

UNIVERSITY OF TWENTE

Alia Programming Language

Fedor Beets
s1227874
Campuslaan 27

Joost van Doorn
s1095005
Dalsteindreef 2404, Diemen

July 9, 2014

Introduction

The Alia programming language was built for the final project of the compiler construction course at the University of Twente. In this final project a programming language is specified and implemented in antlr (ANother Tool for Language Recognition). A compiler is built to translate code into a type of machine instructions. The type of machine instructions can vary from language to language. Over the first half of compiler construction course several parts of the design process of building a compiler are individually learned and tested. In the final project this is brought to culmination by going through the entire process from the specification, to in the end a usable programming language.

This document serves as a specification of the Alia Programming Language (Alia for short) as well as an explanation of the inner workings of the Alia compiler. In it we will give a short description of the Alia Programming Language in the context of programming languages, explain some of the problems faced during the construction of the Alia compiler, and our solutions. Give a specification of the Alia language with the help of the syntax, context-constraints and semantics. Also in this document are the transformations that show how the symbols of the language are turned into JVM instructions, a description of all the auxiliary Java code made for the compiler. A set of tests is described to give confidence in the correctness of the compiler. Lastly conclusions are drawn from the project.

The Alia source code repository is located at: <https://github.com/JoostvDoorn/VertalerbouwEindproject>

1 Alia Programming Language

The Alia programming language is an expression language with type inference. Alia code is compiled to Java bytecode using Jasmin and can be run using the Java Virtual Machine (JVM). As an expression language every statement has a return type. Functional statements such as print, read and conditional statements all return values. For example the print function may return a value which can be used in an assignment statement.

```
x = print(34) + 1 // x is assigned 35
```

Alia contains compound statement which are a series of statements with the last statement as return value, compound statements are used in conditional statements and can be explicitly used for scoping.

```
x = begin
  y = 3 // Only declared in this scope
  y + 1 // Return value for the compound statement
end
if y = true; y && x < 10 do // y here has a different scope
  print(y) // Print the value of y
  print('t')
end
```

Types in Alia are inferred and do not need explicit declarations. Types can be declared explicitly within an assignment statement, but are not required. In the current form of the programming language, type inference can always deduce the type of a declaration. However in an extended version of Alia with functions and procedures, explicit type declarations will be required in cases where type inference will not

be able to deduce the type of the variable or function return type. Type inference makes programming easier, it reduces the work of the programmer by making explicit type declarations optional, and reduces the amount of code required for variable declarations. Type checking is maintained and the programmer still has the option to add type declarations if it helps clarify the code.

```
x = 54 : int // x is assigned 54, and is explicitly declared as an int
```

2 Problems and solutions

During the construction of the Alia compiler we ran into some problems, as is to be expected during any first time construction of a compiler. In this section we will explain some of the bigger problems that we faced as well as the solutions that we applied.

2.1 Scope definition

Because we do not have explicit declarations we cannot redefine a variable inside a new scope. There would be no way to distinguish between redefining a variable in a new scope and reassigning the value that was given to a variable to be used later. With a new variable that overwrites an old one for a temporary scope you would need to assign a new space in memory. We have decided that we want this in our programming language, this is because if you can redeclare a variable inside a new scope then you are only making it more confusing for yourself as a coder. On the other hand not being able to assign a new value to a variable in a new scope destroys a large part of the functionality of language. For these reasons we have chosen to leave it as is.

2.2 String template expressions

StringTemplate does not allow you to evaluate an expression inside of a string template. This was purposefully implemented to stop you from putting a large part of the logic in the string template itself. The problem that we have with this is that there are issues that are specific to the creation of java bytecode that must now be evaluated in the antlr part of the program, and must be passed in the creation of any possible target platform. The first time this became a problem was what instructions to use for outputting any given number. A naive solution to this would be to always use the instruction for loading a large number like an integer. But java has special instructions for loading the numbers -1 through 5 and loading smaller numbers that fit on a byte or a short, so we wanted to use these. To solve this issue we created a function that calculated what type a number can fit into in the CodeGeneratorAux class which passes a number of booleans wrapped in a NumberType to the string template. We then use conditional templates to emit different instructions depending on the number to be put on the stack. Another issue that resulted from this is, that while our StringTemplate is now relatively clean, more java code is called in the code generator. It is also harder to follow the DRY (Don't repeat yourself) principle, when there is not enough flexibility in the StringTemplate.

2.3 Constants

We chose to put the optimization of replacing all constants by their actual reference in the checker stage. The choice was made because in the checker there is already a list of declared variables, and Java functions

to do this. This made it very easy to implement this feature in the checker, and much more ugly to implement in the code generation. In an ideal world there would be a separate stage in-between the checker and the code generation that optimizes the abstract syntax tree, but this one feature was easily done in the checker. We have also chosen to not support constants that can vary, being defined by another identifier. In the Alia programming language a constant can only be defined by a single primitive type.

3 Syntax, context-constraints and semantics

The syntax of Alia is defined as follows:

```

program = (func_def | (statement end_statement) | \n)*;

statements = (statement (end_statement statements)? | \n statements)?;
statements_cond = (statement (end_statement statements)? | \n statements_cond )?;

statement = (expr_assignment | const_assignment) (; type)?
| while_stmnt
;

end_statement = \n | ";" | EOF;

expr_assignment = (identifier "=") expr_assignment
| expr
;

const_assignment = CONST identifier "=" primitive;

expr = expr1 ((or | "|" | "&") expr1)*;
expr1 = expr2 ((and | "&&") expr2)*;
expr2 = expr3 ((" "> | ">=" | "<" | "<=" | "==" | "!=") ^ expr3)*;
expr3 = expr4 (("+" | "-") ^ expr4)*;
expr4 = expr5 (("*" | "/" | "%") ^ expr5)*;
expr5 = "!" operand | operand | expr_minus | expr_plus;
expr_minus = "-" operand;
expr_plus = "+" operand;
operand = read |
    print |
    if_stmnt |
    "(" expr ")" |
    compound_stmnt |
    primitive |
    func_identifier
;

compound_stmnt = begin statements end;

```

```

primitive = number | character | boolean;

func_identifier = identifier ( "(" exprlist? ")" )?;

while_stmnt = WHILE statements_cond DO statements END;

if_stmnt = IF statements_cond DO statements else_stmnt? END;

else_stmnt = ELSEIF statements_cond DO statements else_stmnt?
| (ELSE statements)
;

print = PRINT "(" exprlist ")" ;
read = READ "(" varlist ")" ;

varlist = identifier ("," identifier)*;
exprlist = expr ("," expr)*;

func_def = DEF identifier "(" varlist ")";

```

Semantics and context constraints:

The semantics and context constraints are defined using the abstract syntax of the Alia language.

Program

```
program = ((statement end_statement) | \n)*;
```

A program is run by executing a sequence of statements.

Statement

```

statements = (statement (end_statement statements)? | \n statements)?;
statements_conditional = (statement (end_statement statements)? | \n statements_cond )?;
end_statement = \n | ";" | EOF;
statement = while_stmnt
| (expr_assignment | const_assignment)
;
while_stmnt = WHILE statements_cond DO statements END;
if_stmnt = IF statements_cond DO statements else_stmnt? END;
else_stmnt = ELSEIF statements_cond DO statements else_stmnt?
| (ELSE statements);
compound_stmnt = begin statements end;

```

- A statements is a set of statements separated by an end statement.
- A conditional statements is a statements that is meant for conditional expressions.

