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) principe, 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.

- A statement can be ended by any of the above separators ($\backslash n$, ;, *EOF*).
- The while statement 'while S1 do S2 end' is executed as follows. The statement S1 is evaluated, if its value is true then S2 is evaluated and the while statement is run again. If the value of S1 is false then the execution is completed. S1 must be of type boolean. This statement is of type void. Declarations made in S1 are valid in S1 and S2. The scope of declarations made in S2 is only S2.
- If statements of the form 'if S1 do S2 (elseif S3 do S4)* (else S5)?' are executed as follows. S1 is executed. If S1 is true, then S2 is evaluated. If S1 is false and there is an S3, then S3 is evaluted and if true S4 is executed. If the evaluated S3 is false there is another elseif statement then it is evaluated, same as an if statement is. If S1, S3 and all other elseifs have evaluated to false, then S5 is executed. If there is no S5, execution has completed. The type os S1 and S3 must be boolean. If there is no else part, the type of the statement is void. If there is an else part, then if all S2, S4, S5 are the same type, then that is the type of the conditional statement. If S2, S4, S5 are not of the same type then the type of the statement is void. Special scope rules apply, a declaration in S1 or any S3 is valid in S2, S4, S5 as long as the declaration precedes the use.
- A compound statement is a closed set of statements. Any assignments made in the statements can not be used outside of the compound statement. The result and type of the compound statement are the same as the last statement in the compound statement.

Assignment

```
expr_assignment = identifier "=" expr_assignment
| expr (: type)?
;

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

- An expression assignment binds one or more identifiers to a value yielded by an expression E. If a type is included then the type and the type of the expression must match. The identifiers can thereafter be used in applied occurrences. The expression assignment yields the value of the expression.
- The expression 'const I = P (:T)?' is executed as follows. I is bound to the value P. If T was included, T and E must be of the same type. The expression is of type P. I can be used in applied occurrences. I can not be assigned a different value at a later time.

Expressions

```
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;
```

- 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' 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 comparitive 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_stmnt |
      "(" expr ")" |
      compound_stmnt |
      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 O E2] =
    expr [E1]
    expr [E2]
    instruction [O] // The specific instruction, e.g. iadd etc.

expr [E1 OC 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
       invokenonvirtual 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 handeled 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) plustTest: 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.

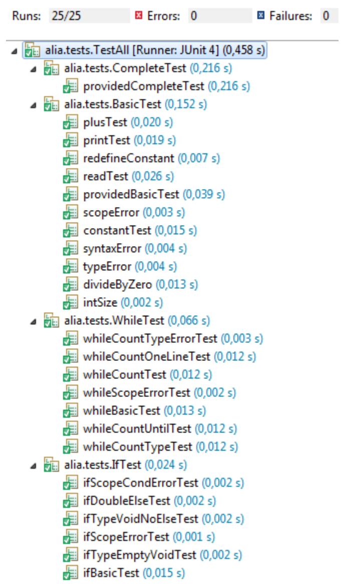
2. WhileTest.java: Contains tests for the while conditional statement.
   (a) providedBasicTest (Syntax): Test with correct syntax which is checked for correct output.
   (b) plustTest (Syntax): Tests a basic arithmetic expression.
   (c) printTest (Syntax): Tests print expressions.
   (d) readTest (Syntax): 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

```
grammar Alia;

options {
    k=1;                                // LL(1) - do not use LL(*)
    language=Java;                      // target language is Java (= default)
    output=AST;                         // build an AST
}

tokens {
    COLON       =   ':'     ;
    NEWLINE       =   '\n'     ;
    COMMA       =   ','     ;
    SEMICOLON   =   ';'     ;
    LPAREN      =   '('     ;
    RPAREN      =   ')'     ;
    LCURLY      =   '{'     ;
    RCURLY      =   '}'     ;
    SQUOTE      =   '\''     ;

    // operators
    BECOMES     =   '='    ;
    PLUS        =   '+'     ;
    PLUS_OP     =   'plusop'     ;
    MINUS       =   '-'     ;
    MINUS_OP    =   'minop'     ;
    TIMES       =   '*'     ;
    DIV         =   '/'     ;

    // comp. operators
    GT          =   '>'     ;
    GE          =   '>='    ;
    LT          =   '<'     ;
    LE          =   '<='    ;
    EQ          =   '=='    ;
    NQ          =   '!='    ;


    AND     =   'and'    ;
    AND_ALT     =   '&&'    ;
    OR        =   'or'    ;
    OR_ALT      =   '||'    ;
    NOT         =   '!'    ;
    MOD         =   '%'    ;

    // types
    INT       =   'int';
    BOOL      =   'boolean';
    CHAR      =   'char';
    STRING      =   'string';

    // keywords
    PROGRAM     =   'program'   ;
    PRINT       =   'print'     ;
    READ        =   'read'     ;
    IF        =   'if'   ;
    ELSE      =   'else'   ;
    ELSEIF    =   'elseif'  ;
    DO          =   'do'        ;
    END         =   'end'        ;
    WHILE       =   'while'     ;
    TRUE    =   'true'    ;
    FALSE   =   'false'    ;
    CONST       = 'const'   ;
    DEF         =   'def'        ;
    BEGIN       =   'begin'        ;

