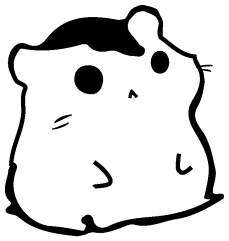
PP Final Project



Campbell

Authors:
Pieter Bos
Sophie Lathouwers

July 6, 2015

Contents

1 Summary **3**

2 Problems and solutions **4**
2.1 Stack size 4
2.2 Access links 4
2.3 Currying and functions as values 4
2.4 ANTLR 5

3 Detailed language description **6**
3.1 Declarations 6
3.2 Assignments 6
3.3 Functions 6
3.4 While loop 7
3.5 If statement 8
3.6 Expressions 8
3.6.1 Boolean literal 8
3.6.2 Call 8
3.6.3 Comparison 9
3.6.4 Dot 9
3.6.5 Haskell 9
3.6.6 Identifier 9
3.6.7 Int literal 9
3.6.8 Math 9
3.6.9 Unary math 10

4 Description of the software **11**
4.1 Campbell 11
4.1.1 Language 11
4.1.2 Parser 12
4.1.3 Roborovski.model 13
4.2 Sprockell 13
4.3 Util 13

5 Test plan and results **14**
5.1 Syntax testing 14
5.2 Concurrency testing 15
5.3 Contextual testing 15
5.4 Semantic testing 17

6 Conclusion **18**

7 Appendix A : Grammar specification **19**

8 Appendix B : Extended test program **23**
8.1 Java to compile code 23
8.2 Campbell file to be compiled 25
8.3 Generated Haskell code 25
8.4 How to debug run 95
8.5 How to run and result 95

Chapter 1

Summary

The main features of Campbell include:

- Currying
- Functions as values
- Classes with bound methods
- Concurrency: joins, locks
- (Nested) Functions (with call-by-reference)
- Mandatory: While loop, If statement, Assignments, Variables, Simple expressions

We were mainly inspired by the programming languages Python[1] and Rust[2]. Our language is called Campbell, named after the Campbell's dwarf hamster[3]. This is because hamsters are great creatures even though they are small and we want our language to be just as great even though it has a limited amount of features. We have also used an intermediate representation called Roborovski[4]. This is named after the Roborovski hamster. This is the smallest breed of hamsters. Just like the hamster, we wanted our intermediate representation to be as minimal as possible.

[1] https://www.python.org/
[2] http://www.rust-lang.org/
[3] https://en.wikipedia.org/wiki/Campbell's_dwarf_hamster
[4] https://en.wikipedia.org/wiki/Roborovski_hamster

Chapter 2

Problems and solutions

2.1 Stack size

The default stack size was quickly found to be too small to accommodate all the locals, function calls and other data we used for non-trivial programs, so the stack size was increased to 65536. In order to achieve this we modified the sprockell codebase.

2.2 Access links

Functions need to be able to access global scope and other scopes if the functions are nested, but the program cannot access these variables when it has no pointer to the scope. We soon realized that access links were made exactly for this purpose and implemented them in our compiler.

2.3 Currying and functions as values

Our language supports currying and functions as values, which means two things.

We need to be able to save functions with an arbitrary amount of arguments saved to support currying. We implemented this in a very simple data structure that saves the location of the function, the access links to the enclosing scope of the function, the number of arguments saved in the data structure and the arguments themselves. When applying more arguments to a saved function a new data structure is allocated and the additional arguments are copied to this data structure as well as the previously saved arguments. The compiler detects when an actual function call must be made, that is when enough arguments are supplied.

Treating functions as values also means that the function might need data from a scope above it that is closed after passing around the value (function), i.e.

closures. We chose not to overcomplicate things for this projects, so all scopes are forever stored on the heap. Further improvements could be to implement a garbage collector or do some form of analysis to determine which scopes do not need to be saved.

2.4 ANTLR

We think we have found a bug in the lexer generator of ANTLR. We have a handwritten lexer that connects to an ANTLR parser to deal with whitespace indenting, but the lexer would not generate any tokens for some orderings of the dummy tokens. We would add a token to CampbellTokens.g4 and for some locations the lexer would work and for other locations the lexer would not even generate the first token. We worked around this by only adding new tokens to the end, which for some reason did work reliably.

Chapter 3

Detailed language description

3.1 Declarations

The syntax of a declaration consists of a type followed by an identifier. An example is: *int i*. Identifier is the name of the variable and it may contain characters from the following set of characters: {a-zA-Z0-9_$}. Type shows what the type of the variable is; this can be a class, integer, boolean or function. All variables must be declared before they can be used. Restrictions to the type that exist are that the given type must exist, so there must be a class defined for it, it must be a function, boolean or integer. Whenever a declaration is made in the scope, memory is allocated for this variable on the heap whenever the scope is entered. No code for the Sprockell is generated for this feature.

3.2 Assignments

An assignment looks like *'expression = expression'*. An example is *'i = 5'* or *clock.time = 1600'*. In the last example you refer to a property of an implemented class. The left hand side must be declared as a variable, property of a class, etc. before a value can be assigned to it. This is one of the restrictions for this feature. Another is that the type of the left hand side expression must be equal to the type of the right hand side expression.

3.3 Functions

The syntax of functions looks like *'fun className identifier (arguments) block'*. "fun" is a keyword that identifies that the declaration of a function starts here. "className" is the name of the return type such as int. "identifier" is the name of the function. Then between brackets arguments can be given; they are

optional. Arguments look like declarations, type followed by an identifier. An example is "int i". A function ends with a block. A block consists of several statements such as loops, assignments, declarations, returns, etc. An example of a function is:
fun int returnSum (int a, int b)
    return (a+b)
This feature can be used in classes and in the general scope. Functions are ideal for returning values or for making your code look neater. It can be executed by defining a function in a class, scope, function, etc. and then calling the function from where the code is executed such as the general scope or from within another function. Calling looks like *'identifier (arguments)'*. The identifier is again the name of the function and arguments are several values, to be used by the function, separated by commas. An example of calling the before mentioned function would be:
returnSum(3, 5)
The call to the function would then return 8 as a result. A function is compiled to a sequence of statements. When the compiler wants to call a function it searches for its definition, which can be a variable or a regularly defined function. Since functions can be partially applied, the compiler also detects whether the program must actually jump to the function or just save the supplied arguments. Internally a function is saved by its address, the access link to the enclosing scope and its current arguments. When the compiler encounters a regular function with no arguments, it simply saves the argument count 0, the address of the function and the access link to the enclosing scope. Whenever more arguments are applied the saved arguments are copied and the new ones are added. When the program actually has to jump to the function, it sets up the return address, original stack pointer, access link and the arguments of the function and jumps there. Functions are saved as pointers to these data structures, so they can be passed around as values within arguments and variables, with any number of arguments applied.

3.4 While loop

The syntax of a while loop looks like:
while condition
    statements
An example is :
while (i <10)
    i = i + 1
This feature can be used in order to loop over a certain set of statements that need to be performed several times. This certain set of statements is put in a new block of statements within the while loop. A restriction of this feature is that the condition must be of type boolean It is compiled to a condition followed by several commands that execute the statements within. It first checks whether the condition is still true with an absolute branch. If this is true it will execute

all statements within the while. If not, it will jump over these statements out of the loop. At the end of all statements, before the end of the loop, there is a jump that refers to the condition again and so the cycle continues.

3.5 If statement

The syntax for an if statement is:
if condition
    statements
else
    statements
The else and the statements that follow are optional. An example of an if is:
if i >= 0
    i = i + 1
else
    i = i - 1
Another example is :
if i == 0
    i = 5
This feature can be used to only perform specific statements under a certain condition. A restriction to this feature is that the condition must be of type boolean. This code is compiled to first a condition check. If so, it will execute the code of the first statements block. At the end of this statements block there is a jump to the end of the if so that the statements in the else block will not be executed. If the condition is false, it will jump to the else statements block and execute this.

3.6 Expressions

There are several types of expressions: boolean literal, call, comparison, dot, identifier, int literal, math and unary math expressions. The dot operator, function calls and parentheses have the highest precedence, followed by the unary math operations, followed by multiplication, division, modulo, left shift and right shift, followed by addition and subtraction, followed by the comparision operators, followed by the boolean binary operators.

3.6.1 Boolean literal

Boolean expressions are *true* or *false*.

3.6.2 Call

Call expressions are used to call a function. Its syntax and how it should be used is further elaborated in section 3.3.

3.6.3 Comparison

Comparison expressions are expressions that evaluate to boolean expressions. It can contain the following operators: *<*, *<=*, *>*, *>=*, *==*, *!=*. An example is: *'i != 1'*.

3.6.4 Dot

Dot expressions are used to refer to properties of expressions. They look like *'expression.property'*. This can be used when objects have certain variables in their classes of which you would like to know their value or if you want to call a function of that class. An example is *'grandmother.age'*. Note that retrieving a function of an object is really just partial application of the class method with *'this'*, so the user can write *'dog.bark(5)'* as *'(dog.bark)(5)'* if they were so inclined.

3.6.5 Haskell

Haskell expressions are used to write inline Haskell in Campbell. It therefore looks like Haskell code. It can be used for several things, an example is writing to the standard out. This feature of the language is meant to implement internal features, analogous to inline assembly in other languages and should never be seen by a regular user. Instead actions that need haskell should be abstracted away in the standard library.

3.6.6 Identifier

Identifier expressions are names of variables or functions. An example would be: *'i'*.

3.6.7 Int literal

Int literal expressions are expressions that evaluate to an integer thus a value from -2^{63} to 2^{63}-1.

3.6.8 Math

Math expressions are expressions that contain one of the following operators: +, -, *, /, %, <<, >>, &, |, ^. "+" is used to add variables. "-" is used to subtract variables. "*" is used to multiply variables. "/" is used to divide variables. % is used for modulus division. <<is the left bitshift operator. >>is the right bitshift operator. These expressions evaluate to integers and only take integers as arguments. & is the binary and operator, | is the binary or operator and ^ is the binary exclusive or operator.

3.6.9 Unary math

Unary math expressions can be either "!" or "-" followed by an expression. "!" expects an expression of type boolean to follow it, which inverts the boolean value. "-" expects an integer to follow it and negates the value.

Chapter 4

Description of the software

On top level we have two packages called examples and main. Examples contains an example file for each feature to help novice users. The main package is divided in three subpackages. The first is the leading package called "campbell". The second package is called "sprockell". The last package is called "util".

4.1 Campbell

Campbell consists of three subpackages: language, parser and roborovski.model.

4.1.1 Language

The language package consists of two other packages: model and types. Model contains classes for all elements in the abstract syntax tree. Types contains classes for all types in Campbell.

Model

The model package in Campbell can be divided into 5 sections. The first section is exceptions and contains the classes CompileException and NotImplementedException. The second section is maps and contains the classes SymbolMap and TypeMap. SymbolMap and TypeMap are maps with elements corresponding to their names and where no duplicates may occur. The third section is interfaces and contains the classes Node and Symbol. Each element in the abstract syntax tree that has a location is a Node. Each element in the abstract syntax tree that has a name and type is a Symbol. The fourth section is a package called scoped. Scoped contains all elements of the language that are a scope in the abstract syntax tree and it contains an interface Scope. Elements of the language that are scoped include: blocks, classes, for loops, functions, if statements, programs, unsafe statements and while loops. A scope can contain new functions and local variables, although you cannot define locals in a class statements, rather these are treated as properties of instances of the class.

The last section is a package called unscoped. This contains all elements of the language that are not a scope in the abstract syntax tree. Elements of the language that are unscoped include: assignments, booleans, call expressions, comparisons, declarations, dot expressions, haskell statements, identifiers, integers, math expressions, return statements and unary math expressions.

Types

Types contains the following classes: BoolType, ClassType, FunctionType, GenericType, IntType, PointerType, PrimitiveType, Type and VoidType. Type is an abstract class that defines two abstract functions that should therefore be defined in all type classes. Type also defines three other functions that are the same for all types: getImplementation, setImplementation and fromContext. All other type classes are types corresponding to their names. BoolType is a type for booleans, it can be used in Campbell as "bool". ClassType is a type made for classes. In Campbell it will look like something like: "<type1, type2 >". FunctionType is a type for functions, as the name might suggest and can be used like "(arg1 ->arg2 ->arg3 ->result)," or "(result)," for example "(int ->int)" or "(void)". GenericType is a type for generic types and has no default implementation. This type is currently not used in the language. IntType is a type for integers and looks like "int". PrimitiveType is an abstract class for the primitive types integers and booleans. VoidType is a type used if a function does not return anything.

