Report Final Project

Programming Paradigms

2020-2021

Huynh Quang Long (S2207931)

Jochem Groen (S2343878)

Summary

For the project we chose to implement a language that is similar to Java.
Here is a list of the features we implemented:

Operators
Our language supports multiplication(*), subtraction(-), addition(+) and it also supports the and(&) and or(|) operators. Next to this we also have the comparison operators namely: (>), (<), (<=),(!=), (>=)and (==)

Statements
We implemented the mandatory part, namely the If-then-else and the While statements.

Arrays
The language also supports multidimensional arrays that are declared in the form of e.g. [3]int a which is a one dimensional array of size 3. A multidimensional array would be declared in the form of [2][3]int b

Functions
Functions in our language are supported and can also be used with arguments. Our implementation supports recursiveness. The language does not support functions that return of type array.

Concurrency
Our language supports concurrency with parbegin parend model. It also offers lock(); and unlock(); statement.

Problems and Solutions

We have countered a bug in making the concurrency work, in particular, the program was stuck when we try to set a begin point. The solution that we came up is to separate the programs in Haskell, which allow us to better look at the code, therefore, I easily found a bug in setting a wrong line number.
Other than that, there is no big problems we faced during the project.

Language description

D-1.1, 1.2 Data types, simple expressions and variables

Variable declaration:

Syntax
(global)? type ID
type: dimension type| int| boolean
where the type can be primitive or array(in that case which means it can have some prefixes before the prime type), dimension is in form an integer
E.g. int a;
Boolean b;
[2][4]int a;

Usage:
It can be used in the program, in any scopes, as long as it is not already declared in the scope. It also overrides the value of the higher scope. The global declaration should be put on the top of the program even before the function declaration part.

Semantics:

It's used to set up the type along with the offset of the variable. The default value is 0 of int, false of boolean, or the array with the default of its primitive type.

Code generation: It does not really generate any code, but the checker will give it a type and an offset and type for the later use

Variable expression:

Syntax:
ID (LOCATION)*
where locations are the real off set of the expression in the array,
E.g. x in x+1, a[1] in b= a[1]

Usage:
It can be used anywhere in the main program, as long as it is declared in the same or higher scope or globally. The number of dimension locations(e.g. [1]) should be less than the declared type. It should comply with the typing rules (mentioned later)

Semantics:
It is used to get the value that it is associated with the variable. It loads the value stored in its offset (stored when declared) to do further computation or assignment

Note: the offset of the Id with locations are mentioned below in the assignment statement.

Code generation:
It loop through its size (calculated during the checking part), starting from its real offset, and copy its contents to the memory associated to the expression by using registers. The way the real offset is calculated will be discussed in the assignment part.

Expression:

Syntax:
Expression Operator Expression
Operator Expression
Integer
Boolean
Array (mentioned later)

The primitive expression is of type integer or boolean (e.g. 9 or true), the compound is expressed by one or two expressions with an operator.

Usage:
The type of the child expressions should be compatible with the context. Here are some rules
Equal: The left and right hand side should be the same type. It produces the type of boolean.
< > >= <= : The left and right hand side should be the int It produces the type of boolean.
Add, Mul, Sub. The expressions should be in the type of integer. It produces the type of integer.
Or, And, Xor: The expression should be in the type of boolean. It produces the type of boolean.

Semantics:
It is used to compute the new expression or printed as output.
Here are some specifics:
Equal: check the content of the left and right hand side.
Add, Mul, Sub: Compute arithmetic operations
Or, And, Xor: Compute boolean operations
< > >= <= : Compare the left hand side to right hand side

Code generation:
It will first generate the code for the child expressions, when the child expressions are loaded to the register/memory, it emits the related operation to produce the output, it would then store it to the register, or on the local memory. For equality, it may loop from the first memory to the last part of memory of the expression to check one by one.

ArrayExpression:

Syntax: “[” expr+ “]”
E.g. [[1 2 3] [4 2 5]], [[i*2,4][2,4-1]]
Usage:
The type of the contents in array must be the same. The type of the array compatible with the context.
Semantics: It is a representation of compound data of the same type expression.

