CS315: Programming Languages

PROJECT - 2

A Boolean Language: RUSSELL

Group - 15
Anıl İlağa - 22002044
Zeynep Begüm Kara - 22003880

CS315 - 01

April 2, 2023

A. Language Design of RUSSELL

a. The Complete BNF Description

Constants

<LC> ::= “{”

<RC> ::= “}”

<LP> ::= “(”

<RP> ::= “)”

<LS> ::= “[”

<RS> ::= “]”

<new_line> ::= \n

<tab> ::= \t

<semicolon> ::= “;”

<comma> ::= “,”

<boolean> ::= true | false

<letter> ::= [a-zA-Z]

<digit> ::= [0-9]

<integer> ::= [0-9]+

<assign> ::= “=”

<and> ::= “&&”

<or> ::= “||”

<not> ::=   “!”

<eql> ::= “==”

<neq> ::=   “!=”

<imply> ::= “=>”

<iff> ::=   “⇔”

<dec_sym> :: “$bool”

<func_> ::= F:

<array_> ::= A:

<while> ::= while

<for_each> ::= for_each

<for_each_sym> ::= “@”

<return> ::= return

<break> ::= break

<if> ::= “if”

<elseif> ::= “elseif”

<else> ::= “else”

<in_> ::= :in:

```
<out_> ::= :out:
<io_sym> ::= "::"
<begin> ::= begin
<end> ::= end
<comment_sym> ::= "--"
```

Program

```
<program> ::= <begin><end> | <begin><stmnt_list><end>
             | <func_list><begin><end>
             | <func_list><begin><stmnt_list><end>
<stmnt_list> ::= <stmnt_list><stmnt>  | <stmnt>
<func_list> ::= <func_list> <func>  | <func>
```

Identifiers and Variables

```
<identifier> ::= <letter> | <identifier><letter>
<func_identifier> ::= <func_><identifier>
<array_identifier> ::= <array_><identifier>
<identifier_list> ::=  <identifier><comma><identifier_list>  | <identifier>
<boolean> ::= <true> | <false>
```

Operators

```
<binary_opr> ::= <and> | <or> |  <imply> | <iff>
```

Expressions

```
<logic_expr> ::= <compound_expr> | <atomic_expr>
<atomic_expr> ::= <identifier> | <array_element> | <boolean>
<compound_expr> ::= <LP><logic_expr><binary_opr><logic_expr><RP>
                  | <not><logic_expr >
```

Statements

```
<stmnt> ::=  <if_stmnt> | <loop_stmnt> | <assign_stmnt> | <io_stmnt>
          | <decl_stmnt> | <function_call> | <comment>  | <return_stmnt>
          | <break_stmnt>
```

Conditional Statements

<if_stmnt> ::=   <if><LP><logic_expr><RP><LC><stmnt_list><RC>
                 | <if><LP><logic_expr><RP><LC><stmnt_list><RC>
                   <else><LC><stmnt_list><RC>
                 | <if><LP> <logic_expr><RP><LC><stmnt_list><RC>
                   <elseif_list><else><LC><stmnt_list><RC>

<elseif_list> ::= <elseif><LP><logic_expr><RP><LC><stmnt_list><RC>
                 |<elseif_list><elseif><LP><logic_expr><RP><LC><stmnt_list><RC>

Declaration Statements

<decl_stmnt> ::= <var_decl> | <arr_decl> | <funct_decl>

<var_decl> ::= <dec_sym><identifier_list><semicolon>
               | <dec_sym> <identifier><assignment_op>
          <logic_expr><semicolon>
               | <dec_sym><identifier> <assignment_op>
               <function_call><semicolon>

<arr_decl> ::= <array_identifier> = <integer><semicolon>
               | <array_identifier> = <integer><LC><logic_expr_list> <RC>;