4.1.2 Parser

Parser is the package that contains the grammar, parser and lexer for Campbell. It contains a file "Campbell.g4" which is the grammar specification of Campbell (See section 7). Another file in this package is "CampbellTokens.g4". This is a token file that is not generated by ANTLR. We have written this in order to be able to write our own lexer and still use ANTLR at the same time. It contains all tokens for the language. This package also contains 3 Java classes. The first class is CampbellLexer. This lexer is not generated by ANTLR but written by us. The lexer tries to match tokens according to the input according to the specified grammar. We have written our own lexer as we wanted to make a language in which blocks are opened and closed with whitespace indenting. Each time a new block is opened, the indent level becomes one higher. When a block is closed, the indent level becomes one lower. The second class in this package is CampbellStreamParser. This class takes an input and creates a new parser for Campbell using the parser generated by ANTLR. The third class in this package is ErrorListener, which converts the ANTLR error to an error that our compiler can deal with.

4.1.3 Roborovski.model

This package contains Java classes used to represent the intermediate representation called Roborovski and two interfaces. Elements that are part of Roborovski are: assignments, binary expressions, blocks, call expressions, constant expressions, dot expressions, functions, function expressions, haskell statements, if statements, programs, return statements, structs, variables, variable expressions and while loops. The most important differences between Roborovski and Campbell is that the generics have been replaced by concrete types on usage, traits have been replaced by concrete implementations on usage and all variables are now referenced to, it is known from which scopes which variables are used. These first two things are not used in the current implementation, since generics do not yet function correctly.

4.2 Sprockell

This package contains tools to help with the conversion from the intermediate representation called Roborovski to Sprockell. The package contains three classes. The first class is SprockellCompute. This is an enum with all possible operators that can be used in the ALU of a Sprockell. The second class is SprockellEmitter. This is the main class used to output a Haskell file. It contains methods for all possible instructions of Sprockell to add them to the Haskell file that is generated. This is a test for the previous mentioned class used to see whether it generates a correct Haskell file. The last class in this package is SprockellRegister. SprockellRegister is an enum containing all registers that can be used in Sprockell.

4.3 Util

The util package contains two classes. The first is the HashList class. HashList is a class that extends LinkedList as to make it possible to test for equality and compare instances by their hash, that is, inequality of the hash implies inequality of the instances. The second class is a test for the previously mentioned class.

Chapter 5

Test plan and results

We have tested our compiler for four different types of errors: syntax, concurrent, contextual and semantic. There are four test classes that can be found in the package campbell.language.test: syntaxTest, concurrencyTest, contextualTest and semanticTest. Not only correctness is tested but also what happens when there is an error in the program.

5.1 Syntax testing

Syntax testing contains several test cases which are all separated into their own method (with Test annotation). They can all be run at once by running the java class syntaxTest. If you want to run a specific test case you can run one case by right-clicking on the method name and then clicking on 'run methodName()' (this is possible in IntelliJ, no guarantees if you are using another IDE). The first test case is tests a function that is declared wrongly. Instead of using the keyword "fun" the programmer has made an error and typed "funnn". In this case the compiler throws, as expected, an exception that it cannot return outside of all functions. This is the case because it cannot find the return address. The second case tests an if statement with an incorrect keyword "iff". The compiler throws, as expected, an exception that there is no implementation of a class. This is correct as it tries to parse iff has a class as it does not recognize the keyword. The third case is a class statement with an incorrect keyword. It tries to parse this as a class as well and comes to the conclusion that it cannot find an implementation of the class and thus it fails, as expected. The fourth case is a correct program. The program contains multiple functions, a while loop, if statements, in-line Haskell, an unsafe statement, returns, a class with a generic argument, declarations, assignments, function calling and mathematical expressions. This program is parsed correctly and thus it is parsed correctly.

5.2 Concurrency testing

Concurrency testing contains three small cases. The first case checks a correct concurrent program that contains locks and a join. It makes four threads and each thread increases a shared counter 100 times. In this case the result is always 400. The code can be generated by running testCorrectConcurrency in the concurrencyTest class. The generated haskell file is called 'correctConcurrency.hs', it is placed in the same package as the concurrencyTest class. If you are in the Sprockell codebase, you can start ghci using your terminal. To run the generated haskell file type 'main' after you have loaded the program. It will then run and after a while will print the result of the counter which should always be 400. The next test case is the same program as the one in the first case only the locks have been removed. In this case a file called 'failConcurrency.hs' is generated by running testFailConcurrency method in this class. It can be run in the same way as the previous test. The result is now different each time it is run. Thus this shows that locks are necessary when using multiple threads and that the multiple threads do actually use shared memory. The last test case for concurrency tests what happens when you try to lock on the same thing twice. In this case a deadlock will occur as can be seen if you run the program in debug mode. How to run a program in debug mode is explained in section 5.4.

5.3 Contextual testing

Contextual testing contains many small test cases from testing declarations, assignments to giving wrong arguments to functions, again all separated in their own method (with Test annotation). The tests can be run all at once by running the contextualTest.java. If you want to run a specific test case you can run one case by right-clicking on the method name and then clicking on 'run methodName()' (this is possible in IntelliJ, no guarantees if you are using another IDE). The first test case checks whether a correct assignment with declaration in the same scope succeeds. The second test case checks what happens when a variable is declared after it has been assigned. This throws a compile exception, as expected, namely "undeclared variable i used". The third test case checks whether a correct assignment with declaration in a scope above its own scope succeeds. The fourth case tests whether an assignment fails when the corresponding declaration is in a scope within its own scope. This throws a compile exception, as expected, namely "no definition of i can be found". The fifth case checks what happens when a value is assigned to a variable of a different type. This throws a compile exception, as expected, namely "Type error: left expression is of type bool whereas right is of type int". The sixth case tests a file containing 5 functions with different return types and some nested succeeds the type check test. The seventh case checks a function that returns something of an incorrect type. As expected a compile exception is thrown, namely: "Type of return expression (bool) does not correspond to the function's contract (int)". The eight case tests what happens when a function is given too many arguments

This will generate a compile exception as the types are not as expected. The ninth case checks what happens when a function is given an argument of a wrong type. This will again generate an expected compile exception as the types of the call expression and the expected types in the function do not comply. The tenth case checks what happens when the result of a function is stored in a variable with the same type as the return type of the function. This test case succeeds as expected. The eleventh case checks what happens when the program contains the adding of an integer to a boolean. As expected a compile exception is thrown: "Incorrect type in expression (i+j)". A similar error occurs when the program tries to "and" an integer and a boolean. As expected a compile exception is thrown: "Incorrect type in expression (i & j)". When "and" is applied to two booleans, the test case succeeds as expected. When the same operand is applied to two integers an exception is thrown namely "Incorrect type in expression (i & j)" as expected. The next test case tries to add two integers and succeeds. The following test case tries two apply the operand "and" to two functions that return a boolean value. The test case succeeds. The test case that tries to add two integers, which are results from functions, also succeeds. The next test case tries to add two booleans and fails. As expected it throws a compile exception: "Cannot apply this operator to the given arguments: a, b". The following test case tests what happens when a function expects arguments but none are given. This generates an expected compile exception as the types for the function do not comply in this case. The next test case tests what happens when a function expects no arguments but one is given. This generates a compile exception as well. In the returning of the type of a call expression is tested whether it has the right amount of arguments. This is not the case and thus the exception is thrown with the message: 'Called function takes up to 0 arguments, but 1 were given.'. The next test case tests what happens when no arguments are given to a class being made whereas it does expect an argument. In this case, as expected it throws an exception because it cannot find the implementation of the class without arguments. The following test case tests the getting of a property in a class. This is a correct test case and succeeds. Then there is a test case that tries to get a non-existing property. We expect a compile exception indicating that the property does not exist. When running this test case we get a compile exception with the message 'Unknown property hamham of type Adder', as expected. Then there is a case that tries to get a nested property. In this case there exist three nested classes which make their nested class upon initialization. Then a method can be called which will call a function or property in their nested class. This is a correct case and as expected it succeeds. The next test case calls a nested function. This is another correct test case and succeeds as well. The last test case tries to use a variable that is declared in the class. In order to refer to values in the class the user should use "this.variableName". In this case the "this." has been omitted. This gives the following expected compile exception: "Try using 'this.total'".

5.4 Semantic testing

At the moment of writing only three test cases have been tested. All test programs can be compiled by running semanticTest.java. After this, you shall have to go to the Sprockell directory with your command line and start ghci. First load the Haskell program that you want to run (infiniteLoop.hs, divideZero.hs or isPrime.hs). You can run the program then by typing 'main' and then enter. If debugging is wanted type the following after you have loaded the correct file: let debug st = (show $ (regbank s) ! PC) ++ " " ++ (show $ (regbank s) ! RegA) ++ " " ++ (show $ (regbank s) ! RegB) ++ " " ++ (show $ (regbank s) ! SP) ++ " " ++ (show $ map (\i ->((localMem s) !!! i)) [0..70]) ++ "\n" where s = head $ sprs $ st To run the program with debugging, type 'runDebug debug ¡cores¿ prog' and enter. The debugging will show the program counter, registers a, b, the stack pointer and some local memory.

The program with an infinite loop kept running on and on. We would have expected to get in trouble when the first number in memory (in debugging mode) would get bigger than 65536. This is because this number represents the first free memory address on the heap and the heap has a size of 65536. However this did not happen. We let the program run until this number was 102805. We expect that it ignores the write instruction if the tries to write to a memory address that does not exist.

The program that tries to divide by zero throws an exception. We expected this as we did not have time to do a run-time check for errors like this. If we would have to prevent this then we would build a solid error handling mechanism. An idea is to make signals when something goes wrong and catch these in specified functions, like C does. Another option would be to introduce exceptions.

The last program checks whether a number is prime. It gets the number 65521 which is a prime. If you want to test it for another number, change the value of n in the file isPrime.ham on line 36. It will print 1 if it is a prime number and 0 if it is not. The listing of this correct test program can be found in chapter 8 Appendix B : Extended Test Program. An example of how a debug run is done for this program can be found in section 8.4. In section 8.5 can be found how to perform a non-debug run and the result of a normal test run can be found here as well.

Chapter 6

Conclusion

Personally I, Sophie, found the module very chaotic in the beginning. Many parts of the module were not finished and the pressure of signing everything off was very big. Personally I would like a style of teaching where the responsibility lies more with the student. I did find the module very challenging and I believe it is one of the modules where I learned the most. It was just a pity that this was partly overshadowed by the chaos. I find Campbell quite easy to write which I find a big benefit for a programming language. It was a pity that we did not have enough time to implement several features such as for loops and traits. However I find it extraordinary that we almost implemented generics in our language, it would have been a working feature if we just had a little bit more time. I would have liked to spend more time on concurrency and optimisations in our language. We mainly focused on getting the bulk of features implemented that we thought were cool nor did we spend any time on optimisations. I believe this is also a very important aspect of compilers and as we did not focus that much on optimisations during the lectures, I would have liked to learn more about this.

I, Pieter, thought this was one of the more interesting modules. However, the pressure was quite high throughout the module, although it did get better after a while. Furthermore the module did not really align with my learning style, since I like to learn the material in my own time and make the schedule myself. The fact that we had to sign off every single exercise was really annoying to me. I think students should certainly be able to plan for themselves at this point, almost two years into their studies at the highest educational level. I think the language we made turned out quite well, although we did get in trouble toward the end and decided not to spend too much time on concurrent programming, which is why the support is so basic. The fact that we could get functions-as-values and currying to work was really nice, though. A lot of layers of the compiler do already support generics and traits, but in the end we had too little time to fully implement them, which is a shame since the time we spent on it is now effectively wasted. In the end I am very satisfied with the feature set of our language.

