Programming Paradigms Project
Potato Programming Language

Group 29

Stijn Dijkstra
Joris Jonkers
Teun Mansfelt

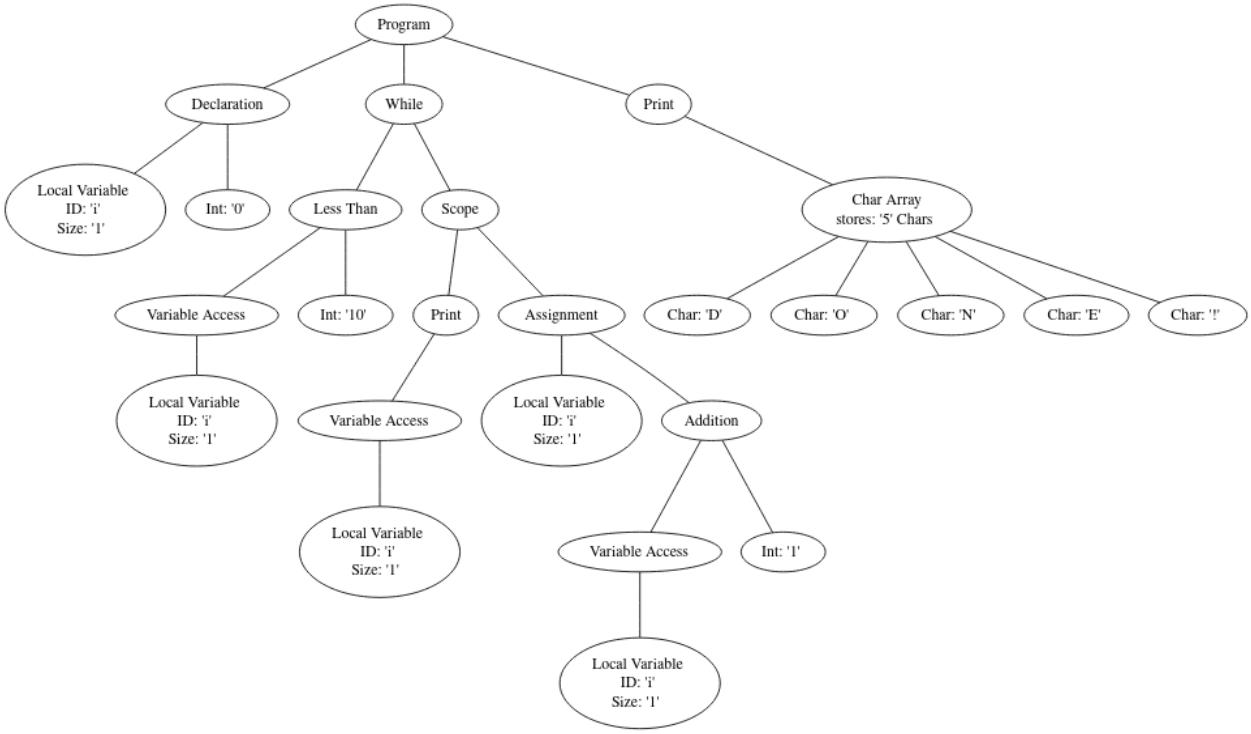

Fig. 1 AST for “simpleWhile.pot”

```
int i = 0;
while (i < 10){
    print i;
    i = i + 1;
}
printNL "DONE!";
```

Fig. 2 “simpleWhile.pot” source code

Main features

Datatypes

Our programming language offers support for integers, characters and booleans and arrays of all of those data types. Arrays are fixed-size defined upon declaration. The language supports 3 types for integer-literals; decimal, binary and hexadecimal. It is also possible to write String literals, which are functionally the same as character-array literals.

Operations

We support various types of basic operations. Integer addition, integer subtraction, integer multiplication, logical OR and logical AND. Both Integers and characters support greater than (or equal) and smaller than (or equal) comparisons. All basic data types can be compared for equality and inequality as well. Our unary operations include logical NOT and integer negation. For arrays we offer concatenation.

