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

Martijn Verkleij & Tim Kerkhoven

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

July 1, 2016

Contents

1 Introduction **4**

2 Summary **5**

3 Problems & Solutions **6**

4 Detailed Language Description **7**
- 4.0.1 Program . . . 7
- 4.0.2 Global . . . 8
- 4.0.3 Procedure . . . 9

4.1 Statements . . . 10
- 4.1.1 Declaration . . . 10
- 4.1.2 If . . . 10
- 4.1.3 While . . . 11
- 4.1.4 Fork . . . 11
- 4.1.5 Join . . . 12
- 4.1.6 Call . . . 12
- 4.1.7 Expression . . . 13
- 4.1.8 Block . . . 13
- 4.1.9 Print . . . 14

4.2 Expressions . . . 14
- 4.2.1 Parentheses . . . 14
- 4.2.2 Assignment . . . 15
- 4.2.3 Variable . . . 15
- 4.2.4 Integer . . . 16
- 4.2.5 Boolean . . . 16
- 4.2.6 Operation . . . 17
- 4.2.7 Unary Operation . . . 17

4.3 Other Features . . . 18
- 4.3.1 Types . . . 18
- 4.3.2 Operators . . . 18
- 4.3.3 Call-by-reference . . . 18
- 4.3.4 Error Handling . . . 18

5 Description of the Software **20**

5.1 ASTBuilder.hs . . . 20

5.2 BasicFunctions.hs . . . 20

5.3 Checker.hs . . . 20
5.4 CodeGen.hs . . . 20
5.5 Constants.hs . . . 20
5.6 FP_ParserGen.hs . . . 20
5.7 Grammar.hs . . . 21
5.8 HardwareTypes.hs . . . 21
5.9 Main.hs . . . 21
5.10 README.md . . . 21
5.11 Simulation.hs . . . 21
5.12 Sprockell.hs . . . 21
5.13 System.hs . . . 21
5.14 Test.hs . . . 21

6 Test Plan & Results **22**
6.1 Implemented Tests . . . 22
6.2 Test Plan . . . 22
6.3 How To Run a Test . . . 23

7 Personal Evaluation **24**
7.1 Martijn . . . 24
7.2 Tim . . . 24

Appendices **25**

A Grammar Specification **26**

B Extended Test Program **30**
B.1 Listing of test program . . . 30
B.2 generated target code of test program . . . 30
B.3 one or more example executions showing correct functioning of the generated code 30

Chapter 1

Introduction

The language designed for this project is called Simple Haskell Language (SHL), with file extension *.shl*.

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.

- Something about concurrency
- Something about call-by-reference
- Something about other stuff

Chapter 4

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 4.0.2 (Global)
PROCEDURE Procedures as defined in 4.0.3 (Procedure)
STATEMENTS Statements as defined in 4.1 (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 4.0.3**
- **Global declarations 4.0.2**
- **Main code**
- **Post-program code**
  Stops auxilary threads.

Global

Syntax

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

TYPE A type as defined in 4.3.1 (Types)
ID A string as defined by 4.2.3 (Variable)
EXPRESSION An expression as defined in 4.2 (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 4.2.4 (Integer) and 4.2.5 (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 4.3.3 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 4.2.3 (Variable)
TYPE A type as defined in 4.3.1 (Types)
VAR A variable as defined in 4.2.3 (Variable)
STATEMENT A statement as defined in 4.1 (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 **??** (**??**)) is used, a variable passed as an argument can be used to write resulting values. One could also use the global variables (see 4.0.2 (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 4.1.6 (Call) and 4.1.4 (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 4.3.1 (Types)
ID A string as defined in 4.2.3 (Variable)
EXPRESSION An expression as defined in 4.2 (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 4.2.4 (Integer) and 4.2.5 (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 4.2 (Expressions)
STATEMENT A statement as defined in 4.1 (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

While

Syntax

`while ( <EXPRESSION> ) <STATEMENT>`

EXPRESSION An expression as defined in 4.2 (Expressions)
STATEMENT A statement as defined in 4.1 (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

Fork

Syntax

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

ID A string as defined by 4.2.3 (Variable)
EXPRESSION An expression as defined by 4.2 (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 4.0.3 (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 only a single thread exists, this will cause the code to enter an infinite loop.