Code generation: The checker will first go through the array and add the children(if the child is also an array, add its children in) node to its content list, The generator will visit its contents emit the computation of its contents from left to right, and load (emitted program does) the computed value to its corresponding memory/register.

D-1.3 Basic statements: Assignments, if and while (mandatory)

Assignments:
Syntax:
ID Location* “=” expression
E.g. a[2]=3;
b=4;

Usage: The type of the identifier (or the compound of id with given locations e.g. a[2] int a[2]=3;) should be the same as the RHS, it also needs to be declared before use. In the

case of an array, the number of dimensions should not be more than the number it declared. (e.g. [2]int a; a[1][0]; is illegal).

Code generation: It first visits and emits the expression of the RHS and loads it to some memory, then loops through the memory of the RHS and writes it to the computed offset.

Note: looping is just to generalize the process which would work for both int/boolean (can be seen as one loop) and the array.

The offset is computed by adding the stored offset to r_arp, if there are locations in the LHS, the program will emit the code for the computation the sum of the multiplication of the location to the corresponding size of its children type.
E.g. let say [4][2]int a is at offset 60, r_arp is 20 and we want to calculate the offset of a[3][1], so the it will add 20 to 60 and the offset of a[3][1] with respect to array a which is 3*2+1 =7. So its offset is 87.

If:

Syntax
If (boolean expr){ stat*} (else{stat*})?

```
if(a < 3) {
out a;
} else {
out a;
}
```

```
If (false){
        Out 2;
}
```

Usage

The if and while statements should be used with a condition inside the () body, this is a boolean expression, the else statement can be used in case of the if statement. Inside The {} block of if (then/else) and while is normal block, it will create a new scope.

Semantics:
If-Statement: The if-statement checks for a certain condition and if this condition holds or returns true, it will execute the code inside the first block, you can add an else statement block of the if statement to execute different code if the condition does not hold. If you do not use the else statement, nothing will be executed if the condition is false.

Code generation: The emitted code will first compute the boolean expression, and branch to then block (this is the set after emit code for else and then blocks) if it's true otherwise to else block. To simplify the program, the our emitted code will generate the else statement before the statement and set the jump to the end of the else statement (We don't need to set the then block jump operation).

While:

Syntax
While (boolean expr){ stat*}

```
while(a < 3) {
        out a;
        a=a+1;
}
```

Usage: mentioned in if part
Sematic: It will check the condition and run the inside block until the condition doesn’t hold, then it will continue with the program.
While-statement:

Code generation: The emitted code will first compute the boolean expression, and branch to the end if the condition is false. We also add the jump statement in the end of the block, to have the program go back to the while loop.

D-1.4 Concurrency (mandatory)

Concurrency:

Syntax

Thread creation

```
Thread {
        Block+
}
```

Lock/unlock: lock();,unlock();

For example:

```
Thread {
{
out 1;
}
{
out 2;
}

}
```

We can also have nested threads with

```
Thread {
{
        Thread{
```

```
            lock();
            Int x;
            Out x;
            unlock();
        }

        Thread{
            lock();
            Int x;
            Out x;
            unlock();
        }

}
```

Usage: it must be used in the top level of the scope, but possible to use it within a thread, the lock () and the unlock() grants the exclusive access to one thread at a time. They cannot be used in the main thread.

Semantics:
It is meant to create multiple threads, using parbegin parend model, the block inside will be executed concurrently, and the main thread will wait until all blocks are executed. It also offers a possibility to lock and unlock, which does not allow 2 or more sibling (the lock from the nested thread will not have any effects) threads to hold it at the same time, one has to wait for the other.

Code generation:
In the scanner phase, the checker will map the parse tree to which the thread belongs to (using inheritance rule, i.e. upon entry it will create a new thread and associate the context to the thread, otherwise the child thread would have the same thread number of

its parent). For the generation phase, it will emit the code to its mapped thread (represented by the program in Sprockell).
The thread creation will be done in 2 steps, first the parent write the 1 to the global memory, then the children wait (a while loop for waiting signal is emitted on the top of the thread) until it receives the signal (1) in the its shared memory cell (BEGINWAITZONE + threadNr). When a child thread finishes it writes 1 to its shared memory cell (ENDWAITZONE + threadNr) then their parent must receive the signal (if read doesn't read 1, it goes back to the condition and check again) for all of children before continuing(parend).