Operator precedence is as expected from conventional mathematics. The 5 levels of precedence from highest to lowest are first parenthesis, then logic NOT and integer negation, then multiplication and concatenation, then addition and subtraction, then logical comparisons and finally logical OR and logical AND.

Structure

A program in our language consists of a list of statements that are executed sequentially. Control flow is done using while-loops and if-statements with optional else-statements. Loops and if statements can be freely nested and always open new scopes so that variable names can be redeclared within a loop. Scopes also consist of statements that are executed sequentially. As noted, they can be used to indicate the body of a control flow structure, but they can also be used independently. When used independently, scopes only serve to indicate how long variables should remain in memory. For visualization purposes we have also featured a graph generation flag in our main class, when this is toggled the AST for the compiled program will also be generated in a .dot file matching the compiled program's title (on by default). One of these ASTs has been placed on the main page of this report. Please keep in mind that these AST's will become difficult to read when working with large arrays. These .dot-files can be loaded in the browser with Graphviz Online, which makes it possible to zoom. Please do take a look at our AST graphs on Graphviz as it will make our data structure easier to understand. The graphs will automatically be built at the content root of our .pot files.

Multithreading

Our language supports multithreading using a fork/join structure. A fork statement consists of "fork" and a new scope which is what will be executed by the newly started thread. The main thread will skip that block and continue normal operation until it reaches a join statement. The main thread will busy-wait at that join statement until the newly started thread has finished execution. We support nested fork/join blocks and our compiler also makes sure that sequential fork/join blocks will reuse Sprockell instances where possible.

Our language supports locks and synchronized blocks that synchronize using those locks. Our locks are not reentrant, so nesting synchronized blocks with the same lock will cause a deadlock. We use synchronized blocks to make sure that the programmer does not have to worry about forgetting to unlock locks.

Language Description

Statement Constituents

Variables

Usage	Semantics
This feature should be used to retrieve a variable that has been declared in the scope or a prior scope. If a variable access takes place in a separate thread it will only retrieve shared variables from the parent scope, otherwise it will throw an unknown identifier error. It is not possible to access a variable after the scope it was declared in was closed.	An operation which requires a variable access will be compiled with the address for the specific variable, this means that the location for this variable is loaded into the top of the stack. When the manipulation or retrieval of the variable occurs from an operation the variable can be written to or retrieved as needed.
Syntax	
type (variable identifier) = *expression* (variable identifier) = *expression* *char 'c' = characterVariableIdentifier*	
Example	
Variable accesses, declarations and assignments are extensively used in the examples for the types of statements	

Literals

Usage	Semantics
Literals are expressions that can be used to initialize or analyze direct declarations for different types or type arrays	Literals are translated at compile time to LiteralExpressions or ArrayExpressions.
Syntax	

[single literal,...] ← *an array literal*
'c' ← *a character literal*
"some string" ← *a string literal synonymous to a character array*
3 ← *an integer literal*
true | false ← *boolean literals*

Statements

Declarations and Assignments

Usage		Semantics
Declarations are used to create a variable in a scope and asssign a value to it simultaniously. This can only take place once for every identifier for every scope. After a declaration a variable can be assigned a different value of a matching type, these assignments can take place only in the same or a scope encapsulated by the scope where it was declared. Shared variables are unique instances which can also be reached from other threads and can therefore also be assigned from other scopes. Declarations of shared an local variables can take place in any scope where the aforementioned conditions apply.		Declarations and assignments are semantically the same. The expression that will be assigned to the variable will be loaded onto the stack. In the case of arrays they will be loaded in reverse order, since were dealing with a stack based processor. The values are then popped one by one and stored at the memory address of the variable.
Syntax		**Result**
Declaration	*type[] identifier = expression;* *type identifier = expression;* lock *identifier;*	an array variable was declared and assigned the expression a variable was declared an assigned the expression a new lock with a identifier was declared
Assignment	*identifier = expression;* *identifier[int literal] = expression;*	A variable's value was reassigned to the expression The element at an index was reassigned the expression

Example Expressions

These declarations and assignments are extensively used in the examples below.