Code Generation

Join

Syntax

```
join ;
```

Example

```
join;
```

Usage

Ensures all thread have are done before continuing. May only be called in the main thread.

Semantics

Blocks execution of the main thread until all other threads have finished their work.

Code Generation

Call

Syntax

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

ID A string as defined by 4.2.3 (Variable)
EXPRESSION An expression as defined by 4.2 (Expressions)

Examples

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

Usage

Execute the called procedure, which must have been declared somewhere. The expressions must have the same types as the procedure as defined in its declaration (see 4.0.3 (Procedure))

Semantics

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

Code Generation

Expression

Syntax

```
<EXPRESSION> ;
```

EXPRESSION An expression as defined in 4.2 (Expressions)

Examples

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

Usage

Allows expressions to be executed as statements, mostly for the purpose of enabling an assignment (see 4.2.2 (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

Block

Syntax

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

STATEMENT A statement as defined in 4.1 (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

Print

Syntax

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

EXPRESSION An expression as defined in 4.2 (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

Expressions

Parentheses

Syntax

```
( <EXPRESSION> )
```

EXPRESSION An expression as defined in 4.2 (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 4.2.6 (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

Assignment

Syntax

```
<ID> = <EXPRESSION>
```

ID A string as defined in 4.2.3 (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

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 4.1.1 (Declaration)) before use. It has a type which is determined upon declaration.

Semantics

Evaluation of a variable returns its value.

Code Generation

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

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

Operation

Syntax

`( <EXPRESSION> <OPERATOR> <EXPRESSION> )`
EXPRESSION An expression as defined in 4.2 (Expressions)
OPERATOR One of the following operators: `==`, `!=`, `&&`, `||`, `<>`, `<=`, `>=`, `<`, `>`, `+`, `-`, `*` (see 4.3.2 (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

Unary Operation

Syntax

`<OPERATOR> <EXPRESSION>`

OPERATOR One of the following operators: `-`, `++`, `-`, `!` (see 4.3.2 (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

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

Error Handling

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

During the parsing phase, the only error which might be thrown is a non-exhaustive pattern 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:

Chapter 5

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 checks an AST and adds information about scopes 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 set of SprIL instructions.

Constants.hs

Constants stores constant values used in the code generation.

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.

Chapter 6

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.

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
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

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 test files have been made 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 4.3.4 (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 7