    FUNC        =   'func' ;
    EXPR_LIST;
    COMPOUND;
    TYPE;
    ID;
    LOCALSIZE;
}

@lexer::header {
package alia;
```

```
}

@header {
package alia;
}


program : (func_def | (statement end_statement) | NEWLINE!)*;

statements : (statement (end_statement statements)? | NEWLINE! statements)?;
statements_cond : statement (end_statement statements)? | NEWLINE! statements_cond;
statement : (expr_assignment | const_assignment) (COLON^ type)?
      | while_stmnt;
end_statement : NEWLINE! | SEMICOLON! | EOF!;

// Syntactic predicate to recognize assignments
// Syntactic predicates can be easily left out if we do not allow expr as statements
expr_assignment : (IDENTIFIER BECOMES) => (IDENTIFIER BECOMES^) expr_assignment |
      expr ;

const_assignment : CONST^ IDENTIFIER BECOMES primitive;

expr : expr1 ((OR | OR_ALT)^ expr1)*;
expr1 : expr2 ((AND | AND_ALT)^ expr2)*;
expr2 : expr3 ((GT | GE | LT | LE | EQ | NQ)^ expr3)*;
expr3 : expr4 ((PLUS | MINUS)^ expr4)*;
expr4 : expr5 ((TIMES | DIV | MOD)^ expr5)*;
expr5 : NOT^ operand | operand | expr_minus | expr_plus;
expr_minus : MINUS operand -> ^(MINUS_OP operand);
expr_plus : PLUS operand -> ^(PLUS_OP operand);
operand : read |
        print |
        if_stmnt |
        LPAREN! expr RPAREN! |
        compound_stmnt |
        primitive |
        func_identifier;

compound_stmnt : BEGIN statements END -> ^(COMPOUND statements);

primitive : NUMBER | CHAR_EXPR | boolean_expr;

char_expr : SQUOTE! LETTER SQUOTE!;

func_identifier : IDENTIFIER
        (LPAREN^ exprlist? RPAREN)?;

while_stmnt : WHILE statements_cond DO statements END -> ^(WHILE statements_cond ^(DO statements));

if_stmnt : IF statements_cond DO statements else_stmnt? END ->
      ^(IF statements_cond ^(DO statements?) else_stmnt?);

else_stmnt
    : ELSEIF statements_cond DO statements else_stmnt? ->
      ^(ELSEIF statements_cond ^(DO statements?) else_stmnt?)
  | (ELSE^ statements)
  ;

print : PRINT^ LPAREN! exprlist RPAREN!;
read : READ^ LPAREN! varlist RPAREN!;

varlist : IDENTIFIER (COMMA! IDENTIFIER)*;
exprlist : expr (COMMA! expr)*;

func_def : DEF IDENTIFIER LPAREN! varlist RPAREN! statements END;

// Lexer rules

boolean_expr : TRUE | FALSE;

type : CHAR | INT | BOOL;

CHAR_EXPR : SQUOTE LETTER SQUOTE;

IDENTIFIER
    :   LETTER (LETTER | DIGIT)*
    ;

NUMBER
    :   DIGIT+
    ;
```

```

COMMENT
    :   ('//' .* '\n' | '/*' .* '*/')
            { $channel=HIDDEN; }

    ;

WS
    :   (' ' | '\t' | '\f' | '\r')+
            { $channel=HIDDEN; }
    ;

fragment LETTER :   LOWER | UPPER ;
fragment DIGIT  :   ('0'..'9') ;
fragment LOWER  :   ('a'..'z') ;
fragment UPPER  :   ('A'..'Z') ;
```

A.3 Checker

```
tree grammar AliaChecker;

options {
    k=1;                                // LL(1) - do not use LL(*)
    tokenVocab=Alia;                    // import tokens from Calc.tokens
    ASTLabelType=CommonTree;            // AST nodes are of type CommonTree
    superClass=CheckerAux;
    output=AST;
}

@header {
package alia;
import alia.types.*;
import alia.symtab.SymbolTable;
import alia.symtab.IdEntry;
import java.util.Set;
import java.util.HashSet;
}

// Alter code generation so catch-clauses get replaced with this action.
// This disables ANTLR error handling: AliaExceptions are propagated upwards.
@rulecatch {
    catch (RecognitionException e) {
      if(!e.getMessage().equals("")) {
      System.err.println("Exception!:"+e.getMessage());
    }
    throw (new AliaException(""));
    }
}

@members {

}

program
    :   { symTab.openScope(); }
      (statement)+
      { symTab.closeScope(); }
      -> LOCALSIZE[getLocalSize()] (statement)+
    ;

statements returns [_Type type = new _Void()]
  : (t=statement
    { $type = $t.type; }
  )*;

statement returns [_Type type = new _Void()]
    :   ^(WHILE {symTab.openScope();} stat=statements {symTab.openScope();}
          ^(DO statements) {symTab.closeScope();symTab.closeScope();}  )
    { checkBoolType($stat.type, $stat.tree); }
    |   t=expr
  { $type = $t.type; }
    ;
```

