Programming Paradigms

MOD08

University of Twente
Dennis de Weerdt (s1420321)
Ruben Groot Roessink (s1468642)

Contents

Main features

Our programming language is not that hard, it is mainly influenced by the C++ syntax.

The programming language which does not really have a name yet contains various features, some required by the project description others implemented mostly for the challenge.
It has the following data types:

- Integers
- Booleans
- Arrays
- Pointers
- Enumerations

The language also implements if-, while- and for-loops and concurrency via so-called lock-blocks. Statements placed in lock-blocks are executed by at most one Sprockell simultaneously; thus providing mutual exclusion.

There is support for functions, although these can only exist at the top level. (There are no nested procedures.) The entry point for the program is the function “main”. That function can then call others (or itself, recursively). Functions are pass-by-value by nature. The presence of pointers, however, means that the programmer may opt for pass-by-reference by declaring their function to accept a pointer to the actual argument.

Arrays are bound to a single dimension only. They can be created of any non-array type, and will be allocated either in the shared or local memory, depending on the declaration. (The “shared” keyword is used to denote variables of any sort which should be stored in the shared memory.)

Pointers can point to any type, including other pointers. As with all other variables, they may be shared or local. Pointers can have an arbitrary ‘depth’, thus enabling pass-by-reference mechanics. Integers are pointers are interoperable, so it is possible to assign an integer to a pointer and vice-versa. If a pointer is assigned to an integer, that integer will hold the address pointed to by the pointer. In the opposite case, the pointer shall interpret the integer’s value as an address.

Enumerated types are strongly typed at compile-time, and replaced by constant integers at run-time. Enums can hold any number of constants (at least one), and are fully interoperable with pointers and arrays.

Finally, we have developed a small standard library for our language, providing such features as printing variables, type conversions and testing arrays for equality.

Problems and solutions

During the implementation of the product we encountered a few problems. In this part we are going to show the problems we encountered and how we resolved these problems.

While implementing the lock we encountered a slight problem. The end result of the program was sometimes one or two beneath the value that we expected. We were trying to run the following program (in pseudocode).

```
program test {
    global int val = 0;

    main() {
        lock;
        for (int i = 0; i < 3; i++) {
            val++;
        }
        unlock;

        lock;
        for (int i = 0; i < 3; i++) {
            val--;
        }

        lock;
        out.println(val);
        unlock;
    }
}
```

This program is very simple. But when ran on 10 Sprockells concurrently we saw that out.println(val) sometimes returned a value which was not divisible by 3, which is weird because we thought that it would only return multiples of 3 due to the amount of iterations in the for-loop.

After a lot of research we found that because every Sprockell set the global variable val again the value would sometimes be overwritten while another Sprockell was already in the for-loop. We solved this by automatically locking every Sprockell while the Sprockell with id 0 was still initializing the global variables and that every Sprockell which did not have id 0 would skip the global variables. The locks now finally work.

Language description

The following features are implemented in the program.

Comments

Syntax

    /* Whatever you want to write */

Usage

    This feature should be used to add comments to a program.

Semantics.

    Everything within a /* */ block is skipped by the program and therefore not executed.

Code generated

    There is no code generated because everything within a /* */ is skipped.

For-loop

Syntax

for (**[declaration]** ; **[expression]** ; **[assignment]**) **[block]**

e.g. for (int i = 0; i < 5; i = i + 1)

Usage

This feature should be used for iteration over arrays and for execution of a certain block a certain number of times.

Semantics

In the for-loop a local variable is created ([declaration]). The statements in the block are executed any number of times until the expression resolves to false. The assignment can be used to in a way make the expression resolve to false the next iteration.

Code generation

code generated for example for(int i = 0; i < 3; i = i + 1)

```
Const 0 RegD                    {-Declaration of i(=0); For loop declaration}
Const 0 RegA
Store RegA (Deref RegD)
Load (Addr 0) RegA              {-For loop condition-}
Push RegA
Const 3 RegA
Pop RegB
Compute GtE RegB RegA RegA      {-i Lt 3-}
Branch RegA (Rel 17)            {-Break from for loop-}
        Code inside the block of the for-loop is placed here.
Const 0 RegD                    {-i = i+1; For loop assignment-}
Load (Addr 0) RegA
Push RegA
Const 1 RegA
Pop RegB
```

```
Compute Add RegB RegA RegA      {-i Add 1-}
Store RegA (Deref RegD)
Jump (Rel (0-21))               {-Back to for loop-}
```

While-loop

Syntax

while (**[expression]**) **[block]**
e.g. while (x < 3)

Usage

This feature should be used to execute a certain block of code until the expression resolves to false.

Semantics

Inside the while-loop a certain expression is evaluated and until it resolves to false the code inside the block of the loop is executed over and over again.

Code generation

```
while (val < 3) {}

Load (Addr 0) RegA
Push RegA
Const 3 RegA
Pop RegB
Compute GtE RegB RegA RegA      {-val Lt 3-}
Branch RegA (Rel 9)
        Code here is the code that is inside the block of the while-loop
Jump (Rel (0-13))
```

If-statement

Syntax

if (**[expression]**) **[block]**
Optionally also
else if (**[expression]**) **[block]**
and
else **[block]**
e.g. if (i < 3) **[block]** else if (i < 10) **[block]** else **[block]**

Usage

This feature should be used when the program can be in different states and the program should execute different blocks of codes depending on the state.

Semantics

When inside the if-statement the first comparison (in the expression) is made and when it resolves to true the first block is executed. If the expression resolves to false an else-if-statement can be executed when it is present or not (if neither an else-if-statement or else-statement are present the program continues with the code outside the if-statement). More than one else-if-statement is also a possibility. The expression of the according else-if-statement is resolved and when it resolves to true the corresponding block of code is executed. When this resolves to false an else-statement can be executed when present, otherwise the program continues with it's normal workflow.

Code generated

```
if (val < 3) {}
else if (val < 10) {}
else {}

Load (Addr 0) RegA
Push RegA
Const 3 RegA
Pop RegB
Compute Lt RegB RegA RegA     {-val Lt 3-}
Branch RegA (Rel ??)
Load (Addr 0) RegA
Push RegA
Const 10 RegA
Pop RegB
Compute Lt RegB RegA RegA     {-val Lt 10-}
Branch RegA (Rel ??)
Jump (Rel ??)
        Block of if-statement is here.
Jump (Rel ??)
        Block of else-if statement is here.
Jump (Rel ??)
        Block of else-statement is here.
```

The jumps are not entirely correct, because another test program was used to determine which code was generated. The overall picture is hopefully clear though.

Lock-statement

Syntax

lock (**[id]**) **[block]**

e.g. lock (val_lock) **[block]**

Usage

This feature can be used to synchronize a block of code, because for example a shared variable may not be altered by different Sprockells at the same time to guarantee correctness of the program.

Semantics

The lock-statement is opened by the keyword 'lock'. Then an identifier (or name) must be provided for this lock, so that more instances of lock can be in the same program. The block is executed synchronized.

Code generated

```
lock (val_lock) {}

TestAndSet (Addr 2)                   {-Locking val_lock; Function main-}
Receive RegE
Branch RegE (Rel 2)
Jump (Rel (0-3))
        Code inside the block is executed here.
Write Zero (Addr 2)                   {-Unlocking val_lock-}
```

Unary expressions

Syntax

The following tokens can appear in a unary expression [-, !, &, *]

A unary expression is one of this tokens followed by another expression (so it is recursive).

e.g. - 3

e.g. - (3 + 4)

e.g. ! true

Usage

The first two tokens, - and ! are used to invert a number and a boolean value respectively.

The other two tokens, & and *. & is used to return the memory address of a certain variable, whereas * is used to return the value in a certain memory address (these were used for the implementation of pointer).

Semantics

- is used to negate a number.

! is used to invert a boolean value.

& is used to return the memory address of a certain variable.

* is used to return the value in a certain memory address.

Code generated

```
-i
Const 0 RegD              {-i = -i-}
Push RegD
Load (Addr 0) RegA
Push RegA
Pop RegA
Compute Sub Zero RegA RegA
Pop RegD
Store RegA (Deref RegD)

! true
Const 0 RegD              {-i = !i-}
Push RegD
```

```
Load (Addr 0) RegA
Const 1 RegB
Compute Xor RegB RegA RegA
Pop RegD
Store RegA (Deref RegD)
```

&x

```
Const <addr> RegA
Push RegA
```

*x

```
Load (Deref <addr>) RegA
Push RegA
```

Binary expressions

Syntax

The following tokens can be used in binary expressions [AND, OR, XOR, <, >, ==, <=, >=, !=, *, /, -, +, %]

A binary consists of a left hand side and a right hand side with one of the above tokens in between.

e.g. true AND true

e.g. false OR true

e.g. 3 < 9

e.g. 6 == 6

e.g. 8 / 4

Usage

The binary expressions can be used to either compare certain values, calculate new values and operators on boolean values.

Semantics

AND, OR and XOR are boolean operators and resolve to true if the expressions on both sides resolve to true, if one or two of the expressions resolve to true and if one and only one expression of the two resolves to true.

<, > , ==, <=, >=, != are comparators and this binary expression resolves to true if the following conditions hold.

<, resolves to true if the left hand side is lesser than the right hand side.

>, resolves to true if the left hand side is greater than the right hand side.

==, resolves to true if the left hand side and the right hand side both resolve to the same value.

<=, resolves to true if the left hand side is lesser than or equal to the right hand side.

>=, resolves to true if the left hand side is greater than or equal to the right hand side.

*, /, -, +, % are used to calculate new values.

*, multiplication
/, division
-, addition
+, subtraction
%, module operation

Code generated

Case of boolean operators
bool i = true AND false;

```
Const 0 RegD                      {-Declaration of i(=true AND false); -}
Push RegD
Const 1 RegA
Push RegA
Push Zero
Pop RegA
Pop RegB
Compute And RegB RegA RegA        {-true And false-}
Pop RegD
Store RegA (Deref RegD)
```

Case of comparators
bool i = 6 == 9;

```
Const 0 RegD                      {-Declaration of i(=6==9);-}
Push RegD
Const 6 RegA
Push RegA
Const 9 RegA
Pop RegB
Compute Equal RegB RegA RegA      {-6 Equal 9-}
Pop RegD
Store RegA (Deref RegD)
```

Case of mathematical operators
int i = 6 % 3;

```
Const 0 RegD                      {-Declaration of i(=6%3);-}
Push RegD
Const 6 RegA
Push RegA
Const 3 RegA
Pop RegB
Compute Mod RegB RegA RegA        {-6 Mod 3-}
```

```
Pop RegD
Store RegA (Deref RegD)
```

Par expression

Syntax

The par expression is an expression which is calculated entirely before the result is used in other operations.

(**[expression]**)

e.g. (2 + 3)

Usage/Semantics

The par expression can be used to define a block of code that needs to be calculated before the result is used in overlaying structures.

