Module 8 Programming Paradigms

Final Project Report

Group 22

Aliaksei Kouzel (s2648563)

Denis Krylov (s2808757)

Serkan Akin (s2727218)

5th July, 2023

Contents

Summary

Our language supports the following:

- integers, chars, booleans, strings and arrays
- for and while loops
- if/else conditions
- parallel execution (can be nested)
- synchronization using the locking mechanism
- functions (can be nested)
- soft division (in "/demo/math.as")

Concurrency

You can parallelize blocks of data, specifying the number of processes. The implementation is similar to parbegin/parend, so that when you wrap your code in the “parallel” block, the main process invites busy waiting workers (processes) to this block, and waits until they are finished. This approach allows many parallel blocks in a sequence, as workers can be reused.

Moreover, it supports nested parallelization, and it uses partial synchronization so that the same workers are not being used by asynchronically running processes.

Libraries

As our language supports functions, we have written the following libraries:

/demo/math.as	for mathematical operations (incl. soft division)
/demo/fib.as	for calculating the fibonacci sequence

Problems and Solutions

Not enough memory

The main problems for the code generation were related to the memory restrictions of Sprockell, to be exact we had 8 blocks for the shared memory, and 32 blocks for the local memory. Because of that, recursive functions could execute only 4-5 iterations, leading to the local memory being fully filled.

Also, we can support only a limited number of parallel processes as each process requires a block in the shared memory. Although, we partially solved this problem by reusing workers across different parallel blocks.

Expressions producing arrays

At the beginning, we designed our compiler to store the results of each expression in a single register. It was convenient at that time, however, when we started working on arrays, we understood that it does not work well with this approach. Therefore, arrays cannot be passed to functions as arguments, as well as cannot be returned. Also, we cannot copy arrays from variables to variables and cannot compare them.

Nested scoping

For supporting the nested scoping, we build a static table for variables at each scope, and reserve a register storing the ARP value. Also, the last variable at each scope is the ARP register which should be used by the next scope to access the upper variables. It is located at the end, so that we always assume it is located at offset ARP - 1.

Scope counting in Elaborator

While counting scopes in the elaboration phase, one of the difficulties was to compute and assign scopes correctly to the variables when they are created or used. It was needed to control that the variable in a neighbor scope, for example, won't be called by the compiler. That's why the elaboration phase needed to prevent it. The first version of the scope counter had a bug which incorrectly determined the previous scope and as a solution, the scope map was created that contains current scope as a key and previous as a value. It helps to go back to the correct scope and also keep track of the neighboring scope. In addition, the previous scope was saved to correctly increment the scope (for example, if the previous scope was (1,0), the next one will be (1,1), not (1,0) again).

Detailed Language Description

Basic Types: Int, Char and Bool

Syntax

```
Bool: true | false
Int: any number
Char: any symbol ASCII that wrapped in ' '
```

Usage

```
print(5 + 5);
print(true || false);
print('a');
```

Semantics

Checks that the type is primitive (Bool, Int or Char) before printing. Bool is converted to 0 (false) and 1 (true) for the compiler.

Expressions

Syntax

```
FunCall:       foo()
Ternary:       if expr ? thenExpr : elseExpr
ArrAccess:     arr[0]
Both:          x && y
OneOf:         x || y
Eq:            x == y
MoreEq:        x >= y
LessEq:        x <= y
More:          x > y
Less:          x < y
Add:           x + y
Sub:           x - y
Mult:          x * y
Neg:           -x
Var:           some variable
Fixed:         any integer, boolean, char, string or array
```

Usage

```
let x: Int = 2 + 2;
let y: Int = (2 + 2) * 2;
let z: Bool = true || true && false;
let b: Bool = false;
let r: Bool = z == b;
let c: Char = 'c';
```

In total we have 16 possible expressions: 3 non-basic (function call, ternary operation, array access), 2 logical (&& and ||), 5 comparison operations (==, >=, <=, >, <), 3 algebraic (addition, subtraction and multiplication) and 3 basics that represent a variable and a (negative) constant (number, boolean or char).

Semantics

To analyze the expressions above, the left-associativity was used. When algebraic operations need to be analyzed, it's checked that the both expressions between operations have the same type and this type is Int. When the logical operations need to be analyzed, it's checked that the both expressions in these operations have the same time and this type is Bool. For comparison operations it's only required the types on both sides be the same. When it's value or variable, and it is assigned to some variable, it's checked that the types of the variables or variable and value are the same.