Scopes

Usage	Semantics
Scopes are used in order to separate different layers of statements, after a new scope is opened it becomes possible to redeclare the variables declared in its parent scope. Scopes are also used in if statements and while loops to indicate which statements must be executed if true/false and to indicate which statements to loop over respectively. Furthermore scopes are also used to indicate what statements should be run concurrently in a separate thread. The root scope is called a Program in our grammar and also in our generated AST graphs.	`There are no real semantics for scopes as describe the way a program will be compiled and can't be linked to any operation in the generated code.`
Syntax	
{ *Statements* }	

Example Expressions

`Scopes are extensively used in the examples for while and if statements below.`

Fork

Usage	Semantics
A fork is used to start a certain scope of statements in a separate thread. It is intialized with the shared variables such that it can access and modify these variables.	`When our program performs a fork, the instructions of the parallel scope are written to a new Sprockell instance. The new thread starts by polling the shared memory at its own sprockell ID. When this location is set to 1 by the calling (main) thread, the sprockell instance will start executing its instructions. After it has finished it will`
Syntax	
fork *parallelscope statements join*	

	set the shared memory at its sprockell ID back to 0. This also indicates to the calling thread that the thread has finished its task. When the calling thread reaches a join it will wait for that address to become 0 it will not wait if the thread has finished and thus already set the memory at its address to 0.

Synchronized Block

Usage	Semantics
A synchronized block is used to ensure that a certain labelled scope of code is mutually exclusive for any other threads that try to access the scope of code with the same identifier. Typical use would involve placing the critical section of your code within synchronized blocks for sections which can interfere with eachothers functioning.	When a thread enters a synchronized block it polls the value at the memory address of the provided lock using a TestAndSet instruction. As long as this value is 1 and the TestAndSet return 0 it will busy wait. After acquiring the lock and executing the instruction in the block it will reset the value at the address of the lock to 0.
Syntax	
sync *identifier* scope	

While

Usage	Semantics
This feature should be used for executing a block of statements repeatedly until a condition is no longer met. Expression should be of type boolean. Make sure that all variables in this expression are declared beforehand.	The expression is evaluated, if it is evaluated to false, a jump will be made to the next statement after the while-block. If the expression is evaluated to true, the statements in this scope will be executed and after that a jump will be made back to the start of the while-block, starting a repeat of the whole process.
Syntax	
while (*expression*) *scope*	

Example	Generated code
`int x = 1;` `int y = 10;` `int i = 1;` `while (i <= y) {` `  x = x * i;` `  i = i + 1;` `}`	*conditionCode* **Pop regA** **Branch regA (Rel 2)** **Jump (Rel (***loopCode.size + 2***))** *loopCode* **Jump (Rel (***-(conditionCode.size + loopCode.size + 2)***))**

if(condition**){**true scope**}else{**else scope**}**

Usage	Semantics
This should be used to do branching, the block of statements is executed only if a condition is met. An optional else block may be added, which is executed only if the the condition is *not* met. Expression should be of type boolean. Make sure that all variables in this expression are declared beforehand.	Without else: The expression is evaluated. If it is evaluated to true, no jump is made. If it is evaluated to false, a jump will be made to the end of the if-block. With else: The expression is evaluated. If it is evaluated to true, no jump is made. If it is evaluated to false, a jump is made to the start of the else-block. Additionally, a jump instruction is
Syntax	
`if (`*expression*`) `*scope* `if (`*expression*`) `*scope*` else `*scope*	

	inserted at the end of the if-block that jumps to the end of the else-block.