Code generated

```
int i = ( 3 + 5 );
Const 0 RegD                        {-Declaration of i(=(3+5));-}
Push RegD
Const 3 RegA
Push RegA
Const 5 RegA
Pop RegB
Compute Add RegB RegA RegA          {-3 Add 5-}
Pop RegD
Store RegA (Deref RegD)
```

ConstArrayExpression

Syntax

[id] '[' **[number]** ']'

Usage/Semantics

Retrieve a value from the array given by **[id]** at the constant index given by **[number]**. There is no compile-time or run-time check to see if the index lies within the bounds of the array. Out-of-bounds values will result in undefined behaviour.

Code Generated

```
Load (Addr 0) RegD
Const [number] RegA
Compute Add RegA RegD RegD
Load (Deref RegD) RegA
Push RegA
```

ExprArrayExpression

Syntax

[id] '[' **[expr]** ']'

Usage/Semantics

Retrieve a value from the array given by **[id]** at the index which is the result of the expression **[expr]**. The expression must have an integer result. There is no compile-time or run-time check to see if the index lies within the bounds of the array. Out-of-bounds values will result in undefined behaviour.

Code Generated

```
Load (Addr 0) RegE
    Evaluate the expression, store the result in RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Push RegA
```

ArrayLiteralExpr

Syntax

‘[‘ **[expr1]** (‘,’ **[expr2]** ... ‘,’ **[exprN]**) ‘]’ (N >= 1)

Usage

Define an array literal for use in assignments or as function parameter.

Semantics

The result of the expression is an anonymous array (it has no **[id]**, therefore no name), which is meant to either be assigned to a variable (of the corresponding array type) or passed as an argument to a function.

Code Generation

(**base** is the address of the first element, **ptr** is the address of the pointer which points to that address. Both are calculated at compile-time.)

```
For each expression in the list:
    Evaluate the expression, storing the result in RegA
    Store RegA (Addr (base + index * element size))
Const base RegA
Store RegA (Addr ptr)
Push RegA
```

Declaration/Assign/Call statement

Syntax

[declaration] ;

[assignment] ;

[call] ;

e.g. **[decl]** ;

Usage

The declaration and assign statements are used to put a semicolon after declarations, assignments and calls.

Semantics

This feature adds a semicolon after declaration, assign statements and calls.

Code generated

```
int i = 9;
```

```
Const 0 RegD                    {-Declaration of i(=0);}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
```

In/Out statement

Syntax

OUT (**[expression]**);
IN (**[id]**);
e.g. out(9);
e.g. in(val_name);

Usage

The input and output statements to get input from outside the program and to give output to outside of the program.

Semantics

Opened by the keywords OUT and IN respectively, these function have their arguments in their brackets. They are closed by a semicolon.

Code generated

```
Const 9 RegA
Out RegA
```

Return statement

Syntax

RETURN **[expression]** ;
RETURN ;
The amount of expression this function gets can be zero or one.
e.g. return int i = 9;

Usage

This function can be used to return a certain value.

Semantics

This feature is used to return a certain value by a function. It is opened by the keyword RETURN and closed by a semicolon. It may or may not return a certain expression.

Code generated

```
Pop RegB                        {-Get Result addr-}
Pop RegC                        {-Get Return addr-}
Compute Lt RegB Zero RegD       {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)         {-Store Result-}
```

Declaration

Syntax

[type] [id] = expr

[type] [id] = [**[number]**]
SHARED **[type] [id]** = expr
SHARED **[type] [id]** = [**[number]**]
e.g. shared int val = 0
e.g. int val = 9

Usage

This feature is used to declare a variable.

Semantics

First the question arises whether the variable is global or not, this is indicated by the keyword SHARED. This is followed by the type of the variable and the identifier (or name) of the variable. The equal sign means that the right hand side is assigned to the left hand side. So a variable with a certain type is set with the value of an expression or a value of an array, indicated by the square brackets and a number which is corresponding with the place in the array.

Code generated

```
int i = 9;
Const 0 RegD            {-Declaration of i(=9); Function main-}
Push RegD
Const 9 RegA
Pop RegD
Store RegA (Deref RegD)
```

Assign

Syntax

[derefId] = **[expression]**
[arrayVal] = **[expression]**
e.g. *ptr = newValue();
e.g. tempArray[4] = 9;

Usage

The feature assign is used to assign a new value to an already existing variable.

Semantics

The variable in which the new value needs to be stalled can be received using the dereference of the name of an variable or is the position in an array. The value that this variable needs to become is calculated in an expression.

Code generator

```
i = 5;
Const 0 RegD            {-i = 5-}
Push RegD
Const 5 RegA
Pop RegD
Store RegA (Deref RegD)
```

Program

Syntax

[progdef] [decl] ; **[func] [decl]** ;
[progdef] [decl] ; **[func] [func]**

e.g.

```
program test;
shared int val = 5;
def int main () {
}
```

The definition above is a bit unclear. But a program always needs [progdef] and one function (namely function main) all of the other declarations and functions can vary in number from 0 till infinity.

Usage

This feature is used to define a program.

Semantics

A program starts with a program definition (or progdef) this is followed by any number of declarations outside of the scope of the main function of the program.
Then the main function is declared. A program must always have a main method. After that any number of declarations or functions can be defined.

Code generator

The top-level code is scattered somewhat, so this overview is more schematic than most.

First, create the initial activation record. The return address is the end of the program, the result address is an invalid value, indicating that the result is to be ignored.

```
Const <address of EndProg instruction> RegA
Push RegA
Const -1 RegA -- Don't store the result of the main function.
Push RegA
```

<shared and local variable declarations and initial assignments>

The following code is to ensure that shared variables are only initialized once. All Sprockells except the one with SPID 0 have to wait for a signal which Sprockell 0 will provide once it has finished initialization. ***<init_addr>*** *is an address in shared memory determined at compile-time, known to all Sprockells, which will store this signal.*

```
Branch SPID (Rel 4)        -- SPID 0 will continue
TestAndSet <init_addr>     -- Set the signal
Receive Zero               -- Don't care about the result
Jump (Rel 5)               -- Jump past the wait loop
Read <init_addr>           -- See if the signal was set
Receive RegA               -- Receive it
Branch RegA (Rel 2)        -- If so, skip the next line...
```

```
Jump (Rel -3)                  -- ...Else, go back to the Read instruction

Jump (Abs <address of main function>)
<code contained in functions>
EndProg
```

Progdef

Syntax

PROGRAM **[id]** ;
PROGRAM **[id]** (**[number]**) ;
e.g. program test (5);

Usage

This feature is used to define the name of a program and on how many sprockells this program should be ran separately.

Semantics

A progdef starts with the word ‘program’ and is followed by a name. The number between brackets is optional, but is implemented to let the program now on how many sprockells it must be run.

Code generator

No code is generated for **Progdef.** *The name and, optionally, the core count are stored, however, and retrieved when writing the output file.*

Typedparams

Syntax

(**[type] [id]**)
(**[type] [id]** , **[type] [id]**)
The number of [type] [id] declarations is unlimited.
e.g. (int temp, bool val)

Usage

This feature is used in the definition of functions to specify which types it needs.

Semantics

Every parameter that a function needs to receive to fulfill its purpose has a type and a name.

Code generator

No code is generated for **Typedparams**. *The information contained in these statements is only used during the elaboration phase.*

Params

Syntax

(**[id]**)
(**[id]** , **[id]**)
The number of [id] declarations is unlimited.
e.g. (temp, bool)

Usage

This feature is used in a function call which gives certain parameters to a function.

Semantics

Every parameter that a function needs to receive to fulfill its purpose has a type and a name.

Code generator

```
Const 0 RegD            {-i = test(4)-}
Push RegD
Const 4 RegA
Store RegA (Addr 2)
```

Description of the software

The following files were implemented. We mainly used JAVA and ANTLR with it for the program.

Package frontend

FrontEnd.java
FrontEnd.java can be run on the command line. It takes a source file containing a program as argument and compiles and runs the program.

Package parsing

Checker.java
Checker extends BaseGrammarBaseVisitor. This class is used to check whether a program is correctly defined and does check whether the syntax used in the program is correct according to the grammar rules or not..

Func.java
Func is used to define a function.

Generator.java
Generator is used to calculate the value of a certain program after it is checked on syntax errors in Checker.java.

Primitive.java
Defines primitive types for variables.

Scope.java
Scope defines scopes which are different kind of levels inside a program.

Type.java
Defines the different types of the variables used in the program.

BaseGrammar.g4
In BaseGrammar.g4 all the rules are defined, which are used to build parse trees and calculate correct values for a certain program in our programming language.

Package tests

CheckerTest.java
CheckerTest.java is used to test whether our implementation of Checker.java is correct in contrast with our ideas of what it should do.

ScopeTest.java
CheckerTest.java is used to test whether our implementation of Checker.java is correct in contrast with our ideas of what it should do.

SprilTest.java
SprilTest.java is used to test whether our implementation of Spril.java is correct in contrast with our ideas of what it should do.

TestRunner.java
TestRunner runs all the tests in the test package without you having to manually execute them yourself.

TypeTest.java
TypeTest..java is used to test whether our implementation of Type.java is correct in contrast with our ideas of what it should do.

Package translation

Int.java
A wrapper of an int.

MemAddr.java
Defines the class MemAddr, which is an implementation of a real life memory address.

OpCode.java
Defines the different OpCodes used by generator to create a haskell file that can be executed.

OpType.java
Holds different type of operands used in Spril instructions.

Operand.java
Interface Operands is the interface for an operand used in Spril instructions.

Operator.java
In Operator.java different operators are specified that can be used in the Spril instruction.

OpType.java
Enum OpType defines the different types of variables that can be used by the Spril instruction.

Program.java
Program defines a Program, this class is passed to OuputDebug.java and is used to define the program-part in the haskell file.

Register.java
Enum containing different operants for uses in registers.

Spril.java
The definition of an instruction in the Spril language.

Target.java
The definition of a jump target. It can be either absolute, relative or indirect (using a register).

Package write

Output.java
Mostly deprecated, used to define a haskell file containing a program with Spril instructions. Output.java makes a program without debug functionalities.

OutputDebug.java
Used by the compiler to write a haskell file containing a program with Spril instructions. This class also writes debug functionalities to the class.

ProgramRunner.java
ProgramRunner starts a thread by running the haskell file and compiling it to an .exe program. It also runs the .exe file.

Test plan and result

Testing Framework

To automate our tests, we have written a Java class which parses, compiles and executes source files under the ‘tests’ directory. Those tests in the ‘bad’ directory are expected to fail, whereas the ones in the ‘good’ directory should compile correctly and print a single ‘0’ to their standard output upon success. A bad test is considered a failure if it compiles normally, and good tests fail if they print anything other than a ‘0’, including nothing at all. This method of inter-process communication is quite rudimentary, but it is enough to signal a ‘yes’ or ‘no’, which, in turn, is sufficient for our purposes.

Syntactic errors

Correct parsing of syntactically correct programs is considered to be demonstrated by the fact that the semantic tests run correctly.
There are several incorrect programs included that cannot be correctly parsed. These are:

emptyEnum: An enum declaration without constants.
invalidArray1: An empty literal array.
invalidArray2: An array of negative size.
negativeCoreCount: A program which is to be executed on a negative number of cores.
noProgDef: A program without a program definition.