- ## Assignment

```
const_assignment = CONST identifier "=" primitive (: type)?
```

- ## Expressions

6

- The expressions 'or', '||', 'and', '&&' preceded by E1 and followed by E2 are evaluated by performing a logical or (True iff E1 or E2) in case of 'or' and '||'. In the case of 'and' and '&&' it is evaluated by performing a logical and on the two expressions (True iff E1 and E2). E1 and E2 must be of type boolean. The type of the expression is Boolean. These are the logical operators.
- The expression 'E1 == E2' is true iff E1 equals E2. 'E1 != E2' is true iff E1 is not equal to E2. 'E1 <= E2' is true iff E1 is smaller than or equal to E2. 'E1 >= E2' is true iff E1 greater than or equal to E2. 'E1 > E2' is true iff E1 is greater than E2. 'E1 < E2' is true iff E1 is smaller than E2. Of all these comparative operators, E1 and E2 must be of the same type. The type of the expressions is Boolean. These are the comparative operators.
- The expression 'E1 + E2' is executed as E1 plus E2. 'E1 - E2' is E1 minus E2. 'E1 * E2' is E1 times E2. 'E1 / E2' is E1 divided by E2, E2 is not allowed to be zero. '
- The operator O in '!O' is inverted. O must be of type boolean, the expression is of type boolean. The '+O' and '-O' are executed as follows. For '+O' nothing is done. For '-O' the operand is negated. O must be of type Int, the expression is of type Int. These are the unary operators.
- The previous expressions have the following priority, from highest to lowest. Unary operators (-, +, !), then *, /, % after those + and -. Then comes comparative operators (<, <=, >=, >, ==, <>) then comes the logical and (&& or 'and') then comes logical or || or 'or'.

Operands

```
operand = READ "(" varlist ")" |
          PRINT "(" exprlist ")" |
          if_stmt |
          "(" expr ")" |
          compound_stmt |
          primitive |
          identifier
;
```

- The expression 'read VL' is executed as follows. The variable list evaluated. For every variable a line is read from the input, the first character of this line is assigned as value to the variable. The type of the expression is the type of VL.
- The expression 'print EL' is executed as follows. The expression list EL is evaluated. All evaluated expressions are then written to the output. The type of the expression is the type of EL.
- If statements are explained under statements.
- An operand can carry another expression as long as that expression is surrounded by brackets. The type and result are the same as the expression.
- Compound statements are explained under statements.
- A primitive is one of the three primitive types NUMBER, CHARACTER and BOOLEAN.
- The identifier operand points to a the value or variable bound to I. The operand I must have been previously declared. The type of the operand is the type of that value or variable.

Lists

```
varlist = identifier ("," identifier)*;  
exprlist = expr ("," expr)*;
```

- The list 'I (I)*' evaluates to a list of identifiers. If there is one identifier the type of the list is the type of that identifier, and the result is its value. If there are 2 or more, the type is void and there is no result.
- The list 'E (E)*' evaluates to a list of expressions. None of the expressions may be void. If there is one expression, the list of the type of E, and the value is E. If there are 2 or more, the type is void, and thus there is no result.

Types

```
primitive = NUMBER  
| CHARACTER  
| BOOLEAN  
;
```

- The operand 'N' evaluates to a number. N can be no larger than 2147483647 and no smaller than -2147483648. N is of type Int.
- The operand 'C' evaluates to a character. C is of type Char.
- The operand 'B' evaluates to a boolean, either true or false. C is of type Bool.

In Alia there are 4 types. 'int', 'char', 'boolean' and 'void'. If something is of type void it is an empty value that cannot be used.

4 Translation rules

The translation rules for Alia to Java bytecode are shown here. Some details have been abstracted away in favor of readability, these details include specific label names, translation rules which are dependent on the type of the expression (such as print and read), and some specific rules where pop statements are included.

Pop lines A pop line is included after every statement that returns a value but has no higher expression using it. The amount of variables generated on the stack are counted by the compiler and after each complete statement the leftover expressions are popped.

Translation rules

```
execute [I = E]  
    expr [E]  
    istore a // address of variable I  
    identifier [I]
```



```

expr [while C do S end] =
    goto COND
    WHILE:
    execute [S]
    COND:
    execute [C]
    ifne WHILE

expr [if C do S E end] =
    execute [C]
    ifeq ELSE
    execute [S]
    goto NEXT
    ELSE:
    exprElse [E]
    NEXT:

exprElse [elseif C do S E]
    execute [C]
    ifeq ELSE
    execute [S]
    goto NEXT
    ELSE:
    exprElse [E]
    NEXT:

exprElse [else S] =
    execute [S]

expr [E1 0 E2] =
    expr [E1]
    expr [E2]
    instruction [0] // The specific instruction, e.g. iadd etc.

expr [E1 0C E2] =
    expr [E1]
    expr [E2]
    if_icmp $+7 // Go to iconst_1 if it is true, this line contains the specific instruction
    iconst_0
    goto $+4 // Go to the line after iconst_1
    iconst_1

expr [-E]
    expr [E]
    ineg

```

```

expr [+E]
  expr [E]

expr [not E]
  expr[E]
  ifeq $+7
  iconst_0
  goto $+4
  iconst_1

expr [begin S end]
  execute [S]

print [S] =
  getstatic java/lang/System/out Ljava/io/PrintStream;
  execute [S]
  invokevirtual java/io/PrintStream/println(T)V

expr [print(S)] =
  getstatic java/lang/System/out Ljava/io/PrintStream;
  execute [S]
  istore_1
  iload_1
  invokevirtual java/io/PrintStream/println(T)V
  iload_1

expr [print(S, L)] =
  print [S]
  executePrint [L]

executePrint [S, L]
  print [S]
  executePrint [L]

executePrint [S]
  print [S]

read [] =
  getstatic ClassName/in Ljava/io/BufferedReader;
  invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
  invokestatic java/lang/Type/parseType(Ljava/lang/String;)T

execute [read(I)] =
  read []
  istore_1

```

```

    iload_1
    istore a ; address of variable I
    execute [S]
    iload_1

execute [read(I, L)] =
    read []
    istore a ; address of variable I
    exprRead [L]

exprRead [I, L]
    read []
    istore a ; address of variable I
    exprRead [L]

exprRead [I]
    read []
    istore a ; address of variable I

execute [S \n S] =
    execute [S]
    execute [S]

execute [S ; S] =
    execute [S]
    execute [S]

execute [S] =
    expr [S]

identifier [I] =
    iload a // address of variable I

operand [I] =
    identifier [I]

operand [N] =
    number [N] // iconst n
operand [C] =
    bipush C

operand [true] =
    iconst_1

operand [false] =
    iconst_0

```

```

program [S] =
  .class public filename.j // target file
  .super java/lang/Object

  .method public \<init\>()V
    aload_0
    invokevirtual java/lang/Object/\<init\>()V
    return
  .end method

  .method public static main([Ljava/lang/String;)V
    .limit stack stackMax // stackMax = maximum size of the stack
    .limit locals localSize // localSize = amount of local variables required

    execute [S]

    return
  .end method

```

5 Java-code

All Alia related code is located in the alia package, the alia package is structured in the following way:

alia Contains the .g files, auxiliary classes and antlr generated classes.

symtab Contains the classes needed for the symbol table.

tests Contains the test code.

types Contains the type classes used in the checker and for code generation.

The main file of the compiler is Alia.java, it is responsible for calling all the antlr generated classes to compile and run code.

5.1 CheckerAux

The checker uses an auxiliary class CheckerAux that handles a large portion of the logic of the checking, such as if two types are the same. This class also declares variables and constants into the symbol table. CheckerAux also has methods to access the symbolTable so that it throws AliaExceptions instead of more general exceptions. The symbol table has a HashMap of Names, IdEntries and a scopestack that has all identifiers declared on a scope. Like every symbolTable it keeps track of what identifiers have been declared on what levels. The IdEntries also store information about whether the identifier is a constant and what type it is.

Most of the logic for type checking is implemented in CheckerAux, to do the type checking a set of type classes are used, such as _Int and _Bool. All of these classes inherit from _Type and have a string with their typename. We chose to make all types into distinct classes instead of an enum because this will allow

for extension of say the `_Int` class with a `_Float` class or of the `_Char` class with a `_String` class. In this way we can more easily add additional types to Alia and a future `_Long` and `_Float` could be compared using inheritance.

5.2 CodeGeneratorAux

The code generation makes use of `CodeGeneratorAux`. This separates some of the logic from the antlr files. In particular `CodeGeneratorAux` calculates what kind of java type can be used for any given number, this choice is explained in the problems section. To do this it uses the `NumberType` class, which acts as a container for a number of booleans so that they can be passed more elegantly. The other part that `CodeGeneratorAux` takes care of is the logic for the stack management, incrementing and decrementing the amount that is still to be pushed off the stack in the code generation.

5.3 Error handling

For error handling `AliaException` and `AliaTypeException` are used. These exceptions are thrown in the checker when ever a type is violated. If there is a syntactical mistake then the classes generated by antlr will throw exceptions. For run time errors standard java exceptions are also used.