Chapter 7

Appendix A : Grammar specification

```
grammar Campbell;
import CampbellTokens;

program
    : topLevelStatement+ EOF
    ;

topLevelStatement
    : statement # normalStatement
    | SHARED decl # sharedDecl
    | THREADS INT # cores
    ;

statement
    : haskell
    | nop
    | fun
    | unsafe
    | whileNode
    | ifNode
    | impl
    | trait
    | returnNode
    | decl
    | assign
    | classNode
    | forNode
    | expr
```

```
    ;

haskell
    : HASKELL
    ;

nop
    : NOP
    ;

fun
    : FUN className IDENTIFIER PAREN_OPEN
      (decl (COMMA decl)*)? PAREN_CLOSE block?
    ;

unsafe
    : UNSAFE block
    ;

whileNode
    : WHILE expr block
    ;

ifNode
    : IF expr block (ELSE block)?
    ;

impl
    : IMPL className (OF classList)? block
    ;

trait
    : TRAIT className (OF classList)? block
    ;

returnNode
    : RETURN expr
    ;

decl
    : className IDENTIFIER
    ;

assign
    : expr EQUALS expr
    | decl EQUALS expr
```

```
    ;

classNode
    : CLASS className block
    ;

forNode
    : FOR expr IN expr block
    ;

expr
    : expr AND expr0 # and
    | expr OR expr0 # or
    | expr XOR expr0 # xor
    | expr0 # superExpr
    ;

expr0
    : expr0 LTE expr1 # lte
    | expr0 GTE expr1 # gte
    | expr0 EQ expr1 # eq
    | expr0 NEQ expr1 # neq
    | expr0 BROKET_OPEN expr1 # lt
    | expr0 BROKET_CLOSE expr1 # gt
    | expr1 # simpleExpr
    ;

expr1
    : expr1 PLUS expr2 # add
    | expr1 MINUS expr2 # subtract
    | expr2 # simpleExpr1
    ;

expr2
    : expr2 STAR expr3 # multiply
    | expr2 SLASH expr3 # divide
    | expr2 PERCENT expr3 # modulo
    | expr2 LSH expr3 # lsh
    | expr2 RSH expr3 # rsh
    | expr3 # simpleExpr2
    ;

expr3
    : MINUS expr # negate
    | NOT expr # not
    | expr4 # simpleExpr3
```

```
    ;

expr4
    : INT exprAddon* # int
    | bool exprAddon* # boolExpr
    | IDENTIFIER exprAddon* # id
    | PAREN_OPEN expr PAREN_CLOSE exprAddon* # paren
    ;

block
    : OPEN_BLOCK statement+ CLOSE_BLOCK
    ;

className
    : IDENTIFIER (BROKET_OPEN className
      (COMMA className)* BROKET_CLOSE)? # classNameClass
    | PAREN_OPEN className (ARROW className)* PAREN_CLOSE #classNameFunc
    ;

classList
    : className
    | PAREN_OPEN className COMMA className
      (COMMA className)* PAREN_CLOSE
    ;

exprAddon
    : PAREN_OPEN (expr (COMMA expr)*)? PAREN_CLOSE # call
    | DOT IDENTIFIER # get
    ;

bool
    : TRUE # true
    | FALSE # false
    ;
```

Chapter 8

Appendix B : Extended test program

8.1 Java to compile code

```
package campbell.language.test;

import campbell.language.model.scoped.Program;
import org.junit.Test;
import sprockell.SprockellEmitter;

import java.io.FileWriter;
import java.io.IOException;
import java.net.URL;

import static campbell.language.model.scoped.Program.parseFrom;

/**
 * This class will test the file "isPrime.ham"
 *
 * Used for the listing of the extended test program
 */
public class isPrimeTest {
    String[] files = {"isPrime.ham",
            "src/main/java/campbell/language/test/isPrime.hs"};

    /**
     * Method that compiles a program from a
     * given input to a given output
     * @param input - .ham file to be compiled
     * @param output - .hs file (path) to be made
```

```
     * @throws IOException
     */
    public static void compileProgram(URL input, String output)
            throws IOException {
        Program p = parseFrom(input.openStream());
        p.setScope(null);
        p.findDefinitions();
        p.findImpls();
        p.checkType();

        System.out.println(p);

        campbell.roborovski.model.Program program
                = p.toRoborovski();

        program.compile(
                new SprockellEmitter(
                        new FileWriter(output)
                )
        );
    }


    /**
     * Test case that generates a file with an algorithm
     * to check whether a given number is prime
     *
     * Number to check for primeness should be entered
     * in the .ham file
     *
     * After this test the Haskell file should be run to
     * check whether it is correct
     */
    @Test
    public void generateIsPrime() {
        try {
            URL input = this.getClass().getResource(
                    files[0]
            );

            compileProgram(input, files[1]);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

8.2 Campbell file to be compiled

```
fun void putc(int c)
    unsafe
        c
        \Write RegA (Addr 0x1000000)

fun void writeNumRecursive(int num)
    if num > 0
        int digit
        writeNumRecursive(num / 10)
        digit = num - (num / 10) * 10
        putc(digit + 48)

fun void writeNum(int num)
    if num > 0
        writeNumRecursive(num)
    else
        if num < 0
            putc(45)
            writeNumRecursive(-num)
        else
            putc(48)
    putc(10)

fun bool isPrime(int n)
    int i
    i = 2

    while i < n
        if n % i == 0
            return false
        i = i + 1

        return true

int n
n = 65521

if isPrime(n)
    writeNum(1)
else
    writeNum(0)
```

8.3 Generated Haskell code

```
import Sprockell.System

prog = [Nop,
    -- 1
    Const (1) RegA,
    -- 2
    Store RegA (Addr 0),
    -- 3
    Jump (Abs 1278),
    -- 4
    Load (Addr 0) RegA,
    -- 5
    Const (3) RegB,
    -- 6
    Compute Add RegA RegB RegB,
    -- 7
    Store RegB (Addr 0),
    -- 8
    Push SP,
    -- 9
    Pop RegB,
    -- 10
    Const (1) RegC,
    -- 11
    Compute Add RegA RegC SP, -- >>> ENTER SCOPE
    -- 12
    Store RegB (Deref SP),
    -- 13
    Load (Addr 0) RegA,
    -- 14
    Const (3) RegB,
    -- 15
    Compute Add RegA RegB RegB,
    -- 16
    Store RegB (Addr 0),
    -- 17
    Push SP,
    -- 18
    Pop RegB,
    -- 19
    Const (1) RegC,
    -- 20
    Compute Add RegA RegC SP, -- >>> ENTER SCOPE
    -- 21
    Store RegB (Deref SP),
    -- 22
```

```
Push SP,
-- 23
Pop RegB,
-- 24
Const (0) RegA,
-- 25
Compute Add RegB RegA RegB,
-- 26
Const (2) RegA,
-- 27
Branch RegA (Abs 29),
-- 28
Jump (Abs 33),
-- 29
Load (Deref RegB) RegB,
-- 30
Const (1) RegC,
-- 31
Compute Sub RegA RegC RegA,
-- 32
Jump (Abs 27),
-- 33
Const (3) RegA,
-- 34
Compute Add RegB RegA RegA,
-- 35
Load (Deref RegA) RegA,
-- 36
Push RegA, -- SP: c
-- 37
Pop Zero,
-- 38
Write RegA (Addr 0x1000000), -- Raw Haskell
-- 39
Load (Deref SP) SP,
-- 40
Load (Deref SP) SP,
-- 41
Const (0) RegA,
-- 42
Push RegA, -- Const: 0
-- 43
Pop RegA, -- Return value
-- 44
Push SP,
-- 45
```

```
Pop RegC,
-- 46
Const (0) RegD,
-- 47
Branch RegD (Abs 49),
-- 48
Jump (Abs 53),
-- 49
Load (Deref RegC) RegC,
-- 50
Const (1) RegB,
-- 51
Compute Sub RegD RegB RegD,
-- 52
Jump (Abs 47),
-- 53
Const (2) RegD,
-- 54
Compute Add RegC RegD RegC,
-- 55
Load (Deref RegC) RegB, -- Return address
-- 56
Push SP,
-- 57
Pop RegC,
-- 58
Const (0) RegD,
-- 59
Branch RegD (Abs 61),
-- 60
Jump (Abs 65),
-- 61
Load (Deref RegC) RegC,
-- 62
Const (1) RegE,
-- 63
Compute Sub RegD RegE RegD,
-- 64
Jump (Abs 59),
-- 65
Const (1) RegD,
-- 66
Compute Add RegC RegD RegC,
-- 67
Load (Deref RegC) SP, -- Restore SP
-- 68
```

```
Jump (Ind RegB), -- Jump to return address
-- 69
Load (Addr 0) RegA,
-- 70
Const (6) RegB,
-- 71
Compute Add RegA RegB RegB,
-- 72
Store RegB (Addr 0),
-- 73
Push SP,
-- 74
Pop RegB,
-- 75
Const (4) RegC,
-- 76
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 77
Store RegB (Deref SP),
-- 78
Push SP,
-- 79
Pop RegB,
-- 80
Const (0) RegA,
-- 81
Compute Add RegB RegA RegB,
-- 82
Const (1) RegA,
-- 83
Branch RegA (Abs 85),
-- 84
Jump (Abs 89),
-- 85
Load (Deref RegB) RegB,
-- 86
Const (1) RegC,
-- 87
Compute Sub RegA RegC RegA,
-- 88
Jump (Abs 83),
-- 89
Const (3) RegA,
-- 90
Compute Add RegB RegA RegA,
-- 91
```

```
Load (Deref RegA) RegA,
-- 92
Push RegA, -- SP: num
-- 93
Const (0) RegA,
-- 94
Push RegA, -- Const: 0
-- 95
Pop RegB,
-- 96
Pop RegA,
-- 97
Compute Gt RegA RegB RegA, -- compute GreaterThan
-- 98
Push RegA,
-- 99
Pop RegA,
-- 100
Branch RegA (Abs 102), -- if true
-- 101
Jump (Abs 379), -- if false
-- 102
Load (Addr 0) RegA,
-- 103
Const (6) RegB,
-- 104
Compute Add RegA RegB RegB,
-- 105
Store RegB (Addr 0),
-- 106
Push SP,
-- 107
Pop RegB,
-- 108
Const (4) RegC,
-- 109
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 110
Store RegB (Deref SP),
-- 111
Load (Addr 0) RegA,
-- 112
Const (7) RegB,
-- 113
Compute Add RegA RegB RegB,
-- 114
```

```
Store RegB (Addr 0),
-- 115
Push SP,
-- 116
Pop RegB,
-- 117
Const (4) RegC,
-- 118
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 119
Store RegB (Deref SP),
-- 120
Push SP,
-- 121
Pop RegB,
-- 122
Const (0) RegA,
-- 123
Compute Add RegB RegA RegB,
-- 124
Const (3) RegA,
-- 125
Branch RegA (Abs 127),
-- 126
Jump (Abs 131),
-- 127
Load (Deref RegB) RegB,
-- 128
Const (1) RegC,
-- 129
Compute Sub RegA RegC RegA,
-- 130
Jump (Abs 125),
-- 131
Const (3) RegA,
-- 132
Compute Add RegB RegA RegA,
-- 133
Load (Deref RegA) RegA,
-- 134
Push RegA, -- SP: num
-- 135
Const (10) RegA,
-- 136
Push RegA, -- Const: 10
-- 137
```

```
Pop RegB,
-- 138
Pop RegA,
-- 139
Compute Div RegA RegB RegA, -- compute Divide
-- 140
Push RegA,
-- 141
Load (Addr 0) RegA,
-- 142
Const (3) RegB,
-- 143
Compute Add RegA RegB RegB,
-- 144
Store RegB (Addr 0),
-- 145
Push RegA,
-- 146
Const (0) RegB,
-- 147
Store RegB (Deref RegA),
-- 148
Const (1) RegB,
-- 149
Compute Add RegA RegB RegA,
-- 150
Const (69) RegB,
-- 151
Store RegB (Deref RegA),
-- 152
Const (1) RegB,
-- 153
Compute Add RegA RegB RegA,
-- 154
Push SP,
-- 155
Pop RegC,
-- 156
Const (2) RegD,
-- 157
Compute Add RegC RegD RegC,
-- 158
Const (4) RegB,
-- 159
Branch RegB (Abs 161),
-- 160
```

```
Jump (Abs 165),
-- 161
Load (Deref RegC) RegC,
-- 162
Const (1) RegD,
-- 163
Compute Sub RegB RegD RegB,
-- 164
Jump (Abs 159),
-- 165
Store RegC (Deref RegA),
-- 166
Pop RegA, -- Actual call
-- 167
Load (Deref RegA) RegB,
-- 168
Const (4) RegC,
-- 169
Compute Add RegB RegC RegB,
-- 170
Load (Addr 0) RegC,
-- 171
Compute Add RegC RegB RegB,
-- 172
Store RegB (Addr 0),
-- 173
Push RegC,
-- 174
Pop RegB,
-- 175
Const (2) RegD,
-- 176
Compute Add RegA RegD RegD,
-- 177
Load (Deref RegD) RegD,
-- 178
Store RegD (Deref RegB),
-- 179
Const (1) RegD,
-- 180
Compute Add RegB RegD RegB,
-- 181
Const (1) RegD,
-- 182
Compute Add SP RegD RegD,
-- 183
```