<function_signature> ::= <function_identifier><LP><parameter_list><RP>

<parameter_list> ::= <parameter> | <parameter><comma><parameter_list>

<parameter> ::= <logic_expr> | <array_identifier>

Assignment Statements

<assign_stmnt> ::= <var_assign_stmnt> | <arr_assign_stmnt>

<var_assign_stmnt> ::= <identifier> = <logic_expr><semicolon>
                       | <identifier><assignment_op><func_call>

<arr_assign_stmnt> ::= <array_element> = <logic_expr><semicolon>
                       |<array_identifier>=
                       <LC><logic_expr_list><RC><semicolon>

Loop Statements

```
<loop_stmnt> ::= <while_stmnt> | <for_each_stmnt>
<while_stmnt> ::= <while><LP><logic_expr><RP> <LC> <stmnt_list><RC>
<for_each_stmnt> ::= <for_each><LP><identifier><for_each_sym>
                     <array_identifier><RP> <LC> <stmnt_list><RC>
                    | <for_each><LP><identifier><for_each_sym>
                     <integer><LC><logic_expr_list><RC><RP> <LC>
                     <stmnt_list><RC>
```

Input / Output Statements

```
<io_stmnt> ::= <in_stmnt> | <out_stmnt>
<funct_decl> ::= <function_signature><LC><stmnt_list> <RC>
<in_stmnt> ::= <identifier><in_><semicolon>
<out_stmnt> ::= <out_><io_expr><semicolon>
               |<variable_out__stmnt>
<variable_out__stmnt> ::= <identifier><out_><semicolon>
<io_expr> ::= <io>  | <io> <io_sym> <io_expr>
<io> :: = <string> | <logic_expr>
<char> ::= <upper_case_char> | <lower_case_char>
<string> ::= “” | “<char>” | “<str><char>”
```

Return Statement

```
<return_stmnt> ::= <return> <logic_expr><semicolon>
<break_stmnt> ::= <break> <semicolon>
```

Function Calls

```
<func_call> ::= <function_signature><semicolon>
```

Data Structures

```
<logic_expr_list> ::=  <fill_with>
                    | <fill_with> , <logic_expr_terms>
                    | <logic_expr_terms> , <fill_with>
                    | <logic_expr_terms> , <fill_with> , <logic_expr_terms>
```

| <logic_expr_terms>

<logic_expr_terms> ::= <boolean> | <logic_expr_terms>, <boolean>

<fill_with> ::= …<boolean>

<array_element> ::= <array_identifier><LS><numeric><RS>

Comments

<comment> ::= <comment_sym><str><comment_sym>

b. List of Features and Explanations

Non Trivial Constants

```
<dec_sym> ::= $bool
```

Defined to indicate declaration of variables. Although only boolean type variables exist in the language , for readability purposes we have chosen the symbol $bool.

```
<func_> ::= F:
```

Defined to distinguish function identifiers, syntax has been chosen similar to common mathematical notation of functions.

```
<array_> ::= A:
```

Defined to distinguish array identifiers, syntax has been chosen for consistency with <func_>.

```
<for_each_sym> ::= @
```

Defined to indicate iterating element in for_each loop. *for_each (x @ A:arr)* means for each element at A:arr. Such syntax has been chosen to distinguish identifiers and array_identifiers.

```
<in_> ::= :in:
```

Defined to get input stream. Such syntax has been chosen for concatenation, and shortcuts. See i/o statements for further explanation.

```
<out_> ::= :out:
```

Defined to display output stream. Such syntax has been chosen for concatenation, and shortcuts. See i/o statements for further explanation.

```
<io_sym> ::= “::”
```

Defined to concatenate input and output streams, syntax has been chosen for consistency with <in_> and <out_>.

```
<comment_sym> ::= “--”
```

Defined to indicate beginning and end of comment statements, a simple syntax has been chosen for increasing writing ability.

Program

```
<program> ::= <begin><end>
            | <begin><stmnt_list><end>
            | <func_list><begin><end>
            | <func_list><begin><stmnt_list><end>
```

Start variable of this grammar is the non-terminal <program>. Statements to be executed must be declared inside curly braces. User-defined functions, if any, must be defined before this structure.

```
<stmnt_list> ::= <stmnt_list><stmnt> | <stmnt>
```

At least one <stmnt> defines a <stmnt_list>. <stmnt_list> cannot be empty. Observe that this is a left recursive definition. As a design choice, if both left and right recursion is possible, we will use left recursive grammars for consistency.

```
<func_list> ::= <func_list><func> | <func>
```

At least one <func> defines a <func_list>.