5.4 Decorated AST

After the checking phase has been completed a decorated AST is returned. The decorated AST stores the type information that was found in the corresponding nodes, such as for all binary expressions. We also store the identifying numbers for all applied usages of identifiers (except for constants which are replaced), these ascending numbers are gotten from the `IdEntries` using `CheckerAux` and are stored with the nodes, for later use in the code generation.

6 Tests

The Alia programming language has been thoroughly tested using a collection of test programs. These test programs have been designed to check the correct workings of parser, checker and compiler of the Alia programming language. Unit tests have been build using JUnit, and are located in the `src/tests/` folder of the Alia project. There are three type of errors which the compiler should check for.

1. Syntax: Incorrect syntax and typos should be reported.
2. Context: The context checker should type check the program, make sure all variables are declared before use, and enforce scoping rules.
3. Semantics: Runtime errors such as division by zero should be adequately handled by the compiler.

6.1 Tests

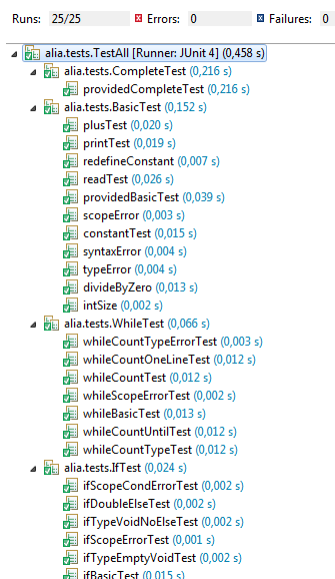
The tests have been constructed based on the requirements formulated in the compiler construction reader.

1. `BasicTest.java`: Contains tests for the basic expression language.

- (a) providedBasicTest: Test with correct syntax which is checked for correct output.
 - (b) plusTest: Tests a basic arithmetic expression.
 - (c) printTest: Tests print expressions.
 - (d) readTest: Tests read expressions.
 - (e) constantTest: Tests constants.
 - (f) redefineConstant (Context): Checks if constants cannot be redefined.
 - (g) syntaxError (Syntax): Checks if syntax errors are properly detected.
 - (h) typeError (Context): Checks if type errors are properly detected.
 - (i) intSize (Context): Check if use of numbers above the maximum integer size are detected.
 - (j) scopeError (Context): Checks if references out of scope are valid.
 - (k) divideByZero (Semantics): Checks if divide by zero triggers a runtime error.
2. WhileTest.java: Contains tests for the while conditional statement.
- (a) providedBasicTest (Syntax): Test with correct syntax which is checked for correct output.
 - (b) plusTest (Syntax): Tests a basic arithmetic expression.
 - (c) printTest (Syntax): Tests print expressions.
 - (d) readTest (Syntax): Tests read expressions.
 - (e) constantTest (Syntax): Tests constants.
 - (f) redefineConstant (Context): Checks if constants cannot be redefined.
 - (g) syntaxError (Syntax): Checks if syntax errors are properly detected.
 - (h) typeError (Context): Checks if type errors are properly detected.
 - (i) intSize (Context): Check if use of numbers above the maximum integer size are detected.
 - (j) scopeError (Context): Checks if references out of scope are valid.
 - (k) divideByZero (Semantics): Checks if divide by zero triggers a runtime error.
3. IfTest.java: Contains test for the if conditional statement.
- (a) ifBasicTest (Syntax): Basic if statement test.
 - (b) ifTypeVoidNoElseTest (Context): Tests if type is void when else statement is not present.
 - (c) ifScopeErrorTest (Context): Tests the scope rules of the if statement.
 - (d) ifScopeCondErrorTest (Context): Tests the scope rules of the condition of the if statement.
 - (e) ifTypeEmptyVoidTest (Context): Tests if type is void when the if statement is empty.
 - (f) ifDoubleElseTest (Syntax): Test with two many else statements, checks correct syntax error.
4. CompleteTest.java: Contains all language constructs of the Alia programming language.
- (a) providedCompleteTest (Syntax): Test program with all language constructs.

6.2 Test results

The test program will output whether or not the tests have executed successfully. The tests results can be seen in the image below. All tests have been successfully run using JUnit. This gives a relatively high certainty that the programming language is correct. See appendix A.6 for an example run of the test program.



7 Conclusion

In this report we have described the Alia programming language. We have specified the features, the syntax, the context constraints and the semantics of the Alia language. Some of the problems that we faced during the construction as well as their solutions have been elaborated. We have also detailed for you the extra java classes constructed for the compiler and the full array of tests that have been made to verify the correctness of the language. Together these give you a good understanding of how the Alia programming language works both in programming and under the hood. For conclusions on the programming language itself. Alia is a language that contains all the functionality of the basic expression language, along with conditional statement and a while statement. Alia also features type inference, though procedures and functions have not yet been added. The extra functionality of functions and procedures is fairly major and for future work these should be the first to be added. The language offers a large amount of freedom as to what you want to write down, such as not having to hardly put any end of line delimiters except newlines and also not having to declare types for variables or constants.

The construction of the Alia compiler was a very interesting learning experience. As everyone knows you put quite a lot of time into the final project of compiler construction, but you get rewarded with a new level of understanding of how programming languages work. It has been fun to be able to define your own programming language, and it is great to have an understanding of every stage of the compilation process, from code to actually executable instructions. Weighing all the gains against the time invested it was a very positive learning experience and definitely adds value to an education in computer science.

A Appendices

A.1 Responsibilities

The following table makes clear who was responsible for what parts of the report.

Part	Person
Title page	Joost
Introduction	Fedor
Description	Joost
Problems	Fedor
Syntax, Context, Semantics	Fedor
Translation Rules	Joost
Java Code	Fedor
Tests	Joost
Conclusion	Fedor