```
Store RegD (Deref RegB),
-- 184
Const (1) RegD,
-- 185
Compute Add RegB RegD RegB,
-- 186
Const (214) RegD, -- Calculate return address
-- 187
Store RegD (Deref RegB),
-- 188
Const (1) RegD,
-- 189
Compute Add RegB RegD RegB,
-- 190
Pop RegE,
-- 191
Store RegE (Deref RegB),
-- 192
Const (1) RegE,
-- 193
Compute Add RegB RegE RegB,
-- 194
Load (Deref RegA) RegD,
-- 195
Push RegA,
-- 196
Const (3) RegE,
-- 197
Compute Add RegA RegE RegA,
-- 198
Branch RegD (Rel (2)),
-- 199
Jump (Rel (8)),
-- 200
Load (Deref RegA) RegE,
-- 201
Store RegE (Deref RegB),
-- 202
Const (1) RegE,
-- 203
Compute Add RegA RegE RegA,
-- 204
Compute Add RegB RegE RegB,
-- 205
Compute Sub RegD RegE RegD,
-- 206
```

```
Jump (Rel (-8)),
-- 207
Pop RegD,
-- 208
Push RegC,
-- 209
Pop SP,
-- 210
Const (1) RegE,
-- 211
Compute Add RegE RegD RegD,
-- 212
Load (Deref RegD) RegD,
-- 213
Jump (Ind RegD),
-- 214
Push RegA,
-- 215
Pop Zero,
-- 216
Push SP,
-- 217
Pop RegB,
-- 218
Const (0) RegA,
-- 219
Compute Add RegB RegA RegB,
-- 220
Const (0) RegA,
-- 221
Branch RegA (Abs 223),
-- 222
Jump (Abs 227),
-- 223
Load (Deref RegB) RegB,
-- 224
Const (1) RegC,
-- 225
Compute Sub RegA RegC RegA,
-- 226
Jump (Abs 221),
-- 227
Const (1) RegA,
-- 228
Compute Add RegB RegA RegA,
-- 229
```

```
Nop,
-- 230
Push RegA, -- SP-ref: digit
-- 231
Push SP,
-- 232
Pop RegB,
-- 233
Const (1) RegA,
-- 234
Compute Add RegB RegA RegB,
-- 235
Const (3) RegA,
-- 236
Branch RegA (Abs 238),
-- 237
Jump (Abs 242),
-- 238
Load (Deref RegB) RegB,
-- 239
Const (1) RegC,
-- 240
Compute Sub RegA RegC RegA,
-- 241
Jump (Abs 236),
-- 242
Const (3) RegA,
-- 243
Compute Add RegB RegA RegA,
-- 244
Load (Deref RegA) RegA,
-- 245
Push RegA, -- SP: num
-- 246
Push SP,
-- 247
Pop RegB,
-- 248
Const (2) RegA,
-- 249
Compute Add RegB RegA RegB,
-- 250
Const (3) RegA,
-- 251
Branch RegA (Abs 253),
-- 252
```

```
Jump (Abs 257),
-- 253
Load (Deref RegB) RegB,
-- 254
Const (1) RegC,
-- 255
Compute Sub RegA RegC RegA,
-- 256
Jump (Abs 251),
-- 257
Const (3) RegA,
-- 258
Compute Add RegB RegA RegA,
-- 259
Load (Deref RegA) RegA,
-- 260
Push RegA, -- SP: num
-- 261
Const (10) RegA,
-- 262
Push RegA, -- Const: 10
-- 263
Pop RegB,
-- 264
Pop RegA,
-- 265
Compute Div RegA RegB RegA, -- compute Divide
-- 266
Push RegA,
-- 267
Const (10) RegA,
-- 268
Push RegA, -- Const: 10
-- 269
Pop RegB,
-- 270
Pop RegA,
-- 271
Compute Mul RegA RegB RegA, -- compute Multiply
-- 272
Push RegA,
-- 273
Pop RegB,
-- 274
Pop RegA,
-- 275
```

```
Compute Sub RegA RegB RegA, -- compute Subtract
-- 276
Push RegA,
-- 277
Pop RegA,
-- 278
Pop RegB,
-- 279
Store RegA (Deref RegB), -- assign
-- 280
Push SP,
-- 281
Pop RegB,
-- 282
Const (0) RegA,
-- 283
Compute Add RegB RegA RegB,
-- 284
Const (0) RegA,
-- 285
Branch RegA (Abs 287),
-- 286
Jump (Abs 291),
-- 287
Load (Deref RegB) RegB,
-- 288
Const (1) RegC,
-- 289
Compute Sub RegA RegC RegA,
-- 290
Jump (Abs 285),
-- 291
Const (1) RegA,
-- 292
Compute Add RegB RegA RegA,
-- 293
Load (Deref RegA) RegA,
-- 294
Push RegA, -- SP: digit
-- 295
Const (48) RegA,
-- 296
Push RegA, -- Const: 48
-- 297
Pop RegB,
-- 298
```

```
Pop RegA,
-- 299
Compute Add RegA RegB RegA, -- compute Add
-- 300
Push RegA,
-- 301
Load (Addr 0) RegA,
-- 302
Const (3) RegB,
-- 303
Compute Add RegA RegB RegB,
-- 304
Store RegB (Addr 0),
-- 305
Push RegA,
-- 306
Const (0) RegB,
-- 307
Store RegB (Deref RegA),
-- 308
Const (1) RegB,
-- 309
Compute Add RegA RegB RegA,
-- 310
Const (4) RegB,
-- 311
Store RegB (Deref RegA),
-- 312
Const (1) RegB,
-- 313
Compute Add RegA RegB RegA,
-- 314
Push SP,
-- 315
Pop RegC,
-- 316
Const (2) RegD,
-- 317
Compute Add RegC RegD RegC,
-- 318
Const (4) RegB,
-- 319
Branch RegB (Abs 321),
-- 320
Jump (Abs 325),
-- 321
```

```
Load (Deref RegC) RegC,
-- 322
Const (1) RegD,
-- 323
Compute Sub RegB RegD RegB,
-- 324
Jump (Abs 319),
-- 325
Store RegC (Deref RegA),
-- 326
Pop RegA, -- Actual call
-- 327
Load (Deref RegA) RegB,
-- 328
Const (4) RegC,
-- 329
Compute Add RegB RegC RegB,
-- 330
Load (Addr 0) RegC,
-- 331
Compute Add RegC RegB RegB,
-- 332
Store RegB (Addr 0),
-- 333
Push RegC,
-- 334
Pop RegB,
-- 335
Const (2) RegD,
-- 336
Compute Add RegA RegD RegD,
-- 337
Load (Deref RegD) RegD,
-- 338
Store RegD (Deref RegB),
-- 339
Const (1) RegD,
-- 340
Compute Add RegB RegD RegB,
-- 341
Const (1) RegD,
-- 342
Compute Add SP RegD RegD,
-- 343
Store RegD (Deref RegB),
-- 344
```

```
Const (1) RegD,
-- 345
Compute Add RegB RegD RegB,
-- 346
Const (374) RegD, -- Calculate return address
-- 347
Store RegD (Deref RegB),
-- 348
Const (1) RegD,
-- 349
Compute Add RegB RegD RegB,
-- 350
Pop RegE,
-- 351
Store RegE (Deref RegB),
-- 352
Const (1) RegE,
-- 353
Compute Add RegB RegE RegB,
-- 354
Load (Deref RegA) RegD,
-- 355
Push RegA,
-- 356
Const (3) RegE,
-- 357
Compute Add RegA RegE RegA,
-- 358
Branch RegD (Rel (2)),
-- 359
Jump (Rel (8)),
-- 360
Load (Deref RegA) RegE,
-- 361
Store RegE (Deref RegB),
-- 362
Const (1) RegE,
-- 363
Compute Add RegA RegE RegA,
-- 364
Compute Add RegB RegE RegB,
-- 365
Compute Sub RegD RegE RegD,
-- 366
Jump (Rel (-8)),
-- 367
```

```
Pop RegD,
-- 368
Push RegC,
-- 369
Pop SP,
-- 370
Const (1) RegE,
-- 371
Compute Add RegE RegD RegD,
-- 372
Load (Deref RegD) RegD,
-- 373
Jump (Ind RegD),
-- 374
Push RegA,
-- 375
Pop Zero,
-- 376
Load (Deref SP) SP,
-- 377
Load (Deref SP) SP,
-- 378
Jump (Abs 389), -- then end
-- 379
Load (Addr 0) RegA,
-- 380
Const (3) RegB,
-- 381
Compute Add RegA RegB RegB,
-- 382
Store RegB (Addr 0),
-- 383
Push SP,
-- 384
Pop RegB,
-- 385
Const (1) RegC,
-- 386
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 387
Store RegB (Deref SP),
-- 388
Load (Deref SP) SP,
-- 389
Load (Deref SP) SP,
-- 390
```

```
Const (0) RegA,
-- 391
Push RegA, -- Const: 0
-- 392
Pop RegA, -- Return value
-- 393
Push SP,
-- 394
Pop RegC,
-- 395
Const (0) RegD,
-- 396
Branch RegD (Abs 398),
-- 397
Jump (Abs 402),
-- 398
Load (Deref RegC) RegC,
-- 399
Const (1) RegB,
-- 400
Compute Sub RegD RegB RegD,
-- 401
Jump (Abs 396),
-- 402
Const (2) RegD,
-- 403
Compute Add RegC RegD RegC,
-- 404
Load (Deref RegC) RegB, -- Return address
-- 405
Push SP,
-- 406
Pop RegC,
-- 407
Const (0) RegD,
-- 408
Branch RegD (Abs 410),
-- 409
Jump (Abs 414),
-- 410
Load (Deref RegC) RegC,
-- 411
Const (1) RegE,
-- 412
Compute Sub RegD RegE RegD,
-- 413
```

```
Jump (Abs 408),
-- 414
Const (1) RegD,
-- 415
Compute Add RegC RegD RegC,
-- 416
Load (Deref RegC) SP, -- Restore SP
-- 417
Jump (Ind RegB), -- Jump to return address
-- 418
Load (Addr 0) RegA,
-- 419
Const (5) RegB,
-- 420
Compute Add RegA RegB RegB,
-- 421
Store RegB (Addr 0),
-- 422
Push SP,
-- 423
Pop RegB,
-- 424
Const (3) RegC,
-- 425
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 426
Store RegB (Deref SP),
-- 427
Push SP,
-- 428
Pop RegB,
-- 429
Const (0) RegA,
-- 430
Compute Add RegB RegA RegB,
-- 431
Const (1) RegA,
-- 432
Branch RegA (Abs 434),
-- 433
Jump (Abs 438),
-- 434
Load (Deref RegB) RegB,
-- 435
Const (1) RegC,
-- 436
```

```
Compute Sub RegA RegC RegA,
-- 437
Jump (Abs 432),
-- 438
Const (3) RegA,
-- 439
Compute Add RegB RegA RegA,
-- 440
Load (Deref RegA) RegA,
-- 441
Push RegA, -- SP: num
-- 442
Const (0) RegA,
-- 443
Push RegA, -- Const: 0
-- 444
Pop RegB,
-- 445
Pop RegA,
-- 446
Compute Gt RegA RegB RegA, -- compute GreaterThan
-- 447
Push RegA,
-- 448
Pop RegA,
-- 449
Branch RegA (Abs 451), -- if true
-- 450
Jump (Abs 562), -- if false
-- 451
Load (Addr 0) RegA,
-- 452
Const (5) RegB,
-- 453
Compute Add RegA RegB RegB,
-- 454
Store RegB (Addr 0),
-- 455
Push SP,
-- 456
Pop RegB,
-- 457
Const (3) RegC,
-- 458
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 459
```

```
Store RegB (Deref SP),
-- 460
Load (Addr 0) RegA,
-- 461
Const (5) RegB,
-- 462
Compute Add RegA RegB RegB,
-- 463
Store RegB (Addr 0),
-- 464
Push SP,
-- 465
Pop RegB,
-- 466
Const (3) RegC,
-- 467
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 468
Store RegB (Deref SP),
-- 469
Push SP,
-- 470
Pop RegB,
-- 471
Const (0) RegA,
-- 472
Compute Add RegB RegA RegB,
-- 473
Const (3) RegA,
-- 474
Branch RegA (Abs 476),
-- 475
Jump (Abs 480),
-- 476
Load (Deref RegB) RegB,
-- 477
Const (1) RegC,
-- 478
Compute Sub RegA RegC RegA,
-- 479
Jump (Abs 474),
-- 480
Const (3) RegA,
-- 481
Compute Add RegB RegA RegA,
-- 482
```