Examples	Result	Generated Code
`int test = 0;` `if (true) {` `  test = 1;` `}`	`test = 1;`	`Load (ImmValue 0) regA` `Store regA (DirAddr 0)` `Load (ImmValue 1) regA` `Branch regA (Rel (2))` `Jump (Rel (3))` `Load (ImmValue 1) regA` `Store regA (DirAddr 0) ← memory location of int test`
`int x = 0;` `if (false) {` `  x = 1;` `} else {` `  x = 2;` `}`	`test = 2;`	`Load (ImmValue 0) regA` `Store regA (DirAddr 0)` `Load (ImmValue 0) regA` `Branch regA (Rel (4))` `Load (ImmValue 2) regA` `Store regA (DirAddr 0)` `Jump (Rel (3))` `Load (ImmValue 1) regA` `Store regA (DirAddr 0) ← memory location of int test`

Binary Expressions

Usage	Semantics
Binary expressions are what make it possible to do calculations, comparisons and boolean operations on two operands. They are not statements by themselves, as the value that results from evaluating an expression is not stored anywhere unless as expressions make up the constituent parts of statements. Whilst compiling the types are checked to be correct for the binary expression type and a type error is thrown if these do not match. The expected types are listed with their operands in a table below.	First, the right operand is evaluated and the result of this evaluation is placed at the top of the stack. After this the left operand is evaluated and placed at the top of the stack. These results are then popped from the stack and placed into registers where we can do our operation. The result of this operation is then pushed onto the stack for use by a statement.
Syntax	
expression operator expression	

Example Expressions	Result	Generated Code
bool test = true && false;	*test* = false	Load (ImmValue 1) regA Push regA Load (ImmValue 0) regA Pop regB Compute And regB regA regA Store regA (DirAddr 0) ← memory location of bool test
bool test = true || false;	*test* = true	Load (ImmValue 1) regA Push regA Load (ImmValue 0) regA Pop regB Compute Or regB regA regA Store regA (DirAddr 0) ← memory location of bool test
bool test = true || 1;	type error	————————————
int test = 3 + 3;	*test* = 6	Load (ImmValue 3) regA Push regA Load (ImmValue 3) regA Pop regB Compute Add regB regA regA Store regA (DirAddr 0) ← memory location of int test
int test = 3 - 'a';	type error	————————————
int test = 3 - 3;	*test* = 0	Load (ImmValue 3) regA Push regA Load (ImmValue 3) regA Pop regB Compute Sub regB regA regA Store regA (DirAddr 0) ← memory location of int test
int test = 3 * 3;	*test* = 9	Load (ImmValue 3) regA Push regA Load (ImmValue 3) regA Pop regB Compute Mul regB regA regA Store regA (DirAddr 0) ← memory location of int test
bool test = 'a' == 'a';	*test* = true	Load (ImmValue 97) regA Push regA Load (ImmValue 97) regA Pop regB

		`Compute Equal regB regA regA` `Store regA (DirAddr 0) ← memory location of bool test`
bool test = 1 == 3;	*test* = false	`Load (ImmValue 1) regA` `Push regA` `Load (ImmValue 3) regA` `Pop regB` `Compute Equal regB regA regA` `Store regA (DirAddr 0) ← memory location of bool test`
bool test = 'a' == 1;	type error	————————————
bool test = 'a' < 'b';	*test* = true	`Load (ImmValue 97) regA` `Push regA` `Load (ImmValue 98) regA` `Pop regB` `Compute Lt regB regA regA` `Store regA (DirAddr 0) ← memory location of bool test`
bool test = 'a' < 3;	type error	————————————

Binary Expression Operators

Operation	**Symbols**	**Left operand type**	**Right operand type**	**Result type**
Logical	'&&', '||'	boolean	boolean	boolean
Mathematical	'+', '-', '*'	integer	integer	integer
Equality	'==','!='	boolean | integer | character	*same as left operand*	*same as left operand*
Orderable Equality	'<=', '>=', '<', '>'	integer | character	*same as left operand*	*same as left operand*

Unary Expressions

Usage	Semantics
Unary expressions are expressions that are applied to a single operand, they make it possible to flip the sign for either booleans. The permitted operand types for the differing expressions are listed below	First, the operand is evaluated and the result of this evaluation is placed at the top of the stack. After this depending on the unary expression this value is either subracted from zero, xor it with true, or evaluate the prioritized expressions in the case of paranthesis. The result from this operation is then pushed onto the stack for use by a statement.
Syntax	
operator expression *operator expression operator* (in case of parenthesis)	