A.2 Lexer and parser

```
1  grammar Alia;
2
3  options {
4      k=1;                                // LL(1) - do not use LL(*)
5      language=Java;                      // target language is Java (= default)
6      output=AST;                          // build an AST
7  }
8
9  tokens {
10     COLON      = ':' ;
11     NEWLINE    = '\n' ;
12     COMMA      = ',' ;
13     SEMICOLON  = ';' ;
14     LPAREN     = '(' ;
15     RPAREN     = ')' ;
16     LCURLY     = '{' ;
17     RCURLY     = '}' ;
18     SQUOTE     = '\'' ;
19
20     // operators
21     BECOMES    = '=' ;
22     PLUS       = '+' ;
23     PLUS_OP    = 'plusop' ;
24     MINUS      = '-' ;
25     MINUS_OP   = 'minop' ;
26     TIMES      = '*' ;
27     DIV        = '/' ;
28
29     // comp. operators
30     GT         = '>' ;
31     GE         = '>=' ;
32     LT         = '<' ;
33     LE         = '<=' ;
34     EQ         = '==' ;
35     NQ         = '!=' ;
36
37     AND        = 'and' ;
38     AND_ALT    = '&&' ;
39     OR         = 'or' ;
40     OR_ALT     = '||' ;
41     NOT        = '!' ;
42     MOD        = '%' ;
43
44     // types
45     INT        = 'int' ;
46     BOOL       = 'boolean' ;
47     CHAR       = 'char' ;
48     STRING     = 'string' ;
49
50     // keywords
51     PROGRAM    = 'program' ;
52     PRINT      = 'print' ;
53     READ       = 'read' ;
54     IF         = 'if' ;
55     ELSE       = 'else' ;
56     ELSEIF     = 'elseif' ;
57     DO         = 'do' ;
58     END        = 'end' ;
59     WHILE      = 'while' ;
60     TRUE       = 'true' ;
61     FALSE      = 'false' ;
62     CONST      = 'const' ;
63     DEF        = 'def' ;
64     BEGIN      = 'begin' ;
65
66     FUNC       = 'func' ;
67     EXPR_LIST;
68     COMPOUND;
69     TYPE;
70     ID;
71     LOCALSIZE;
72 }
73
74
75 @lexer::header {
76     package alia;
```

```

77 }
78
79 @header {
80 package alia;
81 }
82
83
84 program : (func_def | (statement end_statement) | NEWLINE!)*;
85
86 statements : (statement (end_statement statements)? | NEWLINE! statements)?;
87 statements_cond : statement (end_statement statements)? | NEWLINE! statements_cond;
88 statement : (expr_assignment | const_assignment) (COLON~ type)?
89             | while_stmnt;
90 end_statement : NEWLINE! | SEMICOLON! | EOF!;
91
92 // Syntactic predicate to recognize assignments
93 // Syntactic predicates can be easily left out if we do not allow expr as statements
94 expr_assignment : (IDENTIFIER BECOMES) => (IDENTIFIER BECOMES~) expr_assignment |
95                 expr ;
96
97 const_assignment : CONST~ IDENTIFIER BECOMES primitive;
98
99 expr : expr1 ((OR | OR_ALT)~ expr1)*;
100 expr1 : expr2 ((AND | AND_ALT)~ expr2)*;
101 expr2 : expr3 ((GT | GE | LT | LE | EQ | NQ)~ expr3)*;
102 expr3 : expr4 ((PLUS | MINUS)~ expr4)*;
103 expr4 : expr5 ((TIMES | DIV | MOD)~ expr5)*;
104 expr5 : NOT~ operand | operand | expr_minus | expr_plus;
105 expr_minus : MINUS operand -> ~(MINUS_OP operand);
106 expr_plus : PLUS operand -> ~(PLUS_OP operand);
107 operand : read |
108           print |
109           if_stmnt |
110           LPAREN! expr RPAREN! |
111           compound_stmnt |
112           primitive |
113           func_identifier;
114
115 compound_stmnt : BEGIN statements END -> ~(COMPOUND statements);
116
117 primitive : NUMBER | CHAR_EXPR | boolean_expr;
118
119 char_expr : SQUOTE! LETTER SQUOTE!;
120
121 func_identifier : IDENTIFIER
122                 (LPAREN~ exprlist? RPAREN)?;
123
124 while_stmnt : WHILE statements_cond DO statements END -> ~(WHILE statements_cond ~(DO statements));
125
126 if_stmnt : IF statements_cond DO statements else_stmnt? END ->
127           ~(IF statements_cond ~(DO statements?) else_stmnt?);
128
129 else_stmnt
130 : ELSEIF statements_cond DO statements else_stmnt? ->
131   ~(ELSEIF statements_cond ~(DO statements?) else_stmnt?)
132   | (ELSE~ statements)
133   ;
134
135 print : PRINT~ LPAREN! exprlist RPAREN!;
136 read : READ~ LPAREN! varlist RPAREN!;
137
138 varlist : IDENTIFIER (COMMA! IDENTIFIER)*;
139 exprlist : expr (COMMA! expr)*;
140
141 func_def : DEF IDENTIFIER LPAREN! varlist RPAREN! statements END;
142
143 // Lexer rules
144
145 boolean_expr : TRUE | FALSE;
146
147 type : CHAR | INT | BOOL;
148
149 CHAR_EXPR : SQUOTE LETTER SQUOTE;
150
151 IDENTIFIER
152 : LETTER (LETTER | DIGIT)*
153 ;
154
155 NUMBER
156 : DIGIT+
157 ;

```

```

158
159
160 COMMENT
161 :    ('/' '/' .* '\n' | '/' '*' .* '*' '/')
162      { $channel=HIDDEN; }
163
164 ;
165
166 WS
167 :    (' ' | '\t' | '\f' | '\r')+
168      { $channel=HIDDEN; }
169 ;
170
171 fragment LETTER :    LOWER | UPPER ;
172 fragment DIGIT  :    ('0'..'9') ;
173 fragment LOWER  :    ('a'..'z') ;
174 fragment UPPER  :    ('A'..'Z') ;

```

A.3 Checker

```

1  tree grammar AliaChecker;
2
3  options {
4      k=1;                                // LL(1) - do not use LL(*)
5      tokenVocab=Alia;                    // import tokens from Calc.tokens
6      ASTLabelType=CommonTree;            // AST nodes are of type CommonTree
7      superClass=CheckerAux;
8      output=AST;
9  }
10
11  @header {
12  package alia;
13  import alia.types.*;
14  import alia.symtab.SymbolTable;
15  import alia.symtab.IdEntry;
16  import java.util.Set;
17  import java.util.HashSet;
18  }
19
20  // Alter code generation so catch-clauses get replaced with this action.
21  // This disables ANTLR error handling: AliaExceptions are propagated upwards.
22  @rulecatch {
23      catch (RecognitionException e) {
24          if(!e.getMessage().equals("")) {
25              System.err.println("Exception!:"+e.getMessage());
26          }
27          throw (new AliaException(""));
28      }
29  }
30
31  @members {
32  }
33
34
35  program
36  :    { symTab.openScope(); }
37      (statement)+
38      { symTab.closeScope(); }
39      -> LOCALSIZE[getLocalSize()] (statement)+
40      ;
41
42  statements returns [_Type type = new _Void()]
43  : (t=statement
44   { $type = $t.type; }
45  )*;
46
47  statement returns [_Type type = new _Void()]
48  :  ~(WHILE {symTab.openScope();} stat=statements {symTab.openScope();})
49     ~(DO statements) {symTab.closeScope();symTab.closeScope();} )
50     { checkBoolType($stat.type, $stat.tree); }
51     | t=expr
52     { $type = $t.type; }
53     ;

```

```

54
55
56 expr returns [_Type type]
57 :   to=operand
58 {
59     $type = $to.type;
60 }
61 |   ^(c=OR t1=expr t2=expr)
62 |   ^(c=OR_ALT t1=expr t2=expr)
63 |   ^(c=AND t1=expr t2=expr)
64 |   ^(c=AND_ALT t1=expr t2=expr)
65 {
66     checkEqualType($t1.type, $t2.type, $t1.tree);
67     checkBoolType($t1.type, $t1.tree);
68     $type = new _Bool();
69     String typename = String.valueOf($type);
70 }
71 -> ^($c expr expr TYPE[typename])
72 |   ^(c=EQ t1=expr t2=expr)
73 |   ^(c=NQ t1=expr t2=expr)
74 |   ^(c=LE t1=expr t2=expr)
75 |   ^(c=GE t1=expr t2=expr)
76 |   ^(c=GT t1=expr t2=expr)
77 |   ^(c=LT t1=expr t2=expr)
78 {
79     checkEqualType($t1.type, $t2.type, $t1.tree);
80     $type = new _Bool();
81     String typename = String.valueOf($type);
82 }
83 -> ^($c expr expr TYPE[typename])
84 |   ^(c=PLUS te1=expr te2=expr)
85 |   ^(c=MINUS te1=expr te2=expr)
86 |   ^(c=TIMES te1=expr te2=expr)
87 |   ^(c=DIV te1=expr te2=expr)
88 |   ^(c=MOD te1=expr te2=expr)
89 {
90     checkMathType($te1.type, $te2.type, $te1.tree);
91     $type = new _Int();
92     String typename = String.valueOf($type);
93 }
94 -> ^($c expr expr TYPE[typename])
95 | ^ (PRINT te=exprlist)
96 {
97     $type = $te.type;
98     String typename = String.valueOf($type);
99 }
100 -> ^ (PRINT TYPE[typename] exprlist)
101 | ^ (READ tv=varlist)
102 {
103     $type = $tv.type;
104     String typename = String.valueOf($type);
105 }
106 -> ^ (READ TYPE[typename] varlist)
107 | ^ (c=(NOT) to=operand)
108 {
109     $type = $to.type;
110     String typename = String.valueOf($type);
111     checkBoolType($to.type, $to.tree);
112 }
113 -> ^($c operand TYPE[typename])
114 | ^ (c=( PLUS_OP | MINUS_OP ) o=operand)
115 {
116     $type = $o.type;
117     String typename = String.valueOf($type);
118     checkEqualType($o.type, new _Int(), $o.tree);
119 }
120 -> ^($c operand TYPE[typename])
121 | ^ (IF
122 {
123     symTab.openScope(); // Open scope for conditional statements, the scope is the same for the
IF and ELSEIF conditions
124 }
125     t=statements
126     {
127         symTab.openScope(); // Open scope for the first statement
128     }
129     ^ (DO
130     ts=statements
131     {
132         checkBoolType($t.type, $ts.tree);
133         symTab.closeScope(); // Close scope for the first statement