```
Load (Deref RegA) RegA,
-- 483
Push RegA, -- SP: num
-- 484
Load (Addr 0) RegA,
-- 485
Const (3) RegB,
-- 486
Compute Add RegA RegB RegB,
-- 487
Store RegB (Addr 0),
-- 488
Push RegA,
-- 489
Const (0) RegB,
-- 490
Store RegB (Deref RegA),
-- 491
Const (1) RegB,
-- 492
Compute Add RegA RegB RegA,
-- 493
Const (69) RegB,
-- 494
Store RegB (Deref RegA),
-- 495
Const (1) RegB,
-- 496
Compute Add RegA RegB RegA,
-- 497
Push SP,
-- 498
Pop RegC,
-- 499
Const (2) RegD,
-- 500
Compute Add RegC RegD RegC,
-- 501
Const (4) RegB,
-- 502
Branch RegB (Abs 504),
-- 503
Jump (Abs 508),
-- 504
Load (Deref RegC) RegC,
-- 505
```

```
Const (1) RegD,
-- 506
Compute Sub RegB RegD RegB,
-- 507
Jump (Abs 502),
-- 508
Store RegC (Deref RegA),
-- 509
Pop RegA, -- Actual call
-- 510
Load (Deref RegA) RegB,
-- 511
Const (4) RegC,
-- 512
Compute Add RegB RegC RegB,
-- 513
Load (Addr 0) RegC,
-- 514
Compute Add RegC RegB RegB,
-- 515
Store RegB (Addr 0),
-- 516
Push RegC,
-- 517
Pop RegB,
-- 518
Const (2) RegD,
-- 519
Compute Add RegA RegD RegD,
-- 520
Load (Deref RegD) RegD,
-- 521
Store RegD (Deref RegB),
-- 522
Const (1) RegD,
-- 523
Compute Add RegB RegD RegB,
-- 524
Const (1) RegD,
-- 525
Compute Add SP RegD RegD,
-- 526
Store RegD (Deref RegB),
-- 527
Const (1) RegD,
-- 528
```

```
Compute Add RegB RegD RegB,
-- 529
Const (557) RegD, -- Calculate return address
-- 530
Store RegD (Deref RegB),
-- 531
Const (1) RegD,
-- 532
Compute Add RegB RegD RegB,
-- 533
Pop RegE,
-- 534
Store RegE (Deref RegB),
-- 535
Const (1) RegE,
-- 536
Compute Add RegB RegE RegB,
-- 537
Load (Deref RegA) RegD,
-- 538
Push RegA,
-- 539
Const (3) RegE,
-- 540
Compute Add RegA RegE RegA,
-- 541
Branch RegD (Rel (2)),
-- 542
Jump (Rel (8)),
-- 543
Load (Deref RegA) RegE,
-- 544
Store RegE (Deref RegB),
-- 545
Const (1) RegE,
-- 546
Compute Add RegA RegE RegA,
-- 547
Compute Add RegB RegE RegB,
-- 548
Compute Sub RegD RegE RegD,
-- 549
Jump (Rel (-8)),
-- 550
Pop RegD,
-- 551
```

```
Push RegC,
-- 552
Pop SP,
-- 553
Const (1) RegE,
-- 554
Compute Add RegE RegD RegD,
-- 555
Load (Deref RegD) RegD,
-- 556
Jump (Ind RegD),
-- 557
Push RegA,
-- 558
Pop Zero,
-- 559
Load (Deref SP) SP,
-- 560
Load (Deref SP) SP,
-- 561
Jump (Abs 897), -- then end
-- 562
Load (Addr 0) RegA,
-- 563
Const (5) RegB,
-- 564
Compute Add RegA RegB RegB,
-- 565
Store RegB (Addr 0),
-- 566
Push SP,
-- 567
Pop RegB,
-- 568
Const (3) RegC,
-- 569
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 570
Store RegB (Deref SP),
-- 571
Load (Addr 0) RegA,
-- 572
Const (5) RegB,
-- 573
Compute Add RegA RegB RegB,
-- 574
```

```
Store RegB (Addr 0),
-- 575
Push SP,
-- 576
Pop RegB,
-- 577
Const (3) RegC,
-- 578
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 579
Store RegB (Deref SP),
-- 580
Push SP,
-- 581
Pop RegB,
-- 582
Const (0) RegA,
-- 583
Compute Add RegB RegA RegB,
-- 584
Const (3) RegA,
-- 585
Branch RegA (Abs 587),
-- 586
Jump (Abs 591),
-- 587
Load (Deref RegB) RegB,
-- 588
Const (1) RegC,
-- 589
Compute Sub RegA RegC RegA,
-- 590
Jump (Abs 585),
-- 591
Const (3) RegA,
-- 592
Compute Add RegB RegA RegA,
-- 593
Load (Deref RegA) RegA,
-- 594
Push RegA, -- SP: num
-- 595
Const (0) RegA,
-- 596
Push RegA, -- Const: 0
-- 597
```

```
Pop RegB,
-- 598
Pop RegA,
-- 599
Compute Lt RegA RegB RegA, -- compute LessThan
-- 600
Push RegA,
-- 601
Pop RegA,
-- 602
Branch RegA (Abs 604), -- if true
-- 603
Jump (Abs 798), -- if false
-- 604
Load (Addr 0) RegA,
-- 605
Const (5) RegB,
-- 606
Compute Add RegA RegB RegB,
-- 607
Store RegB (Addr 0),
-- 608
Push SP,
-- 609
Pop RegB,
-- 610
Const (3) RegC,
-- 611
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 612
Store RegB (Deref SP),
-- 613
Load (Addr 0) RegA,
-- 614
Const (5) RegB,
-- 615
Compute Add RegA RegB RegB,
-- 616
Store RegB (Addr 0),
-- 617
Push SP,
-- 618
Pop RegB,
-- 619
Const (3) RegC,
-- 620
```

```
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 621
Store RegB (Deref SP),
-- 622
Const (45) RegA,
-- 623
Push RegA, -- Const: 45
-- 624
Load (Addr 0) RegA,
-- 625
Const (3) RegB,
-- 626
Compute Add RegA RegB RegB,
-- 627
Store RegB (Addr 0),
-- 628
Push RegA,
-- 629
Const (0) RegB,
-- 630
Store RegB (Deref RegA),
-- 631
Const (1) RegB,
-- 632
Compute Add RegA RegB RegA,
-- 633
Const (4) RegB,
-- 634
Store RegB (Deref RegA),
-- 635
Const (1) RegB,
-- 636
Compute Add RegA RegB RegA,
-- 637
Push SP,
-- 638
Pop RegC,
-- 639
Const (2) RegD,
-- 640
Compute Add RegC RegD RegC,
-- 641
Const (6) RegB,
-- 642
Branch RegB (Abs 644),
-- 643
```

```
Jump (Abs 648),
-- 644
Load (Deref RegC) RegC,
-- 645
Const (1) RegD,
-- 646
Compute Sub RegB RegD RegB,
-- 647
Jump (Abs 642),
-- 648
Store RegC (Deref RegA),
-- 649
Pop RegA, -- Actual call
-- 650
Load (Deref RegA) RegB,
-- 651
Const (4) RegC,
-- 652
Compute Add RegB RegC RegB,
-- 653
Load (Addr 0) RegC,
-- 654
Compute Add RegC RegB RegB,
-- 655
Store RegB (Addr 0),
-- 656
Push RegC,
-- 657
Pop RegB,
-- 658
Const (2) RegD,
-- 659
Compute Add RegA RegD RegD,
-- 660
Load (Deref RegD) RegD,
-- 661
Store RegD (Deref RegB),
-- 662
Const (1) RegD,
-- 663
Compute Add RegB RegD RegB,
-- 664
Const (1) RegD,
-- 665
Compute Add SP RegD RegD,
-- 666
```

```
Store RegD (Deref RegB),
-- 667
Const (1) RegD,
-- 668
Compute Add RegB RegD RegB,
-- 669
Const (697) RegD, -- Calculate return address
-- 670
Store RegD (Deref RegB),
-- 671
Const (1) RegD,
-- 672
Compute Add RegB RegD RegB,
-- 673
Pop RegE,
-- 674
Store RegE (Deref RegB),
-- 675
Const (1) RegE,
-- 676
Compute Add RegB RegE RegB,
-- 677
Load (Deref RegA) RegD,
-- 678
Push RegA,
-- 679
Const (3) RegE,
-- 680
Compute Add RegA RegE RegA,
-- 681
Branch RegD (Rel (2)),
-- 682
Jump (Rel (8)),
-- 683
Load (Deref RegA) RegE,
-- 684
Store RegE (Deref RegB),
-- 685
Const (1) RegE,
-- 686
Compute Add RegA RegE RegA,
-- 687
Compute Add RegB RegE RegB,
-- 688
Compute Sub RegD RegE RegD,
-- 689
```

```
Jump (Rel (-8)),
-- 690
Pop RegD,
-- 691
Push RegC,
-- 692
Pop SP,
-- 693
Const (1) RegE,
-- 694
Compute Add RegE RegD RegD,
-- 695
Load (Deref RegD) RegD,
-- 696
Jump (Ind RegD),
-- 697
Push RegA,
-- 698
Pop Zero,
-- 699
Const (0) RegA,
-- 700
Push RegA, -- Const: 0
-- 701
Push SP,
-- 702
Pop RegB,
-- 703
Const (1) RegA,
-- 704
Compute Add RegB RegA RegB,
-- 705
Const (5) RegA,
-- 706
Branch RegA (Abs 708),
-- 707
Jump (Abs 712),
-- 708
Load (Deref RegB) RegB,
-- 709
Const (1) RegC,
-- 710
Compute Sub RegA RegC RegA,
-- 711
Jump (Abs 706),
-- 712
```

```
Const (3) RegA,
-- 713
Compute Add RegB RegA RegA,
-- 714
Load (Deref RegA) RegA,
-- 715
Push RegA, -- SP: num
-- 716
Pop RegB,
-- 717
Pop RegA,
-- 718
Compute Sub RegA RegB RegA, -- compute Subtract
-- 719
Push RegA,
-- 720
Load (Addr 0) RegA,
-- 721
Const (3) RegB,
-- 722
Compute Add RegA RegB RegB,
-- 723
Store RegB (Addr 0),
-- 724
Push RegA,
-- 725
Const (0) RegB,
-- 726
Store RegB (Deref RegA),
-- 727
Const (1) RegB,
-- 728
Compute Add RegA RegB RegA,
-- 729
Const (69) RegB,
-- 730
Store RegB (Deref RegA),
-- 731
Const (1) RegB,
-- 732
Compute Add RegA RegB RegA,
-- 733
Push SP,
-- 734
Pop RegC,
-- 735
```

```
Const (2) RegD,
-- 736
Compute Add RegC RegD RegC,
-- 737
Const (6) RegB,
-- 738
Branch RegB (Abs 740),
-- 739
Jump (Abs 744),
-- 740
Load (Deref RegC) RegC,
-- 741
Const (1) RegD,
-- 742
Compute Sub RegB RegD RegB,
-- 743
Jump (Abs 738),
-- 744
Store RegC (Deref RegA),
-- 745
Pop RegA, -- Actual call
-- 746
Load (Deref RegA) RegB,
-- 747
Const (4) RegC,
-- 748
Compute Add RegB RegC RegB,
-- 749
Load (Addr 0) RegC,
-- 750
Compute Add RegC RegB RegB,
-- 751
Store RegB (Addr 0),
-- 752
Push RegC,
-- 753
Pop RegB,
-- 754
Const (2) RegD,
-- 755
Compute Add RegA RegD RegD,
-- 756
Load (Deref RegD) RegD,
-- 757
Store RegD (Deref RegB),
-- 758
```

```
Const (1) RegD,
-- 759
Compute Add RegB RegD RegB,
-- 760
Const (1) RegD,
-- 761
Compute Add SP RegD RegD,
-- 762
Store RegD (Deref RegB),
-- 763
Const (1) RegD,
-- 764
Compute Add RegB RegD RegB,
-- 765
Const (793) RegD, -- Calculate return address
-- 766
Store RegD (Deref RegB),
-- 767
Const (1) RegD,
-- 768
Compute Add RegB RegD RegB,
-- 769
Pop RegE,
-- 770
Store RegE (Deref RegB),
-- 771
Const (1) RegE,
-- 772
Compute Add RegB RegE RegB,
-- 773
Load (Deref RegA) RegD,
-- 774
Push RegA,
-- 775
Const (3) RegE,
-- 776
Compute Add RegA RegE RegA,
-- 777
Branch RegD (Rel (2)),
-- 778
Jump (Rel (8)),
-- 779
Load (Deref RegA) RegE,
-- 780
Store RegE (Deref RegB),
-- 781
```