Code Generation

- For binary expressions (Add, Sub, Mult, etc.), we compute the 1st expression, put the result onto the stack, compute the 2nd expression, put both results in free registers, and finally apply one of the Sprockell built-in instructions.
- For the variable (Var), we find the ARP in the static table of the scope where the variable is located. Then, we load the value that is located at ARP + offset, and put it in a free register.
- For values (Fixed), we just load a value to a free register.
- For the array access (ArrAccess), we do the same as for just a variable, but we also apply an additional offset, which is an element index.
- For the function call (FunCall), we first evaluate the arguments, put them onto the stack. Then, we load the function address from a variable, and the ARP of the scope where the function was declared. After that, we update the current ARP, save the static link and the return address as function variables. Next, we pop arguments from the stack, load them to the function's context, and jump by the function address. Finally, when we jump back from the function, we load the return value and put it in a free register.
- For the ternary operator (Ternary), it is the same as for the if/else condition (see below).
- For the negation (Neg), we use the NEq Sprockell instruction.

Scopes

Syntax

```
// scope (0,0)
{
   // scope (1,0)
   {
       // scope (2,0)
   }

   {
       // scope (2,1)
   }
}
```

Usage

```
let x: Int;
{
   x = 5; // No errors as "x" was declared in (0,0)
   {
       let y: Bool;
   }

   {
       x = 10; // No errors as "x" was declared in (1,0)
       y = false; // Undeclared variable error: (2,1)
                  // doesn't see from (2,0)
   }
}
```

Semantics

In the example above, the syntax to declare a new scope was used (two curly brackets). Everything in between automatically gets into a new (nested) scope. Each scope is defined as (Int, Int). The first number determines the depth of the scope relatively to the main scope (0,0). The second number describes the number of the scope in the same depth (neighbor scopes). Neighbor scopes don't see variables declared inside them but they both see all variables declared in themselves and wrapping scopes.

Code Generation

When we enter a new scope, we allocate a block in memory based on the scope's size (sum of all variable sizes), and put the current ARP to the end of the current memory block.

If-Else Statement

Syntax

```
if x < y { ... } else { ... }
```

Usage

```
let x: Bool = true;

if x == false {
   let x: Int = 0;
   print(x);
} else {
   print("x is true");
}
```

The If-statement is used to check a particular condition and then, depending on the result (true or false), execute one of them. In terms of scopes, an If-statement is equivalent to two neighbor scopes “then” and “else”.

Semantics

This feature contains the checking whether the expression given to `if` can be a boolean. It will also check whether there is a valid script for the `if` and `else.`

Code Generation

For the if/else statement, we evaluate the condition, and branch to the else block if the result is false, otherwise we continue to the if block where we optionally put a jump instruction to jump over the else block if it exists.

Also, these blocks are new scopes and thus we apply the same rules as for simple scopes (see above).

While & For Loops

Syntax

```
while true { … }
```

Usage

```
let x: Int = 5;
let y: Int = 0;

while x > y {
   print("x is still larger y");
}
```

While loop is needed if some code needs to be executed while some condition is holding. In terms of scopes, while loop is equivalent to one scope.

Semantics

This feature contains the checking whether the expression given to `while` can be a boolean. It will also check whether there is a valid script for the body of the loop.

Code Generation

For the while loop, we just need to compute the condition and branch over the body if the result is false. At the end, we add a jump instruction to the beginning.

Also, this block is a new scope and thus we apply the same rules as for simple scopes (see above).

Arrays

Syntax

```
let arr: Int[4] = [1,2,3,4];
```

Usage

```
let arr: Int[4] = [1, 2, 3, 4];
arr[0] = 0;
```

In the example above, the array can be defined by specifying the type and the size. After, each element of the array can be accessed using its index.

Semantics

This feature contains the checking that all variables in the array have the same type and this type is equal to the defined type (in the example above it is **Int**).