Identifiers and Variables

```
<identifier> ::= <letter> | <identifier><letter>
```

In this grammar, <identifier> is the name of a boolean variable, or simply a variable since there only exist boolean type variables. Identifiers can only contain letters and they cannot be empty. Even though convention in first order logic is to use a single uppercase letter to represent atomic propositions, this language allows users to have lowercase letters and identifier names longer than single letter. As a result, one can define as many as necessary propositions and give them meaningful names, which increases expressivity and readability where overall simplicity is decreasing.

```
<func_identifier> ::= <func_><identifier>
```

<func_identifier> is the name of a function. Function identifiers always begin with <func_>, i. e. "F:" and followed by an identifier. Having a restriction in function identifiers allows us to distinguish function identifiers from variable identifiers or array identifiers. Even though this decreases the writability and simplicity, it increases reliability by distinguishing those structures.

```
<array_identifier> ::= <array_><identifier>
```

<array_identifier> is the name of a data structure, array. Array identifiers always begin with <array_>, i. e. "A:" and followed by an identifier. Having a restriction in array identifiers allows us to distinguish array identifiers from variable identifiers or function identifiers. Even though this decreases the writability and simplicity, it increases reliability by distinguishing those structures.

```
<identifier_list> ::=  <identifier><comma> <identifier_list>
                    |  <identifier>
```

One <identifier> or consecutive <identifier>s separated by a comma defines a <identifier_list>. <identifier_list> cannot be empty.

```
      <boolean> ::= <true> | <false>
```

Boolean variables, which is the only type in RUSSELL language, has only two values: true or false.

Operators

```
       <binary_opr> ::= <and> | <or> | <imply> | <iff>
```

This language has binary operations *and, or, implication* and *if and only if.*

Expressions

```
      <logic_expr> ::= <compound_expr> | <atomic_expr>
```

Logical expressions can be classified as *compound expressions* and *atomic expressions.* In first-order logic, this distinction respectively corresponds to prepositions with <logic_opr> and prepositions with no <logic_opr>.

```
      <atomic_expr> ::= <identifier> | <array_element> | <boolean>
```

An atomic expression, which represents prepositions with no connectives, is either an identifier, element of an array or a truth value. In both cases they do not include any <logic_opr>.

```
<compound_expr> ::= <LP><logic_expr><binary_opr><logic_expr><RP>
                  | <not><logic_expr>
```

A compound expression is either a preposition with <binary_opr> that connects two logical expressions and such expressions are bounded by left, (, and right,), parentheses, or it is a negation of a proposition, whose truth value is opposite of the previous one. Observe that operation <not> does not require parentheses, which increases readability and writability.

Statements

```
<stmnt> ::= <if_stmnt> | <loop_stmnt> | <assign_stmnt> | <io_stmnt>
          | <decl_stmnt> | <function_call> | <comment>
          | <return_stmnt> | <break_stmnt>
```

A statement is either a conditional statement, or a loop statement, or an assignment statement, or an input/output statement or a declaration statement, or a function call or a comment or a return statement, or a break statement. For simplicity, we introduce <comment> as a statement. Note that a <comment> can be considered as a statement that does nothing.