In all these cases, it is expected that ANTLR signals an error. Tests show that this is indeed what happens.

Contextual errors

As with syntactic errors, correct handling of correct programs is demonstrated by the semantic tests. The following test programs contain errors of this category:

duplicateFunc: A duplicate definition within the source file.
duplicateStdlib: This program defines a function which exists in the standard library.
duplicateVar1 and 2: Duplicate definition of variables
invalidArray3: Tries to get an int from a bool array.
invalidIf: Tries to use an int as an expression in an if-statement.
invalidLoop1 and 2: Same as above, but in while and for loops.
invalidPointer1: Tries to dereference something that is not a pointer.
invalidPointer2: Tries to return a pointer to an int while an int is expected.
noMain: A program without a main function.
noReturn1 and 2: Define functions without return statements.
typeMismatch1 through 4: All mix up different types.
undeclaredEnumConst: Tries to reference an enum constant that does not exist
undeclaredFunc: Tries to call a non-existing function.
undeclaredVar1 and 2: Try to reference non-existing variables.

Semantic errors

The following programs were used to test correctness:

simpleTest: A very simple test program that was mainly used to test the testing framework. Despite this, the test does cover a number of aspects, if not comprehensively: if-statements, arithmetic, function calls and the out-statement are included.

algTest: A program implementing two simple, well-known algorithms: computation of a factorial and of the greatest common divisor of two integers. In both cases, a number of computed results are tested against actual values.

bank versions 1 and 2: These programs test the locking mechanism. *bankV1* has ten Sprockells modify a shared account in such a way that all changes cancel each other out. The test is considered successful if the account has the original value at the end of the test. *bankV2* has an array of ten accounts, one for each Sprockell, and all Sprockells transfer 'money' from their account to all others. Again, the end result should be that the final state of the accounts is identical to the initial state.

enumTest: This program tests the use of enums in a variety of circumstances. This test shows that enums can be used as intended both by themselves and in conjunction with arrays and pointers.

primeTest: This program tests an array of integers for primality, and compares the results against a constant array containing the correct results.

stdLibTest: This program tests standard library functionality.

Conclusions

Module

The module on itself was interesting and we have learned a lot about compilers and how they work. The workload in the module was so high that it became harder than some modules that were just hard to do because the subjects in the other modules was really hard.

Due to the enormous workload some not mandatory courses where skipped and we think a lot of people had some trouble because they had to do Concurrent Programming in a few days.

Project

We thought the project was fun to do and that it is a good way to put the skills gained in Compiler Construction into practice. We didn’t do much functional programming, but the option was there and it would have been interesting to do, if we had felt he would have had enough time to do so.

Results

When we started out we planned to capitalise on the fact that reading and writing to local memory executes in a single clock cycle by using that memory for nearly everything. This worked well for a while, but eventually it became clear that a more conventional approach would have been better. At that point it was too late to redo almost all of the code generation, so we had to stick with what we had. In the end, we are satisfied with the result, flawed as it is. Unfortunately, this method of allocation must be regarded as a failed experiment, but we believe the rest of the language is of decent quality.

Appendix A

Grammar specification

```
grammar BaseGrammar;

program
:
        progdef
        (
                (
                        decl
                        | enumDecl
                ) SEMI
        )* func
        (
                (
                        (
                                decl
                                | enumDecl
                        ) SEMI
                )
                | func
        )*
;

progdef
:
        PROGRAM ID
        (
                LBRACE NUMBER RBRACE
        )? SEMI
;

block
:
        LCURL stat* RCURL
;

topLevelBlock
:
        block
;

func
:
        DEF type ID typedparams topLevelBlock
;

typedparams
:
        LBRACE
        (
                type ID
                (
                        COMMA type ID
                )*
        )? RBRACE
;

params
:
        LBRACE
        (
                expr
                (
                        COMMA expr
                )*
        )? RBRACE
;

call
:
        ID params
;

expr
:
        MINUS expr # negNumExpr
        | EXCLAMATION expr # negBoolExpr
        | AMP expr # refExpr
        | TIMES expr # derefExpr
        | expr boolOp expr # boolOpExpr
        | expr comp expr # compExpr
        | expr TIMES expr # multExpr
        | expr DIV expr # divExpr
        | expr MINUS expr # minExpr
        | expr PLUS expr # plusExpr
        | expr MOD expr # modExpr
        | LBRACE expr RBRACE # parExpr
        | call # callExpr
        | ID LSQUARE NUMBER RSQUARE # constArrayExpr
        | arrayVal # exprArrayExpr
        | LSQUARE expr
        (
                COMMA expr
        )* RSQUARE # arrayLiteralExpr
        | ID # idExpr
        | NUMBER # numExpr
        | TRUE # trueExpr
        | FALSE # falseExpr
        | SPID # spidExpr
        | TYPE DOT TYPE # enumExpr
;

stat
:
        LOCK LBRACE ID RBRACE block # lockStat
        | decl SEMI # declStat
        | assign SEMI # assignStat
        | IF expr block
        (
                ELSE IF expr block
        )*
        (
                ELSE block
        )? # ifStat
        | WHILE expr block # whileStat
        | call SEMI # callStat
        | block # blockStat
        | FOR LBRACE decl SEMI expr SEMI assign RBRACE block # forStat
        | OUT LBRACE expr RBRACE SEMI # outStat
        | IN LBRACE ID RBRACE SEMI # inStat
        | RETURN expr? SEMI # returnStat
;

decl
:
        SHARED? type ID EQ
```

```
	(
		expr
		| type LSQUARE NUMBER RSQUARE
	)
;

assign
:
	(
		derefID
		| arrayVal
	) EQ expr
;

arrayVal
:
	ID LSQUARE expr RSQUARE
;

type
:
	INT
	| BOOL
	| VOID
	| TYPE
	| type TIMES
	| type LSQUARE RSQUARE
;

enumDecl
:
	ENUM TYPE COLON TYPE
	(
		COMMA TYPE
	)*
;

comp
:
	LT
	| GT
	| EQ EQ
	| LE
	| GE
	| NE
;

boolOp
:
	AND
	| OR
	| XOR
	| NAND
	| NOR
	| NXOR
;

prefix
:
	MINUS
	| NOT
;

derefID
:
	TIMES derefID
```

```
        | ID
;

////////////////////////////////////////////////////////////////////////////////

INT
:
        I N T
;

BOOL
:
        B O O L
;

VOID
:
        V O I D
;

STRING
:
        S T R I N G
;

ENUM
:
        E N U M
;

FOR
:
        F O R
;

WHILE
:
        W H I L E
;

IF
:
        I F
;

ELSE
:
        E L S E
;

TRUE
:
        T R U E
;

FALSE
:
        F A L S E
;

AND
:
        A N D
;

OR
```

```
:
       O R
;

XOR
:
       X O R
;

NAND
:
       N A N D
;

NOR
:
       N O R
;

NXOR
:
       N X O R
;

NOT
:
       N O T
;

RETURN
:
       R E T U R N
;

BREAK
:
       B R E A K
;

DEF
:
       D E F
;

LOCK
:
       L O C K
;

UNLOCK
:
       U N L O C K
;

SHARED
:
       S H A R E D
;

PROGRAM
:
       P R O G R A M
;

SPID
:
        '$' S P I D
;

OUT
:
        O U T
;

IN
:
        I N
;

UNDERSCORE
:
        '_'
;

DOT
:
        '.'
;

COMMA
:
        ','
;

SEMI
:
        ';'
;

COLON
:
        ':'
;

EXCLAMATION
:
        '!'
;

LBRACE
:
        '('
;

RBRACE
:
        ')'
;

LCURL
:
        '{'
;

RCURL
:
        '}'
;

LSQUARE
:
        '['
;

RSQUARE
:
        ']'
;

PLUS
:
        '+'
;

MINUS
:
        '-'
;

TIMES
:
        '*'
;

DIV
:
        '/'
;

MOD
:
        '%'
;

AMP
:
        '&'
;

SQUOT
:
        '\''
;

DQUOT
:
        '"'
;

BACKSLASH
:
        '\\'
;

TIDLE
:
        '~'
;

HAT
:
        '^'
;

GT
:
        '>'
;
```

```
LT
:
	'<'
;

EQ
:
	'='
;

ARROW
:
	MINUS GT
;

GE
:
	GT EQ
;

LE
:
	LT EQ
;

NE
:
	EXCLAMATION EQ
;

TYPE
:
	UCASE
	(
		LETTER
		| DIGIT
		| UNDERSCORE
	)*
;

ID
:
	LCASE
	(
		LETTER
		| DIGIT
		| UNDERSCORE
	)*
;

NUMBER
:
	NONZERO DIGIT*
	| ZERO
;

fragment
LCASE
:
	[a-z]
;

fragment
UCASE
:
        [A-Z]
;

fragment
LETTER
:
        LCASE
        | UCASE
;

fragment
ZERO
:
        '0'
;

fragment
NONZERO
:
        [1-9]
;

fragment
DIGIT
:
        ZERO
        | NONZERO
;

fragment
A
:
        [aA]
;

fragment
B
:
        [bB]
;

fragment
C
:
        [cC]
;

fragment
D
:
        [dD]
;

fragment
E
:
        [eE]
;

fragment
F
:
        [fF]
;

fragment
```

```
G
:
        [gG]
;

fragment
H
:
        [hH]
;

fragment
I
:
        [iI]
;

fragment
J
:
        [jJ]
;

fragment
K
:
        [kK]
;

fragment
L
:
        [lL]
;

fragment
M
:
        [mM]
;

fragment
N
:
        [nN]
;

fragment
O
:
        [oO]
;

fragment
P
:
        [pP]
;

fragment
Q
:
        [qQ]
;

fragment
```

```
R
:
        [rR]
;

fragment
S
:
        [sS]
;

fragment
T
:
        [tT]
;

fragment
U
:
        [uU]
;

fragment
V
:
        [vV]
;

fragment
W
:
        [wW]
;

fragment
X
:
        [xX]
;

fragment
Y
:
        [yY]
;

fragment
Z
:
        [zZ]
;

COMMENT
:
        DIV TIMES .*? TIMES DIV -> skip
;

WS
:
        [ \r\n\t] -> skip
;
```

Types

-int: A 32-bit integer.

```
      int x = 3;
      int y = 0;
      int z = -8;
      int sum = x + y + z;          /* sum == -5 */
```

-bool: A boolean variable, internally represented as an int with the value 0(false) or 1(true).

```
      bool x = true;
      bool y = false;
      bool z = x OR y;              /* z == true */
```

-Pointer: A container type which stores an address to another variable, rather than that variable itself.

```
      int x = 5;
      int* ptr = &x;
      *ptr = 4;                     /* x == 4 */
      int** ptrToPtr = &ptr;
      **ptrToPtr = 9;               /* x == 9 */
```

-Array: A container type which stores some number of other variables. Arrays are fixed-size, all elements must be of the same type, and the size must be a compile-time constant. Indices are zero-based. Multidimensional arrays are not supported.

```
      int[] oneTwoThree = [1,2,3];
      int[] tenZeroes = int[10];
      bool[] bools = [true, false, true];
      int x = oneTwoThree[1] + tenZeroes[4]; /* x == 2 */
      int y = oneTwoThree[x];               /* y == 3 */
      bools[0] = x == y;                    /* bools[0] == false */
```

-void: A pseudo-type indicating the absence of a return type. May only be used in function declarations.

```
      def void doSomething(int argument) {
            /* perform some task */
            return;
      }
```

Note that even for void functions, an explicit return statement is required.