Functions

Syntax

```
fun incr(x: Int) -> Int { ... }
```

Usage

```
let x: Int = 0;
fun incr(x: Int) -> Int {
   x = x + 1;
   return x;
}

print(incr(x));
```

In the example above, the function that increments a variable is defined. It takes a number of arguments with specified type and returns a type

Semantics

The feature contains the checking of multiple conditions:

1. If the function has a return type, it should have a return keyword at the end of the body.
2. If the function doesn't have a return type, it shouldn't have a return keyword at the end of the body.
3. Checks that all argument's types are correct and match to types in the body.
4. After the return statement, expressions are not checked because they will never be executed in the backend.

Code Generation

For the function definition, the compiler stores its PC in a variable, so that future calls can find and jump to the function code, and then it compiles the function body. Next, it loads the return address from a variable that was assigned by the caller and restores the ARP register from the static link. Finally, it jumps to the caller's code by the loaded return address.

Standard Functions

Syntax

```
// Stores the current process ID
set_process_id(id);

// Print functions
print_str(string);
print(expr);

// Synchronizers
lock(num);
unlock(num);
```

Usage

```
let msg: String = "I love compilers!";
print_str(msg);

let pid: Int;
set_process_id(pid);

if pid == 0 {
   // execute some code
}
```

Two functions were used to provide the example of basic functions available in the language. The function “print_str()” printed a variable “msg” of type String, while set_process_id() set the ID of the process to the “pid” variable.

Description of the Software

Application files (/app)

Main.hs	API for the compiler that can be run using “stack run”

Source files (/src)

Compiler.hs	compiles AST into the SpriL language
Elaborator.hs	handles errors before compilation
Lexer.hs	defines language tokens & basic parsers
Parser.hs	parses the source code into AST
PreCompiler.hs	collects information about the program before compilation
Runner.hs	runs/debugs the program, and prints its compiled code
SprockellExt.hs	manages memory, registers, IO, variables and processes

Test files (/test)

ContextualTest.hs	checks if compilation errors are handled correctly
SemanticsTest.hs	checks the behavior of some /demo programs
SyntaxTest.hs	checks if the source code is parsed correctly
Main.hs	combines all tests that can be run using “stack test”

Demo files (/test)

banking.as	an elementary banking system, consisting of several processes
basic.as	demonstrates the basic functionality of the compiler
fib.as	algorithms for the fibonacci sequence
functions.as	an example of nested functions as well as nested variables
math.as	a basic mathematical library with support for division, etc
parallels.as	an example of the nested parallel execution
peterson.as	an implementation for the Peterson' algorithm
scopes.as	a fairly complex example of nested scoping with variables

Test Plan and Results

In order to make sure that our compiler works correctly, we had to come up with an extensive test plan and make sure that we were able to pass all the tests. In this part, we will present our test plan and its results to you.

Testing for Syntax Errors

Our first tests were focused on checking whether we were able to catch syntax errors. We have created multiple test cases which contain correct and incorrect syntax.

Correct Syntax

To verify whether an input was given correctly we try to parse it and if we do not have any errors parsing we pass the test. In order, to test every functionality of the compiler, we have created several testing codes to be parsed and checked.

Wrong Syntax

To verify whether an input was given incorrectly we try to parse it and try to catch an error. Once we successfully catch an error we are able to verify that indeed whenever a code with invalid syntax is given a syntax error will be thrown.

Testing for Contextual Errors

To test for contextual errors multiple test cases have been created. The tests were created such that they can test for multiple scenarios where a contextual error can occur.

We have created multiple test cases where we check for different scenarios with different error types. Here are some example cases that we have checked:

- Using variables that have not been declared.
- Using wrong data types in the assignment of a variable.
- Using a variable that does have a value yet.
- Creating global variables in the wrong scopes.
- Creating variables that have been already declared with the same name.

We have also checked whether code that contains no errors can be parsed and can pass the elaborator without any errors. Multiple test cases for this have also been added to our automated tests.

Testing for Semantic Errors

The final testing was done on semantic errors. While building our compiler, we have already created multiple files to test our language on. Using these and some new ones, we have created the semantic tests. Using semantic errors we can ensure that the code that we input is going to be executed in the right way and give the right output. Some examples of which functionalities we check in our system are:

- Declaring functions with or without arguments.
- Defining the return types of the functions.
- Recursive calls to functions.
- Simple functionalities like assigning values to variables, conditions, and arithmetic operations that we support.
- Nested function declarations.
- Concurrency implementations.

The way our semantic tests work is we first calculate the expected values of the processes and then we run our programs. The results are easily comparable with the expected values, thus we can be confident that our language can support all these functionalities and many more.