```
Const (1) RegE,
-- 782
Compute Add RegA RegE RegA,
-- 783
Compute Add RegB RegE RegB,
-- 784
Compute Sub RegD RegE RegD,
-- 785
Jump (Rel (-8)),
-- 786
Pop RegD,
-- 787
Push RegC,
-- 788
Pop SP,
-- 789
Const (1) RegE,
-- 790
Compute Add RegE RegD RegD,
-- 791
Load (Deref RegD) RegD,
-- 792
Jump (Ind RegD),
-- 793
Push RegA,
-- 794
Pop Zero,
-- 795
Load (Deref SP) SP,
-- 796
Load (Deref SP) SP,
-- 797
Jump (Abs 895), -- then end
-- 798
Load (Addr 0) RegA,
-- 799
Const (5) RegB,
-- 800
Compute Add RegA RegB RegB,
-- 801
Store RegB (Addr 0),
-- 802
Push SP,
-- 803
Pop RegB,
-- 804
```

```
Const (3) RegC,
-- 805
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 806
Store RegB (Deref SP),
-- 807
Load (Addr 0) RegA,
-- 808
Const (5) RegB,
-- 809
Compute Add RegA RegB RegB,
-- 810
Store RegB (Addr 0),
-- 811
Push SP,
-- 812
Pop RegB,
-- 813
Const (3) RegC,
-- 814
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 815
Store RegB (Deref SP),
-- 816
Const (48) RegA,
-- 817
Push RegA, -- Const: 48
-- 818
Load (Addr 0) RegA,
-- 819
Const (3) RegB,
-- 820
Compute Add RegA RegB RegB,
-- 821
Store RegB (Addr 0),
-- 822
Push RegA,
-- 823
Const (0) RegB,
-- 824
Store RegB (Deref RegA),
-- 825
Const (1) RegB,
-- 826
Compute Add RegA RegB RegA,
-- 827
```

```
Const (4) RegB,
-- 828
Store RegB (Deref RegA),
-- 829
Const (1) RegB,
-- 830
Compute Add RegA RegB RegA,
-- 831
Push SP,
-- 832
Pop RegC,
-- 833
Const (2) RegD,
-- 834
Compute Add RegC RegD RegC,
-- 835
Const (6) RegB,
-- 836
Branch RegB (Abs 838),
-- 837
Jump (Abs 842),
-- 838
Load (Deref RegC) RegC,
-- 839
Const (1) RegD,
-- 840
Compute Sub RegB RegD RegB,
-- 841
Jump (Abs 836),
-- 842
Store RegC (Deref RegA),
-- 843
Pop RegA, -- Actual call
-- 844
Load (Deref RegA) RegB,
-- 845
Const (4) RegC,
-- 846
Compute Add RegB RegC RegB,
-- 847
Load (Addr 0) RegC,
-- 848
Compute Add RegC RegB RegB,
-- 849
Store RegB (Addr 0),
-- 850
```

```
Push RegC,
-- 851
Pop RegB,
-- 852
Const (2) RegD,
-- 853
Compute Add RegA RegD RegD,
-- 854
Load (Deref RegD) RegD,
-- 855
Store RegD (Deref RegB),
-- 856
Const (1) RegD,
-- 857
Compute Add RegB RegD RegB,
-- 858
Const (1) RegD,
-- 859
Compute Add SP RegD RegD,
-- 860
Store RegD (Deref RegB),
-- 861
Const (1) RegD,
-- 862
Compute Add RegB RegD RegB,
-- 863
Const (891) RegD, -- Calculate return address
-- 864
Store RegD (Deref RegB),
-- 865
Const (1) RegD,
-- 866
Compute Add RegB RegD RegB,
-- 867
Pop RegE,
-- 868
Store RegE (Deref RegB),
-- 869
Const (1) RegE,
-- 870
Compute Add RegB RegE RegB,
-- 871
Load (Deref RegA) RegD,
-- 872
Push RegA,
-- 873
```

```
Const (3) RegE,
-- 874
Compute Add RegA RegE RegA,
-- 875
Branch RegD (Rel (2)),
-- 876
Jump (Rel (8)),
-- 877
Load (Deref RegA) RegE,
-- 878
Store RegE (Deref RegB),
-- 879
Const (1) RegE,
-- 880
Compute Add RegA RegE RegA,
-- 881
Compute Add RegB RegE RegB,
-- 882
Compute Sub RegD RegE RegD,
-- 883
Jump (Rel (-8)),
-- 884
Pop RegD,
-- 885
Push RegC,
-- 886
Pop SP,
-- 887
Const (1) RegE,
-- 888
Compute Add RegE RegD RegD,
-- 889
Load (Deref RegD) RegD,
-- 890
Jump (Ind RegD),
-- 891
Push RegA,
-- 892
Pop Zero,
-- 893
Load (Deref SP) SP,
-- 894
Load (Deref SP) SP,
-- 895
Load (Deref SP) SP,
-- 896
```

```
Load (Deref SP) SP,
-- 897
Const (10) RegA,
-- 898
Push RegA, -- Const: 10
-- 899
Load (Addr 0) RegA,
-- 900
Const (3) RegB,
-- 901
Compute Add RegA RegB RegB,
-- 902
Store RegB (Addr 0),
-- 903
Push RegA,
-- 904
Const (0) RegB,
-- 905
Store RegB (Deref RegA),
-- 906
Const (1) RegB,
-- 907
Compute Add RegA RegB RegA,
-- 908
Const (4) RegB,
-- 909
Store RegB (Deref RegA),
-- 910
Const (1) RegB,
-- 911
Compute Add RegA RegB RegA,
-- 912
Push SP,
-- 913
Pop RegC,
-- 914
Const (2) RegD,
-- 915
Compute Add RegC RegD RegC,
-- 916
Const (2) RegB,
-- 917
Branch RegB (Abs 919),
-- 918
Jump (Abs 923),
-- 919
```

```
Load (Deref RegC) RegC,
-- 920
Const (1) RegD,
-- 921
Compute Sub RegB RegD RegB,
-- 922
Jump (Abs 917),
-- 923
Store RegC (Deref RegA),
-- 924
Pop RegA, -- Actual call
-- 925
Load (Deref RegA) RegB,
-- 926
Const (4) RegC,
-- 927
Compute Add RegB RegC RegB,
-- 928
Load (Addr 0) RegC,
-- 929
Compute Add RegC RegB RegB,
-- 930
Store RegB (Addr 0),
-- 931
Push RegC,
-- 932
Pop RegB,
-- 933
Const (2) RegD,
-- 934
Compute Add RegA RegD RegD,
-- 935
Load (Deref RegD) RegD,
-- 936
Store RegD (Deref RegB),
-- 937
Const (1) RegD,
-- 938
Compute Add RegB RegD RegB,
-- 939
Const (1) RegD,
-- 940
Compute Add SP RegD RegD,
-- 941
Store RegD (Deref RegB),
-- 942
```

```
Const (1) RegD,
-- 943
Compute Add RegB RegD RegB,
-- 944
Const (972) RegD, -- Calculate return address
-- 945
Store RegD (Deref RegB),
-- 946
Const (1) RegD,
-- 947
Compute Add RegB RegD RegB,
-- 948
Pop RegE,
-- 949
Store RegE (Deref RegB),
-- 950
Const (1) RegE,
-- 951
Compute Add RegB RegE RegB,
-- 952
Load (Deref RegA) RegD,
-- 953
Push RegA,
-- 954
Const (3) RegE,
-- 955
Compute Add RegA RegE RegA,
-- 956
Branch RegD (Rel (2)),
-- 957
Jump (Rel (8)),
-- 958
Load (Deref RegA) RegE,
-- 959
Store RegE (Deref RegB),
-- 960
Const (1) RegE,
-- 961
Compute Add RegA RegE RegA,
-- 962
Compute Add RegB RegE RegB,
-- 963
Compute Sub RegD RegE RegD,
-- 964
Jump (Rel (-8)),
-- 965
```

```
Pop RegD,
-- 966
Push RegC,
-- 967
Pop SP,
-- 968
Const (1) RegE,
-- 969
Compute Add RegE RegD RegD,
-- 970
Load (Deref RegD) RegD,
-- 971
Jump (Ind RegD),
-- 972
Push RegA,
-- 973
Pop Zero,
-- 974
Load (Deref SP) SP,
-- 975
Const (0) RegA,
-- 976
Push RegA, -- Const: 0
-- 977
Pop RegA, -- Return value
-- 978
Push SP,
-- 979
Pop RegC,
-- 980
Const (0) RegD,
-- 981
Branch RegD (Abs 983),
-- 982
Jump (Abs 987),
-- 983
Load (Deref RegC) RegC,
-- 984
Const (1) RegB,
-- 985
Compute Sub RegD RegB RegD,
-- 986
Jump (Abs 981),
-- 987
Const (2) RegD,
-- 988
```

```
Compute Add RegC RegD RegC,
-- 989
Load (Deref RegC) RegB, -- Return address
-- 990
Push SP,
-- 991
Pop RegC,
-- 992
Const (0) RegD,
-- 993
Branch RegD (Abs 995),
-- 994
Jump (Abs 999),
-- 995
Load (Deref RegC) RegC,
-- 996
Const (1) RegE,
-- 997
Compute Sub RegD RegE RegD,
-- 998
Jump (Abs 993),
-- 999
Const (1) RegD,
-- 1000
Compute Add RegC RegD RegC,
-- 1001
Load (Deref RegC) SP, -- Restore SP
-- 1002
Jump (Ind RegB), -- Jump to return address
-- 1003
Load (Addr 0) RegA,
-- 1004
Const (6) RegB,
-- 1005
Compute Add RegA RegB RegB,
-- 1006
Store RegB (Addr 0),
-- 1007
Push SP,
-- 1008
Pop RegB,
-- 1009
Const (3) RegC,
-- 1010
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1011
```

```
Store RegB (Deref SP),
-- 1012
Push SP,
-- 1013
Pop RegB,
-- 1014
Const (0) RegA,
-- 1015
Compute Add RegB RegA RegB,
-- 1016
Const (0) RegA,
-- 1017
Branch RegA (Abs 1019),
-- 1018
Jump (Abs 1023),
-- 1019
Load (Deref RegB) RegB,
-- 1020
Const (1) RegC,
-- 1021
Compute Sub RegA RegC RegA,
-- 1022
Jump (Abs 1017),
-- 1023
Const (1) RegA,
-- 1024
Compute Add RegB RegA RegA,
-- 1025
Nop,
-- 1026
Push RegA, -- SP-ref: i
-- 1027
Const (2) RegA,
-- 1028
Push RegA, -- Const: 2
-- 1029
Pop RegA,
-- 1030
Pop RegB,
-- 1031
Store RegA (Deref RegB), -- assign
-- 1032
Push SP,
-- 1033
Pop RegB,
-- 1034
```

```
Const (0) RegA,
-- 1035
Compute Add RegB RegA RegB,
-- 1036
Const (0) RegA,
-- 1037
Branch RegA (Abs 1039),
-- 1038
Jump (Abs 1043),
-- 1039
Load (Deref RegB) RegB,
-- 1040
Const (1) RegC,
-- 1041
Compute Sub RegA RegC RegA,
-- 1042
Jump (Abs 1037),
-- 1043
Const (1) RegA,
-- 1044
Compute Add RegB RegA RegA,
-- 1045
Load (Deref RegA) RegA,
-- 1046
Push RegA, -- SP: i
-- 1047
Push SP,
-- 1048
Pop RegB,
-- 1049
Const (1) RegA,
-- 1050
Compute Add RegB RegA RegB,
-- 1051
Const (1) RegA,
-- 1052
Branch RegA (Abs 1054),
-- 1053
Jump (Abs 1058),
-- 1054
Load (Deref RegB) RegB,
-- 1055
Const (1) RegC,
-- 1056
Compute Sub RegA RegC RegA,
-- 1057
```

```
Jump (Abs 1052),
-- 1058
Const (3) RegA,
-- 1059
Compute Add RegB RegA RegA,
-- 1060
Load (Deref RegA) RegA,
-- 1061
Push RegA, -- SP: n
-- 1062
Pop RegB,
-- 1063
Pop RegA,
-- 1064
Compute Lt RegA RegB RegA, -- compute LessThan
-- 1065
Push RegA,
-- 1066
Pop RegA,
-- 1067
Branch RegA (Abs 1069), -- while true
-- 1068
Jump (Abs 1249), -- while false
-- 1069
Load (Addr 0) RegA,
-- 1070
Const (5) RegB,
-- 1071
Compute Add RegA RegB RegB,
-- 1072
Store RegB (Addr 0),
-- 1073
Push SP,
-- 1074
Pop RegB,
-- 1075
Const (3) RegC,
-- 1076
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1077
Store RegB (Deref SP),
-- 1078
Push SP,
-- 1079
Pop RegB,
-- 1080
```