Tokens

The following tokens are declared:

VOID	- Indicator of a void method..
INT	- Can be used to declare an integer.
BOOL	- Can be used to declare a boolean.

FOR - Used to define a for-statement.

WHILE - Used to define a while-statement.
IF - Used to define an if-statement.
ELSE - Used to define an else-statement.

TRUE - Used as a value of a boolean.
FALSE - Used as a value of a boolean.

AND - Defines an AND operator which performs logical disjunction.
OR - Defines an OR operator which performs logical conjunction.
XOR - Defines an XOR operator which performs the exclusive OR operation.
NOT - Defines an NOT operator which performs logical inversion.
RETURN - Defines a return statement.
DEF - Defines a function declaration.
LOCK - Defines a lock statement.
SHARED - Defines a shared variable.
PROGRAM - Defines a program.
SPID - Defines a Sprockell id.
OUT - Defines an OUT-statement, which prints a number to stdout.
The following declarations define a name for a certain token or tokens.

UNDERSCORE	=	_
DOT	=	.
COMMA	=	,
SEMI	=	;
EXCLAMATION	=	!
LBRACE	=	(
RBRACE	=	)
LCURL	=	{
RCURL	=	}
LSQUARE	=	[
RSQUARE	=	]
PLUS	=	+
MINUS	=	-
TIMES	=	*
DIV	=	/
MOD	=	%
AMP	=	&
SQUOT	=	\
DQUOT	=	“
BACKSLASH	=	\\
TIDLE	=	~
HAT	=	^
GT	=	>
LT	=	<

EQ	=	=
ARROW	=	->
GE	=	>=
LE	=	<=
NE	=	!=

Types of variables are defined as a string starting with a capital letter, whereas identifiers (or names) of variables always start with a lowercase letter. A number is defined as a sequence of digits starting with a digit which is not zero.

The fragments are used to define lowercase letters, uppercase letters, letters in general, zero's and nonzero's and digits. The rest of the fragments are used to define every letter in the alphabet respectively.

Comments must always start with /* and end with */, everything between these tokens is skipped by the compiler.

Rules

program - The declaration of program.
progdef decl* func (decl|func)*

There must always be at least one function, *main*. It may be followed and/or preceded by any number of other functions or top-level variable declarations.

progdef - The declaration of the program definition:
PROGRAM **id** (**number**) ;
PROGRAM **id** ;

If a **number** is provided, the program will be executed on that number of Sprockells. If no **number** is given, then the program will be executed on a single Sprockell. It is a compile-time error if **number** is present but less than 1.

block - The declaration of a block in the program:
{ }
{ **stat** }
A block contains some number (possibly zero) of statements. These statements will be evaluated sequentially.

topLevelBlock - The declaration of the top level block of a function.
block

> **topLevelBlock** is mainly meant for bookkeeping. The declaration and use is identical to that of **block**.

func - The declaration of a function.
DEF **type** ID **typedparams topLevelBlock**

Examples:
```
def int main() {
     return 0;
}

def void doSomething(int* arg1, bool arg2) {
     return;
}
def int[] intToString(int i) {
     return [i];
}

def bool* makePtr(bool b) {
     return &b;
}
```

typeparams - The declaration of parameters with their type. Used as the input of a function, because the input needs to be of certain types. It can contain any amount of variables.
'(' (**type** ID)* ')'

params - The declaration of parameters without explicit type. Used in function calls, because the type of a variable is inferred. It can contain any amount of variables.
(**expr***)

call - The declaration of a function call.
ID **params**
Examples:
```
main()                      /* main() */
someFunc1(5, [1,2,3])       /* someFunc1(int i, int[] arr) */
someFunc2(&boolVar)         /* someFunc2(bool* b) */
someFunc3(*intPtr)          /* someFunc3(int i) */
```

expr - The declaration of an expression, it can be of either one of these types:

negNumExpr - Defines the negation of a number.
- expr
```
int x = -(5 + 4);
int y = --x; /* not a pre-decrement but double negation */
int z = -intProducingFunction();
```

negBoolExpr - Defines the negation of a boolean.
! **expr**

```
bool x = !true;
bool y = !x;
bool z = !boolProducingFunction();
```

refExpr - Returns the address of the value of the contained expression.
& **expr**

```
int* ptr = &i;
int** ptrptr = &ptr;
```

derefExpr - Returns the value stored at the address given by the expression.
* **expr**

```
int x = *ptr;
int* p = *ptrptr;
int y = **ptrptr;
```

*It is a compile-time error if **expr** does not evaluate to a pointer type*

multExpr - Defines the multiplication of a number with another number.
expr * expr
*It is a compile-time error if either **expr** does not evaluate to an* int

divExpr - Defines the division of a number by another number.
expr / **expr**
*It is a compile-time error if either **expr** does not evaluate to an* int

minExpr - Defines the subtraction of a number with another number.
expr - **expr**
*It is a compile-time error if either **expr** does not evaluate to an* int

plusExpr - Defines the addition of a number by another number.
expr + **expr**
*It is a compile-time error if either **expr** does not evaluate to an* int

modExpr - Defines a modulo on a number with another number.
expr % **expr**
*It is a compile-time error if either **expr** does not evaluate to an* int

boolOpExpr - Defines an operation on *bool* values. (**AND, XOR, NOT,** etc).
expr boolOp expr
*It is a compile-time error if either **expr** does not evaluate to an* int

compExpr - Defines a comparison of two numbers.
expr comp expr
*It is a compile-time error if either **expr** does not evaluate to an* int

parExpr - Defines a nested expression, eaning that the expression is surrounded by brackets.
(**expr**)
```
      (9 / 3) * (2 + (4 % 2))
```

callExpr - Defines a call to a function.
call
It is a compile-time error if the called function returns void. Void *functions may only be called as a statement, not as an expression.*

constArrayExpr - Queries an array at a constant index.
ID [NUMBER]
```
      arr[3];
      arr[0];
```

exprArrayExpr - Queries an array at an index which is the result of an expression
arrayVal
It is a compile-time error if the expression does not evaluate to an int.

arrayLiteralExpr - Defines a literal array
'[' expr (, expr)* ']'
```
      [1,2,3]
      [true, false]
      [&x, &y, &z]
      [3 + 2, 5 % 3, intProducingFunction()]
```
It is a compile-time error if not all expressions evaluate to the same type, or if any expression has the void *pseudo-type.*

enumExpr - Specifies a previously defined enum constant.
TYPE DOT TYPE
```
      Enum.CONSTANT
      Days.MONDAY
      Primitive.BOOL
```
It is a compile-time error if either the enum is undeclared, or the constant was not declared within the enum.

idExpr - Defines the identifier (or name) of a variable.
ID

numExpr - Defines a number.
NUM

trueExpr - Defines the boolean value true.
TRUE

falseExpr - Defines the boolean value false.
FALSE

stat - The declaration of a statement. It can be either on of the following statements:

lockStat - Defines a lock statement.
LOCK (ID) **block**

```
LOCK(lock_name) { /* protected operations */ }
```

declStat - Defines a declaration statement. Declarations of a variable.
decl ;

assignStat - Defines an assign statement. Assignments of a variable.
assign ;

ifStat - Defines an if-(else-)statement. The amount of ELSE IF's can be unlimited.
IF **expr block**
IF **expr block** ELSE IF **expr block**
IF **expr block** ELSE IF **expr block** ELSE **block**
IF **expr block** ELSE **block**

```
if (x AND y) {
      ...
}

if (!x) {
      ...
} else {
      ...
}

if (x > y) {
      ...
} else if (y > x) {
      ...
} else {
      ...
}
```

```
      if (x == y) {
            ...
      } else if (x > 5) {
            ...
      }
```
It is a compile-time error if any ***expr*** *does not evaluate to a* bool.

whileStat - Defines a while-statement. The statements in **block** are executed if and only if **expr** holds. The programmer should ensure that the value of **expr** eventually becomes false. If **expr** is false from the start, **block** is never executed.
WHILE **expr block**
```
      while (i > 0) {
            ...
            i = i - 1;
      }

      while (x) { /* x is a bool */
            /* some ops that may change x */
      }

      while (someFunctionWithSideEffects()){}
      /* calls the function until it returns false */
```
It is a compile-time error if ***expr*** *does not evaluate to a* bool.

callStat - Defines a call-statement. If the function returns a (non-*void*) value, it is discarded.
call ;

blockStat - Defines a block-statement, the statements in this block are executed in a lower lexical scope.
block

forStat - Defines a for-statement. **decl** is performed exactly once. **block** is then executed for as long as **expr** hold (possibly never). After each execution of **block**, **assign** is performed.
FOR (**decl** ; **expr** ; **assign**) **block**
```
      for (int i = 1; i <= 10; i = i + 1) {
            sum = sum + i;
      }

      for (bool b = true; b; b = b) {
            performSomeOperation(&b);
```

```
        }
```

It is a compile-time error if:
-The declared variable already exists within the current scope.
*-**expr** does not evaluate to a* bool*.*

outStat - Defines an out-statement. Prints the expression that it gets as a parameter. Note that *bool* values are printed as their internal representations (0/1). In addition, note that multi-digit and/or negative values cannot be printed correctly this way. See the standard library for functions that can print such values.
OUT (**expr**) ;

```
out(3);          /* prints 3 (ASCII 51) */
out(func());     /* evaluates func and prints the result */
out(true);       /* prints 1 (ASCII 49) */
out(-1);         /* prints / (ASCII 47) */
out(15);         /* prints ? (ASCII 63) */
```

These last two are not errors, but likely undesired.

returnStat - Defines a return-statement.
RETURN ;
RETURN **expr** ;

```
return;
return 3;
return func();
return [1,2,3];
```

*It is a compile-time error if the type of **expr** does not match the declared return type of the function in which the return statement appears.*
It is a compile-time error if no return statement is present in the body of a function.

decl - The declaration of a variable. It may or may not contain the keyword SHARED, depending on if the variable should be global or not.
type ID = **expr** *(1)*
type ID = **type** [NUMBER] *(2)*
SHARED **type** ID = **expr** *(3)*
SHARED **type** ID = **type** [NUMBER] *(4)*

```
        int i = 3;
        bool b = i == 3;
        shared int* globalIntPtr = &i;
        /* the address i is stored in global memory, i itself is not.
*/      shared bool b2 = false;
```

It is a compile-time error if:

*-The **type** in variations 1 or 3 to the left of the = is* void.
*-The **expr** in variations 1 or 3 does not evaluate to the **type** on the left.*
*-Either **type** in variations 2 or 4 is* void.
*-The left-hand **type** in variations 2 or 4 is not an Array type.*
*-The left-hand **type** in variations 2 or 4 is the primitive or Pointer type wrapped by the right-hand **type**.*
-The NUMBER *in variations 2 or 4 is negative.*
-A variable with the same name has previously been declared within the same scope, regardless of type and location. (All variations)

enumDecl - Declares a new enum. Both the type name and all constants must start with a capital letter.
ENUM TYPE COLON TYPE (‘,’ TYPE)*

```
    enum Types: PRIMITIVE, POINTER, ARRAY, ENUM
    enum CaPiTaLiZaTiOn: DOES, Not, MAT_ter
```

assign - The assignment of a variable. Sets the value of a variable.
derefID = **expr**
arrayVal = **expr**

```
    i = 4;
    b = true;
    *ptr = func();
    arr[idx * 2] = *ptr;
```

*It is a compile time error if **expr** does not evaluate to the declared type of **derefID / arrayVal**.*

arrayVal - A query of a specific index in an array, given by **expr**. The programmer must ensure that the value of **expr** lies within the bounds of the array. Accessing an out-of-bounds index will result in undefined behaviour.
ID [**expr**]

```
    arr[1];
    arr[3*3];
    arr[*ptr];
    arr[func()];
    arr[boolToInt(true)]; /* standard library function */
```

*It is a compile-time error if **expr** does not evaluate to an* int, *or if it evaluates to a negative value.*

type - Defines the different types that can be used in our programming language.