How to Run The Tests

To run all the tests except for semantic errors, it is simply enough to run “stack test” in the terminal. To see the results of the semantics tests the following steps can be executed in order:

```
stack ghci test/SemanticsTest.hs
runSemanticsTests
```

Contributions

Deciding on the syntax	Together
Scanner	Aliaksei Kouzel
Parser	Aliaksei Kouzel
Function Extension	Aliaksei Kouzel
Code Generation	Aliaksei Kouzel
Code Examples	Aliaksei Kouzel
Type Checking	Denis Krylov
Label Checking	Denis Krylov
Testing	Serkan Akin
Concurrency Extension	Aliaksei Kouzel, Serkan Akin
Test Automation	Aliaksei Kouzel, Serkan Akin
Report	Together

Conclusions

First of all, a lot of time was spent on research on how to start implementing the language. The main dilemma was whether to work with Haskell or with Java. As a result, Haskell was chosen because this choice looked more simple and logical to us.

One of the struggles we had was the fact that failing the project will lead to the need of extending the time of our studies, so we were stressed quite a bit. But at the same time we were truly excited about the fact of creating our own programming language that we tried our best and put a lot of effort into.

As a conclusion, we are happy about our language. We wished to implement custom data types or error catching. Also, it would be nice to convert the code to the actual Assembly code instead of using Sprockell. However, it’s nice to have what we already have and we hope that our effort and time working were worth it.

Appendix

Here is our grammar in haskell:

```
type FunName = String
type VarName = String
type ArrSize = Integer

type VarDef  = (VarName, DataType)
type ArgsDef = [VarDef]

type Script  = [Statement]

data Statement
    = VarDecl       VarDef (Maybe Expr)
    | GlVarDecl     VarDef (Maybe Expr)
    | VarAssign     VarName Expr
    | ArrInsert       VarName Integer Expr
    | FunDef          FunName ArgsDef (Maybe DataType) Script
    | ForLoop         VarDef LoopIter Script
    | WhileLoop     Expr Script
    | Condition       Expr Script (Maybe Script)
    | Parallel          Integer Script
    | InScope         Script
    | ReturnVal      Expr
    | Action            Expr

data LoopIter
    = IterRange Expr Expr

data Value
    = Bool    Bool
    | Char    Char
    | Int        Integer
    | Arr       [Value]
```

```
data DataType
   = CharType
    | BoolType
    | IntType
    | ArrType DataType ArrSize

data Expr
    = FunCall       FunName [Expr]
    | Ternary         Expr Expr Expr
    | ArrAccess    VarName Integer
    | Both             Expr Expr
    | OneOf          Expr Expr
    | Eq                Expr Expr
    | NotEq          Expr Expr
    | MoreEq        Expr Expr
    | LessEq        Expr Expr
    | More            Expr Expr
    | Less            Expr Expr
    | Add              Expr Expr
    | Sub              Expr Expr
    | Mult             Expr Expr
    | Neg             Expr
    | Var              VarName
    | Fixed          Value
```

Here is an example of a test program that implements soft division:

```
fun div(a: Int, b: Int) -> Int {
    if b <= 0 || b > a {
        error("invalid dividend");
    }
    let k: Int = 0;
    while a >= b {
        a = a - b;
        k = k + 1;
    }
    return k;
}

print(div(100, 3)); // result: 33
```