Conditional Statements

```
<if_stmnt>::= <if><LP><logic_expr><RP><LC><stmnt_list><RC>
            |<if><LP><logic_expr><RP><LC><stmnt_list><RC>
             <else><LC><stmnt_list><RC>
            |<if><LP><logic_expr><RP><LC><stmnt_list><RC>
             <elseif_list><else><LC><stmnt_list><RC>
```

Any conditional statement, or as it generally called as an *if statement*, at least consists of <if> followed by an <logic_expr> enclosed by parentheses and an <stmnt_list> enclosed by curly braces. Therefore it is possible to write any number of statements inside those parentheses. A <if_stmnt> can also have a <else> block following the <RC> or it can also have an <elseif_list> between the first <RC> and <else>.

```
<elseif_list>::= <elseif><LP><logic_expr><RP><LC><stmnt_list><RC>
     | <elseif_list><elseif><LP><logic_expr><RP><LC><stmnt_list><RC>
```

This non-terminal is introduced to have a sequence of one or more <elseif> statements, each followed by a <logic_expr> and a block of <stmnt_list> enclosed in parentheses and curly braces, respectively. It provides additional conditions to check and reduces nested ifs or code repetitions, which is good for writability.

Declaration Statements

```
     <decl_stmnt> ::= <var_decl> | <arr_decl> | <funct_decl>
```

Declaration statements can be variable declarations, array declarations, or function declarations.

```
<var_decl> ::= <dec_sym><identifier_list><semicolon>
             | <dec_sym><identifier> =
<logic_expr><semicolon>
             | <dec_sym><identifier> = <func_call><semicolon>
```

In our language variables can be declared by just passing a declaration symbol $bool (since we don't have any other type) and identifier. Also, you can initialize the variable in the declaration statement.

```
<arr_decl> ::= <array_identifier> = <integer><semicolon>
             | <array_identifier> =
<integer><LC><logic_expr_list><RC><semicolon>
```

An array is declared by passing the <array_identifier> (A:<identifier>), assignment operator, and a numeric value that will be the size of the array.

```
<integer> ::= <integer><digit> | <digit>
```

Integer values consist of at least one digit and it is defined with left recursion.

```
<funct_decl> ::= <function_signature><LC><stmnt_list><RC>
```

A function is declared by a signature followed by statements which are inside curly braces.

```
<function_signature> ::= <function_identifier><LP><parameter_list><RP>
```

Function signature contains the identifier "F: functionName" and the parameters for that function inside braces.

```
<parameter_list> ::= <parameter><comma><parameter_list> | <parameter>
```

A parameter list is the plural form of a parameter followed by commas.

```
      <parameter>::= <logic_expr> | <array_identifier>
```

A function parameter can be a boolean or an array. On the contrary with C like languages, The language does not allow passing a function as parameters for reliability concerns.

Assignment Statements

```
<assign_stmnt> ::= <var_assign_stmnt> | <arr_assign_stmnt>
```

Two types of assignment statements are variable assignment (boolean) and array assignment.

```
<var_assign_stmnt> ::= <identifier> = <logic_expr><semicolon>
                     | <identifier> = <func_call>
```

Variables can be assigned by the variable name followed by the expression to be assigned.

```
<arr_assign_stmnt> ::= <array_element> = <logic_expr><semicolon>
      | <array_identifier> = <LC><logic_expr_list><RC><semicolon>
```

An element of an array can be changed by passing the array identifier and the expression.

Loop Statements

```
<loop_stmnt> ::= <while_stmnt> | <for_each_stmnt>
```

The RUSSEL language has two loop statements that are while loop and for each loop.

```
<while_stmnt> ::= <while><LP><logic_expr><RP><LC><stmnt_list><RC>
```

Statement type for while loops are similar to traditional while loops in commonly used languages to increase readability and writability purposes.

```
 <for_each_stmnt> ::= <for_each><LP><identifier><for_each_sym>
                      <array_identifier><RP><LC><stmnt_list><RC>
                     |<for_each><LP><identifier><for_each_sym>
<integer><LC><logic_expr_list><RC><RP><LC><stmnt_list><RC>
```

The language has a for loop statement to go through each element of an array. Again, syntax has chosen to be intuitively.

```
<break_stmnt> ::= <break><semicolon>
```

Break statement to get out of the loop. This statement can be used to avoid infinite loops.