```
expr returns [_Type type]
    :   to=operand
    {
      $type = $to.type;
      }
    |   (^(c=OR t1=expr t2=expr)
    |   ^(c=OR_ALT t1=expr t2=expr)
    |   ^(c=AND t1=expr t2=expr)
    |   ^(c=AND_ALT t1=expr t2=expr))
      {
          checkEqualType($t1.type, $t2.type, $t1.tree);
          checkBoolType($t1.type, $t1.tree);
          $type = new _Bool();
          String typename = String.valueOf($type);
      }
      -> ^($c expr expr TYPE[typename])
    |   (^(c=EQ t1=expr t2=expr)
    |   ^(c=NQ t1=expr t2=expr)
    |   ^(c=LE t1=expr t2=expr)
    |   ^(c=GE t1=expr t2=expr)
    |   ^(c=GT t1=expr t2=expr)
    |   ^(c=LT t1=expr t2=expr))
      {
          checkEqualType($t1.type, $t2.type, $t1.tree);
          $type = new _Bool();
          String typename = String.valueOf($type);
      }
      -> ^($c expr expr TYPE[typename])
    |   (^(c=PLUS te1=expr te2=expr)
    |   ^(c=MINUS te1=expr te2=expr)
    |   ^(c=TIMES te1=expr te2=expr)
    |   ^(c=DIV te1=expr te2=expr)
    |   ^(c=MOD te1=expr te2=expr))
      {
        checkMathType($te1.type, $te2.type, $te1.tree);
        $type = new _Int();
          String typename = String.valueOf($type);
      }
      -> ^($c expr expr TYPE[typename])
    | ^(PRINT te=exprlist)
      {
        $type = $te.type;
          String typename = String.valueOf($type);
        }
      -> ^(PRINT TYPE[typename] exprlist)
    | ^(READ tv=varlist)
      {
        $type = $tv.type;
          String typename = String.valueOf($type);
        }
      -> ^(READ TYPE[typename] varlist)
  | ^(c=(NOT) to=operand)
      {
          $type = $to.type;
          String typename = String.valueOf($type);
          checkBoolType($to.type, $to.tree);
        }
      -> ^($c operand TYPE[typename])
    | ^(c=( PLUS_OP | MINUS_OP ) o=operand)
      {
          $type = $o.type;
          String typename = String.valueOf($type);
          checkEqualType($o.type, new _Int(), $o.tree);
      }
      -> ^($c operand TYPE[typename])
    |   ^(IF
        {
          symTab.openScope(); // Open scope for conditional statements, the scope is the same for the
    IF and ELSEIF conditions
        }
        t=statements
        {
          symTab.openScope(); // Open scope for the first statement
        }
        ^(DO
        ts=statements
        {
          checkBoolType($t.type, $ts.tree);
          symTab.closeScope(); // Close scope for the first statement
```

```
          }
          )
          texp=else_stmnt?
          {
            symTab.closeScope(); // Close scope for the conditional statements
            checkBoolType($t.type, $t.tree);
            $type = checkTypesIf($ts.type,$texp.type);
          }
        )
      |   ^(COLON ^(BECOMES id=IDENTIFIER t1=expr) typ=type)
          {
            _Type declType = checkEqualType($t1.type, $typ.type, $t1.tree);
            declare($id.text, declType, $t1.tree);
          $type = declType;

            String typename = String.valueOf($type);
            String identifier = String.valueOf(getIdentifier($id.text, $id.tree));
          }
          -> ^(BECOMES ^(IDENTIFIER TYPE[typename] ID[identifier]) expr)
      |   ^(BECOMES id=IDENTIFIER t1=expr)
          {
            declare($id.text, $t1.type, $t1.tree);
          $type = $t1.type;
          checkNotVoid($type, $t1.tree);

            String typename = String.valueOf($type);
            String identifier = String.valueOf(getIdentifier($id.text, $id.tree));
          }
        -> ^(BECOMES ^(IDENTIFIER TYPE[typename] ID[identifier]) expr)
      |   ^(COMPOUND
        { // symTab.openScope
          symTab.openScope();
        }
        t=statements)
          {
            // closeScope
            symTab.closeScope();
          $type = $t.type;
            String typename = String.valueOf($type);
          }
        -> ^(COMPOUND TYPE[typename] statements)
       |   ^(CONST id=IDENTIFIER BECOMES prim=primitive (COLON typ=type)?)
            { _Type declType = checkEqualType($prim.type, $typ.type, $prim.tree);
              declareConst($id.text, declType, prim, $prim.tree);
              $type = declType;
              String typename = String.valueOf($type);
              String identifier = String.valueOf(getIdentifier($id.text, $prim.tree));

              }
        -> // constants are not used after checking fase, thus they are removed
      ;

else_stmnt returns [_Type type]
    :
      ^(ELSEIF t=statements
        ^(DO
          {
            symTab.openScope(); // Open scope for this elseif statement
          }
        ts=statements
        )
          te=else_stmnt?
          {
            checkBoolType($t.type, $t.tree);
            $type = checkTypesIf($ts.type, $te.type);
            symTab.closeScope();
          }
      )
    | ^(ELSE
          {
            symTab.openScope(); // Open scope for the else statement
          }
        ts=statements
          {
            $type = $ts.type;
            symTab.closeScope(); // Open scope for the else statement
          }
        )
    ;
```