Which results in “Sprockell 0 says 33” in the terminal. Here is the compiled code:

```
0: Load (ImmValue 0) 2
1: Compute Equal 0 1 3
2: Branch 3 (Rel 14)
3: WriteInstr 0 (IndAddr 1)
4: ReadInstr (IndAddr 1)
5: Receive 3
6: Compute GtE 3 0 4
7: Branch 4 (Rel 2)
8: EndProg
9: Compute Equal 3 0 4
10: Branch 4 (Rel 5)
11: Load (ImmValue 1) 4
12: Compute Equal 3 4 4
13: Branch 4 (Rel 2)
14: Jump (Ind 3)
15: Jump (Abs 4)
16: Compute Add 0 2 3
17: Load (ImmValue 0) 4
18: Compute Add 4 3 3
19: Compute Add 0 9 4
```

```
20: Load (ImmValue 5) 5
21: Compute Add 4 5 4
22: Store 4 (IndAddr 3)
23: Jump (Rel 174)
24: Compute Add 0 2 4
25: Load (ImmValue 4) 5
26: Compute Add 4 5 5
27: Load (IndAddr 5) 3
28: Push 3
29: Load (ImmValue 0) 3
30: Pop 4
31: Compute LtE 4 3 3
32: Push 3
33: Compute Add 0 2 4
34: Load (ImmValue 4) 5
35: Compute Add 4 5 5
36: Load (IndAddr 5) 3
37: Push 3
38: Compute Add 0 2 4
39: Load (ImmValue 3) 5
40: Compute Add 4 5 5
41: Load (IndAddr 5) 3
42: Pop 4
43: Compute Gt 4 3 3
44: Pop 4
45: Compute Or 4 3 3
46: Compute Equal 3 0 3
47: Branch 3 (Rel 59)
48: Compute Add 0 2 3
49: Load (ImmValue 6) 4
50: Compute Add 4 3 4
51: Store 2 (IndAddr 4)
52: Load (ImmValue 7) 3
53: Compute Add 3 2 2
54: Load (ImmValue 101) 4
55: WriteInstr 4 (DirAddr 65537)
56: Load (ImmValue 114) 4
57: WriteInstr 4 (DirAddr 65537)
58: Load (ImmValue 114) 4
```

```
59: WriteInstr 4 (DirAddr 65537)
60: Load (ImmValue 111) 4
61: WriteInstr 4 (DirAddr 65537)
62: Load (ImmValue 114) 4
63: WriteInstr 4 (DirAddr 65537)
64: Load (ImmValue 58) 4
65: WriteInstr 4 (DirAddr 65537)
66: Load (ImmValue 32) 4
67: WriteInstr 4 (DirAddr 65537)
68: Load (ImmValue 105) 4
69: WriteInstr 4 (DirAddr 65537)
70: Load (ImmValue 110) 4
71: WriteInstr 4 (DirAddr 65537)
72: Load (ImmValue 118) 4
73: WriteInstr 4 (DirAddr 65537)
74: Load (ImmValue 97) 4
75: WriteInstr 4 (DirAddr 65537)
76: Load (ImmValue 108) 4
77: WriteInstr 4 (DirAddr 65537)
78: Load (ImmValue 105) 4
79: WriteInstr 4 (DirAddr 65537)
80: Load (ImmValue 100) 4
81: WriteInstr 4 (DirAddr 65537)
82: Load (ImmValue 32) 4
83: WriteInstr 4 (DirAddr 65537)
84: Load (ImmValue 100) 4
85: WriteInstr 4 (DirAddr 65537)
86: Load (ImmValue 105) 4
87: WriteInstr 4 (DirAddr 65537)
88: Load (ImmValue 118) 4
89: WriteInstr 4 (DirAddr 65537)
90: Load (ImmValue 105) 4
91: WriteInstr 4 (DirAddr 65537)
92: Load (ImmValue 100) 4
93: WriteInstr 4 (DirAddr 65537)
94: Load (ImmValue 101) 4
95: WriteInstr 4 (DirAddr 65537)
96: Load (ImmValue 110) 4
97: WriteInstr 4 (DirAddr 65537)
```

```
98: Load (ImmValue 100) 4
99: WriteInstr 4 (DirAddr 65537)
100: Load (ImmValue 10) 4
101: WriteInstr 4 (DirAddr 65537)
102: EndProg
103: Load (ImmValue (-1)) 3
104: Compute Add 2 3 3
105: Load (IndAddr 3) 2
106: Load (ImmValue 0) 3
107: Compute Add 0 2 4
108: Load (ImmValue 5) 5
109: Compute Add 5 4 5
110: Store 3 (IndAddr 5)
111: Compute Add 0 2 4
112: Load (ImmValue 3) 5
113: Compute Add 4 5 5
114: Load (IndAddr 5) 3