Input/Output Statements

```
<io_stmnt> ::= <in_stmnt> | <out_stmnt>
```

An I/O statement is either an input statement or an output statement.

```
<in_stmnt> ::= <identifier><in_><semicolon>
```

As a design choice, this language only accepts only boolean type inputs. An input statement assigns user input to the identifier before the input indicator :in:. Even though it might decrease writability, we believe that it is a good practice to write reliable programs.

```
<out_stmnt> ::= <out_><io_expr><semicolon>|<variable_out_stmnt>
```

Output statements either begin with a constant ":out:" and might contain I/O expressions followed by a semicolon or it is a shortcut to display variables.

```
<variable_out_stmnt> ::= <identifier><out_><semicolon>
```

This statement is a shortcut to commonly used notation in displaying variables. It is called by an identifier :out: followed by a semicolon and it will print out the identifier and its value. To be more precise:

P :out: ;   prints   p = true

```
<io_expr> ::= <io>  | <io> <io_sym> <io_expr>
```

These expressions can contain just one expression or expressions followed by io constant "::". This concatenation property helps programmers to write output statements efficiently.

```
<io> :: = <string> | <logic_expr>
```

Either strings or expressions can be used in io expressions.

```
<string> ::= "<str><char>" | "<char>" | ""
```

String literals are strings surrounded by quotation marks.

```
<str> ::= <str><char> | <char>
```

Strings are chars following each other. We have distinguished strings and string literals to avoid nested quotation marks in recursive definitions.

Return Statement

```
<return_stmnt> ::= <return><logic_expr><semicolon>
```

Return statements can be used to return logic expressions.

Function Calls

```
<func_call> ::= <function_signature><semicolon>
```

Parameters are passed in the function signature to call a function.

Data Structures

```
<array_assign> ::= <array_identifier><assign><numeric><LC>
                   <logic_expr_list> <RC> <semicolon>
```

Assigning values inside an array by putting all the data inside curly braces.

```
<logic_expr_list> ::=  <fill_with>
       | <fill_with>, <logic_expr_terms>
       | <logic_expr_terms>, <fill_with>
       | <logic_expr_terms>, <fill_with>, <logic_expr_terms>
       | <logic_expr_terms>
```

A:myArray = {true, false, ...true, false, false, true, false};

A logical expression list may or may not contain a "<fill_with>" structure which means to assign the same value through the end of the array if no other value passed after or until the last few elements.

```
<logic_expr_terms> ::= <boolean> | <logic_expr_terms>, <boolean>
```

Logical expression terms are one boolean or booleans separated by commas.

```
<array_element> ::= <array_identifier><LS><numeric><RS>
```

Accessing a specific element of an array is done by passing the numeric value inside square brackets.

Comments

```
<comment> ::= <comment_sym><str>
```

Comments begin and end with a comment symbol which allows line comments or comment blocks.

B. Language Evaluation

RUSSELL programming language is designed to be used in applications of first-order logic. It is named after the memory of the famous logician, mathematician, and philosopher *Bertrand Russell* (1872 - 1970).

The crucial aspect of the language is that it only contains boolean types, true and false. Therefore, this language is designed to be very compact, yet quite ***expressive*** in terms of propositions. Note that every variable in this language can be considered as a proposition, whose truth value is either true or false. Recall that in first-order logic as well as being *atomic* statements, propositions might be *compound*, i.e. they might be combinations of atomic statements connected by logical operations. RUSSELL also allows these propositions. Even though it increases writability, for ***readability and reliability*** purposes, such propositions are bounded by left, (, and right,), parentheses.

Even though we know that each of the following logical operation sets {and, not}, {or, not}, {implication, not}, {nand (or as logicians called it "the Sheffer stroke")}, and {nor} is *expressively complete*, meaning that every compound proposition can be expressed by only using logical operations in one of these small sets, we have commonly used logical operations *and, or, iff,* and *imply* in RUSSELL for ***writability*** purposes. The purpose of our design choice is to avoid complex and long expressions. However, we do not include every operator such nand or nor in order not to decrease ***readability*** too much. Also, even though negation of a proposition forms a compound statement, our definition does not require parentheses in such statements to increase simplicity.