Personal Evaluation

Martijn

Tim

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

«««< HEAD

Listing of test program

generated target code of test program

one or more example executions showing correct functioning of the generated code

======= 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

```
1  global bool flag_0 = false;
2  global bool flag_1 = false;
3  global int turn = 0;
4  global int i = 0;
5
6  procedure p_0() {
7      flag_0 = true;
8      turn = 1;
9      while ((flag_1 && (turn == 1))) {
10         // wait
11     }
12     // begin critical section
13     int j = 5;
14     while ((j > 0)) {
15         i = ++i;
16         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
```

```
55  WriteInstr 0 (DirAddr 1)
56  Jump (Ind 2)
57  ComputeI Add 1 30 3
58  WriteInstr 0 (IndAddr 3)
59  Jump (Abs 9)
60  Load (ImmValue 1) 2
61  Compute Sub 7 2 2
62  Load (ImmValue 1) 5
63  ComputeI Gt 5 0 6
64  Branch 6 (Rel 7)
65  Load (IndAddr 2) 3
66  Compute Add 7 5 6
67  Store 3 (IndAddr 6)
68  Compute Incr 5 0 5
69  ComputeI Add 2 3 2
70  Jump (Rel (-7))
71  Compute Add 7 0 4
72  ComputeI Add 4 1 4
73  Store 7 (IndAddr 4)
74  Compute Add 4 0 7
75  Load (ImmValue 1) 6
76  Push 6
77  Load (ImmValue 33) 2
78  TestAndSet (IndAddr 2)
79  Receive 3
80  Branch 3 (Rel 2)
81  Jump (Rel (-4))
82  Load (ImmValue 34) 4
83  Pop 6
84  WriteInstr 6 (IndAddr 4)
85  WriteInstr 0 (IndAddr 2)
86  Pop 0
87  Load (ImmValue 1) 6
88  Push 6
89  Load (ImmValue 39) 2
90  TestAndSet (IndAddr 2)
91  Receive 3
92  Branch 3 (Rel 2)
93  Jump (Rel (-4))
94  Load (ImmValue 40) 4
95  Pop 6
96  WriteInstr 6 (IndAddr 4)
97  WriteInstr 0 (IndAddr 2)
98  Pop 0
99  Load (ImmValue 35) 2
100 TestAndSet (IndAddr 2)
101 Receive 3
102 Branch 3 (Rel 2)
```

```
103 Jump (Rel (-4))
104 Load (ImmValue 36) 4
105 ReadInstr (IndAddr 4)
106 Receive 5
107 Push 5
108 WriteInstr 0 (IndAddr 2)
109 Load (ImmValue 39) 2
110 TestAndSet (IndAddr 2)
111 Receive 3
112 Branch 3 (Rel 2)
113 Jump (Rel (-4))
114 Load (ImmValue 40) 4
115 ReadInstr (IndAddr 4)
116 Receive 5
117 Push 5
118 WriteInstr 0 (IndAddr 2)
119 Load (ImmValue 1) 6
120 Push 6
121 Pop 3
122 Pop 2
123 Compute Equal 2 3 4
124 Push 4
125 Pop 3
126 Pop 2
127 Compute And 2 3 4
128 Push 4
129 Pop 6
130 ComputeI Xor 6 1 6
131 Branch 6 (Rel 7)
132 Compute Add 7 0 4
133 ComputeI Add 4 1 4
134 Store 7 (IndAddr 4)
135 Compute Add 4 0 7
136 Load (IndAddr 7) 7
137 Jump (Rel (-38))
138 Load (ImmValue 5) 6
139 Push 6
140 Compute Add 7 0 6
141 ComputeI Add 6 1 6
142 Pop 5
143 Store 5 (IndAddr 6)
144 Compute Add 7 0 6
145 ComputeI Add 6 1 6
146 Load (IndAddr 6) 5
147 Push 5
148 Load (ImmValue 0) 6
149 Push 6
150 Pop 3
```

```
151 Pop 2
152 Compute Gt 2 3 4
153 Push 4
154 Pop 6
155 ComputeI Xor 6 1 6
156 Branch 6 (Rel 45)
157 Compute Add 7 0 4
158 ComputeI Add 4 2 4
159 Store 7 (IndAddr 4)
160 Compute Add 4 0 7