115: Push 3
116: Compute Add 0 2 4
117: Load (ImmValue 4) 5
118: Compute Add 4 5 5
119: Load (IndAddr 5) 3
120: Pop 4
121: Compute GtE 4 3 3
122: Compute Equal 3 0 3
123: Branch 3 (Rel 48)
124: Compute Add 0 2 3
125: Load (ImmValue 6) 4
126: Compute Add 4 3 4
127: Store 2 (IndAddr 4)
128: Load (ImmValue 7) 3
129: Compute Add 3 2 2
130: Load (ImmValue (-1)) 5
131: Compute Add 2 5 5
132: Load (IndAddr 5) 4
133: Load (ImmValue 3) 5
134: Compute Add 4 5 5
135: Load (IndAddr 5) 3
136: Push 3
```

```
137: Load (ImmValue (-1)) 5
138: Compute Add 2 5 5
139: Load (IndAddr 5) 4
140: Load (ImmValue 4) 5
141: Compute Add 4 5 5
142: Load (IndAddr 5) 3
143: Pop 4
144: Compute Sub 4 3 3
145: Load (ImmValue (-1)) 5
146: Compute Add 2 5 5
147: Load (IndAddr 5) 4
148: Load (ImmValue 3) 5
149: Compute Add 5 4 5
150: Store 3 (IndAddr 5)
151: Load (ImmValue (-1)) 5
152: Compute Add 2 5 5
153: Load (IndAddr 5) 4
154: Load (ImmValue 5) 5
155: Compute Add 4 5 5
156: Load (IndAddr 5) 3
157: Push 3
158: Load (ImmValue 1) 3
159: Pop 4
160: Compute Add 4 3 3
161: Load (ImmValue (-1)) 5
162: Compute Add 2 5 5
163: Load (IndAddr 5) 4
164: Load (ImmValue 5) 5
165: Compute Add 5 4 5
166: Store 3 (IndAddr 5)
167: Load (ImmValue (-1)) 3
168: Compute Add 2 3 3
169: Load (IndAddr 3) 2
170: Jump (Rel (-59))
171: Compute Add 0 2 4
172: Load (ImmValue 5) 5
173: Compute Add 4 5 5
174: Load (IndAddr 5) 3
175: Compute Add 0 2 4
```

```
176: Load (ImmValue 0) 5
177: Compute Add 5 4 5
178: Store 3 (IndAddr 5)
179: Compute Add 0 2 4
180: Load (ImmValue 1) 5
181: Compute Add 4 5 5
182: Load (IndAddr 5) 3
183: Compute Add 0 2 4
184: Load (ImmValue 2) 5
185: Compute Add 4 5 5
186: Load (IndAddr 5) 2
187: Jump (Ind 3)
188: Compute Add 0 2 4
189: Load (ImmValue 1) 5
190: Compute Add 4 5 5
191: Load (IndAddr 5) 3
192: Compute Add 0 2 4
193: Load (ImmValue 2) 5
194: Compute Add 4 5 5
195: Load (IndAddr 5) 2
196: Jump (Ind 3)
197: Load (ImmValue 100) 3
198: Push 3
199: Load (ImmValue 3) 3
200: Push 3
201: Compute Add 0 2 4
202: Load (ImmValue 0) 5
203: Compute Add 4 5 5
204: Load (IndAddr 5) 3
205: Compute Add 0 2 4
206: Compute Add 0 2 5
207: Load (ImmValue 1) 6
208: Compute Add 6 5 6
209: Store 4 (IndAddr 6)
210: Compute Add 0 2 4
211: Load (ImmValue 2) 5
212: Compute Add 5 2 2
213: Compute Add 0 2 5
214: Load (ImmValue 2) 6
```

```
215: Compute Add 6 5 6
216: Store 4 (IndAddr 6)
217: Pop 5
218: Compute Add 0 2 6
219: Load (ImmValue 4) 7
220: Compute Add 7 6 7
221: Store 5 (IndAddr 7)
222: Pop 5
223: Compute Add 0 2 6
224: Load (ImmValue 3) 7
225: Compute Add 7 6 7
226: Store 5 (IndAddr 7)
227: Compute Add 0 2 5
228: Load (ImmValue 1) 6
229: Compute Add 6 5 5
230: Compute Add 0 9 6
231: Load (ImmValue 5) 7
232: Compute Add 6 7 6
233: Store 6 (IndAddr 5)
234: Jump (Ind 3)
235: Load (ImmValue 2) 4
236: Compute Add 2 4 4
237: Load (IndAddr 4) 3
238: WriteInstr 3 (DirAddr 65536)
239: Load (ImmValue (-1)) 3
240: WriteInstr 3 (DirAddr 1)
241: EndProg
```