INT	- Integer declaration
BOOL	- Boolean declaration
VOID	- Void declaration

TYPE	- User-defined type declaration -- *Only used for enums*
type *	- Declares a pointer. In contrast to arrays, multi-level pointers *are* supported.
type [ ]	- Declares an array. Note that multi-dimensional arrays are not supported.

val - Defines a value a variable can have.

NUMBER	- Declaration of a number
derefID	- *See below for the independent definition of* ***derefID***
TRUE	- Declaration of boolean value true
FALSE	- Declaration of boolean value false
$SPID	- The value stored in the register SPID in the local Sprockell.
arrayVal	- *See above for the independent definition of* ***arrayVal****.*

comp - Compare operators.

<	- Lesser than
>	- Greater than
==	- Equal to
<=	- Lesser than or equal to
>=	- Greater than or equal to
!=	- Not equal to

boolOp - Boolean operators.

AND	- And operation
OR	- Or operation
XOR	- eXclusive OR operation

prefix - This is used to negate either a number or a boolean.

MINUS	- Negate a number
NOT	- Negate a boolean value

derefID - An ID, possibly preceded by some number of dereferencing operators (‘*’)
*** derefID**
ID

```
x          /* int   */
*ptr       /* int*  */
**ptrptr   /* int** */
```

It is a compile-time error if the number of dereferencing operators exceeds the depth of the pointer chain indicated by ID:

```
**ptr /* error if ptr is an int* like above */
```

Appendix B

Extended test program

Program:

```
program extended;

int[] bases = [1,2,3,4,5,6,7,8,9,10];
int[] expected = [1, 2, 6, 25, 120, 720, 5040, 40320, 362880, 3628800];

enum Result : SUCCESS, FAILURE;

def int main() {
	for (int i = 0; i < 10; i = i + 1) {
		int val = bases[i];
		compute(&val);
		bases[i] = val;
	}
	Result res = Result.FAILURE;
	if (arraysEqual(expected, bases, 10)) {
		res = Result.SUCCESS;
	}
	bool b = processResult(&res, bases);
	if (b) {
		out(0);
	}
	return 0;
}

def void compute(int* ptr) {
	int fac = 1;
	for (int i = 1; i <= *ptr; i = i + 1) {
		fac = fac * i;
	}
	*ptr = fac;
	return;
}

def bool processResult(Result* result, int[] arr) {
	if (*result == Result.FAILURE) {
		printArray(arr, 10);
		return false;
	}
	return true;
}
```

Generated Code (not including standard library, see Appendix D):

```
 Const 1174 RegA              {-Initial Return Addr-}
 Push RegA
 Const (0-1) RegA             {-Initial Result Addr-}
 Push RegA
 Const 67 RegA                {-Base addr for array bases-}
```

```
Store RegA (Addr 0)
Const 1 RegA
Store RegA (Addr 67)
Const 2 RegA
Store RegA (Addr 68)
Const 3 RegA
Store RegA (Addr 69)
Const 4 RegA
Store RegA (Addr 70)
Const 5 RegA
Store RegA (Addr 71)
Const 6 RegA
Store RegA (Addr 72)
Const 7 RegA
Store RegA (Addr 73)
Const 8 RegA
Store RegA (Addr 74)
Const 9 RegA
Store RegA (Addr 75)
Const 10 RegA
Store RegA (Addr 76)
Const 77 RegA                  {-Base addr for array expected-}
Store RegA (Addr 1)
Const 1 RegA
Store RegA (Addr 77)
Const 2 RegA
Store RegA (Addr 78)
Const 6 RegA
Store RegA (Addr 79)
Const 25 RegA
Store RegA (Addr 80)
Const 120 RegA
Store RegA (Addr 81)
Const 720 RegA
Store RegA (Addr 82)
Const 5040 RegA
Store RegA (Addr 83)
Const 40320 RegA
Store RegA (Addr 84)
Const 362880 RegA
Store RegA (Addr 85)
Const 3628800 RegA
Store RegA (Addr 86)
Branch SPID (Rel 4)
TestAndSet (Addr 1)
Receive Zero
Jump (Rel 5)
Read (Addr 1)
Receive RegA
Branch RegA (Rel 2)
Jump (Rel (0-3))
Jump (Abs 57)                  {-Jump To Main Function-}
Const 52 RegD                  {-Declaration of i(=0); For loop declaration; Function main-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
```

```
Load (Addr 52) RegA           {-For loop condition-}
Push RegA
Const 10 RegA
Pop RegB
Compute GtE RegB RegA RegA    {-i Lt 10-}
Branch RegA (Rel 35)          {-Break from for loop-}
Const 53 RegD                 {-Declaration of val(=bases[i]); For loop body-}
Push RegD
Load (Addr 0) RegE
Load (Addr 52) RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Pop RegD
Store RegA (Deref RegD)
Load (Addr 53) RegA
Const 53 RegA                 {-Create pointer to val-}
Store RegA (Addr 10)
Const 84 RegA                 {-Return addr-}
Push RegA
Const 54 RegA                 {-Result addr-}
Push RegA
Jump (Abs 153)                {-Jump to function compute-}
Const 0 RegD                  {-bases[i] = val-}
Load (Deref RegD) RegD        {-Get base addr for array bases-}
Load (Addr 52) RegA
Compute Add RegA RegD RegD    {-Compute target addr-}
Push RegD
Load (Addr 53) RegA
Pop RegD
Store RegA (Deref RegD)
Const 52 RegD                 {-i = i+1; For loop assignment-}
Push RegD
Load (Addr 52) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Add RegB RegA RegA    {-i Add 1-}
Pop RegD
Store RegA (Deref RegD)
Jump (Rel (0-39))             {-Back to for loop-}
Const 54 RegD                 {-Declaration of res(=Result.FAILURE)-}
Push RegD
Const 1 RegA
Pop RegD
Store RegA (Deref RegD)
Load (Addr 1) RegA
Store RegA (Addr 24)
Load (Addr 0) RegA
Store RegA (Addr 25)
Const 10 RegA
Store RegA (Addr 26)
Const 118 RegA                {-Return addr-}
Push RegA
Const 55 RegA                 {-Result addr-}
Push RegA
Jump (Abs 803)                {-Jump to function arraysEqual-}
Load (Addr 55) RegA
```

```
Branch RegA (Rel 2)
Jump (Rel 6)
Const 54 RegD                {-res = Result.SUCCESS-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Const 57 RegD                {-Declaration of b(=processResult(&res,bases))-}
Push RegD
Load (Addr 54) RegA
Const 54 RegA                {-Create pointer to res-}
Store RegA (Addr 37)
Load (Addr 0) RegA
Store RegA (Addr 38)
Const 138 RegA               {-Return addr-}
Push RegA
Const 56 RegA                {-Result addr-}
Push RegA
Jump (Abs 200)               {-Jump to function processResult-}
Load (Addr 56) RegA
Pop RegD
Store RegA (Deref RegD)
Load (Addr 57) RegA
Branch RegA (Rel 2)
Jump (Rel 3)
Const 0 RegA
Out RegA
Const 0 RegA
Pop RegB                     {-Get Result addr-}
Pop RegC                     {-Get Return addr-}
Compute Lt RegB Zero RegD    {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)      {-Store Result-}
Jump (Ind RegC)              {-return-}
Const 11 RegD                {-Declaration of fac(=1); Function compute-}
Push RegD
Const 1 RegA
Pop RegD
Store RegA (Deref RegD)
Const 12 RegD                {-Declaration of i(=1); For loop declaration-}
Push RegD
Const 1 RegA
Pop RegD
Store RegA (Deref RegD)
Load (Addr 12) RegA          {-For loop condition-}
Push RegA
Load (Addr 10) RegA
Load (Deref RegA) RegA       {-Dereference pointer ptr-}
Pop RegB
Compute Gt RegB RegA RegA    {-i LtE *ptr-}
Branch RegA (Rel 20)         {-Break from for loop-}
Const 11 RegD                {-fac = fac*i; For loop body-}
Push RegD
Load (Addr 11) RegA
Push RegA
Load (Addr 12) RegA
Pop RegB
```

```
Compute Mul RegB RegA RegA    {-fac Mul i-}
Pop RegD
Store RegA (Deref RegD)
Const 12 RegD                 {-i = i+1; For loop assignment-}
Push RegD
Load (Addr 12) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Add RegB RegA RegA    {-i Add 1-}
Pop RegD
Store RegA (Deref RegD)
Jump (Rel (0-25))             {-Back to for loop-}
Const 10 RegD                 {-*ptr = fac-}
Load (Deref RegD) RegD
Push RegD
Load (Addr 11) RegA
Pop RegD
Store RegA (Deref RegD)
Pop RegB                      {-Get Result addr-}
Pop RegC                      {-Get Return addr-}
Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
Branch RegD (Rel 2)
Jump (Ind RegC)               {-return-}
Load (Addr 37) RegA           {-Function processResult-}
Load (Deref RegA) RegA        {-Dereference pointer result-}
Push RegA
Const 1 RegA
Pop RegB
Compute Equal RegB RegA RegA  {-*result Equal Result.FAILURE-}
Branch RegA (Rel 2)
Jump (Rel 18)
Load (Addr 38) RegA
Store RegA (Addr 39)
Const 10 RegA
Store RegA (Addr 40)
Const 217 RegA                {-Return addr-}
Push RegA
Const 39 RegA                 {-Result addr-}
Push RegA
Jump (Abs 901)                {-Jump to function printArray-}
Push Zero
Pop RegA
Pop RegB                      {-Get Result addr-}
Pop RegC                      {-Get Return addr-}
Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)       {-Store Result-}
Jump (Ind RegC)               {-return-}
Const 1 RegA
Pop RegB                      {-Get Result addr-}
Pop RegC                      {-Get Return addr-}
Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)       {-Store Result-}
Jump (Ind RegC)               {-return-}
```

The result with the current variables is that the program prints ‘0’, indicating success.
If the expected values are changed to erroneous ones, the program will print the (correctly) calculated array instead.