Example Expressions	Result	Generated Code
`int test = ((3*3)-3);`	`test = 6`	`Load (ImmValue 3) regA` `Push regA` `Load (ImmValue 3) regA` `Pop regB` `Compute Mul regB regA regA ← 3*3` `Push regA` `Load (ImmValue 3) regA` `Pop regB` `Compute Sub regB regA regA ← (3*3)-3` `Store regA (DirAddr 0) ← memory address of test`
`bool test = !true;`	`test = false`	`Load (ImmValue 84) regE` `Load (ImmValue 70) regF` `Load (ImmValue 1) regA` `Load (ImmValue 1) regB` `Compute Xor regA regB regA` `Store regA (DirAddr 0) ← memory address of test`
`int test = -3;`	`test = -3`	`Load (ImmValue 84) regE` `Load (ImmValue 70) regF` `Load (ImmValue 3) regA` `Compute Sub reg0 regA regA ← 0 - 3 into reg A`

		`Store regA (DirAddr 0) ← memory address of test`

Unary Expression Operators

Operation	Symbols	Operand type	Result type
Logic Not	'!' *operator*	boolean	boolean
Negation	'-' *operator*	integer	integer
Parenthesis	'(' *Operator* ')'	boolean | integer | character	*same as operand*

Array Expressions

Usage

Arrays are fixed size and can be declared using expressions. All elements of arrays must be of the same type, which means that their constituent expressions must also match in type. When there are multiple types of expressions in an array a type error will be thrown by the compiler. It is possible to declare a character array using string declaration syntax the resulting type for a variable declared in ths manner will be a character array. It is also possible to initialize array without giving the array explicit values, in this case the array will contain null values.

Semantics

For arrays we reserve as much memory as it required to store every object in the array. Accessing and modifying an array is done by first evaluating an expression which wishes to read or write to an array element and retrieving the memory location of the array that is to be manipulated. Which is done by evaluation of the expression. If we then want to modify or retrieve this array element our program has the location in memory for this element allowing us to retrieve its value, do some operations on this value and then if desired write back a modified value into the array element's memory location.

Syntax

Declaration	*type*[] *id* = [*expression*, *expression*... , *expression*]; *type*[] *id* = new *type*(*expression*); *string id* = "*example*"; *string id = new char(3);*
Access	*variable = id*[*index*];
Element Assignment	*id*[*index*] = *expression;*
String Element Assignment	*id*[*index*] = *expression;*
Length of array	*int len* = length *id*
Array concatenation	*string test = id ++ id*

Example Expressions	Result	Generated Code

string test = "ab"; char[] test = “ab”;	test = “ab”	Load (ImmValue 84) regE Load (ImmValue 70) regF Load (ImmValue 98) regA Push regA Load (ImmValue 97) regA Store regA (DirAddr 0) ← store “b” in diraddr 0 Pop regA Store regA (DirAddr 1) ← store “a” in diraddr 1
char[] test = new char(2);	test = “ “	Load (ImmValue 84) regE Load (ImmValue 70) regF Load (ImmValue 32) regA Push regA Load (ImmValue 32) regA Store regA (DirAddr 0) ← store “ “ in diraddr 0 Pop regA Store regA (DirAddr 1) ← store “ “ in diraddr 1
string testa = “a”; string testb = “b”; string testc = testa ++ testb;	test_c = “ab”	Load (ImmValue 84) regE Load (ImmValue 70) regF Load (ImmValue 97) regA Store regA (DirAddr 0) Load (ImmValue 98) regA Store regA (DirAddr 1) Load (DirAddr 1) regA Push regA Load (DirAddr 0) regA Store regA (DirAddr 2) ← store “b” in diraddr 2 Pop regA Store regA (DirAddr 3) ← store “a” in diraddr 3
string teststr = “abc”; int len = length teststr;	teststr = “abc” len = 3;	Load (ImmValue 84) regE Load (ImmValue 70) regF Load (ImmValue 99) regA Push regA Load (ImmValue 98) regA Push regA Load (ImmValue 97) regA Store regA (DirAddr 0) Pop regA Store regA (DirAddr 1) Pop regA Store regA (DirAddr 2)