161 Load (ImmValue 37) 2
162 TestAndSet (IndAddr 2)
163 Receive 3
164 Branch 3 (Rel 2)
165 Jump (Rel (-4))
166 Load (ImmValue 38) 4
167 ReadInstr (IndAddr 4)
168 Receive 5
169 Push 5
170 WriteInstr 0 (IndAddr 2)
171 Pop 2
172 Compute Incr 2 0 4
173 Push 4
174 Load (ImmValue 37) 2
175 TestAndSet (IndAddr 2)
176 Receive 3
177 Branch 3 (Rel 2)
178 Jump (Rel (-4))
179 Load (ImmValue 38) 4
180 Pop 6
181 WriteInstr 6 (IndAddr 4)
182 WriteInstr 0 (IndAddr 2)
183 Pop 0
184 Compute Add 7 0 6
185 Load (IndAddr 6) 6
186 ComputeI Add 6 1 6
187 Load (IndAddr 6) 5
188 Push 5
189 Pop 2
190 Compute Decr 2 0 4
191 Push 4
192 Compute Add 7 0 6
193 Load (IndAddr 6) 6
194 ComputeI Add 6 1 6
195 Pop 2
196 Store 2 (IndAddr 6)
197 Push 2
198 Pop 0
```

```
199 Load (IndAddr 7) 7
200 Jump (Rel (-56))
201 Load (ImmValue 0) 6
202 Push 6
203 Load (ImmValue 33) 2
204 TestAndSet (IndAddr 2)
205 Receive 3
206 Branch 3 (Rel 2)
207 Jump (Rel (-4))
208 Load (ImmValue 34) 4
209 Pop 6
210 WriteInstr 6 (IndAddr 4)
211 WriteInstr 0 (IndAddr 2)
212 Pop 0
213 Load (IndAddr 7) 7
214 Load (ImmValue 0) 2
215 Compute Sub 7 2 2
216 ComputeI Add 0 1 5
217 ComputeI Gt 5 0 6
218 Branch 6 (Rel 23)
219 Compute Add 7 5 6
220 Load (IndAddr 6) 4
221 Load (IndAddr 2) 3
222 Compute Lt 3 0 6
223 Branch 6 (Rel 2)
224 Store 4 (IndAddr 3)
225 Compute Incr 2 0 2
226 Load (IndAddr 2) 3
227 Compute Lt 3 0 6
228 Branch 6 (Rel 10)
229 Compute Add 3 0 6
230 TestAndSet (IndAddr 6)
231 Receive 6
232 Branch 6 (Rel 2)
233 Jump (Rel (-4))
234 ComputeI Add 3 1 3
235 WriteInstr 4 (IndAddr 3)
236 ComputeI Sub 3 1 3
237 WriteInstr 0 (IndAddr 3)
238 Compute Incr 5 0 5
239 ComputeI Add 2 2 2
240 Jump (Rel (-23))
241 Compute Decr 7 0 2
242 Load (IndAddr 2) 6
243 Load (IndAddr 7) 7
244 Jump (Ind 6)
245 Load (ImmValue 1) 2
246 Compute Sub 7 2 2
```

```
247 Load (ImmValue 1) 5
248 ComputeI Gt 5 0 6
249 Branch 6 (Rel 7)
250 Load (IndAddr 2) 3
251 Compute Add 7 5 6
252 Store 3 (IndAddr 6)
253 Compute Incr 5 0 5
254 ComputeI Add 2 3 2
255 Jump (Rel (-7))
256 Compute Add 7 0 4
257 ComputeI Add 4 1 4
258 Store 7 (IndAddr 4)
259 Compute Add 4 0 7
260 Load (ImmValue 1) 6
261 Push 6
262 Load (ImmValue 35) 2
263 TestAndSet (IndAddr 2)
264 Receive 3
265 Branch 3 (Rel 2)
266 Jump (Rel (-4))
267 Load (ImmValue 36) 4
268 Pop 6
269 WriteInstr 6 (IndAddr 4)
270 WriteInstr 0 (IndAddr 2)
271 Pop 0
272 Load (ImmValue 0) 6
273 Push 6
274 Load (ImmValue 39) 2
275 TestAndSet (IndAddr 2)
276 Receive 3
277 Branch 3 (Rel 2)
278 Jump (Rel (-4))
279 Load (ImmValue 40) 4
280 Pop 6
281 WriteInstr 6 (IndAddr 4)
282 WriteInstr 0 (IndAddr 2)
283 Pop 0
284 Load (ImmValue 33) 2
285 TestAndSet (IndAddr 2)
286 Receive 3
287 Branch 3 (Rel 2)
288 Jump (Rel (-4))
289 Load (ImmValue 34) 4
290 ReadInstr (IndAddr 4)
291 Receive 5
292 Push 5
293 WriteInstr 0 (IndAddr 2)
294 Load (ImmValue 39) 2
```

```
295 TestAndSet (IndAddr 2)