Appendix C

Overview of standard library functions

printInt / intToStr

These methods transform a number into a printable string. The string is represented as an

int array containing the digits. The difference is that *printInt* prints the result, whereas *intToStr* returns it.

Parameters:

int i - The integer to transform

Returns:

void for *printInt*

int[] containing digits for *intToStr*

Example:

```
int i = 1234;
int[] arr = intToStr(i); /* arr == [1, 2, 3, 4] */
i = -42;
printInt(i); /* prints "-42" (no quotes) */
```

printBool / boolToStr

These methods transform a boolean into either “TRUE” or “FALSE”. As above, *printBool* p

prints the result, *boolToStr* returns it.

Parameters:

bool b - The bool to transform

Returs:

void for *printBool*

int[] containing one of the ASCII strings “TRUE” or “FALSE”, shifted so they are correctly printed by the **out** statement.

Example:

```
bool b = true;
int[] arr = boolToStr(b) /* arr == [36,34,37,21] */
b = false;
printBool(b); /* prints "FALSE" (no quotes) */
```

shiftArray

Shifts all non-zero values from the end of an array to the start. This method exists mainly for use in *intToStr*.

Parameters:

int[] arr - The array to shift.

Returns:

Shifted int[]

Example:

```
int[] arr = [0,0,0,4,3,2];
int[] shifted = shiftArray(arr);
printArray(shifted); /* prints "[4, 3, 2, 0, 0, 0]" */
```

boolToInt

Converts a bool to an int.

Parameters:

bool b - The bool to convert.

Returns:

1 if b is true, 0 otherwise.

Example:

```
bool w = true;
bool x = false;
int y = boolToInt(w); /* y == 1 */
int z = boolToInt(z); /* z == 0 */
```

intToBool

Converts an int to a bool. Note that for the purpose of this function, all non-zero numbers are considered to correspond to the boolean value *true*.

Parameters:

int i - The int to convert.

Returns:

False if i == 0, True otherwise.

Example:

```
int i = 0;
int j = 1;
int k = 5;
bool x = intToBool(i); /* x == false */
bool y = intToBool(j); /* y == true */
bool z = intToBool(k); /* z == true */
```

arraysEqual

Tests two arrays for equality up to a given index. If the index exceeds the length of either array, the behaviour of this function is undefined. Note that both arrays need to be of the same type.

Parameters:

(int/bool)[] x - The first array.
(int/bool)[] y - The second array.
int len - The amount of elements to compare.

Returns:

True if and only if the first *len* elements of *x* are equal to the first *len* elements of *y*.

Example:

```
int[] x = [1,2,3,4];
int[] y = [1,2,3,4,5,6];
int[] z = [1,2,0,4];
arraysEqual(x, y, 4); /* true */
arraysEqual(x, y, 6); /* undefined: out of bounds for x */
arraysEqual(x, z, 4); /* false: x[2] != z[2] */
```

printArray

Prints some amount of elements of an array to stdout. If the specified amount is greater than the length of the array, this function's behaviour is undefined.

Parameters:

(int/bool)[] arr - The array to print.
int len - The amount of elements to print.

Returns:

void

Example:

```
printArray([1,2,3,4]); /* prints "[1, 2, 3, 4]" */
```

flipArray

Mirrors an array (in-place), between its first element and a given value.

Parameters:

(int/bool)[] arr - The array to mirror.
int len - The length of the array.

Returns:

void

Example:

```
int[] arr = [1,2,3,4];
flipArray(arr);
printArray(arr); /* prints "[4,3,2,1]" */
```