		Load (ImmValue 3) regA Store regA (DirAddr 3) ← store value 3 into diraddr 3
string teststr = “aaa”; teststr[1] = ‘b’;	testr = “aba"	Load (ImmValue 84) regE Load (ImmValue 70) regF Load (ImmValue 97) regA Push regA Load (ImmValue 97) regA Push regA Load (ImmValue 97) regA Store regA (DirAddr 0) Pop regA Store regA (DirAddr 1) Pop regA Store regA (DirAddr 2) Load (ImmValue 98) regA ← load value ‘b’ Push regA Load (ImmValue 1) regA ← array index Load (ImmValue 0) regB ← array memory address Compute Add regA regB regB ← array element address Pop regA Store regA (IndAddr regB) ← store ‘b’ into array elem

Array Expression Operators

Operation	Symbols	Left operand type	Right operand type	Result type
Length	‘length’ *operator*	Array	————————	Int
Array Element Access	*operator* ‘[‘ *operator* ‘]’	Array of any type	Integer	Type of array
Concatenation	*operator* ‘++’ *operator*	Array of any type	Array of same type as left operand	Array of same type as left operand

Conclusion

Looking back we are happy to see that we have created a language with well functioning basic structures and extensive error handling and reporting. At first we had underestimated how much

work each feature would require and had built a grammar that offered all the features we wanted from a language. Halfway through we came to the conclusion that starting from a minimalist implementation and building from a working version would be much easier to expand on. For that reason we stripped out some functionality, during that process we learned reconsidered our approach and structure and improved it. Things like our compile-time approach to memory allocation turns out to limit flexibility in array operations.

A big realization we made was how much easier it is to generate code that heavily relies on the stack. It lends itself very well to recursively generating code since all generated code is self-containing and can be easily inserted into a bigger block of code.

Overall we learned that we should have put more effort into creating a well-defined and attainable grammar at the start. Most of our difficulties came from us having to work independently, as some of our group members were absent due to covid and a fractured arm at separate times. We also struggled a bit with discussions because all three of us have strong opinions about code structure as some of us prefer imperative programming and some object focussed programming. However we feel that we were able to combine these styles of coding by using objects in a tree structure with a large file that generates this tree.

We are confident that what we delivered is well-structured and would be easy to extend at a later time.

Due to our personal circumstances we finished our completed grammar and compiler at a late date were unfortunately unable to add automatic JUNIT tests, however we have included a large set of broken and functioning programs which highlight our compilers error recognition and handling. Which can be run through setting the relevant flags in our main.java.

The project is not completed, we really needed one more day in order to add the testing and extend on the report due to our personal circumstances.