```
operand returns [_Type type]
    :   id=identifier
        { $type = $id.type; }
    |   n=NUMBER
      { $type = new _Int();checkInt(n);  }
    |   c=CHAR_EXPR
      { $type = new _Char(); }
    |   b=(TRUE | FALSE)
      { $type = new _Bool(); }
    ;

identifier returns [_Type type]
  :
  id=IDENTIFIER
    {
      $type = getType($id.text, $id.tree);
            String typename = String.valueOf($type);
            String identifier = String.valueOf(getIdentifier($id.text, $id.tree));
          Boolean constant = retrieve($id.text, $id.tree).isConstant();
          Token value = getConstant($id.text);
    }
  -> {constant && typename.equals("int") }? ^(NUMBER[value])
  -> {constant && typename.equals("char") }? ^(CHAR_EXPR[value])
  -> {constant && typename.equals("bool") && value.getText().equals("true")  }? ^(TRUE[value])
  -> {constant && typename.equals("bool") && value.getText().equals("false")  }? ^(FALSE[value])
  -> ^(IDENTIFIER TYPE[typename] ID[identifier])
  ;


varlist returns [_Type type]
  : t=identifier
    {
      $type = $t.type;
    }
    (identifier
      {
        $type = new _Void();
      }

    )*
  ;

exprlist returns [_Type type]
    : tl=exprentry
    {
      checkNotVoid($tl.type, $tl.tree);
      $type = $tl.type;
    }
    (t=exprentry
      {
        checkNotVoid($t.type, $t.tree);
        $type = new _Void();
      }
    )*
  ;

exprentry returns [_Type type]
  : t=expr
  {
    $type = $t.type;
        String typename = String.valueOf($t.type);
  } -> TYPE[typename] expr
  ;

type returns [_Type type]
    :   INT
        { $type = new _Int(); }
    |   CHAR
        { $type = new _Char(); }
    |   BOOL
        { $type = new _Bool(); }
    ;

primitive returns [_Type type] :
      NUMBER {$type = new _Int();}
    | CHAR_EXPR
        { $type = new _Char(); }
    | boolean_expr
        { $type = new _Bool(); }
    ;
boolean_expr : TRUE | FALSE;
```

A.4 Code Generator

```
tree grammar AliaCodeGeneratorStringTemplate;

options {
    k=1;                              // LL(1) - do not use LL(*)
    tokenVocab=Alia;                  // import tokens from Calc.tokens
    output = template;
    ASTLabelType=CommonTree;          // AST nodes are of type CommonTree
    superClass=CodeGeneratorAux;
}

@header {
package alia;
import alia.symtab.SymbolTable;
import alia.symtab.IdEntry;
import java.util.Set;
import java.util.HashSet;
}

program
    :
      localSize=LOCALSIZE (s+=exprPop)+
            ->  file(instructions={$s},stackMax={getStackMax()},localSize={$localSize},classname={getProgramClass()})
    ;

statements @init { startExpression(); }
       @after { endExpression(); }
  : (s+=exprPopInterleaved)* ->  statements(instructions={$s});

statementsPop @init { startExpression(); }
       @after { endExpression(); }
  : (s+=exprPop)* ->  statements(instructions={$s});

exprPopInterleaved @init { String pop = ""; }
  : {pop=pops(endExpression());startExpression();}
    s=expr ->  exprPopInterleaved(instruction={$s.st}, pop={pop});

exprPop @init { String pop = ""; }
  : {startExpression();}
    s=expr
    {pop=pops(endExpression());} ->  exprPop(instruction={$s.st}, pop={pop});

expr @init { }
   @after {}
    :   o=operand             -> statement(instruction={$o.st})
    |   ^(OR t1=expr t2=expr t=TYPE) {decStack();}      -> binexpr(x={$t1.st}, y={$t2.st}, instr={"or"})
    |   ^(OR_ALT t1=expr t2=expr t=TYPE) {decStack();}    -> binexpr(x={$t1.st}, y={$t2.st}, instr={"or"})
    |   ^(AND t1=expr t2=expr t=TYPE) {decStack();}       -> binexpr(x={$t1.st}, y={$t2.st}, instr={"and"})
    |   ^(AND_ALT t1=expr t2=expr t=TYPE) {decStack();}     -> binexpr(x={$t1.st}, y={$t2.st}, instr={"and"})
    |   ^(EQ t1=expr t2=expr t=TYPE) {decStack();}      -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"eq"})
    |   ^(NQ t1=expr t2=expr t=TYPE) {decStack();}      -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"ne"})
    |   ^(LE t1=expr t2=expr t=TYPE) {decStack();}      -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"le"})
    |   ^(GE t1=expr t2=expr t=TYPE) {decStack();}      -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"ge"})
    |   ^(GT t1=expr t2=expr t=TYPE) {decStack();}      -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"gt"})
    |   ^(LT t1=expr t2=expr t=TYPE) {decStack();}      -> binexprcomp(x={$t1.st}, y={$t2.st}, instr={"lt"})
    |   ^(PLUS t1=expr t2=expr t=TYPE) {decStack();}      -> binexpr(x={$t1.st}, y={$t2.st}, instr={"add"})
    |   ^(MINUS t1=expr t2=expr t=TYPE) {decStack();}       -> binexpr(x={$t1.st}, y={$t2.st}, instr={"sub"})
    |   ^(TIMES t1=expr t2=expr t=TYPE) {decStack();}       -> binexpr(x={$t1.st}, y={$t2.st}, instr={"mul"})
    |   ^(DIV t1=expr t2=expr t=TYPE) {decStack();}       -> binexpr(x={$t1.st}, y={$t2.st}, instr={"div"})
    |   ^(MOD t1=expr t2=expr t=TYPE) {decStack();}       -> binexpr(x={$t1.st}, y={$t2.st}, instr={"rem"})
```