```

```

134     }
135     )
136     texp=else_stmnt?
137     {
138         symTab.closeScope(); // Close scope for the conditional statements
139         checkBoolType($t.type, $t.tree);
140         $type = checkTypesIf($ts.type, $texp.type);
141     }
142 )
143 | ~(COLON ~(BECOMES id=IDENTIFIER t1=expr) typ=type)
144 {
145     _Type declType = checkEqualType($t1.type, $typ.type, $t1.tree);
146     declare($id.text, declType, $t1.tree);
147     $type = declType;
148
149     String typename = String.valueOf($type);
150     String identifier = String.valueOf(getIdentifier($id.text, $id.tree));
151 }
152 -> ~(BECOMES ~(IDENTIFIER TYPE[typename] ID[identifier]) expr)
153 | ~(BECOMES id=IDENTIFIER t1=expr)
154 {
155     declare($id.text, $t1.type, $t1.tree);
156     $type = $t1.type;
157     checkNotVoid($type, $t1.tree);
158
159     String typename = String.valueOf($type);
160     String identifier = String.valueOf(getIdentifier($id.text, $id.tree));
161 }
162 -> ~(BECOMES ~(IDENTIFIER TYPE[typename] ID[identifier]) expr)
163 | ~(COMPOUND
164 { // symTab.openScope
165     symTab.openScope();
166 }
167 t=statements)
168 {
169     // closeScope
170     symTab.closeScope();
171     $type = $t.type;
172     String typename = String.valueOf($type);
173 }
174 -> ~(COMPOUND TYPE[typename] statements)
175 | ~(CONST id=IDENTIFIER BECOMES prim=primitive (COLON typ=type)?)
176 { _Type declType = checkEqualType($prim.type, $typ.type, $prim.tree);
177     declareConst($id.text, declType, prim, $prim.tree);
178     $type = declType;
179     String typename = String.valueOf($type);
180     String identifier = String.valueOf(getIdentifier($id.text, $prim.tree));
181 }
182 }
183 -> // constants are not used after checking fase, thus they are removed
184 ;
185
186 else_stmnt returns [_Type type]
187 :
188 ~(ELSEIF t=statements
189     ~(DO
190     {
191         symTab.openScope(); // Open scope for this elseif statement
192     }
193     ts=statements
194 )
195     te=else_stmnt?
196     {
197         checkBoolType($t.type, $t.tree);
198         $type = checkTypesIf($ts.type, $te.type);
199         symTab.closeScope();
200     }
201 )
202 | ~(ELSE
203 {
204     symTab.openScope(); // Open scope for the else statement
205 }
206 ts=statements
207 {
208     $type = $ts.type;
209     symTab.closeScope(); // Open scope for the else statement
210 }
211 )
212 ;
213
214

```

```

215 operand returns [_Type type]
216 :   id=identifier
217   { $type = $id.type; }
218   |   n=NUMBER
219   { $type = new _Int();checkInt(n); }
220   |   c=CHAR_EXPR
221   { $type = new _Char(); }
222   |   b=(TRUE | FALSE)
223   { $type = new _Bool(); }
224 ;
225
226 identifier returns [_Type type]
227 :
228   id=IDENTIFIER
229   {
230     $type = getType($id.text, $id.tree);
231     String typename = String.valueOf($type);
232     String identifier = String.valueOf(getIdentifier($id.text, $id.tree));
233     Boolean constant = retrieve($id.text, $id.tree).isConstant();
234     Token value = getConstant($id.text);
235   }
236   -> {constant && typename.equals("int") }? ~(NUMBER[value])
237   -> {constant && typename.equals("char") }? ~(CHAR_EXPR[value])
238   -> {constant && typename.equals("bool") && value.getText().equals("true") }? ~(TRUE[value])
239   -> {constant && typename.equals("bool") && value.getText().equals("false") }? ~(FALSE[value])
240   -> ~(IDENTIFIER TYPE[typename] ID[identifier])
241 ;
242
243
244 varlist returns [_Type type]
245 : t=identifier
246 {
247   $type = $t.type;
248 }
249 (identifier
250 {
251   $type = new _Void();
252 }
253 )*
254 ;
255
256
257 exprlist returns [_Type type]
258 : tl=exprentry
259 {
260   checkNotVoid($tl.type, $tl.tree);
261   $type = $tl.type;
262 }
263 (t=exprentry
264 {
265   checkNotVoid($t.type, $t.tree);
266   $type = new _Void();
267 }
268 )*
269 ;
270
271 exprentry returns [_Type type]
272 : t=expr
273 {
274   $type = $t.type;
275   String typename = String.valueOf($t.type);
276 } -> TYPE[typename] expr
277 ;
278
279 type returns [_Type type]
280 :   INT
281   { $type = new _Int(); }
282   |   CHAR
283   { $type = new _Char(); }
284   |   BOOL
285   { $type = new _Bool(); }
286 ;
287
288 primitive returns [_Type type] :
289   NUMBER {$type = new _Int();}
290   | CHAR_EXPR
291   { $type = new _Char(); }
292   | boolean_expr
293   { $type = new _Bool(); }
294 ;
295 boolean_expr : TRUE | FALSE;