D-1.6 Procedures/functions (optional)

Syntax

Function declaration

ID (type ID (, type ID)*)) {stat+}

```
f(boolean a) {
        return ;
}

Int add(int a, int b) {
return a + b;
}
```

Function expression

ID ((ID: epxr)*)

```
out f(a:true);
out add(a:1 b:2);
```

Usage:

You must declare the return statement (if it doesnt return anything then just leave as return;)somewhere in the function declaration, the return type must be the same as the function type function and pass the arguments with a colon, like add(a:1 b:2). You don’t need a comma between the arguments. It is possible to give arrays as arguments to the function but our language does not support returning arrays so the function cannot return an array. It is possible that you don't put all values of the params in, in that case, the default value is used.

Semantics:
This feature makes a function that if called executes a certain piece of code without repetition in the main program, possibly on given arguments, it also supports the recursiveness, where the function can be revisited in one call.

Code generation: For the declaration we emit the normal code block, the parameters act as normal variables, which have its own offset type. Upon return, it will get the return_address to jump in the address r_arp-1, then jump to the address.

For the function expression, it calculates the expression that maps to the parameters, then loads them to its offset, it also calculates the jump distance by getting the current highest offset of the function. It loads the return address to the memory, which indicates the line after the jump. It adds the jump distance to the r_arp and subtracts it after the return address.

We also have a register namely R_EXPR_MEM_POINT which will add the total amount currently used memory to itself, by doing that it can differentiate the old expression parse tree and the new expression parse tree, that will make the recursiveness possible,

You may make use of your grammar specification as a basis for this description, but note that not every grammar rule necessarily corresponds to a language feature.

Description of the software

The global size is 100 and 200 (its used to calculate fib function)of local
We split the local memory into 2 parts, first is for offset second is for memory of the expressions. We associate the expression to a local memory, which is mapped to the expression context. In the global memory 70-80 is the lock zone, 80-90 is the beginWaitZone, 90-100 is the EndWaitZone

The checker class (extending baselistener) is responsible for getting, setting, types, offsets of variable or expression node. It also mapped the context node to the thread nr (same for function). Specifically, the children thread will get the number of its parent thread unless it enters a parallel block, then it will be associated with the new thread. It also maintains the symbol table in form of Deque<Scope>. When open a new Scope, it will add a new Scope the Deque the new Scope continue with the old scope offset. When closing scope the deque will be popped. The symbol table(the deque described above) is mapped to specific thread or function, default is threadNr 0.

The generator is responsible for generating the result. It extends the class baseVisitor, it will emit the code upon visit.

Result: Contains error list, symbol table and other necessary information for the generation phase, e.g. function type, global scope, thread scope, check if the function returns value. This is used to pass the information from the elaboration phase to the generation phase.

Compiler: write the file to the directory, run and get the output from sprockell

ParseException: throw any error related to the elaboration and the lexing phase. During the checking parse, the error is added by addErrors method.
ErrorListener: Our own listener, generate nicely handle the error
Type: Represents the type of expression, id, function which includes void (function only), int , Boolean and array.

Model Package:
AddrImmDI: Represents the AddrImmDI (address and immediate value) in Sprockell
Num: Represents numbers and registers
Opcode: Represents the opcode of the operation
Op: Represents the operation
Operand: abstract class that represents the operands of the operation
Operator: Represents operators in Sprockell
Program: Represents a thread or a program in Sprockell.

Test plan and Results

For testing we made multiple folders that contain different kinds of testing. We have a folder with samples. These are example codes that we wrote for every feature of the language. For every feature we have a folder that contains two folders. One for examples of correct usage of the language feature, and one for incorrect usage. Most of these contain multiple examples to test different aspects of that feature and check for different errors. The testing of these samples goes almost automatically. For all these samples, we made tests that check if the behavior is correct. We made contextual tests,

semantic tests and syntax tests. For all the features, we made tests in all three of these categories. If you want to run these tests, then you go to the test folder and you can right click on the “java” file and then run “All tests”. This will run all the JUNIT tests. If you want to manually check the output of the tests you can also go to resources and in the file Sprockell.sprockell. There are folders with ErrorOutput and OutPut. These contain the output of the tests so you can see what the compiler gave back as result or what kind of error per test. You can write your own file and run it under the main method the Compiler class. There is small unit tests for checking typing, offset and how the scope works. There is a main in the compiler class which allows you to run the program manually.