```
Const (0) RegA,
-- 1081
Compute Add RegB RegA RegB,
-- 1082
Const (2) RegA,
-- 1083
Branch RegA (Abs 1085),
-- 1084
Jump (Abs 1089),
-- 1085
Load (Deref RegB) RegB,
-- 1086
Const (1) RegC,
-- 1087
Compute Sub RegA RegC RegA,
-- 1088
Jump (Abs 1083),
-- 1089
Const (3) RegA,
-- 1090
Compute Add RegB RegA RegA,
-- 1091
Load (Deref RegA) RegA,
-- 1092
Push RegA, -- SP: n
-- 1093
Push SP,
-- 1094
Pop RegB,
-- 1095
Const (1) RegA,
-- 1096
Compute Add RegB RegA RegB,
-- 1097
Const (1) RegA,
-- 1098
Branch RegA (Abs 1100),
-- 1099
Jump (Abs 1104),
-- 1100
Load (Deref RegB) RegB,
-- 1101
Const (1) RegC,
-- 1102
Compute Sub RegA RegC RegA,
-- 1103
```

```
Jump (Abs 1098),
-- 1104
Const (1) RegA,
-- 1105
Compute Add RegB RegA RegA,
-- 1106
Load (Deref RegA) RegA,
-- 1107
Push RegA, -- SP: i
-- 1108
Pop RegB,
-- 1109
Pop RegA,
-- 1110
Compute Mod RegA RegB RegA, -- compute Modulo
-- 1111
Push RegA,
-- 1112
Const (0) RegA,
-- 1113
Push RegA, -- Const: 0
-- 1114
Pop RegB,
-- 1115
Pop RegA,
-- 1116
Compute Equal RegA RegB RegA, -- compute Equals
-- 1117
Push RegA,
-- 1118
Pop RegA,
-- 1119
Branch RegA (Abs 1121), -- if true
-- 1120
Jump (Abs 1170), -- if false
-- 1121
Load (Addr 0) RegA,
-- 1122
Const (3) RegB,
-- 1123
Compute Add RegA RegB RegB,
-- 1124
Store RegB (Addr 0),
-- 1125
Push SP,
-- 1126
```

```
Pop RegB,
-- 1127
Const (1) RegC,
-- 1128
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1129
Store RegB (Deref SP),
-- 1130
Load (Addr 0) RegA,
-- 1131
Const (3) RegB,
-- 1132
Compute Add RegA RegB RegB,
-- 1133
Store RegB (Addr 0),
-- 1134
Push SP,
-- 1135
Pop RegB,
-- 1136
Const (1) RegC,
-- 1137
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1138
Store RegB (Deref SP),
-- 1139
Const (0) RegA,
-- 1140
Push RegA, -- Const: 0
-- 1141
Pop RegA, -- Return value
-- 1142
Push SP,
-- 1143
Pop RegC,
-- 1144
Const (4) RegD,
-- 1145
Branch RegD (Abs 1147),
-- 1146
Jump (Abs 1151),
-- 1147
Load (Deref RegC) RegC,
-- 1148
Const (1) RegB,
-- 1149
```

```
Compute Sub RegD RegB RegD,
-- 1150
Jump (Abs 1145),
-- 1151
Const (2) RegD,
-- 1152
Compute Add RegC RegD RegC,
-- 1153
Load (Deref RegC) RegB, -- Return address
-- 1154
Push SP,
-- 1155
Pop RegC,
-- 1156
Const (4) RegD,
-- 1157
Branch RegD (Abs 1159),
-- 1158
Jump (Abs 1163),
-- 1159
Load (Deref RegC) RegC,
-- 1160
Const (1) RegE,
-- 1161
Compute Sub RegD RegE RegD,
-- 1162
Jump (Abs 1157),
-- 1163
Const (1) RegD,
-- 1164
Compute Add RegC RegD RegC,
-- 1165
Load (Deref RegC) SP, -- Restore SP
-- 1166
Jump (Ind RegB), -- Jump to return address
-- 1167
Load (Deref SP) SP,
-- 1168
Load (Deref SP) SP,
-- 1169
Jump (Abs 1180), -- then end
-- 1170
Load (Addr 0) RegA,
-- 1171
Const (3) RegB,
-- 1172
```

```
Compute Add RegA RegB RegB,
-- 1173
Store RegB (Addr 0),
-- 1174
Push SP,
-- 1175
Pop RegB,
-- 1176
Const (1) RegC,
-- 1177
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1178
Store RegB (Deref SP),
-- 1179
Load (Deref SP) SP,
-- 1180
Push SP,
-- 1181
Pop RegB,
-- 1182
Const (0) RegA,
-- 1183
Compute Add RegB RegA RegB,
-- 1184
Const (1) RegA,
-- 1185
Branch RegA (Abs 1187),
-- 1186
Jump (Abs 1191),
-- 1187
Load (Deref RegB) RegB,
-- 1188
Const (1) RegC,
-- 1189
Compute Sub RegA RegC RegA,
-- 1190
Jump (Abs 1185),
-- 1191
Const (1) RegA,
-- 1192
Compute Add RegB RegA RegA,
-- 1193
Nop,
-- 1194
Push RegA, -- SP-ref: i
-- 1195
```

```
Push SP,
-- 1196
Pop RegB,
-- 1197
Const (1) RegA,
-- 1198
Compute Add RegB RegA RegB,
-- 1199
Const (1) RegA,
-- 1200
Branch RegA (Abs 1202),
-- 1201
Jump (Abs 1206),
-- 1202
Load (Deref RegB) RegB,
-- 1203
Const (1) RegC,
-- 1204
Compute Sub RegA RegC RegA,
-- 1205
Jump (Abs 1200),
-- 1206
Const (1) RegA,
-- 1207
Compute Add RegB RegA RegA,
-- 1208
Load (Deref RegA) RegA,
-- 1209
Push RegA, -- SP: i
-- 1210
Const (1) RegA,
-- 1211
Push RegA, -- Const: 1
-- 1212
Pop RegB,
-- 1213
Pop RegA,
-- 1214
Compute Add RegA RegB RegA, -- compute Add
-- 1215
Push RegA,
-- 1216
Pop RegA,
-- 1217
Pop RegB,
-- 1218
```

```
Store RegA (Deref RegB), -- assign
-- 1219
Const (1) RegA,
-- 1220
Push RegA, -- Const: 1
-- 1221
Pop RegA, -- Return value
-- 1222
Push SP,
-- 1223
Pop RegC,
-- 1224
Const (2) RegD,
-- 1225
Branch RegD (Abs 1227),
-- 1226
Jump (Abs 1231),
-- 1227
Load (Deref RegC) RegC,
-- 1228
Const (1) RegB,
-- 1229
Compute Sub RegD RegB RegD,
-- 1230
Jump (Abs 1225),
-- 1231
Const (2) RegD,
-- 1232
Compute Add RegC RegD RegC,
-- 1233
Load (Deref RegC) RegB, -- Return address
-- 1234
Push SP,
-- 1235
Pop RegC,
-- 1236
Const (2) RegD,
-- 1237
Branch RegD (Abs 1239),
-- 1238
Jump (Abs 1243),
-- 1239
Load (Deref RegC) RegC,
-- 1240
Const (1) RegE,
-- 1241
```

```
Compute Sub RegD RegE RegD,
-- 1242
Jump (Abs 1237),
-- 1243
Const (1) RegD,
-- 1244
Compute Add RegC RegD RegC,
-- 1245
Load (Deref RegC) SP, -- Restore SP
-- 1246
Jump (Ind RegB), -- Jump to return address
-- 1247
Load (Deref SP) SP,
-- 1248
Jump (Abs 1032), -- while end
-- 1249
Load (Deref SP) SP,
-- 1250
Const (0) RegA,
-- 1251
Push RegA, -- Const: 0
-- 1252
Pop RegA, -- Return value
-- 1253
Push SP,
-- 1254
Pop RegC,
-- 1255
Const (0) RegD,
-- 1256
Branch RegD (Abs 1258),
-- 1257
Jump (Abs 1262),
-- 1258
Load (Deref RegC) RegC,
-- 1259
Const (1) RegB,
-- 1260
Compute Sub RegD RegB RegD,
-- 1261
Jump (Abs 1256),
-- 1262
Const (2) RegD,
-- 1263
Compute Add RegC RegD RegC,
-- 1264
```

```
Load (Deref RegC) RegB, -- Return address
-- 1265
Push SP,
-- 1266
Pop RegC,
-- 1267
Const (0) RegD,
-- 1268
Branch RegD (Abs 1270),
-- 1269
Jump (Abs 1274),
-- 1270
Load (Deref RegC) RegC,
-- 1271
Const (1) RegE,
-- 1272
Compute Sub RegD RegE RegD,
-- 1273
Jump (Abs 1268),
-- 1274
Const (1) RegD,
-- 1275
Compute Add RegC RegD RegC,
-- 1276
Load (Deref RegC) SP, -- Restore SP
-- 1277
Jump (Ind RegB), -- Jump to return address
-- 1278
Load (Addr 0) RegA,
-- 1279
Const (6) RegB,
-- 1280
Compute Add RegA RegB RegB,
-- 1281
Store RegB (Addr 0),
-- 1282
Push SP,
-- 1283
Pop RegB,
-- 1284
Const (3) RegC,
-- 1285
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1286
Store RegB (Deref SP),
-- 1287
```

```
Push SP,
-- 1288
Pop RegB,
-- 1289
Const (0) RegA,
-- 1290
Compute Add RegB RegA RegB,
-- 1291
Const (0) RegA,
-- 1292
Branch RegA (Abs 1294),
-- 1293
Jump (Abs 1298),
-- 1294
Load (Deref RegB) RegB,
-- 1295
Const (1) RegC,
-- 1296
Compute Sub RegA RegC RegA,
-- 1297
Jump (Abs 1292),
-- 1298
Const (1) RegA,
-- 1299
Compute Add RegB RegA RegA,
-- 1300
Nop,
-- 1301
Push RegA, -- SP-ref: n
-- 1302
Const (65521) RegA,
-- 1303
Push RegA, -- Const: 65521
-- 1304
Pop RegA,
-- 1305
Pop RegB,
-- 1306
Store RegA (Deref RegB), -- assign
-- 1307
Push SP,
-- 1308
Pop RegB,
-- 1309
Const (0) RegA,
-- 1310
```

```
Compute Add RegB RegA RegB,
-- 1311
Const (0) RegA,
-- 1312
Branch RegA (Abs 1314),
-- 1313
Jump (Abs 1318),
-- 1314
Load (Deref RegB) RegB,
-- 1315
Const (1) RegC,
-- 1316
Compute Sub RegA RegC RegA,
-- 1317
Jump (Abs 1312),
-- 1318
Const (1) RegA,
-- 1319
Compute Add RegB RegA RegA,
-- 1320
Load (Deref RegA) RegA,
-- 1321
Push RegA, -- SP: n
-- 1322
Load (Addr 0) RegA,
-- 1323
Const (3) RegB,
-- 1324
Compute Add RegA RegB RegB,
-- 1325
Store RegB (Addr 0),
-- 1326
Push RegA,
-- 1327
Const (0) RegB,
-- 1328
Store RegB (Deref RegA),
-- 1329
Const (1) RegB,
-- 1330
Compute Add RegA RegB RegA,
-- 1331
Const (1003) RegB,
-- 1332
Store RegB (Deref RegA),
-- 1333
```

```
Const (1) RegB,
-- 1334
Compute Add RegA RegB RegA,
-- 1335
Push SP,
-- 1336
Pop RegC,
-- 1337
Const (2) RegD,
-- 1338
Compute Add RegC RegD RegC,
-- 1339
Const (0) RegB,
-- 1340
Branch RegB (Abs 1342),
-- 1341
Jump (Abs 1346),
-- 1342
Load (Deref RegC) RegC,
-- 1343
Const (1) RegD,
-- 1344
Compute Sub RegB RegD RegB,
-- 1345
Jump (Abs 1340),
-- 1346
Store RegC (Deref RegA),
-- 1347
Pop RegA, -- Actual call
-- 1348
Load (Deref RegA) RegB,
-- 1349
Const (4) RegC,
-- 1350
Compute Add RegB RegC RegB,
-- 1351
Load (Addr 0) RegC,
-- 1352
Compute Add RegC RegB RegB,
-- 1353
Store RegB (Addr 0),
-- 1354
Push RegC,
-- 1355
Pop RegB,
-- 1356
```