```
    | ^(WHILE cond=statements {decStack();} ^(DO t2=statementsPop))  -> whilestmt(expr={$cond.st},
    statement={$t2.st}, labelCond={newLabel()}, labelWhile={newLabel()})
    | ^(PRINT t=TYPE te=TYPE fexp=expr (exp+=exprPrint)*) {decStackIfVoid(getType($t.toString()));}
                   -> printstmt(firststatement={$fexp.st},statements={$exp},type={getType($t.toString()
    )},t={getType($te.toString()).T})
    | ^(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()})
    | ^(NOT o=operand t=TYPE)                    -> unarynot(x={$o.st}, instr={"not"})
    | ^(PLUS_OP o=operand t=TYPE)                  -> unaryplus(x={$o.st}, instr={"plus"})
    | ^(MINUS_OP o=operand t=TYPE)                 -> unarymin(x={$o.st}, instr={"neg"})
    |   { startExpression(); } ^(IF
        stif1=statements
        ^(DO stif2=statements)
        (elsestmnts=elseif)?
      ) { decStack();endExpression(); }                                             -> ifstmnt(cond={
    $stif1.st}, statements={$stif2.st}, elseStmnts={elsestmnts}, labelElse={newLabel()}, labelNext={
    newLabel()})
    |   ^(BECOMES ^(id=IDENTIFIER t=TYPE a=ID) {incStack();} t1=expr {decStack();}) -> assign(var={$id
    },addr={$a}, expr={$t1.st})
    |   ^(COMPOUND t=TYPE s=statements)                -> statements(instructions={$s.st})
    ;
elseif @init { decStack(); }
   :
    ^(ELSEIF stelseif1=statements
          ^(DO   stelseif2=statements)  {decStack();}
          elsestmnts=elseif)            ->  elseifstmnt(cond={$stelseif1.st}, statements={$stelseif2.st
    }, elseStmnts={elsestmnts}, labelElse={newLabel()}, labelNext={newLabel()})
  | ^(ELSE stelse=statements)                      -> elsemaybestmnt(statements={$stelse.st})
      ;
operand @init {incStack();}
    :   i=identifier       -> statement(instruction={$i.st})
    |   n=NUMBER                 -> number(n={$n.toString()}, numberType={whatNumber(Integer.parseInt(
    $n.toString()))})
    |   c=CHAR_EXPR              -> character(c={(int) c.toString().charAt(1)})
    |   b=(TRUE | FALSE)         -> boolean(b={$b.toString().equals("true")})
    ;

exprPrint @init {decStack();} :
  t=TYPE exp=expr -> printexpr(statements={$exp.st},t={getType($t.toString()).T})
  ;

varRead @init {incStack();decStack();} :
  ^(id=IDENTIFIER t=TYPE a=ID) -> readvar(var={$id},addr={$a},type={getType($t.toString())},classname={
    getProgramClass()})
  ;

identifier
  : ^(id=IDENTIFIER t=TYPE a=ID)           -> identifier(addr={$a})
  ;

varlist
  : s+=identifier
    (s+=identifier)*
    ->  statements(instructions={$s});

exprlist
    : s+=expr
    (s+=expr)*
    ->  statements(instructions={$s})
  ;

type
    :   INTEGER
    |   CHAR
    |   BOOL
    ;
```

A.5 String templates

```
group tmg;

```

```
file(instructions,stackMax,localSize,classname) ::= <<
; Jasmin JBC assembler code generated by AliaCodeGenerator
.class public <classname>
.super java/lang/Object

.field private static in Ljava/io/BufferedReader;

.method static public \<clinit\>()V
  .limit stack 5

  new java/io/BufferedReader
  dup
  new java/io/InputStreamReader
  dup
  getstatic java/lang/System/in Ljava/io/InputStream;
  invokespecial java/io/InputStreamReader/\<init\>(Ljava/io/InputStream;)V
  invokespecial java/io/BufferedReader/\<init\>(Ljava/io/Reader;)V
  putstatic <classname>/in Ljava/io/BufferedReader;
  return

.end method

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

.method public static main([Ljava/lang/String;)V
   .limit stack <stackMax>
   .limit locals <localSize>

   <instructions; separator="\n">

   return
.end method
>>

statements(instructions) ::= <<
<instructions; separator="\n">
>>

exprPopInterleaved(instruction, pop) ::= <<
<pop>
<instruction>
>>

exprPop(instruction, pop) ::= <<
<instruction>
<pop>
>>

statement(instruction) ::= <<
<instruction>
>>

whilestmt(statement, expr, labelCond, labelWhile) ::= <<
goto COND<labelCond>    ; Jump to while condition
WHILE<labelWhile>:
<statement>
COND<labelCond>:
<expr>    ; Execute condition
ifne WHILE<labelWhile>    ; Jump to start of inner while statement
>>

printstmt(firststatement, statements, type, t) ::= <<
getstatic java/lang/System/out Ljava/io/PrintStream;
<firststatement; separator="\n">
<if(type._void)>
<else>
istore_1
iload_1
<endif>

invokevirtual java/io/PrintStream/println(<t>)V

<statements; separator="\n">

<if(type._void)>
<else>
iload_1 ; repush the value to the stack if it is used again
```