Conclusions

We feel like we managed to implement quite a lot of good features and next to making the mandatory parts work we also managed to implement some nice extra features that sometimes gave an extra challenge but was nice working on knowing that it was optional. It is fun to realise that we actually made our own programming language and can do some cool things with it. By making this we also realise how complicated the standard languages like java are. Overall we feel like we delivered a good project in the end.

This module was one of the hardest modules so far in this study so this final project was obviously not easy. I liked how much we learnt in this module and that it was a really a completely new aspect of computer science with programming languages that were

nothing like the languages we have learned so far. This however, made it really hard to understand sometimes and could feel like too much information in too little time once in a while.

Appendices

Grammar

```
grammar G71;

/** G71 Program. */
program
   :  globalDecl* function* mainProg  ;


mainProg: stat*;
function: type? ID LPAR params? RPAR block;
globalDecl: GLOBAL type ID SEMI;

/** Grouped sequence of statements, it also creates a new scope */
block
   : LBRACE stat* RBRACE
   ;

/** Special block that create a new thread executing what is inside */
parBlock: block;
```

```
/** Statement. */
stat: ID exprDimension* ASS expr SEMI              #assStat
   | type ID SEMI                                  #declStat
   | THREAD LBRACE parBlock+ RBRACE                #parallelStat
   | LOCK LPAR RPAR   SEMI                         #lockStat
   | UNLOCK LPAR RPAR SEMI                         #unlockStat
   | IF LPAR expr RPAR  block (ELSE block)?        #ifStat
   | WHILE LPAR expr RPAR block                    #whileStat
   | OUT expr SEMI                                 #outStat
   | RETURN expr?  SEMI                            #returnStat
   | expr SEMI                                     #exprStat
   | block                                         #blockStat
   ;

params:
   type ID (COMMA type ID)*
   ;

dimension:
   LBRACK NUM RBRACK
   ;

exprDimension:
   LBRACK expr RBRACK
   ;

/** Expression. */
expr: NOT expr                #notExpr
   | SUB expr                 #negExpr
   | expr mulOp expr          #mulExpr
   | expr addOp expr          #addExpr
```

```
  | expr booleanOp expr       #boolExpr
  | expr compOp expr          #compExpr
  | expr EQUAL expr           #eqExpr
  | LPAR expr RPAR            #parExpr
  | ID exprDimension*         #idExpr
  | NUM                       #numExpr
  | LBRACK expr+ RBRACK       #arrayExpr
  | ID LPAR paramMap* RPAR    #funcExpr
  | TRUE                      #trueExpr
  | FALSE                     #falseExpr
  ;

paramMap:
  ID COLON expr
  ;
addOp:
  ADD|SUB
  ;

mulOp:
  MUL
  ;

booleanOp:   AND | OR ;
compOp:  NEQ | GT | GTE | LT | LTE ;
/** Data type. */
type: INTEGER
  | BOOLEAN
  | dimension type
  ;
```

```
BOOLEAN: 'boolean';
INTEGER: 'int' ;
ELSE:    'else' ;
FALSE:   'false';
IF:     'if';
THEN:    'then' ;
WHILE:   'while' ;
TRUE:    'true' ;
ASS:    '=';

//Built in binary operators
MUL: '*';
ADD: '+';
SUB: '-';
EQUAL: '==';
NEQ: '!=';
GT: '>';
GTE: '>=';
LT: '<';
LTE: '<=';
AND: '&';
OR: '|';

//Prefix operator
NOT: '!';

//Used to print to console
OUT: 'out';

THREAD: 'thread';
GLOBAL: 'global';
```

```
LBRACE: '{';
LPAR:   '(';
LBRACK:  '[';
RBRACE: '}';
RPAR:   ')';
RBRACK:  ']';
SEMI:   ';';
COLON: ':';
COMMA :',';


LOCK: 'lock';
UNLOCK: 'unlock';
RETURN: 'return';


ID: LETTER (LETTER | DIGIT)*;
NUM: DIGIT (DIGIT)*;


fragment LETTER: [a-zA-Z];
fragment DIGIT: [0-9];

COMMENT:   '//' ~[\r\n]* -> skip;
WS: [ \t\r\n]+ -> skip;
```