```

A.4 Code Generator

```
1  tree grammar AliaCodeGeneratorStringTemplate;
2
3  options {
4      k=1;                                // LL(1) - do not use LL(*)
5      tokenVocab=Alia;                    // import tokens from Calc.tokens
6      output = template;
7      ASTLabelType=CommonTree;           // AST nodes are of type CommonTree
8      superClass=CodeGeneratorAux;
9  }
10
11  @header {
12  package alia;
13  import alia.syntab.SymbolTable;
14  import alia.syntab.IdEntry;
15  import java.util.Set;
16  import java.util.HashSet;
17  }
18
19  program
20      :
21          localSize=LOCALSIZE (s+=exprPop)+
22          -> file(instructions={$s},stackMax={getStackMax()},localSize={$localSize},classname={
23              getProgramClass()})
24      ;
25
26  statements @init { startExpression(); }
27      @after { endExpression(); }
28      : (s+=exprPopInterleaved)* -> statements(instructions={$s});
29
30  statementsPop @init { startExpression(); }
31      @after { endExpression(); }
32      : (s+=exprPop)* -> statements(instructions={$s});
33
34  exprPopInterleaved @init { String pop = ""; }
35      : {pop=pops(endExpression());startExpression();}
36      s=expr -> exprPopInterleaved(instruction={$s.st}, pop={pop});
37
38  exprPop @init { String pop = ""; }
39      : {startExpression();}
40      s=expr
41      {pop=pops(endExpression());} -> exprPop(instruction={$s.st}, pop={pop});
42
43  expr @init { }
44      @after { }
45      : o=operand -> statement(instruction={$o.st})
46      | ^((OR t1=expr t2=expr t=TYPE) {decStack();} -> binexpr(x={$t1.st}, y={$t2.st}, instr={"or"
47      | ^((OR_ALT t1=expr t2=expr t=TYPE) {decStack();} -> binexpr(x={$t1.st}, y={$t2.st}, instr={"or
48      | ^((AND t1=expr t2=expr t=TYPE) {decStack();} -> binexpr(x={$t1.st}, y={$t2.st}, instr={"
49      | ^((AND_ALT t1=expr t2=expr t=TYPE) {decStack();} -> binexpr(x={$t1.st}, y={$t2.st}, instr={"
50      | ^((EQ t1=expr t2=expr t=TYPE) {decStack();} -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"
51      | ^((NE t1=expr t2=expr t=TYPE) {decStack();} -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"
52      | ^((LE t1=expr t2=expr t=TYPE) {decStack();} -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"
53      | ^((GE t1=expr t2=expr t=TYPE) {decStack();} -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"
54      | ^((GT t1=expr t2=expr t=TYPE) {decStack();} -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"
55      | ^((LT t1=expr t2=expr t=TYPE) {decStack();} -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"
56      | ^((PLUS t1=expr t2=expr t=TYPE) {decStack();} -> binexpr(x={$t1.st}, y={$t2.st}, instr={"
57      | ^((MINUS t1=expr t2=expr t=TYPE) {decStack();} -> binexpr(x={$t1.st}, y={$t2.st}, instr={"
58      | ^((TIMES t1=expr t2=expr t=TYPE) {decStack();} -> binexpr(x={$t1.st}, y={$t2.st}, instr={"
59      | ^((DIV t1=expr t2=expr t=TYPE) {decStack();} -> binexpr(x={$t1.st}, y={$t2.st}, instr={"
60      | ^((MOD t1=expr t2=expr t=TYPE) {decStack();} -> binexpr(x={$t1.st}, y={$t2.st}, instr={"
61      rem"));
```

```

60 | ^ (WHILE cond=statements {decStack();} ^ (DO t2=statementsPop)) -> whilestmt(expr={ $cond.st},
statement={ $t2.st}, labelCond={newLabel()}, labelWhile={newLabel()})
61 | ^ (PRINT t=TYPE te=TYPE fexp=expr (exp+=exprPrint*)) {decStackIfVoid(getType($t.toString()));}
-> printstmt(firststatement={ $fexp.st}, statements={ $exp}, type={getType($t.toString())})
62 | ^ (READ t=TYPE ^ (id=IDENTIFIER t=TYPE a=ID) {incStack();} (v+=varRead*)) {decStackIfVoid(getType(
$t.toString()));} -> readstmt(statements={ $v}, addr={ $a}, type={getType($t.
toString())}, t={getType($t.toString()).T}, void={ $t.toString().equals("void")}, classname={
getProgramClass()})
63 | ^ (NOT o=operand t=TYPE) -> unarynot(x={ $o.st}, instr={"not"})
64 | ^ (PLUS_OP o=operand t=TYPE) -> unaryplus(x={ $o.st}, instr={"plus"})
65 | ^ (MINUS_OP o=operand t=TYPE) -> unarymin(x={ $o.st}, instr={"neg"})
66 | { startExpression(); } ^ (IF
67 | stif1=statements
68 | ^ (DO stif2=statements)
69 | (elsestmnts=elseif)?
70 | ) { decStack();endExpression(); } -> ifstmt(cond={
$stif1.st}, statements={ $stif2.st}, elseStmnts={elsestmnts}, labelElse={newLabel()}, labelNext={
newLabel()})
71 | ^ (BECOMES ^ (id=IDENTIFIER t=TYPE a=ID) {incStack();} t1=expr {decStack();}) -> assign(var={ $id
}, addr={ $a}, expr={ $t1.st})
72 | ^ (COMPOUND t=TYPE s=statements) -> statements(instructions={ $s.st})
73 ;
74 elseif @init { decStack(); }
75 :
76 ^ (ELSEIF stelseif1=statements
77 ^ (DO stelseif2=statements) {decStack();}
78 elsestmnts=elseif) -> elseifstmt(cond={ $stelseif1.st}, statements={ $stelseif2.st
}, elseStmnts={elsestmnts}, labelElse={newLabel()}, labelNext={newLabel()})
79 | ^ (ELSE stelse=statements) -> elsemaybestmnt(statements={ $stelse.st})
80 ;
81 operand @init {incStack();}
82 : i=identifier -> statement(instruction={ $i.st})
83 | n=NUMBER -> number(n={ $n.toString()}, numberType={whatNumber(Integer.parseInt(
$n.toString()))})
84 | c=CHAR_EXPR -> character(c={({int} c.toString().charAt(1)})
85 | b=(TRUE | FALSE) -> boolean(b={ $b.toString().equals("true")})
86 ;
87
88 exprPrint @init {decStack();} :
89 t=TYPE exp=expr -> printexpr(statements={ $exp.st}, t={getType($t.toString()).T})
90 ;
91
92 varRead @init {incStack();decStack();} :
93 ^ (id=IDENTIFIER t=TYPE a=ID) -> readvar(var={ $id}, addr={ $a}, type={getType($t.toString())}, classname={
getProgramClass()})
94 ;
95
96 identifier
97 : ^ (id=IDENTIFIER t=TYPE a=ID) -> identifier(addr={ $a})
98 ;
99
100 varlist
101 : s+=identifier
102 (s+=identifier)*
103 -> statements(instructions={ $s});
104
105 exprlist
106 : s+=expr
107 (s+=expr)*
108 -> statements(instructions={ $s})
109 ;
110
111 type
112 : INTEGER
113 | CHAR
114 | BOOL
115 ;

```

A.5 String templates

```

1 group tmg;
2

```



```

3 file(instructions,stackMax,localSize,classname) ::= <<
4 ; Jasmin JBC assembler code generated by AliaCodeGenerator
5 .class public <classname>
6 .super java/lang/Object
7
8 .field private static in Ljava/io/BufferedReader;
9
10 .method static public \<clinit\>()V
11 .limit stack 5
12
13     new java/io/BufferedReader
14     dup
15     new java/io/InputStreamReader
16     dup
17     getstatic java/lang/System/in Ljava/io/InputStream;
18     invokespecial java/io/InputStreamReader/\<init\>(Ljava/io/InputStream;)V
19     invokespecial java/io/BufferedReader/\<init\>(Ljava/io/Reader;)V
20     putstatic <classname>/in Ljava/io/BufferedReader;
21     return
22
23 .end method
24
25 .method public \<init\>()V
26     aload_0
27     invokenonvirtual java/lang/Object/\<init\>()V
28     return
29 .end method
30
31 .method public static main([Ljava/lang/String;)V
32     .limit stack <stackMax>
33     .limit locals <localSize>
34
35     <instructions; separator="\n">
36
37     return
38 .end method
39 >>
40
41 statements(instructions) ::= <<
42 <instructions; separator="\n">
43 >>
44
45 exprPopInterleaved(instruction, pop) ::= <<
46 <pop>
47 <instruction>
48 >>
49
50 exprPop(instruction, pop) ::= <<
51 <instruction>
52 <pop>
53 >>
54
55 statement(instruction) ::= <<
56 <instruction>
57 >>
58
59 whilestmt(statement, expr, labelCond, labelWhile) ::= <<
60 goto COND<labelCond> ; Jump to while condition
61 WHILE<labelWhile>:
62 <statement>
63 COND<labelCond>:
64 <expr> ; Execute condition
65 ifne WHILE<labelWhile> ; Jump to start of inner while statement
66 >>
67
68 printstmt(firststatement, statements, type, t) ::= <<
69 getstatic java/lang/System/out Ljava/io/PrintStream;
70 <firststatement; separator="\n">
71 <if(type._void)>
72 <else>
73 istore_1
74 iload_1
75 <endif>
76
77 invokevirtual java/io/PrintStream/println(<t>)V
78
79 <statements; separator="\n">
80
81 <if(type._void)>
82 <else>
83 iload_1 ; repush the value to the stack if it is used again

```

```

84 <endif>
85
86 >>
87
88 printexpr(statements, t) ::= <<
89   getstatic java/lang/System/out Ljava/io/PrintStream;
90   <statements; separator="\n">
91   invokevirtual java/io/PrintStream/println(<t>)V ; add right constant pool reference bytes for println
92 >>
93
94 readstmt(statements, addr, void, type, classname) ::= <<
95   getstatic <classname>/in Ljava/io/BufferedReader;
96   invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
97
98   <if(type._int)>
99     invokestatic java/lang/Integer/parseInt(Ljava/lang/String;)I
100   <elseif(type._bool)>
101     invokestatic java/lang/Boolean/parseBoolean(Ljava/lang/String;)Z
102   <elseif(type._char)>
103     iconst_0
104     invokevirtual java/lang/String/charAt(I)C
105   <endif>
106
107
108   <if(type._void)>
109   <else>
110     istore_1
111     iload_1
112   <endif>
113
114   istore <addr> ; store value
115   <statements>
116   <if(void)>
117   <else>
118     iload_1 ; repush the value to the stack if it is used again
119   <endif>
120 >>
121
122 readvar(var, addr, expr, classname) ::= <<
123   getstatic <classname>/in Ljava/io/BufferedReader;
124   invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
125
126   <if(type._int)>
127     invokestatic java/lang/Integer/parseInt(Ljava/lang/String;)I
128   <elseif(type._bool)>
129     invokestatic java/lang/Boolean/parseBoolean(Ljava/lang/String;)Z
130   <elseif(type._char)>
131     iconst_0
132     invokevirtual java/lang/String/charAt(I)C
133   <endif>
134
135
136   istore <addr> ; store value into <var>
137
138 >>
139
140 identifier(addr) ::= <<
141   iload <addr>
142 >>
143
144 number(n, numberType) ::= <<
145   <if(numberType.lessthanfive)>
146     iconst_<n>
147   <elseif(numberType.minusone)>
148     iconst_m1
149   <elseif(numberType.byteType)>
150     bipush <n>
151   <elseif(numberType.shortType)>
152     sipush <n>
153   <else>
154     ldc <n>
155   <endif>
156 >>
157
158 character(c) ::= <<
159   bipush <c> ; Char
160 >>
161
162 boolean(b) ::= <<
163   iconst_<if(b)>1<else>0<endif> ; Bool
164 >>