```
<endif>

>>

printexpr(statements, t) ::= <<
getstatic java/lang/System/out Ljava/io/PrintStream;
<statements; separator="\n">
invokevirtual java/io/PrintStream/println(<t>)V ; add right constant pool reference bytes for println
>>

readstmt(statements, addr, void, type, classname) ::= <<
getstatic <classname>/in Ljava/io/BufferedReader;
invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;

<if(type._int)>
invokestatic java/lang/Integer/parseInt(Ljava/lang/String;)I
<elseif(type._bool)>
invokestatic java/lang/Boolean/parseBoolean(Ljava/lang/String;)Z
<elseif(type._char)>
iconst_0
invokevirtual java/lang/String/charAt(I)C
<endif>


<if(type._void)>
<else>
istore_1
iload_1
<endif>

istore <addr>                   ; store value
<statements>
<if(void)>
<else>
iload_1 ; repush the value to the stack if it is used again
<endif>
>>

readvar(var, addr, expr, classname) ::= <<
getstatic <classname>/in Ljava/io/BufferedReader;
invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;

<if(type._int)>
invokestatic java/lang/Integer/parseInt(Ljava/lang/String;)I
<elseif(type._bool)>
invokestatic java/lang/Boolean/parseBoolean(Ljava/lang/String;)Z
<elseif(type._char)>
iconst_0
invokevirtual java/lang/String/charAt(I)C
<endif>


istore <addr>                   ; store value into <var>

>>

identifier(addr) ::= <<
iload <addr>
>>

number(n, numberType) ::= <<
<if(numberType.lessthanfive)>
iconst_<n>
<elseif(numberType.minusone)>
iconst_m1
<elseif(numberType.byteType)>
bipush <n>
<elseif(numberType.shortType)>
sipush <n>
<else>
ldc <n>
<endif>
>>

character(c) ::= <<
bipush <c> ; Char
>>

boolean(b) ::= <<
iconst_<if(b)>1<else>0<endif> ; Bool
>>
```

```
assign(var, addr, expr) ::= <<
<expr>
istore <addr>                   ; store value into <var>
iload <addr>            ; put value on the stack
>>

binexpr(x, y, instr) ::= <<
<x> ; expr1
<y> ; expr2
i<instr>
>>

binexprcomp(x, y, instr) ::= <<
<x>
<y>
if_icmp<instr> $+7 ; Go to iconst_1 if it is true
iconst_0
goto $+4 ; Go to the line after iconst_1
iconst_1
>>


unarynot(x, instr) ::=  <<
<x>   ; if x is 0 make it 1, if x is 1 make it 0
ifeq $+7 ; Go to iconst_1 if it is false
iconst_0
goto $+4 ; Go to the line after iconst_1
iconst_1 ; if original was 0, load 1
>>

unaryplus(x, instr) ::= <<  ; does nothing, is feature
<x>
>>

unarymin(x, instr) ::=  <<
<x>
i<instr>
>>

ifstmnt(cond, statements, elseStmnts, labelElse, labelNext) ::= <<
  <cond>
  ifeq ELSE<labelElse>
  <statements>
  goto NEXT<labelNext>
  ELSE<labelElse>:
  <elseStmnts; separator="\n">
  NEXT<labelNext>:
>>

elseifstmnt(cond, statements, elseStmnts, labelElse, labelNext) ::= <<
  <cond>
  ifeq ELSE<labelElse>
  <statements>
  goto NEXT<labelNext>
  ELSE<labelElse>:
  <elseStmnts; separator="\n">
  NEXT<labelNext>:
>>

elsemaybestmnt(statements)           ::= <<
<statements>
>>
```

A.6 Example test program

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

A.6.1 Alia code

```
ivar = begin
    ivar1 = ivar2 = 0
    read(ivar1, ivar2);
    print(ivar1, ivar2);
    const iconst1 = 1;
    const iconst2 = 2;
    ivar2 = ivar1 = +16 + 2 * -8;
    print(ivar1 < ivar2 && iconst1 <= iconst2,iconst1 * iconst2 > ivar2 - ivar1);
    ivar1 < read(ivar2) && iconst1 <= iconst2;
    ivar2 = print(ivar2) + 1;
  end + 1
bvar = begin
    bvar = false
    read(bvar);
    print(bvar);
    bvar = 12 / 5 * 5 + 12 % 5 == 12 && 6 >= 6;
    const bconst = true;
    print(!false && bvar == bconst || true != false);
  end && true;
cvar = begin
    cvar1 = 'c'
    read(cvar1);
    const cconst = 'c';
    cvar2 = 'z';
    print('a', cvar1 == cconst && cvar2 != 'b' || !true);
    'b';
  end;
print(ivar, bvar, cvar);

i = 0
z = 0
while x = 5; x > i do
  print(i)
  if z == 1 do
    z = 0
  elseif z == -1 do
    z = 1
  else
    z = -1
    i = i + 1
```