296 Receive 3
297 Branch 3 (Rel 2)
298 Jump (Rel (-4))
299 Load (ImmValue 40) 4
300 ReadInstr (IndAddr 4)
301 Receive 5
302 Push 5
303 WriteInstr 0 (IndAddr 2)
304 Load (ImmValue 0) 6
305 Push 6
306 Pop 3
307 Pop 2
308 Compute Equal 2 3 4
309 Push 4
310 Pop 3
311 Pop 2
312 Compute And 2 3 4
313 Push 4
314 Pop 6
315 ComputeI Xor 6 1 6
316 Branch 6 (Rel 7)
317 Compute Add 7 0 4
318 ComputeI Add 4 1 4
319 Store 7 (IndAddr 4)
320 Compute Add 4 0 7
321 Load (IndAddr 7) 7
322 Jump (Rel (-38))
323 Load (ImmValue 5) 6
324 Push 6
325 Compute Add 7 0 6
326 ComputeI Add 6 1 6
327 Pop 5
328 Store 5 (IndAddr 6)
329 Compute Add 7 0 6
330 ComputeI Add 6 1 6
331 Load (IndAddr 6) 5
332 Push 5
333 Load (ImmValue 0) 6
334 Push 6
335 Pop 3
336 Pop 2
337 Compute Gt 2 3 4
338 Push 4
339 Pop 6
340 ComputeI Xor 6 1 6
341 Branch 6 (Rel 45)
342 Compute Add 7 0 4
```

```
343 ComputeI Add 4 2 4
344 Store 7 (IndAddr 4)
345 Compute Add 4 0 7
346 Load (ImmValue 37) 2
347 TestAndSet (IndAddr 2)
348 Receive 3
349 Branch 3 (Rel 2)
350 Jump (Rel (-4))
351 Load (ImmValue 38) 4
352 ReadInstr (IndAddr 4)
353 Receive 5
354 Push 5
355 WriteInstr 0 (IndAddr 2)
356 Pop 2
357 Compute Decr 2 0 4
358 Push 4
359 Load (ImmValue 37) 2
360 TestAndSet (IndAddr 2)
361 Receive 3
362 Branch 3 (Rel 2)
363 Jump (Rel (-4))
364 Load (ImmValue 38) 4
365 Pop 6
366 WriteInstr 6 (IndAddr 4)
367 WriteInstr 0 (IndAddr 2)
368 Pop 0
369 Compute Add 7 0 6
370 Load (IndAddr 6) 6
371 ComputeI Add 6 1 6
372 Load (IndAddr 6) 5
373 Push 5
374 Pop 2
375 Compute Decr 2 0 4
376 Push 4
377 Compute Add 7 0 6
378 Load (IndAddr 6) 6
379 ComputeI Add 6 1 6
380 Pop 2
381 Store 2 (IndAddr 6)
382 Push 2
383 Pop 0
384 Load (IndAddr 7) 7
385 Jump (Rel (-56))
386 Load (ImmValue 0) 6
387 Push 6
388 Load (ImmValue 35) 2
389 TestAndSet (IndAddr 2)
390 Receive 3
```

```
391 Branch 3 (Rel 2)
392 Jump (Rel (-4))
393 Load (ImmValue 36) 4
394 Pop 6
395 WriteInstr 6 (IndAddr 4)
396 WriteInstr 0 (IndAddr 2)
397 Pop 0
398 Load (IndAddr 7) 7
399 Load (ImmValue 0) 2
400 Compute Sub 7 2 2
401 ComputeI Add 0 1 5
402 ComputeI Gt 5 0 6
403 Branch 6 (Rel 23)
404 Compute Add 7 5 6
405 Load (IndAddr 6) 4
406 Load (IndAddr 2) 3
407 Compute Lt 3 0 6
408 Branch 6 (Rel 2)
409 Store 4 (IndAddr 3)
410 Compute Incr 2 0 2
411 Load (IndAddr 2) 3
412 Compute Lt 3 0 6
413 Branch 6 (Rel 10)
414 Compute Add 3 0 6
415 TestAndSet (IndAddr 6)
416 Receive 6
417 Branch 6 (Rel 2)
418 Jump (Rel (-4))
419 ComputeI Add 3 1 3
420 WriteInstr 4 (IndAddr 3)
421 ComputeI Sub 3 1 3
422 WriteInstr 0 (IndAddr 3)
423 Compute Incr 5 0 5
424 ComputeI Add 2 2 2
425 Jump (Rel (-23))
426 Compute Decr 7 0 2
427 Load (IndAddr 2) 6
428 Load (IndAddr 7) 7
429 Jump (Ind 6)
430 Load (ImmValue 4) 2
431 Compute Sub 7 2 2
432 Load (ImmValue 1) 5
433 ComputeI Gt 5 1 6
434 Branch 6 (Rel 7)
435 Load (IndAddr 2) 3
436 Compute Add 7 5 6
437 Store 3 (IndAddr 6)