Appendix A
Grammar Specification

```
lexer grammar PotatoTokens;

// === KEYWORDS
=======================================================
SHARED : 'shared';

IF: 'if';
ELSE: 'else';
WHILE: 'while';

PRINT: 'print';
PRINT_NL: 'printNL';
INPUT: 'userInput';
FORK: 'fork';
JOIN: 'join';
NEW: 'new';
LOCK: 'lock';
SETLOCK: 'setlock';
UNLOCK: 'unlock';
LENGTH: 'length';

// === LITERALS
=======================================================
fragment LETTER    : [a-zA-Z];
fragment DIGIT     : [0-9];
fragment BIN_DIGIT : [01];
fragment HEX_DIGIT : [0-9a-f];
fragment CHARACTER : (~['"]);

// Integer literals
DEC_INT_LITERAL : DIGIT+;           // Decimal integer
BIN_INT_LITERAL : '0b' BIN_DIGIT+;  // Binary integer
HEX_INT_LITERAL : '0x' HEX_DIGIT+;  // Hexadecimal integer

// Boolean literal
BOOL_LITERAL : 'true' | 'false';

// Character Literal
CHAR_LITERAL : '\'' CHARACTER '\'';

// String Literal
STRING_LITERAL : '"' CHARACTER* '"';
```

```
// === DATA TYPES
=====================================================
// Integer types
TYPE_INT : 'int';

// Boolean type
TYPE_BOOL : 'bool';

// Character type
TYPE_CHAR : 'char';

// String type
TYPE_STRING : 'string';

// Identifier
IDENTIFIER : LETTER (LETTER | DIGIT)*;

// === IGNORED TOKENS
================================================
COMMENT    : '//' (~[\n])* -> skip;
WHITESPACE : [ \t\r\n]+ -> skip;

// === PROGRAM
============================================================

program
    : statement* EOF
    ;

// === STATEMENTS
=======================================================
statement
    : declaration ';'
    | assignment ';'
    | if
    | while
    | scope
    | print ';'
    | fork ';'
    | lock ';'
    ;

declaration
    : SHARED? type '[]' IDENTIFIER
          '=' (expression | array_literal)
      #arrayDeclaration
    | SHARED? type IDENTIFIER '=' expression    #primitiveDeclaration
    | SHARED? TYPE_STRING IDENTIFIER
          '=' expression                        #stringDeclaration
```

```
    | LOCK IDENTIFIER                              #lockDeclaration
    ;


lock
    : SETLOCK IDENTIFIER                                 #setLock
    | UNLOCK IDENTIFIER                                  #releaseLock
    ;

assignment
    : IDENTIFIER
          '=' (expression|array_literal)         #assignVariable
    | IDENTIFIER '[' index=expression ']'
          '=' expr=expression                    #assignArrayElement
    ;

if
    : IF '(' expression ')' trueScope=scope (ELSE elseScope=scope)?
    ;

while
    : WHILE '(' expression ')' scope
    ;

scope
    : '{' statement* '}'
    ;

fork
    : FORK parallelScope statement* JOIN
    ;

parallelScope
    : '{' statement* '}'
    ;

print
    : PRINT expression
    | PRINT_NL expression
    ;

// === EXPRESSIONS
============================================================
expression
    : expression_l1                                    #exprLevel1
    | left=expression '==' right=expression_l1         #equals
    | left=expression '!=' right=expression_l1         #notEquals
    | left=expression '<'  right=expression_l1         #lessThan
```

```
    | left=expression '<=' right=expression_l1
#lessThanEquals
    | left=expression '>'  right=expression_l1         #greaterThan
    | left=expression '>=' right=expression_l1
#greaterThanEquals
    | left=expression '&&' right=expression_l1         #logicalAnd
    | left=expression '||' right=expression_l1         #logicalOr
    ;

expression_l1
    : expression_l2                                    #exprLevel2
    | left=expression_l1 '+' right=expression_l2       #addition
    | left=expression_l1 '-' right=expression_l2       #subtraction
    ;

expression_l2
    : expression_l3                                    #exprLevel3
    | left=expression_l2 '*' right=expression_l3
#multiplication
    | left=expression_l3 '++' right=expression_l2      #concatenation
    ;

expression_l3
    : literal                                          #literalValue
    | variable                                         #variableCall
    | input                                            #userInput
    | '(' expression ')'                               #parenthesis
    | LENGTH arrayExpression=expression_l2             #length
    ;

input
    : INPUT request=STRING_LITERAL
    ;

// === TYPES, VARIABLES & LITERALS
============================================
type
    : TYPE_INT                                         #intType
    | TYPE_BOOL                                        #boolType
    | TYPE_CHAR                                        #charType
    ;

variable
    : IDENTIFIER '[' expression ']'                    #arrayAccess
    | IDENTIFIER
#variableAccess
    ;

literal
```

```
    : DEC_INT_LITERAL                                   #decIntLiteral
    | BIN_INT_LITERAL                                   #binIntLiteral
    | HEX_INT_LITERAL                                   #hexIntLiteral
    | BOOL_LITERAL                                      #boolLiteral
    | CHAR_LITERAL                                      #charLiteral
    | STRING_LITERAL                                    #stringLiteral
    ;

array_literal
    : '[' expression (',' expression)* ']'              #arrayLiteral
    | NEW type '(' expression ')'                       #newArray
    ;
```