```
Const (2) RegD,
-- 1357
Compute Add RegA RegD RegD,
-- 1358
Load (Deref RegD) RegD,
-- 1359
Store RegD (Deref RegB),
-- 1360
Const (1) RegD,
-- 1361
Compute Add RegB RegD RegB,
-- 1362
Const (1) RegD,
-- 1363
Compute Add SP RegD RegD,
-- 1364
Store RegD (Deref RegB),
-- 1365
Const (1) RegD,
-- 1366
Compute Add RegB RegD RegB,
-- 1367
Const (1395) RegD, -- Calculate return address
-- 1368
Store RegD (Deref RegB),
-- 1369
Const (1) RegD,
-- 1370
Compute Add RegB RegD RegB,
-- 1371
Pop RegE,
-- 1372
Store RegE (Deref RegB),
-- 1373
Const (1) RegE,
-- 1374
Compute Add RegB RegE RegB,
-- 1375
Load (Deref RegA) RegD,
-- 1376
Push RegA,
-- 1377
Const (3) RegE,
-- 1378
Compute Add RegA RegE RegA,
-- 1379
```

```
Branch RegD (Rel (2)),
-- 1380
Jump (Rel (8)),
-- 1381
Load (Deref RegA) RegE,
-- 1382
Store RegE (Deref RegB),
-- 1383
Const (1) RegE,
-- 1384
Compute Add RegA RegE RegA,
-- 1385
Compute Add RegB RegE RegB,
-- 1386
Compute Sub RegD RegE RegD,
-- 1387
Jump (Rel (-8)),
-- 1388
Pop RegD,
-- 1389
Push RegC,
-- 1390
Pop SP,
-- 1391
Const (1) RegE,
-- 1392
Compute Add RegE RegD RegD,
-- 1393
Load (Deref RegD) RegD,
-- 1394
Jump (Ind RegD),
-- 1395
Push RegA,
-- 1396
Pop RegA,
-- 1397
Branch RegA (Abs 1399), -- if true
-- 1398
Jump (Abs 1497), -- if false
-- 1399
Load (Addr 0) RegA,
-- 1400
Const (5) RegB,
-- 1401
Compute Add RegA RegB RegB,
-- 1402
```

```
Store RegB (Addr 0),
-- 1403
Push SP,
-- 1404
Pop RegB,
-- 1405
Const (3) RegC,
-- 1406
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1407
Store RegB (Deref SP),
-- 1408
Load (Addr 0) RegA,
-- 1409
Const (5) RegB,
-- 1410
Compute Add RegA RegB RegB,
-- 1411
Store RegB (Addr 0),
-- 1412
Push SP,
-- 1413
Pop RegB,
-- 1414
Const (3) RegC,
-- 1415
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1416
Store RegB (Deref SP),
-- 1417
Const (1) RegA,
-- 1418
Push RegA, -- Const: 1
-- 1419
Load (Addr 0) RegA,
-- 1420
Const (3) RegB,
-- 1421
Compute Add RegA RegB RegB,
-- 1422
Store RegB (Addr 0),
-- 1423
Push RegA,
-- 1424
Const (0) RegB,
-- 1425
```

```
Store RegB (Deref RegA),
-- 1426
Const (1) RegB,
-- 1427
Compute Add RegA RegB RegA,
-- 1428
Const (418) RegB,
-- 1429
Store RegB (Deref RegA),
-- 1430
Const (1) RegB,
-- 1431
Compute Add RegA RegB RegA,
-- 1432
Push SP,
-- 1433
Pop RegC,
-- 1434
Const (2) RegD,
-- 1435
Compute Add RegC RegD RegC,
-- 1436
Const (2) RegB,
-- 1437
Branch RegB (Abs 1439),
-- 1438
Jump (Abs 1443),
-- 1439
Load (Deref RegC) RegC,
-- 1440
Const (1) RegD,
-- 1441
Compute Sub RegB RegD RegB,
-- 1442
Jump (Abs 1437),
-- 1443
Store RegC (Deref RegA),
-- 1444
Pop RegA, -- Actual call
-- 1445
Load (Deref RegA) RegB,
-- 1446
Const (4) RegC,
-- 1447
Compute Add RegB RegC RegB,
-- 1448
```

```
Load (Addr 0) RegC,
-- 1449
Compute Add RegC RegB RegB,
-- 1450
Store RegB (Addr 0),
-- 1451
Push RegC,
-- 1452
Pop RegB,
-- 1453
Const (2) RegD,
-- 1454
Compute Add RegA RegD RegD,
-- 1455
Load (Deref RegD) RegD,
-- 1456
Store RegD (Deref RegB),
-- 1457
Const (1) RegD,
-- 1458
Compute Add RegB RegD RegB,
-- 1459
Const (1) RegD,
-- 1460
Compute Add SP RegD RegD,
-- 1461
Store RegD (Deref RegB),
-- 1462
Const (1) RegD,
-- 1463
Compute Add RegB RegD RegB,
-- 1464
Const (1492) RegD, -- Calculate return address
-- 1465
Store RegD (Deref RegB),
-- 1466
Const (1) RegD,
-- 1467
Compute Add RegB RegD RegB,
-- 1468
Pop RegE,
-- 1469
Store RegE (Deref RegB),
-- 1470
Const (1) RegE,
-- 1471
```

```
Compute Add RegB RegE RegB,
-- 1472
Load (Deref RegA) RegD,
-- 1473
Push RegA,
-- 1474
Const (3) RegE,
-- 1475
Compute Add RegA RegE RegA,
-- 1476
Branch RegD (Rel (2)),
-- 1477
Jump (Rel (8)),
-- 1478
Load (Deref RegA) RegE,
-- 1479
Store RegE (Deref RegB),
-- 1480
Const (1) RegE,
-- 1481
Compute Add RegA RegE RegA,
-- 1482
Compute Add RegB RegE RegB,
-- 1483
Compute Sub RegD RegE RegD,
-- 1484
Jump (Rel (-8)),
-- 1485
Pop RegD,
-- 1486
Push RegC,
-- 1487
Pop SP,
-- 1488
Const (1) RegE,
-- 1489
Compute Add RegE RegD RegD,
-- 1490
Load (Deref RegD) RegD,
-- 1491
Jump (Ind RegD),
-- 1492
Push RegA,
-- 1493
Pop Zero,
-- 1494
```

```
Load (Deref SP) SP,
-- 1495
Load (Deref SP) SP,
-- 1496
Jump (Abs 1594), -- then end
-- 1497
Load (Addr 0) RegA,
-- 1498
Const (5) RegB,
-- 1499
Compute Add RegA RegB RegB,
-- 1500
Store RegB (Addr 0),
-- 1501
Push SP,
-- 1502
Pop RegB,
-- 1503
Const (3) RegC,
-- 1504
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1505
Store RegB (Deref SP),
-- 1506
Load (Addr 0) RegA,
-- 1507
Const (5) RegB,
-- 1508
Compute Add RegA RegB RegB,
-- 1509
Store RegB (Addr 0),
-- 1510
Push SP,
-- 1511
Pop RegB,
-- 1512
Const (3) RegC,
-- 1513
Compute Add RegA RegC SP, -- >>> ENTER SCOPE
-- 1514
Store RegB (Deref SP),
-- 1515
Const (0) RegA,
-- 1516
Push RegA, -- Const: 0
-- 1517
```

```
Load (Addr 0) RegA,
-- 1518
Const (3) RegB,
-- 1519
Compute Add RegA RegB RegB,
-- 1520
Store RegB (Addr 0),
-- 1521
Push RegA,
-- 1522
Const (0) RegB,
-- 1523
Store RegB (Deref RegA),
-- 1524
Const (1) RegB,
-- 1525
Compute Add RegA RegB RegA,
-- 1526
Const (418) RegB,
-- 1527
Store RegB (Deref RegA),
-- 1528
Const (1) RegB,
-- 1529
Compute Add RegA RegB RegA,
-- 1530
Push SP,
-- 1531
Pop RegC,
-- 1532
Const (2) RegD,
-- 1533
Compute Add RegC RegD RegC,
-- 1534
Const (2) RegB,
-- 1535
Branch RegB (Abs 1537),
-- 1536
Jump (Abs 1541),
-- 1537
Load (Deref RegC) RegC,
-- 1538
Const (1) RegD,
-- 1539
Compute Sub RegB RegD RegB,
-- 1540
```

```
Jump (Abs 1535),
-- 1541
Store RegC (Deref RegA),
-- 1542
Pop RegA, -- Actual call
-- 1543
Load (Deref RegA) RegB,
-- 1544
Const (4) RegC,
-- 1545
Compute Add RegB RegC RegB,
-- 1546
Load (Addr 0) RegC,
-- 1547
Compute Add RegC RegB RegB,
-- 1548
Store RegB (Addr 0),
-- 1549
Push RegC,
-- 1550
Pop RegB,
-- 1551
Const (2) RegD,
-- 1552
Compute Add RegA RegD RegD,
-- 1553
Load (Deref RegD) RegD,
-- 1554
Store RegD (Deref RegB),
-- 1555
Const (1) RegD,
-- 1556
Compute Add RegB RegD RegB,
-- 1557
Const (1) RegD,
-- 1558
Compute Add SP RegD RegD,
-- 1559
Store RegD (Deref RegB),
-- 1560
Const (1) RegD,
-- 1561
Compute Add RegB RegD RegB,
-- 1562
Const (1590) RegD, -- Calculate return address
-- 1563
```

```
Store RegD (Deref RegB),
-- 1564
Const (1) RegD,
-- 1565
Compute Add RegB RegD RegB,
-- 1566
Pop RegE,
-- 1567
Store RegE (Deref RegB),
-- 1568
Const (1) RegE,
-- 1569
Compute Add RegB RegE RegB,
-- 1570
Load (Deref RegA) RegD,
-- 1571
Push RegA,
-- 1572
Const (3) RegE,
-- 1573
Compute Add RegA RegE RegA,
-- 1574
Branch RegD (Rel (2)),
-- 1575
Jump (Rel (8)),
-- 1576
Load (Deref RegA) RegE,
-- 1577
Store RegE (Deref RegB),
-- 1578
Const (1) RegE,
-- 1579
Compute Add RegA RegE RegA,
-- 1580
Compute Add RegB RegE RegB,
-- 1581
Compute Sub RegD RegE RegD,
-- 1582
Jump (Rel (-8)),
-- 1583
Pop RegD,
-- 1584
Push RegC,
-- 1585
Pop SP,
-- 1586
```

```
    Const (1) RegE,
    -- 1587
    Compute Add RegE RegD RegD,
    -- 1588
    Load (Deref RegD) RegD,
    -- 1589
    Jump (Ind RegD),
    -- 1590
    Push RegA,
    -- 1591
    Pop Zero,
    -- 1592
    Load (Deref SP) SP,
    -- 1593
    Load (Deref SP) SP,
    -- 1594
    Load (Deref SP) SP,
    EndProg]

main = run 1 prog
```

8.4 How to debug run

```
*Main> :l ../../Campbell/campbell/src/main/java/campbell/language/test/isPrime.hs
[1 of 5] Compiling Sprockell.Components ( Sprockell/Components.hs, interpreted )
[2 of 5] Compiling Sprockell.TypesEtc ( Sprockell/TypesEtc.hs, interpreted )
[3 of 5] Compiling Sprockell.Sprockell ( Sprockell/Sprockell.hs, interpreted )
[4 of 5] Compiling Sprockell.System ( Sprockell/System.hs, interpreted )
[5 of 5] Compiling Main             ( ../../Campbell/campbell/src/main/java/campbell/language/test/isP
rime.hs, interpreted )
Ok, modules loaded: Sprockell.System, Main, Sprockell.Components, Sprockell.TypesEtc, Sprockell.Sprock
ell.
*Main> let debug st = (show $ (regbank s) ! PC) ++ " " ++ (show $ (regbank s) ! RegA) ++ " " ++ (show
$ (regbank s) ! RegB) ++ " " ++ (show $ (regbank s) ! SP) ++ " " ++ (show $ map (\i -> ((localMem s) !
!! i)) [0..70]) ++ "\n" where s = head $ sprs $ st
*Main> runDebug debug 1 prog
```

8.5 How to run and result

```
,0,0,0,71]
*Main> :l ../../Campbell/campbell/src/main/java/campbell/language/test/isPrime.hs
[1 of 5] Compiling Sprockell.Components ( Sprockell/Components.hs, interpreted )
[2 of 5] Compiling Sprockell.TypesEtc ( Sprockell/TypesEtc.hs, interpreted )
[3 of 5] Compiling Sprockell.Sprockell ( Sprockell/Sprockell.hs, interpreted )
[4 of 5] Compiling Sprockell.System ( Sprockell/System.hs, interpreted )
[5 of 5] Compiling Main             ( ../../Campbell/campbell/src/main/java/campbell/language/test/isP
rime.hs, interpreted )
Ok, modules loaded: Sprockell.System, Main, Sprockell.Components, Sprockell.TypesEtc, Sprockell.Sprock
ell.
*Main> main
Starting with random seed: 2632667823807622807
1
*Main>
```