```

```

165
166
167 assign(var, addr, expr) ::= <<
168 <expr>
169 istore <addr> ; store value into <var>
170 iload <addr> ; put value on the stack
171 >>
172
173 binexpr(x, y, instr) ::= <<
174 <x> ; expr1
175 <y> ; expr2
176 i<instr>
177 >>
178
179 binexprcomp(x, y, instr) ::= <<
180 <x>
181 <y>
182 if_icmp<instr> $+7 ; Go to iconst_1 if it is true
183 iconst_0
184 goto $+4 ; Go to the line after iconst_1
185 iconst_1
186 >>
187
188
189 unarynot(x, instr) ::= <<
190 <x> ; if x is 0 make it 1, if x is 1 make it 0
191 ifeq $+7 ; Go to iconst_1 if it is false
192 iconst_0
193 goto $+4 ; Go to the line after iconst_1
194 iconst_1 ; if original was 0, load 1
195 >>
196
197 unaryplus(x, instr) ::= << ; does nothing, is feature
198 <x>
199 >>
200
201 unarymin(x, instr) ::= <<
202 <x>
203 i<instr>
204 >>
205
206 ifstmnt(cond, statements, elseStmnts, labelElse, labelNext) ::= <<
207 <cond>
208 ifeq ELSE<labelElse>
209 <statements>
210 goto NEXT<labelNext>
211 ELSE<labelElse>:
212 <elseStmnts; separator="\n">
213 NEXT<labelNext>:
214 >>
215
216 elseifstmnt(cond, statements, elseStmnts, labelElse, labelNext) ::= <<
217 <cond>
218 ifeq ELSE<labelElse>
219 <statements>
220 goto NEXT<labelNext>
221 ELSE<labelElse>:
222 <elseStmnts; separator="\n">
223 NEXT<labelNext>:
224 >>
225
226 elsemaybestmnt(statements) ::= <<
227 <statements>
228 >>

```

A.6 Example test program

The following test is designed to test most of the programming language functionalities.

A.6.1 Alia code

```
1  ivar = begin
2      ivar1 = ivar2 = 0
3      read(ivar1, ivar2);
4      print(ivar1, ivar2);
5      const iconst1 = 1;
6      const iconst2 = 2;
7      ivar2 = ivar1 = +16 + 2 * -8;
8      print(ivar1 < ivar2 && iconst1 <= iconst2, iconst1 * iconst2 > ivar2 - ivar1);
9      ivar1 < read(ivar2) && iconst1 <= iconst2;
10     ivar2 = print(ivar2) + 1;
11 end + 1
12 bvar = begin
13     bvar = false
14     read(bvar);
15     print(bvar);
16     bvar = 12 / 5 * 5 + 12 % 5 == 12 && 6 >= 6;
17     const bconst = true;
18     print(!false && bvar == bconst || true != false);
19 end && true;
20 cvar = begin
21     cvar1 = 'c'
22     read(cvar1);
23     const cconst = 'c';
24     cvar2 = 'z';
25     print('a', cvar1 == cconst && cvar2 != 'b' || !true);
26     'b';
27 end;
28 print(ivar, bvar, cvar);
29
30 i = 0
31 z = 0
32 while x = 5; x > i do
33     print(i)
34     if z == 1 do
35         z = 0
36     elseif z == -1 do
37         z = 1
38     else
39         z = -1
40     i = i + 1
```

```
41 end  
42 end
```

A.6.2 Test results

The test program was run using the following input:

```
30  
-100  
998  
true  
z
```

It resulted in the following output:

```
30  
-100  
false  
true  
998  
true  
true  
a  
false  
1000  
true  
b  
0  
1  
1  
1  
2  
2  
2  
3  
3  
3  
4  
4  
4
```

A.6.3 Jasmin bytecode

```
1 ; Jasmin JBC assembler code generated by
  AliaCodeGenerator
2 .class public Complete
3 .super java/lang/Object
4 .field private static in Ljava/io/BufferedReader
  ;
5
6 .method static public <clinit>()V
7   .limit stack 5
8   new java/io/BufferedReader
9   dup
10  new java/io/InputStreamReader
11  dup
12  getstatic java/lang/System/in Ljava/io/
    InputStream;
13  invokespecial java/io/InputStreamReader/<init>
    >(Ljava/io/InputStream;)V
14  invokespecial java/io/BufferedReader/<init>(
    Ljava/io/Reader;)V
15  putstatic Complete/in Ljava/io/BufferedReader;
    return
16 .end method
17
18 .method public <init>()V
19   aload_0
20   invokenonvirtual java/lang/Object/<init>()V
21   return
22 .end method
23
24 .method public static main([Ljava/lang/String;)V
25   .limit stack 7
26   .limit locals 8
27
28   iconst_0
29   istore 2 ; store value into ivar2
30   iload 2 ; put value on the stack
31   istore 3 ; store value into ivar1
32   iload 3 ; put value on the stack
33   pop
34   getstatic Complete/in Ljava/io/BufferedReader
    ;
35   invokevirtual java/io/BufferedReader/readLine
    ()Ljava/lang/String;
36   invokestatic java/lang/Integer/parseInt(Ljava
    /lang/String;)I
37
38   istore_1
39   iload_1
40   istore 3 ; store value
41   getstatic Complete/in Ljava/io/BufferedReader
    ;
42   invokevirtual java/io/BufferedReader/readLine
    ()Ljava/lang/String;
43   invokestatic java/lang/Integer/parseInt(Ljava
    /lang/String;)I
44
45   istore 2 ; store value into ivar2
46   iload_1 ; repush the value to the stack if it
    is used again
47   pop
48   getstatic java/lang/System/out Ljava/io/
    PrintStream;
49   iload 3
50   invokevirtual java/io/PrintStream/println(I)V
51   getstatic java/lang/System/out Ljava/io/
    PrintStream;
52   iload 2
53   invokevirtual java/io/PrintStream/println(I)V
    ; add right constant pool reference bytes
    for println
54
55   ; does nothing, is feature
56   bipush 16 ; expr1
57   iconst_2 ; expr1
58   bipush 8
59   ineg ; expr2
60   imul ; expr2
61   iadd
62
63   istore 3 ; store value into ivar1
64   iload 3 ; put value on the stack
65   istore 2 ; store value into ivar2
66   iload 2 ; put value on the stack
67   pop
68   getstatic java/lang/System/out Ljava/io/
    PrintStream;
69   iload 3
70   iload 2
71   if_icmplt $+7 ; Go to iconst_1 if it is true
72   iconst_0
73   goto $+4 ; Go to the line after iconst_1
74   iconst_1 ; expr1
75   iconst_1
76   iconst_2
77   if_icmple $+7 ; Go to iconst_1 if it is true
78   iconst_0
79   goto $+4 ; Go to the line after iconst_1
80   iconst_1 ; expr2
81   iand
82   invokevirtual java/io/PrintStream/println(Z)V
83   getstatic java/lang/System/out Ljava/io/
    PrintStream;
84   iconst_1 ; expr1
85   iconst_2
86   ; expr2
87   imul
88   iload 2 ; expr1
89   iload 3 ; expr2
90   isub
91   if_icmpgt $+7 ; Go to iconst_1 if it is true
92   iconst_0
93   goto $+4 ; Go to the line after iconst_1
94   iconst_1
95   invokevirtual java/io/PrintStream/println(Z)V
    ; add right constant pool reference bytes
    for println
96
97   iload 3
98   getstatic Complete/in Ljava/io/BufferedReader
    ;
99   invokevirtual java/io/BufferedReader/readLine
    ()Ljava/lang/String;
100  invokestatic java/lang/Integer/parseInt(Ljava
    /lang/String;)I
101
102  istore_1
103  iload_1
104  istore 2 ; store value
105  iload_1 ; repush the value to the stack if it
    is used again
106  if_icmplt $+7 ; Go to iconst_1 if it is true
107  iconst_0
108  goto $+4 ; Go to the line after iconst_1
109  iconst_1 ; expr1
110  iconst_1
111  iconst_2
112  if_icmple $+7 ; Go to iconst_1 if it is true
113  iconst_0
114  goto $+4 ; Go to the line after iconst_1
115  iconst_1 ; expr2
116  iand
117  pop
118  getstatic java/lang/System/out Ljava/io/
    PrintStream;
119  iload 2
120  istore_1
121  iload_1
122  invokevirtual java/io/PrintStream/println(I)V
123
124  iload_1 ; repush the value to the stack if it
    is used again ; expr1
125  iconst_1 ; expr2
126  iadd
127  istore 2 ; store value into ivar2
128  iload 2 ; put value on the stack ; expr1
129  iconst_1 ; expr2
130  iadd
131  istore 2 ; store value into ivar
```