Test Program

Extended test program for the summing over an array in a function:

```
global int a;
int sum([4]int arr){
   int i;
   int sum;
   while(i<4){
      sum=sum+arr[i];
      i=i+1;
   }
   return sum;
}

a= sum(arr:[1 2 3 6]);
out a;
```

```
import Sprockell
prog0 =
            [Jump (Abs (136))
            , Load (ImmValue (4)) (2)
            , Compute Add (7) (2) (2)
            , Load (IndAddr (2)) (3)
            , Load (ImmValue (1)) (4)
            , Compute Add (4) (2) (2)
            , Load (ImmValue (0)) (4)
            , Compute Add (4) (5) (4)
            , Store (3) (IndAddr (4))
            , Load (ImmValue (4)) (2)
            , Load (ImmValue (1)) (4)
            , Compute Add (4) (5) (4)
            , Store (2) (IndAddr (4))
            , Load (ImmValue (0)) (2)
            , Compute Add (2) (5) (2)
            , Load (IndAddr (2)) (2)
            , Load (ImmValue (1)) (3)
            , Compute Add (3) (5) (3)
```

, Load (IndAddr (3)) (3)
, Compute Lt (2) (3) (2)
, Load (ImmValue (2)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (2)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Branch (2) (Rel (2))
, Jump (Abs (116))
, Load (ImmValue (5)) (2)
, Compute Add (7) (2) (2)
, Load (IndAddr (2)) (3)
, Load (ImmValue (1)) (4)
, Compute Add (4) (2) (2)
, Load (ImmValue (3)) (4)
, Compute Add (4) (5) (4)
, Store (3) (IndAddr (4))
, Load (ImmValue (0)) (2)
, Compute Add (7) (2) (2)
, Load (ImmValue (4)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (4)) (2)
, Compute Add (7) (2) (2)
, Load (IndAddr (2)) (3)
, Load (ImmValue (1)) (4)
, Compute Add (4) (2) (2)
, Load (ImmValue (5)) (4)
, Compute Add (4) (5) (4)
, Store (3) (IndAddr (4))
, Load (ImmValue (5)) (3)
, Compute Add (3) (5) (3)
, Load (IndAddr (3)) (3)
, Load (ImmValue (4)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Load (ImmValue (1)) (4)
, Compute Mul (3) (4) (3)
, Compute Add (2) (3) (2)
, Load (ImmValue (4)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (IndAddr (2)) (3)
, Load (ImmValue (1)) (4)
, Compute Add (4) (2) (2)
, Load (ImmValue (6)) (4)
, Compute Add (4) (5) (4)
, Store (3) (IndAddr (4))
, Load (ImmValue (3)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Load (ImmValue (6)) (3)

, Compute Add (3) (5) (3)
, Load (IndAddr (3)) (3)
, Compute Add (2) (3) (2)
, Load (ImmValue (7)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (5)) (2)
, Compute Add (7) (2) (2)
, Load (ImmValue (7)) (3)
, Compute Add (3) (5) (3)
, Load (IndAddr (3)) (3)
, Store (3) (IndAddr (2))
, Load (ImmValue (1)) (4)
, Compute Add (4) (2) (2)
, Load (ImmValue (4)) (2)
, Compute Add (7) (2) (2)
, Load (IndAddr (2)) (3)
, Load (ImmValue (1)) (4)
, Compute Add (4) (2) (2)
, Load (ImmValue (8)) (4)
, Compute Add (4) (5) (4)
, Store (3) (IndAddr (4))
, Load (ImmValue (1)) (2)
, Load (ImmValue (9)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (8)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Load (ImmValue (9)) (3)
, Compute Add (3) (5) (3)
, Load (IndAddr (3)) (3)
, Compute Add (2) (3) (2)
, Load (ImmValue (10)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (4)) (2)
, Compute Add (7) (2) (2)
, Load (ImmValue (10)) (3)
, Compute Add (3) (5) (3)
, Load (IndAddr (3)) (3)
, Store (3) (IndAddr (2))
, Load (ImmValue (1)) (4)
, Compute Add (4) (2) (2)
, Jump (Abs (1))
, Load (ImmValue (5)) (2)
, Compute Add (7) (2) (2)
, Load (IndAddr (2)) (3)
, Load (ImmValue (1)) (4)
, Compute Add (4) (2) (2)
, Load (ImmValue (11)) (4)
, Compute Add (4) (5) (4)
, Store (3) (IndAddr (4))

