Rel (-10))
Load (ImmValue 37) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 38) 4
ReadInstr (IndAddr 4)
Receive 5
Push 5
WriteInstr 0 (IndAddr 2)
Pop 6
PrintOut 6
TestAndSet (DirAddr 1)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
Load (ImmValue 5) 4
Load (ImmValue 0) 5
WriteInstr 5 (DirAddr 4)
Load (ImmValue 245) 6
Push 6
Pop 5
WriteInstr 5 (DirAddr 3)
WriteInstr 0 (DirAddr 2)
Load (ImmValue 1) 3
ReadInstr (IndAddr 3)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
TestAndSet (DirAddr 1)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
Load (ImmValue 5) 4
```

```
Load (ImmValue 0) 5
WriteInstr 5 (DirAddr 4)
Load (ImmValue 60) 6
Push 6
Pop 5
WriteInstr 5 (DirAddr 3)
WriteInstr 0 (DirAddr 2)
Load (ImmValue 1) 3
ReadInstr (IndAddr 3)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-3))
Compute Equal 0 1 6
Branch 6 (Rel 4)
Load (ImmValue 2) 2
PrintOut 2
EndProg
Load (ImmValue 30) 3
Load (ImmValue 0) 2
ReadInstr (IndAddr 3)
Receive 4
Compute Add 2 4 2
ComputeI NEq 3 33 6
Compute Incr 3 0 3
Branch 6 (Rel (-5))
Compute Equal 2 0 6
Branch 6 (Rel 2)
Jump (Rel (-10))
Load (ImmValue 37) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-4))
Load (ImmValue 38) 4
ReadInstr (IndAddr 4)
Receive 5
Push 5
WriteInstr 0 (IndAddr 2)
Pop 6
PrintOut 6
Compute Add 7 0 6
Load (IndAddr 6) 6
Load (IndAddr 6) 6
ComputeI Add 6 1 6
Load (IndAddr 6) 5
Push 5
Pop 2
Compute Decr 2 0 4
```

```
Push 4
Compute Add 7 0 6
Load (IndAddr 6) 6
Load (IndAddr 6) 6
ComputeI Add 6 1 6
Pop 2
Store 2 (IndAddr 6)
Push 2
Pop 0
Load (IndAddr 7) 7
Jump (Rel (-160))
Load (IndAddr 7) 7
Load (ImmValue 3) 2
Compute Sub 7 2 2
ComputeI Add 0 1 5
ComputeI Gt 5 1 6
Branch 6 (Rel 23)
Compute Add 7 5 6
Load (IndAddr 6) 4
Load (IndAddr 2) 3
Compute Lt 3 0 6
Branch 6 (Rel 2)
Store 4 (IndAddr 3)
Compute Incr 2 0 2
Load (IndAddr 2) 3
Compute Lt 3 0 6
Branch 6 (Rel 10)
Compute Add 3 0 6
TestAndSet (IndAddr 6)
Receive 6
Branch 6 (Rel 2)
Jump (Rel (-4))
ComputeI Add 3 1 3
WriteInstr 4 (IndAddr 3)
ComputeI Sub 3 1 3
WriteInstr 0 (IndAddr 3)
Compute Incr 5 0 5
ComputeI Add 2 2 2
Jump (Rel (-23))
Compute Decr 7 0 2
Load (IndAddr 2) 6
Load (IndAddr 7) 7
Jump (Ind 6)
Load (ImmValue 0) 6
Push 6
Pop 6
Load (ImmValue 33) 2
TestAndSet (IndAddr 2)
```

```
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-3))
Load (ImmValue 34) 4
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Load (ImmValue 0) 6
Push 6
Pop 6
Load (ImmValue 35) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-3))
Load (ImmValue 36) 4
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Load (ImmValue 0) 6
Push 6
Pop 6
Load (ImmValue 39) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-3))
Load (ImmValue 40) 4
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Load (ImmValue 0) 6
Push 6
Pop 6
Load (ImmValue 37) 2
TestAndSet (IndAddr 2)
Receive 3
Branch 3 (Rel 2)
Jump (Rel (-3))
Load (ImmValue 38) 4
WriteInstr 6 (IndAddr 4)
WriteInstr 0 (IndAddr 2)
Load (ImmValue 10) 6
Push 6
Compute Add 7 0 4
ComputeI Add 4 1 4
Load (ImmValue 1) 5
Pop 3
Store 3 (IndAddr 4)
Compute Incr 4 0 4
Load (ImmValue (-1)) 3
```

```
Store 3 (IndAddr 4)
Compute Incr 4 0 4
Load (ImmValue (-1)) 3
Store 3 (IndAddr 4)
Compute Incr 4 0 4
Load (ImmValue 707) 6
Push 6
Pop 5
Store 5 (IndAddr 4)
Compute Incr 4 0 4
Store 7 (IndAddr 4)
Compute Add 4 0 7
Load (ImmValue 430) 6
Push 6
Pop 2
Jump (Ind 2)
Load (ImmValue 1) 2
WriteInstr 2 (DirAddr 0)
EndProg
```

Correct Executions

Every time a value is printed in Peterson's algorithm test, it should be zero:

```
What file do you want to run? Please provide the relative path excluding the extension.
test/peterson
How many Sprockells do you want to use to run this file?
3
Running: test/peterson.shl
On 3 Sprockells
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
>>> 0
```

```
>>> 0
>>> 0
```