```
    end
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

```
; Jasmin JBC assembler code generated by AliaCodeGenerator
.class public Complete
.super java/lang/Object
.field private static in Ljava/io/BufferedReader;

.method static public <clinit>()V
  .limit stack 5
  new java/io/BufferedReader
  dup
  new java/io/InputStreamReader
  dup
  getstatic java/lang/System/in Ljava/io/InputStream;
  invokespecial java/io/InputStreamReader/<init>(Ljava/io/InputStream;)V
  invokespecial java/io/BufferedReader/<init>(Ljava/io/Reader;)V
  putstatic Complete/in Ljava/io/BufferedReader;
  return
.end method

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

.method public static main([Ljava/lang/String;)V
   .limit stack 7
   .limit locals 8

   iconst_0
   istore 2 ; store value into ivar2
   iload 2 ; put value on the stack
   istore 3 ; store value into ivar1
   iload 3 ; put value on the stack
   pop
   getstatic Complete/in Ljava/io/BufferedReader;
   invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
   invokestatic java/lang/Integer/parseInt(Ljava/lang/String;)I

   istore_1
   iload_1
   istore 3 ; store value
   getstatic Complete/in Ljava/io/BufferedReader;
   invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
   invokestatic java/lang/Integer/parseInt(Ljava/lang/String;)I

   istore 2 ; store value into ivar2
   iload_1 ; repush the value to the stack if it is used again
   pop
   getstatic java/lang/System/out Ljava/io/PrintStream;
   iload 3
   invokevirtual java/io/PrintStream/println(I)V
   getstatic java/lang/System/out Ljava/io/PrintStream;
   iload 2
   invokevirtual java/io/PrintStream/println(I)V ; add right constant pool reference bytes for println

     ; does nothing, is feature
   bipush 16 ; expr1
   iconst_2 ; expr1
   bipush 8
   ineg ; expr2
   imul ; expr2
   iadd
   istore 3 ; store value into ivar1
   iload 3 ; put value on the stack
   istore 2 ; store value into ivar2
   iload 2 ; put value on the stack
   pop
   getstatic java/lang/System/out Ljava/io/PrintStream;
   iload 3
   iload 2
   if_icmplt $+7 ; Go to iconst_1 if it is true
   iconst_0
   goto $+4 ; Go to the line after iconst_1
   iconst_1 ; expr1
   iconst_1
   iconst_2
   if_icmple $+7 ; Go to iconst_1 if it is true
   iconst_0
   goto $+4 ; Go to the line after iconst_1
   iconst_1 ; expr2
   iand
   invokevirtual java/io/PrintStream/println(Z)V
   getstatic java/lang/System/out Ljava/io/PrintStream;
   iconst_1 ; expr1
   iconst_2
    ; expr2
   imul
   iload 2 ; expr1
   iload 3 ; expr2
   isub
   if_icmpgt $+7 ; Go to iconst_1 if it is true
   iconst_0
   goto $+4 ; Go to the line after iconst_1
   iconst_1
   invokevirtual java/io/PrintStream/println(Z)V ; add right constant pool reference bytes for println

   iload 3
   getstatic Complete/in Ljava/io/BufferedReader;
   invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
   invokestatic java/lang/Integer/parseInt(Ljava/lang/String;)I

   istore_1
   iload_1
   istore 2 ; store value
   iload_1 ; repush the value to the stack if it is used again
   if_icmplt $+7 ; Go to iconst_1 if it is true
   iconst_0
   goto $+4 ; Go to the line after iconst_1
   iconst_1 ; expr1
   iconst_1
   iconst_2
   if_icmple $+7 ; Go to iconst_1 if it is true
   iconst_0
   goto $+4 ; Go to the line after iconst_1
   iconst_1 ; expr2
   iand
   pop
   getstatic java/lang/System/out Ljava/io/PrintStream;
   iload 2
   istore_1
   iload_1
   invokevirtual java/io/PrintStream/println(I)V

   iload_1 ; repush the value to the stack if it is used again ; expr1
   iconst_1 ; expr2
   iadd
   istore 2 ; store value into ivar2
   iload 2 ; put value on the stack ; expr1
   iconst_1 ; expr2
   iadd
   istore 2 ; store value into ivar
```

```
    iload 2 ; put value on the stack
    pop
    iconst_0 ; Bool
    istore 3 ; store value into bvar
    iload 3 ; put value on the stack
    pop
    getstatic Complete/in Ljava/io/BufferedReader;
    invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
    invokestatic java/lang/Boolean/parseBoolean(Ljava/lang/String;)Z
    istore_1
    iload_1
    istore 3 ; store value
    iload_1 ; repush the value to the stack if it is used again
    pop
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iload 3
    istore_1
    iload_1
    invokevirtual java/io/PrintStream/println(Z)V
    iload_1 ; repush the value to the stack if it is used again
    pop
    bipush 12 ; expr1
    iconst_5
    ; expr2
    idiv ; expr1
    iconst_5 ; expr2
    imul ; expr1
    bipush 12 ; expr1
    iconst_5 ; expr2
    irem ; expr2
    iadd
    bipush 12
    if_icmpeq $+7 ; Go to iconst_1 if it is true
    iconst_0
    goto $+4 ; Go to the line after iconst_1
    iconst_1 ; expr1