Appendix D

Listing of generated code for standard library

```
Load (Addr 42) RegA             {-Function intToStr-}
 Push RegA
 Const 0 RegA
 Pop RegB
 Compute GtE RegB RegA RegA     {-i GtE 0-}
 Push RegA
 Load (Addr 42) RegA
 Push RegA
 Const 10 RegA
 Pop RegB
 Compute Lt RegB RegA RegA      {-i Lt 10-}
 Pop RegB
 Compute And RegB RegA RegA     {-(i>=0) And (i<10)-}
 Branch RegA (Rel 2)
 Jump (Rel 11)
 Load (Addr 42) RegA
 Store RegA (Addr 87)
 Const 87 RegA                  {-Base addr for (anon) array 540642172-}
 Store RegA (Addr 43)
 Pop RegB                       {-Get Result addr-}
 Pop RegC                       {-Get Return addr-}
 Compute Lt RegB Zero RegD      {-Is Result addr valid?-}
 Branch RegD (Rel 2)
 Store RegA (Deref RegB)        {-Store Result-}
 Jump (Ind RegC)                {-return-}
 Const 88 RegA                  {-Base addr for array res-}
 Store RegA (Addr 44)
 Store Zero (Addr 88)
 Store Zero (Addr 89)
 Store Zero (Addr 90)
 Store Zero (Addr 91)
 Store Zero (Addr 92)
 Store Zero (Addr 93)
 Store Zero (Addr 94)
 Store Zero (Addr 95)
 Store Zero (Addr 96)
 Store Zero (Addr 97)
 Const 45 RegD                  {-Declaration of neg(=i<0)-}
 Push RegD
 Load (Addr 42) RegA
 Push RegA
 Const 0 RegA
 Pop RegB
 Compute Lt RegB RegA RegA      {-i Lt 0-}
 Pop RegD
 Store RegA (Deref RegD)
 Load (Addr 45) RegA
 Branch RegA (Rel 2)
 Jump (Rel 9)
 Const 42 RegD                  {-i = -i-}
 Push RegD
```

```
Load (Addr 42) RegA
Push RegA
Pop RegA
Compute Sub Zero RegA RegA
Pop RegD
Store RegA (Deref RegD)
Const 46 RegD                 {-Declaration of q(=0)-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Const 47 RegD                 {-Declaration of r(=0)-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Const 48 RegD                 {-Declaration of pos(=9)-}
Push RegD
Const 9 RegA
Pop RegD
Store RegA (Deref RegD)
Const 49 RegD                 {-Declaration of first(=true)-}
Push RegD
Const 1 RegA
Pop RegD
Store RegA (Deref RegD)
Load (Addr 49) RegA
Push RegA
Load (Addr 42) RegA
Push RegA
Const 0 RegA
Pop RegB
Compute NEq RegB RegA RegA    {-i NEq 0-}
Pop RegB
Compute Nor RegB RegA RegA    {-first Or (i!=0)-}
Branch RegA (Rel 52)
Const 49 RegD                 {-first = false-}
Push RegD
Push Zero
Pop RegA
Pop RegD
Store RegA (Deref RegD)
Const 46 RegD                 {-q = i/10-}
Push RegD
Load (Addr 42) RegA
Push RegA
Const 10 RegA
Pop RegB
Compute Div RegB RegA RegA    {-i Div 10-}
Pop RegD
Store RegA (Deref RegD)
Const 47 RegD                 {-r = i-(q*10)-}
Push RegD
Load (Addr 42) RegA
Push RegA
Load (Addr 46) RegA
Push RegA
```

```
Const 10 RegA
Pop RegB
Compute Mul RegB RegA RegA    {-q Mul 10-}
Pop RegB
Compute Sub RegB RegA RegA    {-i Sub (q*10)-}
Pop RegD
Store RegA (Deref RegD)
Const 44 RegD                 {-res[pos] = r-}
Load (Deref RegD) RegD        {-Get base addr for array res-}
Load (Addr 48) RegA
Compute Add RegA RegD RegD    {-Compute target addr-}
Push RegD
Load (Addr 47) RegA
Pop RegD
Store RegA (Deref RegD)
Const 48 RegD                 {-pos = pos-1-}
Push RegD
Load (Addr 48) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-pos Sub 1-}
Pop RegD
Store RegA (Deref RegD)
Const 42 RegD                 {-i = q-}
Push RegD
Load (Addr 46) RegA
Pop RegD
Store RegA (Deref RegD)
Jump (Rel (0-60))
Load (Addr 45) RegA
Branch RegA (Rel 2)
Jump (Rel 12)
Const 44 RegD                 {-res[pos] = -3-}
Load (Deref RegD) RegD        {-Get base addr for array res-}
Load (Addr 48) RegA
Compute Add RegA RegD RegD    {-Compute target addr-}
Push RegD
Const 3 RegA
Push RegA
Pop RegA
Compute Sub Zero RegA RegA
Pop RegD
Store RegA (Deref RegD)
Load (Addr 44) RegA
Store RegA (Addr 13)
Const 391 RegA                {-Return addr-}
Push RegA
Const 50 RegA                 {-Result addr-}
Push RegA
Jump (Abs 398)                {-Jump to function shiftArray-}
Load (Addr 50) RegA
Pop RegB                      {-Get Result addr-}
Pop RegC                      {-Get Return addr-}
Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)       {-Store Result-}
```

```
Jump (Ind RegC)                {-return-}
Const 98 RegA                  {-Base addr for array res; Function shiftArray-}
Store RegA (Addr 14)
Store Zero (Addr 98)
Store Zero (Addr 99)
Store Zero (Addr 100)
Store Zero (Addr 101)
Store Zero (Addr 102)
Store Zero (Addr 103)
Store Zero (Addr 104)
Store Zero (Addr 105)
Store Zero (Addr 106)
Store Zero (Addr 107)
Const 15 RegD                  {-Declaration of i(=0)-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Const 16 RegD                  {-Declaration of j(=0)-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Load (Addr 15) RegA
Push RegA
Const 9 RegA
Pop RegB
Compute Lt RegB RegA RegA      {-i Lt 9-}
Push RegA
Load (Addr 13) RegE
Load (Addr 15) RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Push RegA
Const 0 RegA
Pop RegB
Compute Equal RegB RegA RegA   {-arr[i] Equal 0-}
Pop RegB
Compute Nand RegB RegA RegA    {-(i<9) And (arr[i]==0)-}
Branch RegA (Rel 20)
Const 15 RegD                  {-i = i+1-}
Push RegD
Load (Addr 15) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Add RegB RegA RegA     {-i Add 1-}
Pop RegD
Store RegA (Deref RegD)
Const 16 RegD                  {-j = j+1-}
Push RegD
Load (Addr 16) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Add RegB RegA RegA     {-j Add 1-}
Pop RegD
```

```
 Store RegA (Deref RegD)
 Jump (Rel (0-35))
 Load (Addr 15) RegA
 Push RegA
 Const 10 RegA
 Pop RegB
 Compute GtE RegB RegA RegA    {-i Lt 10-}
 Branch RegA (Rel 26)
 Const 14 RegD                 {-res[i-j] = arr[i]-}
 Load (Deref RegD) RegD        {-Get base addr for array res-}
 Load (Addr 15) RegA
 Push RegA
 Load (Addr 16) RegA
 Pop RegB
 Compute Sub RegB RegA RegA    {-i Sub j-}
 Compute Add RegA RegD RegD    {-Compute target addr-}
 Push RegD
 Load (Addr 13) RegE
 Load (Addr 15) RegA
 Compute Add RegE RegA RegD
 Load (Deref RegD) RegA
 Pop RegD
 Store RegA (Deref RegD)
 Const 15 RegD                 {-i = i+1-}
Push RegD
 Load (Addr 15) RegA
 Push RegA
 Const 1 RegA
 Pop RegB
 Compute Add RegB RegA RegA    {-i Add 1-}
 Pop RegD
 Store RegA (Deref RegD)
 Jump (Rel (0-30))
 Load (Addr 14) RegA
 Pop RegB                      {-Get Result addr-}
 Pop RegC                      {-Get Return addr-}
 Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
 Branch RegD (Rel 2)
 Store RegA (Deref RegB)       {-Store Result-}
 Jump (Ind RegC)               {-return-}
 Load (Addr 2) RegA            {-Function printInt-}
 Push RegA
 Const 0 RegA
 Pop RegB
 Compute GtE RegB RegA RegA    {-i GtE 0-}
 Push RegA
 Load (Addr 2) RegA
 Push RegA
 Const 10 RegA
 Pop RegB
 Compute Lt RegB RegA RegA     {-i Lt 10-}
 Pop RegB
 Compute And RegB RegA RegA    {-(i>=0) And (i<10)-}
 Branch RegA (Rel 2)
 Jump (Rel 8)
 Load (Addr 2) RegA
 Out RegA
```

```
Pop RegB                      {-Get Result addr-}
Pop RegC                      {-Get Return addr-}
Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
Branch RegD (Rel 2)
Jump (Ind RegC)               {-return-}
Const 108 RegA                {-Base addr for array res-}
Store RegA (Addr 3)
Store Zero (Addr 108)
Store Zero (Addr 109)
Store Zero (Addr 110)
Store Zero (Addr 111)
Store Zero (Addr 112)
Store Zero (Addr 113)
Store Zero (Addr 114)
Store Zero (Addr 115)
Store Zero (Addr 116)
Store Zero (Addr 117)
Const 4 RegD                  {-Declaration of neg(=i<0)-}
Push RegD
Load (Addr 2) RegA
Push RegA
Const 0 RegA
Pop RegB
Compute Lt RegB RegA RegA     {-i Lt 0-}
Pop RegD
Store RegA (Deref RegD)
Load (Addr 4) RegA
Branch RegA (Rel 2)
Jump (Rel 9)
Const 2 RegD                  {-i = -i-}
Push RegD
Load (Addr 2) RegA
Push RegA
Pop RegA
Compute Sub Zero RegA RegA
Pop RegD
Store RegA (Deref RegD)
Const 5 RegD                  {-Declaration of q(=0)-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Const 6 RegD                  {-Declaration of r(=0)-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Const 7 RegD                  {-Declaration of pos(=9)-}
Push RegD
Const 9 RegA
Pop RegD
Store RegA (Deref RegD)
Const 8 RegD                  {-Declaration of first(=true)-}
Push RegD
Const 1 RegA
Pop RegD
Store RegA (Deref RegD)
```

```
Load (Addr 8) RegA
Push RegA
Load (Addr 2) RegA
Push RegA
Const 0 RegA
Pop RegB
Compute NEq RegB RegA RegA    {-i NEq 0-}
Pop RegB
Compute Nor RegB RegA RegA    {-first Or (i!=0)-}
Branch RegA (Rel 52)
Const 8 RegD                  {-first = false-}
Push RegD
Push Zero
Pop RegA
Pop RegD
Store RegA (Deref RegD)
Const 5 RegD                  {-q = i/10-}
Push RegD
Load (Addr 2) RegA
Push RegA
Const 10 RegA
Pop RegB
Compute Div RegB RegA RegA    {-i Div 10-}
Pop RegD
Store RegA (Deref RegD)
Const 6 RegD                  {-r = i-(q*10)-}
Push RegD
Load (Addr 2) RegA
Push RegA
Load (Addr 5) RegA
Push RegA
Const 10 RegA
Pop RegB
Compute Mul RegB RegA RegA    {-q Mul 10-}
Pop RegB
Compute Sub RegB RegA RegA    {-i Sub (q*10)-}
Pop RegD
Store RegA (Deref RegD)
Const 3 RegD                  {-res[pos] = r-}
Load (Deref RegD) RegD        {-Get base addr for array res-}
Load (Addr 7) RegA
Compute Add RegA RegD RegD    {-Compute target addr-}
Push RegD
Load (Addr 6) RegA
Pop RegD
Store RegA (Deref RegD)
Const 7 RegD                  {-pos = pos-1-}
Push RegD
Load (Addr 7) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-pos Sub 1-}
Pop RegD
Store RegA (Deref RegD)
Const 2 RegD                  {-i = q-}
Push RegD
```

```
Load (Addr 5) RegA
Pop RegD
Store RegA (Deref RegD)
Jump (Rel (0-60))
Load (Addr 4) RegA
Branch RegA (Rel 2)
Jump (Rel 21)
Const 3 RegD                  {-res[pos] = -3-}
Load (Deref RegD) RegD        {-Get base addr for array res-}
Load (Addr 7) RegA
Compute Add RegA RegD RegD    {-Compute target addr-}
Push RegD
Const 3 RegA
Push RegA
Pop RegA
Compute Sub Zero RegA RegA
Pop RegD
Store RegA (Deref RegD)
Const 7 RegD                  {-pos = pos-1-}
Push RegD
Load (Addr 7) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-pos Sub 1-}
Pop RegD
Store RegA (Deref RegD)
Const 9 RegD                  {-Declaration of i(=pos); For loop declaration-}
Push RegD
Load (Addr 7) RegA
Pop RegD
Store RegA (Deref RegD)
Load (Addr 9) RegA            {-For loop condition-}
Push RegA
Const 10 RegA
Pop RegB
Compute GtE RegB RegA RegA    {-i Lt 10-}
Branch RegA (Rel 16)          {-Break from for loop-}
Load (Addr 3) RegE            {-For loop body-}
Load (Addr 9) RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Out RegA
Const 9 RegD                  {-i = i+1; For loop assignment-}
Push RegD
Load (Addr 9) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Add RegB RegA RegA    {-i Add 1-}
Pop RegD
Store RegA (Deref RegD)
Jump (Rel (0-20))             {-Back to for loop-}
Pop RegB                      {-Get Result addr-}
Pop RegC                      {-Get Return addr-}
Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
Branch RegD (Rel 2)
```

```
Jump (Ind RegC)                {-return-}
Load (Addr 58) RegA            {-Function printBool-}
Branch RegA (Rel 2)
Jump (Rel 13)
Const 36 RegA
Store RegA (Addr 118)
Const 34 RegA
Store RegA (Addr 119)
Const 37 RegA
Store RegA (Addr 120)
Const 21 RegA
Store RegA (Addr 121)
Const 118 RegA                 {-Base addr for (anon) array 32863545-}
Store RegA (Addr 59)
Out RegA
Jump (Rel 14)
Const 22 RegA
Store RegA (Addr 122)
Const 17 RegA
Store RegA (Addr 123)
Const 28 RegA
Store RegA (Addr 124)
Const 35 RegA
Store RegA (Addr 125)
Const 21 RegA
Store RegA (Addr 126)
Const 122 RegA                 {-Base addr for (anon) array 662822946-}
Store RegA (Addr 60)
Out RegA
Pop RegB                       {-Get Result addr-}
Pop RegC                       {-Get Return addr-}
Compute Lt RegB Zero RegD      {-Is Result addr valid?-}
Branch RegD (Rel 2)
Jump (Ind RegC)                {-return-}
Load (Addr 20) RegA            {-Function boolToStr-}
Branch RegA (Rel 2)
Jump (Rel 18)
Const 36 RegA
Store RegA (Addr 127)
Const 34 RegA
Store RegA (Addr 128)
Const 37 RegA
Store RegA (Addr 129)
Const 21 RegA
Store RegA (Addr 130)
Const 127 RegA                 {-Base addr for (anon) array 1935365522-}
Store RegA (Addr 21)
Pop RegB                       {-Get Result addr-}
Pop RegC                       {-Get Return addr-}
Compute Lt RegB Zero RegD      {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)        {-Store Result-}