132	iload 2 ; put value on the stack	204	iconst_1 ; Bool ; expr2
133	pop	205	iand
134	iconst_0 ; Bool	206	istore 3 ; store value into bvar
135	istore 3 ; store value into bvar	207	iload 3 ; put value on the stack
136	iload 3 ; put value on the stack	208	pop
137	pop	209	bipush 99 ; Char
138	getstatic Complete/in Ljava/io/BufferedReader	210	istore 4 ; store value into cvar1
	;	211	iload 4 ; put value on the stack
139	invokevirtual java/io/BufferedReader/readLine	212	pop
	(L)Ljava/lang/String;	213	getstatic Complete/in Ljava/io/BufferedReader
140	invokestatic java/lang/Boolean/parseBoolean(;
	Ljava/lang/String;)Z	214	invokevirtual java/io/BufferedReader/readLine
141	istore_1		(L)Ljava/lang/String;
142	iload_1	215	iconst_0
143	istore 3 ; store value	216	invokevirtual java/lang/String/charAt(I)C
144	iload_1 ; repush the value to the stack if it	217	istore_1
	is used again	218	iload_1
145	pop	219	istore 4 ; store value
146	getstatic java/lang/System/out Ljava/io/	220	iload_1 ; repush the value to the stack if it
	PrintStream;		is used again
147	iload 3	221	pop
148	istore_1	222	bipush 122 ; Char
149	iload_1	223	istore 6 ; store value into cvar2
150	invokevirtual java/io/PrintStream/println(Z)V	224	iload 6 ; put value on the stack
151		225	pop
152	iload_1 ; repush the value to the stack if it	226	getstatic java/lang/System/out Ljava/io/
	is used again		PrintStream;
153	pop	227	bipush 97 ; Char
154	bipush 12 ; expr1	228	invokevirtual java/io/PrintStream/println(C)V
155	iconst_5	229	getstatic java/lang/System/out Ljava/io/
156	; expr2		PrintStream;
157	idiv ; expr1	230	iload 4
158	iconst_5 ; expr2	231	bipush 99 ; Char
159	imul ; expr1	232	if_icmpeq \$+7 ; Go to iconst_1 if it is true
160	bipush 12 ; expr1	233	iconst_0
161	iconst_5 ; expr2	234	goto \$+4 ; Go to the line after iconst_1
162	irem ; expr2	235	iconst_1 ; expr1
163	iadd	236	iload 6
164	bipush 12	237	bipush 98 ; Char
165	if_icmpeq \$+7 ; Go to iconst_1 if it is true	238	if_icmpne \$+7 ; Go to iconst_1 if it is true
166	iconst_0	239	iconst_0
167	goto \$+4 ; Go to the line after iconst_1	240	goto \$+4 ; Go to the line after iconst_1
168	iconst_1 ; expr1	241	iconst_1 ; expr2
169	bipush 6	242	iand ; expr1
170	bipush 6	243	iconst_1 ; Bool ; if x is 0 make it 1, if x
171	if_icmpge \$+7 ; Go to iconst_1 if it is true		is 1 make it 0
172	iconst_0	244	ifeq \$+7 ; Go to iconst_1 if it is false
173	goto \$+4 ; Go to the line after iconst_1	245	iconst_0
174	iconst_1 ; expr2	246	goto \$+4 ; Go to the line after iconst_1
175	iand	247	iconst_1 ; if original was 0, load 1 ; expr2
176	istore 3 ; store value into bvar	248	ior
177	iload 3 ; put value on the stack	249	invokevirtual java/io/PrintStream/println(Z)V
178	pop		; add right constant pool reference bytes
179	getstatic java/lang/System/out Ljava/io/		for println
	PrintStream;	250	
180	iconst_0 ; Bool ; if x is 0 make it 1, if x	251	bipush 98 ; Char
	is 1 make it 0	252	istore 4 ; store value into cvar
181	ifeq \$+7 ; Go to iconst_1 if it is false	253	iload 4 ; put value on the stack
182	iconst_0	254	pop
183	goto \$+4 ; Go to the line after iconst_1	255	getstatic java/lang/System/out Ljava/io/
184	iconst_1 ; if original was 0, load 1 ; expr1		PrintStream;
185	iload 3	256	iload 2
186	iconst_1 ; Bool	257	invokevirtual java/io/PrintStream/println(I)V
187	if_icmpeq \$+7 ; Go to iconst_1 if it is true	258	getstatic java/lang/System/out Ljava/io/
188	iconst_0		PrintStream;
189	goto \$+4 ; Go to the line after iconst_1	259	iload 3
190	iconst_1 ; expr2	260	invokevirtual java/io/PrintStream/println(Z)V
191	iand ; expr1		; add right constant pool reference bytes
192	iconst_1 ; Bool		for println
193	iconst_0 ; Bool	261	getstatic java/lang/System/out Ljava/io/
194	if_icmpne \$+7 ; Go to iconst_1 if it is true		PrintStream;
195	iconst_0	262	iload 4
196	goto \$+4 ; Go to the line after iconst_1	263	invokevirtual java/io/PrintStream/println(C)V
197	iconst_1 ; expr2		; add right constant pool reference bytes
198	ior		for println
199	istore_1	264	
200	iload_1	265	iconst_0
201	invokevirtual java/io/PrintStream/println(Z)V	266	istore 5 ; store value into i
202		267	iload 5 ; put value on the stack
203	iload_1 ; repush the value to the stack if it	268	pop
	is used again ; expr1	269	iconst_0

```

270     istore 6 ; store value into z
271     iload 6 ; put value on the stack
272     pop
273     goto COND4 ; Jump to while condition
274     WHILE5:
275     getstatic java/lang/System/out Ljava/io/
        PrintStream;
276     iload 5
277     istore_1
278     iload_1
279     invokevirtual java/io/PrintStream/println(I)V
280
281     iload_1 ; repush the value to the stack if it
        is used again
282     pop
283     iload 6
284     iconst_1
285     if_icmpeq $+7 ; Go to iconst_1 if it is
        true
286     iconst_0
287     goto $+4 ; Go to the line after iconst_1
288     iconst_1
289     ifeq ELSE2
290     iconst_0
291     istore 6 ; store value into z
292     iload 6 ; put value on the stack
293     goto NEXT3
294     ELSE2:
295     iload 6
296     iconst_1
297     ineg
298     if_icmpeq $+7 ; Go to iconst_1 if it is
        true
299     iconst_0
300     goto $+4 ; Go to the line after iconst_1
301     iconst_1

```

```

302     ifeq ELSE0
303     iconst_1
304     istore 6 ; store value into z
305     iload 6 ; put value on the stack
306     goto NEXT1
307     ELSE0:
308     iconst_1
309     ineg
310     istore 6 ; store value into z
311     iload 6 ; put value on the stack
312     pop
313     iload 5 ; expr1
314     iconst_1
315     ; expr2
316     iadd
317     istore 5 ; store value into i
318     iload 5 ; put value on the stack
319     NEXT1:
320     NEXT3:
321     pop
322     COND4:
323     iconst_5
324     istore 7 ; store value into x
325     iload 7 ; put value on the stack
326     pop
327     iload 7
328     iload 5
329     if_icmpgt $+7 ; Go to iconst_1 if it is true
330     iconst_0
331     goto $+4 ; Go to the line after iconst_1
332     iconst_1 ; Execute condition
333     ifne WHILE5 ; Jump to start of inner while
        statement
334
335     return
336 .end method

```