, Load (ImmValue (11)) (3)
, Compute Add (3) (5) (3)
, Load (IndAddr (3)) (3)
, Compute Add (7) (0) (2)
, Load (ImmValue (-1)) (4)
, Compute Add (2) (4) (2)
, Store (3) (IndAddr (2))
, Compute Add (7) (0) (2)
, Load (ImmValue (-2)) (4)
, Compute Add (2) (4) (2)
, Load (IndAddr (2)) (6)
, Jump (Ind (6))
, Load (ImmValue (50)) (4)
, Compute Add (5) (4) (5)
, Load (ImmValue (1)) (2)
, Load (ImmValue (12)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (12)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Load (ImmValue (13)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (2)) (2)
, Load (ImmValue (17)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (17)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Load (ImmValue (14)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (3)) (2)
, Load (ImmValue (18)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (18)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Load (ImmValue (15)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (6)) (2)
, Load (ImmValue (19)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (19)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Load (ImmValue (16)) (4)
, Compute Add (4) (5) (4)

, Store (2) (IndAddr (4))
, Load (ImmValue (2)) (4)
, Compute Add (7) (4) (7)
, Compute Add (7) (0) (2)
, Load (ImmValue (-2)) (4)
, Compute Add (2) (4) (2)
, Load (ImmValue (212)) (6)
, Store (6) (IndAddr (2))
, Load (ImmValue (0)) (3)
, Compute Add (3) (7) (3)
, Load (ImmValue (13)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Store (2) (IndAddr (3))
, Load (ImmValue (1)) (3)
, Compute Add (3) (7) (3)
, Load (ImmValue (14)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Store (2) (IndAddr (3))
, Load (ImmValue (2)) (3)
, Compute Add (3) (7) (3)
, Load (ImmValue (15)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Store (2) (IndAddr (3))
, Load (ImmValue (3)) (3)
, Compute Add (3) (7) (3)
, Load (ImmValue (16)) (2)
, Compute Add (2) (5) (2)
, Load (IndAddr (2)) (2)
, Store (2) (IndAddr (3))
, Load (ImmValue (20)) (4)
, Compute Add (5) (4) (5)
, Jump (Abs (1))
, Compute Add (7) (0) (2)
, Load (ImmValue (-1)) (4)
, Compute Add (2) (4) (2)
, Load (IndAddr (2)) (2)
, Load (ImmValue (-2)) (4)
, Compute Add (7) (4) (7)
, Load (ImmValue (-20)) (4)
, Compute Add (5) (4) (5)
, Load (ImmValue (20)) (4)
, Compute Add (4) (5) (4)
, Store (2) (IndAddr (4))
, Load (ImmValue (0)) (2)
, Load (ImmValue (20)) (3)
, Compute Add (3) (5) (3)
, Load (IndAddr (3)) (3)
, WriteInstr (3) (IndAddr (2))
, Load (ImmValue (1)) (4)
, Compute Add (4) (2) (2)

```
            , Load (ImmValue (0)) (2)
            , ReadInstr (IndAddr (2))
            , Receive (3)
            , Load (ImmValue (1)) (4)
            , Compute Add (4) (2) (2)
            , Load (ImmValue (21)) (4)
            , Compute Add (4) (5) (4)
            , Store (3) (IndAddr (4))
            , Load (ImmValue (21)) (2)
            , Compute Add (2) (5) (2)
            , Load (IndAddr (2)) (2)
            , WriteInstr (2) (DirAddr (65536))
            , EndProg
            ]
main = run [prog0]
```

*Main> Sprockell 0 says 12