Jump (Ind RegC)                {-return-}
Jump (Rel 19)
Const 22 RegA
Store RegA (Addr 131)
Const 17 RegA
```

```
Store RegA (Addr 132)
Const 28 RegA
Store RegA (Addr 133)
Const 35 RegA
Store RegA (Addr 134)
Const 21 RegA
Store RegA (Addr 135)
Const 131 RegA                 {-Base addr for (anon) array 1335050193-}
Store RegA (Addr 22)
Pop RegB                       {-Get Result addr-}
Pop RegC                       {-Get Return addr-}
Compute Lt RegB Zero RegD      {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)        {-Store Result-}
Jump (Ind RegC)                {-return-}
Const 0 RegA
Store RegA (Addr 136)
Const 136 RegA                 {-Base addr for (anon) array 1757293506-}
Store RegA (Addr 23)
Pop RegB                       {-Get Result addr-}
Pop RegC                       {-Get Return addr-}
Compute Lt RegB Zero RegD      {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)        {-Store Result-}
Jump (Ind RegC)                {-return-}
Load (Addr 61) RegA            {-Function boolToInt-}
Branch RegA (Rel 2)
Jump (Rel 8)
Const 1 RegA
Pop RegB                       {-Get Result addr-}
Pop RegC                       {-Get Return addr-}
Compute Lt RegB Zero RegD      {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)        {-Store Result-}
Jump (Ind RegC)                {-return-}
Const 0 RegA
Pop RegB                       {-Get Result addr-}
Pop RegC                       {-Get Return addr-}
Compute Lt RegB Zero RegD      {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)        {-Store Result-}
Jump (Ind RegC)                {-return-}
Load (Addr 51) RegA            {-Function intToBool-}
Push RegA
Const 0 RegA
Pop RegB
Compute Equal RegB RegA RegA  {-i Equal 0-}
Branch RegA (Rel 2)
Jump (Rel 9)
Push Zero
Pop RegA
Pop RegB                       {-Get Result addr-}
Pop RegC                       {-Get Return addr-}
Compute Lt RegB Zero RegD      {-Is Result addr valid?-}
Branch RegD (Rel 2)
Store RegA (Deref RegB)        {-Store Result-}
Jump (Ind RegC)                {-return-}
```

```
 Const 1 RegA
 Pop RegB                      {-Get Result addr-}
 Pop RegC                      {-Get Return addr-}
 Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
 Branch RegD (Rel 2)
 Store RegA (Deref RegB)       {-Store Result-}
 Jump (Ind RegC)               {-return-}
 Const 27 RegD                 {-Declaration of i(=0); For loop declaration; Function
arraysEqual-}
 Push RegD
 Const 0 RegA
 Pop RegD
 Store RegA (Deref RegD)
 Load (Addr 27) RegA           {-For loop condition-}
 Push RegA
 Load (Addr 26) RegA
 Pop RegB
 Compute GtE RegB RegA RegA    {-i Lt len-}
 Branch RegA (Rel 32)          {-Break from for loop-}
 Load (Addr 24) RegE           {-For loop body-}
 Load (Addr 27) RegA
 Compute Add RegE RegA RegD
 Load (Deref RegD) RegA
 Push RegA
 Load (Addr 25) RegE
 Load (Addr 27) RegA
 Compute Add RegE RegA RegD
 Load (Deref RegD) RegA
 Pop RegB
 Compute NEq RegB RegA RegA    {-x[i] NEq y[i]-}
 Branch RegA (Rel 2)
 Jump (Rel 9)
 Push Zero
 Pop RegA
 Pop RegB                      {-Get Result addr-}
 Pop RegC                      {-Get Return addr-}
 Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
 Branch RegD (Rel 2)
 Store RegA (Deref RegB)       {-Store Result-}
 Jump (Ind RegC)               {-return-}
 Const 27 RegD                 {-i = i+1; For loop assignment-}
 Push RegD
 Load (Addr 27) RegA
 Push RegA
 Const 1 RegA
 Pop RegB
 Compute Add RegB RegA RegA    {-i Add 1-}
 Pop RegD
 Store RegA (Deref RegD)
 Jump (Rel (0-36))             {-Back to for loop-}
 Const 1 RegA
 Pop RegB                      {-Get Result addr-}
 Pop RegC                      {-Get Return addr-}
 Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
 Branch RegD (Rel 2)
 Store RegA (Deref RegB)       {-Store Result-}
 Jump (Ind RegC)               {-return-}
 Const 31 RegD                 {-Declaration of i(=0); For loop declaration; Function
arraysEqual-}
 Push RegD
 Const 0 RegA
 Pop RegD
 Store RegA (Deref RegD)
 Load (Addr 31) RegA           {-For loop condition-}
 Push RegA
 Load (Addr 30) RegA
 Pop RegB
 Compute GtE RegB RegA RegA    {-i Lt len-}
 Branch RegA (Rel 32)          {-Break from for loop-}
 Load (Addr 28) RegE           {-For loop body-}
 Load (Addr 31) RegA
 Compute Add RegE RegA RegD
 Load (Deref RegD) RegA
 Push RegA
 Load (Addr 29) RegE
 Load (Addr 31) RegA
 Compute Add RegE RegA RegD
 Load (Deref RegD) RegA
 Pop RegB
 Compute NEq RegB RegA RegA    {-x[i] NEq y[i]-}
 Branch RegA (Rel 2)
 Jump (Rel 9)
 Push Zero
 Pop RegA
 Pop RegB                      {-Get Result addr-}
 Pop RegC                      {-Get Return addr-}
 Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
 Branch RegD (Rel 2)
 Store RegA (Deref RegB)       {-Store Result-}
 Jump (Ind RegC)               {-return-}
 Const 31 RegD                 {-i = i+1; For loop assignment-}
 Push RegD
 Load (Addr 31) RegA
 Push RegA
 Const 1 RegA
 Pop RegB
 Compute Add RegB RegA RegA    {-i Add 1-}
 Pop RegD
 Store RegA (Deref RegD)
 Jump (Rel (0-36))             {-Back to for loop-}
 Const 1 RegA
 Pop RegB                      {-Get Result addr-}
 Pop RegC                      {-Get Return addr-}
 Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
 Branch RegD (Rel 2)
 Store RegA (Deref RegB)       {-Store Result-}
 Jump (Ind RegC)               {-return-}
 Const 43 RegA                 {-Function printArray-}
 Out RegA
 Const 41 RegD                 {-Declaration of i(=0); For loop declaration-}
 Push RegD
 Const 0 RegA
 Pop RegD
 Store RegA (Deref RegD)
```

```
Load (Addr 41) RegA           {-For loop condition-}
Push RegA
Load (Addr 40) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-len Sub 1-}
Pop RegB
Compute GtE RegB RegA RegA    {-i Lt (len-1)-}
Branch RegA (Rel 31)          {-Break from for loop-}
Load (Addr 39) RegE           {-For loop body-}
Load (Addr 41) RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Store RegA (Addr 2)
Const 928 RegA                {-Return addr-}
Push RegA
Const 42 RegA                 {-Result addr-}
Push RegA
Jump (Abs 494)                {-Jump to function printInt-}
Const 4 RegA
Push RegA
Pop RegA
Compute Sub Zero RegA RegA
Out RegA
Const 16 RegA
Push RegA
Pop RegA
Compute Sub Zero RegA RegA
Out RegA
Const 41 RegD                 {-i = i+1; For loop assignment-}
Push RegD
Load (Addr 41) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Add RegB RegA RegA    {-i Add 1-}
Pop RegD
Store RegA (Deref RegD)
Jump (Rel (0-39))             {-Back to for loop-}
Load (Addr 39) RegE
Load (Addr 40) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-len Sub 1-}
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Store RegA (Addr 2)
Const 962 RegA                {-Return addr-}
Push RegA
Const 42 RegA                 {-Result addr-}
Push RegA
Jump (Abs 494)                {-Jump to function printInt-}
Const 45 RegA
Out RegA
Pop RegB                      {-Get Result addr-}
Pop RegC                      {-Get Return addr-}
Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
Branch RegD (Rel 2)
Jump (Ind RegC)               {-return-}
Const 43 RegA                 {-Function printArray-}
Out RegA
Const 19 RegD                 {-Declaration of i(=0); For loop declaration-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Load (Addr 19) RegA           {-For loop condition-}
Push RegA
Load (Addr 18) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-len Sub 1-}
Pop RegB
Compute GtE RegB RegA RegA    {-i Lt (len-1)-}
Branch RegA (Rel 31)          {-Break from for loop-}
Load (Addr 17) RegE           {-For loop body-}
Load (Addr 19) RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Store RegA (Addr 58)
Const 996 RegA                {-Return addr-}
Push RegA
Const 20 RegA                 {-Result addr-}
Push RegA
Jump (Abs 683)                {-Jump to function printBool-}
Const 4 RegA
Push RegA
Pop RegA
Compute Sub Zero RegA RegA
Out RegA
Const 16 RegA
Push RegA
Pop RegA
Compute Sub Zero RegA RegA
Out RegA
Const 19 RegD                 {-i = i+1; For loop assignment-}
Push RegD
Load (Addr 19) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Add RegB RegA RegA    {-i Add 1-}
Pop RegD
Store RegA (Deref RegD)
Jump (Rel (0-39))             {-Back to for loop-}
Load (Addr 17) RegE
Load (Addr 18) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-len Sub 1-}
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Out RegA
Const 45 RegA
Out RegA
Pop RegB                      {-Get Result addr-}
Pop RegC                      {-Get Return addr-}
Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
Branch RegD (Rel 2)
Jump (Ind RegC)               {-return-}
Const 64 RegD                 {-Declaration of i(=0); Function flipArray-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Const 65 RegD                 {-Declaration of j(=len-1)-}
Push RegD
Load (Addr 63) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-len Sub 1-}
Pop RegD
Store RegA (Deref RegD)
Load (Addr 64) RegA
Push RegA
Load (Addr 65) RegA
Pop RegB
Compute GtE RegB RegA RegA    {-i Lt j-}
Branch RegA (Rel 47)
Const 66 RegD                 {-Declaration of t(=arr[j])-}
Push RegD
Load (Addr 62) RegE
Load (Addr 65) RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Pop RegD
Store RegA (Deref RegD)
Const 62 RegD                 {-arr[j] = arr[i]-}
Load (Deref RegD) RegD        {-Get base addr for array arr-}
Load (Addr 65) RegA
Compute Add RegA RegD RegD    {-Compute target addr-}
Push RegD
Load (Addr 62) RegE
Load (Addr 64) RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Pop RegD
Store RegA (Deref RegD)
Const 62 RegD                 {-arr[i] = t-}
Load (Deref RegD) RegD        {-Get base addr for array arr-}
Load (Addr 64) RegA
Compute Add RegA RegD RegD    {-Compute target addr-}
Push RegD
Load (Addr 66) RegA
Pop RegD
Store RegA (Deref RegD)
```

```
Const 64 RegD                 {-i = i+1-}
Push RegD
Load (Addr 64) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Add RegB RegA RegA    {-i Add 1-}
Pop RegD
Store RegA (Deref RegD)
Const 65 RegD                 {-j = j-1-}
Push RegD
Load (Addr 65) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-j Sub 1-}
Pop RegD
Store RegA (Deref RegD)
Jump (Rel (0-51))
Pop RegB                      {-Get Result addr-}
Pop RegC                      {-Get Return addr-}
Compute Lt RegB Zero RegD     {-Is Result addr valid?-}
Branch RegD (Rel 2)
Jump (Ind RegC)               {-return-}
Const 34 RegD                 {-Declaration of i(=0); Function flipArray-}
Push RegD
Const 0 RegA
Pop RegD
Store RegA (Deref RegD)
Const 35 RegD                 {-Declaration of j(=len-1)-}
Push RegD
Load (Addr 33) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA    {-len Sub 1-}
Pop RegD
Store RegA (Deref RegD)
Load (Addr 34) RegA
Push RegA
Load (Addr 35) RegA
Pop RegB
Compute GtE RegB RegA RegA    {-i Lt j-}
Branch RegA (Rel 47)
Const 36 RegD                 {-Declaration of t(=arr[j])-}
Push RegD
Load (Addr 32) RegE
Load (Addr 35) RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Pop RegD
Store RegA (Deref RegD)
Const 32 RegD                 {-arr[j] = arr[i]-}
Load (Deref RegD) RegD        {-Get base addr for array arr-}
Load (Addr 35) RegA
Compute Add RegA RegD RegD    {-Compute target addr-}
Push RegD
```

```
Load (Addr 32) RegE
Load (Addr 34) RegA
Compute Add RegE RegA RegD
Load (Deref RegD) RegA
Pop RegD
Store RegA (Deref RegD)
Const 32 RegD                  {-arr[i] = t-}
Load (Deref RegD) RegD         {-Get base addr for array arr-}
Load (Addr 34) RegA
Compute Add RegA RegD RegD     {-Compute target addr-}
Push RegD
Load (Addr 36) RegA
Pop RegD
Store RegA (Deref RegD)
Const 34 RegD                  {-i = i+1-}
Push RegD
Load (Addr 34) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Add RegB RegA RegA     {-i Add 1-}
Pop RegD
Store RegA (Deref RegD)
Const 35 RegD                  {-j = j-1-}
Push RegD
Load (Addr 35) RegA
Push RegA
Const 1 RegA
Pop RegB
Compute Sub RegB RegA RegA     {-j Sub 1-}
Pop RegD
Store RegA (Deref RegD)
Jump (Rel (0-51))
Pop RegB                       {-Get Result addr-}
Pop RegC                       {-Get Return addr-}
Compute Lt RegB Zero RegD      {-Is Result addr valid?-}
Branch RegD (Rel 2)
Jump (Ind RegC)                {-return-}
EndProg
EndProg
```