438 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
```

```
535 WriteInstr 5 (DirAddr 3)
536 WriteInstr 0 (DirAddr 2)
537 Load (ImmValue 1) 3
538 ReadInstr (IndAddr 3)
539 Receive 6
540 Branch 6 (Rel 2)
541 Jump (Rel (-3))
542 TestAndSet (DirAddr 1)
543 Receive 6
544 Branch 6 (Rel 2)
545 Jump (Rel (-3))
546 Load (ImmValue 5) 4
547 Load (ImmValue 0) 5
548 WriteInstr 5 (DirAddr 4)
549 Load (ImmValue 60) 6
550 Push 6
551 Pop 5
552 WriteInstr 5 (DirAddr 3)
553 WriteInstr 0 (DirAddr 2)
554 Load (ImmValue 1) 3
555 ReadInstr (IndAddr 3)
556 Receive 6
557 Branch 6 (Rel 2)
558 Jump (Rel (-3))
559 Compute Equal 0 1 6
560 Branch 6 (Rel 4)
561 Load (ImmValue 2) 2
562 PrintOut 2
563 EndProg
564 Load (ImmValue 30) 3
565 Load (ImmValue 0) 2
566 ReadInstr (IndAddr 3)
567 Receive 4
568 Compute Add 2 4 2
569 ComputeI NEq 3 33 6
570 Compute Incr 3 0 3
571 Branch 6 (Rel (-5))
572 Compute Equal 2 0 6
573 Branch 6 (Rel 2)
574 Jump (Rel (-10))
575 Load (ImmValue 37) 2
576 TestAndSet (IndAddr 2)
577 Receive 3
578 Branch 3 (Rel 2)
579 Jump (Rel (-4))
580 Load (ImmValue 38) 4
581 ReadInstr (IndAddr 4)
582 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)
```

```
631 Compute Incr 5 0 5
632 ComputeI Add 2 2 2
633 Jump (Rel (-23))
634 Compute Decr 7 0 2
635 Load (IndAddr 2) 6
636 Load (IndAddr 7) 7
637 Jump (Ind 6)
638 Load (ImmValue 0) 6
639 Push 6
640 Pop 6
641 Load (ImmValue 33) 2
642 TestAndSet (IndAddr 2)
643 Receive 3
644 Branch 3 (Rel 2)
645 Jump (Rel (-3))
646 Load (ImmValue 34) 4
647 WriteInstr 6 (IndAddr 4)
648 WriteInstr 0 (IndAddr 2)
649 Load (ImmValue 0) 6
650 Push 6
651 Pop 6
652 Load (ImmValue 35) 2
653 TestAndSet (IndAddr 2)
654 Receive 3
655 Branch 3 (Rel 2)
656 Jump (Rel (-3))
657 Load (ImmValue 36) 4
658 WriteInstr 6 (IndAddr 4)
659 WriteInstr 0 (IndAddr 2)
660 Load (ImmValue 0) 6
661 Push 6
662 Pop 6
663 Load (ImmValue 39) 2
664 TestAndSet (IndAddr 2)
665 Receive 3
666 Branch 3 (Rel 2)
667 Jump (Rel (-3))
668 Load (ImmValue 40) 4
669 WriteInstr 6 (IndAddr 4)
670 WriteInstr 0 (IndAddr 2)
671 Load (ImmValue 0) 6
672 Push 6
673 Pop 6
674 Load (ImmValue 37) 2
675 TestAndSet (IndAddr 2)
676 Receive 3
677 Branch 3 (Rel 2)
678 Jump (Rel (-3))
```

```
679 Load (ImmValue 38) 4
680 WriteInstr 6 (IndAddr 4)
681 WriteInstr 0 (IndAddr 2)
682 Load (ImmValue 10) 6
683 Push 6
684 Compute Add 7 0 4
685 ComputeI Add 4 1 4
686 Load (ImmValue 1) 5
687 Pop 3
688 Store 3 (IndAddr 4)
689 Compute Incr 4 0 4
690 Load (ImmValue (-1)) 3
691 Store 3 (IndAddr 4)
692 Compute Incr 4 0 4
693 Load (ImmValue (-1)) 3
694 Store 3 (IndAddr 4)
695 Compute Incr 4 0 4
696 Load (ImmValue 707) 6
697 Push 6
698 Pop 5
699 Store 5 (IndAddr 4)
700 Compute Incr 4 0 4
701 Store 7 (IndAddr 4)
702 Compute Add 4 0 7
703 Load (ImmValue 430) 6
704 Push 6
705 Pop 2
706 Jump (Ind 2)
707 Load (ImmValue 1) 2
708 WriteInstr 2 (DirAddr 0)
709 EndProg
```