    bipush 6
    bipush 6
    if_icmpge $+7 ; Go to iconst_1 if it is true
    iconst_0
    goto $+4 ; Go to the line after iconst_1
    iconst_1 ; expr2
    iand
    istore 3 ; store value into bvar
    iload 3 ; put value on the stack
    pop
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iconst_0 ; Bool   ; if x is 0 make it 1, if x is 1 make it 0
    ifeq $+7 ; Go to iconst_1 if it is false
    iconst_0
    goto $+4 ; Go to the line after iconst_1
    iconst_1 ; if original was 0, load 1 ; expr1
    iload 3
    iconst_1 ; Bool
    if_icmpeq $+7 ; Go to iconst_1 if it is true
    iconst_0
    goto $+4 ; Go to the line after iconst_1
    iconst_1 ; expr2
    iand ; expr1
    iconst_1 ; Bool
    iconst_0 ; Bool
    if_icmpne $+7 ; Go to iconst_1 if it is true
    iconst_0
    goto $+4 ; Go to the line after iconst_1
    iconst_1 ; expr2
    ior
    istore_1
    iload_1
    invokevirtual java/io/PrintStream/println(Z)V
    iload_1 ; repush the value to the stack if it is used again ; expr1
    iconst_1 ; Bool ; expr2
    iand
    istore 3 ; store value into bvar
    iload 3 ; put value on the stack
    pop
    bipush 99 ; Char
    istore 4 ; store value into cvar1
    iload 4 ; put value on the stack
    pop
    getstatic Complete/in Ljava/io/BufferedReader;
    invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
    iconst_0
    invokevirtual java/lang/String/charAt(I)C
    istore_1
    iload_1
    istore 4 ; store value
    iload_1 ; repush the value to the stack if it is used again
    pop
    bipush 122 ; Char
    istore 6 ; store value into cvar2
    iload 6 ; put value on the stack
    pop
    getstatic java/lang/System/out Ljava/io/PrintStream;
    bipush 97 ; Char
    invokevirtual java/io/PrintStream/println(C)V
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iload 4
    bipush 99 ; Char
    if_icmpeq $+7 ; Go to iconst_1 if it is true
    iconst_0
    goto $+4 ; Go to the line after iconst_1
    iconst_1 ; expr1
    iload 6
    bipush 98 ; Char
    if_icmpne $+7 ; Go to iconst_1 if it is true
    iconst_0
    goto $+4 ; Go to the line after iconst_1
    iconst_1 ; expr2
    iand ; expr1
    iconst_1 ; Bool   ; if x is 0 make it 1, if x is 1 make it 0
    ifeq $+7 ; Go to iconst_1 if it is false
    iconst_0
    goto $+4 ; Go to the line after iconst_1
    iconst_1 ; if original was 0, load 1 ; expr2
    ior
    invokevirtual java/io/PrintStream/println(Z)V ; add right constant pool reference bytes for println
    bipush 98 ; Char
    istore 4 ; store value into cvar
    iload 4 ; put value on the stack
    pop
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iload 2
    invokevirtual java/io/PrintStream/println(I)V
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iload 3
    invokevirtual java/io/PrintStream/println(Z)V ; add right constant pool reference bytes for println
    getstatic java/lang/System/out Ljava/io/PrintStream;
    iload 4
    invokevirtual java/io/PrintStream/println(C)V ; add right constant pool reference bytes for println
    iconst_0
    istore 5 ; store value into i
    iload 5 ; put value on the stack
    pop
    iconst_0
```

```
   istore 6 ; store value into z
   iload 6 ; put value on the stack
   pop
   goto COND4   ; Jump to while condition
   WHILE5:
   getstatic java/lang/System/out Ljava/io/PrintStream;
   iload 5
   istore_1
   iload_1
   invokevirtual java/io/PrintStream/println(I)V

   iload_1 ; repush the value to the stack if it is used again
   pop
     iload 6
     iconst_1
     if_icmpeq $+7 ; Go to iconst_1 if it is true
     iconst_0
     goto $+4 ; Go to the line after iconst_1
     iconst_1
     ifeq ELSE2
     iconst_0
     istore 6 ; store value into z
     iload 6 ; put value on the stack
     goto NEXT3
     ELSE2:
       iload 6
       iconst_1
       ineg
       if_icmpeq $+7 ; Go to iconst_1 if it is true
       iconst_0
       goto $+4 ; Go to the line after iconst_1
       iconst_1
       ifeq ELSE0
       iconst_1
       istore 6 ; store value into z
       iload 6 ; put value on the stack
       goto NEXT1
       ELSE0:
       iconst_1
       ineg
       istore 6 ; store value into z
       iload 6 ; put value on the stack
       pop
       iload 5 ; expr1
       iconst_1
        ; expr2
       iadd
       istore 5 ; store value into i
       iload 5 ; put value on the stack
       NEXT1:
     NEXT3:
   pop
   COND4:
   iconst_5
   istore 7 ; store value into x
   iload 7 ; put value on the stack
   pop
   iload 7
   iload 5
   if_icmpgt $+7 ; Go to iconst_1 if it is true
   iconst_0
   goto $+4 ; Go to the line after iconst_1
   iconst_1   ; Execute condition
   ifne WHILE5    ; Jump to start of inner while statement

   return
.end method
```