Although we know that every variable is a boolean, to distinguish declaration and assignment statements RUSSELL still requires the defined keyword *$bool* in declaration statements because of ***reliability*** concerns. Distinguishing declaration statements from assignment statements, RUSSELL checks if the variable is previously declared to protect misleading declarations in assignments. This design choice prevents users from bugs resulting from basic typos by throwing a syntax error. Additionally, the language doesn't have the *aliasing* feature, which may cause hard-to-detect bugs, as a design choice to have more reliable programs.

The control flow of the program can be changed using conditional statements, break statements in loops, while loop, and for each loop. The syntax of the loops is close to some well-known languages. Therefore, it increases readability, writability, and reliability. Ambiguity of *the dangling else problem* has been solved by introducing curly braces at the beginning and the end of the if-else blocks. Also introducing *else if,* it is aimed to reduce nested if-else statements, which increases writability. Also, "for each loop" gives the ability to the language to iterate over a collection of booleans, an array, easily. The language does not have the functionality of *go to, jump* or *jal* statements to ease the traceability of the program and enhance reliability.

As a data structure, the language provides a one dimensional array whose elements are boolean type. Because first-order logic is interested in the truth values of set propositions and their relations, we might have a large dataset. To efficiently store and retrieve these propositions, this language has an array structure. Recall that the retrieval operation is O(1) in arrays. Due to this structure, we can perform complex and long logical operations easily. Recall that elements of arrays might be only logical expressions, neither any array or a function. One might argue that this choice directly forbids multidimensional arrays, and damages orthogonality; this design choice resulted from reliability concerns. Additionally, since the propositions which will be given to the program might be a long list, instead of writing every value one by one, the programmer can assign one interval in the array as a specified value. To be more precise:

A:myArray = 10{true, true, false, false, …true}

which corresponds to

A:myArray = 10{true, true, false, false, true, true, true, true, true, true}

or another example might be

B:yourArray = 13{false, true, false, true, …false, true, true, true}

which corresponds to

B:yourArray = 13{false, true, false, true, false, false, false, false, false, false, true, true, true}

this notation also includes

C:theirArray = 14{...true}

C:theirArray = 14{ true, true, true, true, true, true, true, true, true, true, true, true, true, true }

This feature of the language increases the writability of it.

C. Revised and Augmented Language Design

The start variable of this language is the variable <russell> Following the language design conventions, it is the left-hand side of the first production rule in BNF description of the language. Also, it is specified in the Yacc as %start russel. A valid program is in the form of

```
<russell> ::= <begin><end>
            | <begin><stmnt_list><end>
            | <func_list><begin><end>
            | <func_list><begin><stmnt_list><end>
```

Execution of a program starts with the right side of the non-terminal <begin>, which corresponds token ***begin***, and ends with the right side of the non-terminal <begin>, which corresponds token ***end***.

Note that, since the language works with only boolean type variables in input statements, it is expected from the user that in such statements one should enter words "*true*" or "*false*". Naturally, the assigned variable takes one of the two boolean values. If this character sequence fails because of any typo, case sensitivity, or some reason, the language will accept any other input as true. Even though this feature may decrease reliability, we believe it makes the language more user-friendly.

It is only possible to declare boolean variables, functions, and arrays in this language. Function declarations and array declarations are designed to be *implicit declarations* whereas variable declarations are designed to be *explicit* and need the reserved word ***$bool*** for reliability purposes.

It is possible to declare functions before the reserved word ***begin*** which indicates the start of execution. As a design choice, the scope of such functions is determined as global, such functions can be called anywhere from between reserved words ***begin*** and ***end***. Also, they can be called from other global functions as long as the function being called has been defined before the function that calls. It is allowed to declare functions inside a block such functions, if blocks, for each block et., such functions are designed to be only visible inside that block. Similarly, one can define boolean variables or arrays in such blocks as well as between reserved words ***begin*** and ***end***. The scope of a variable or an array is determined as static scope.

The parameters of a function in this language can be either a logic expression or an array identifier. The language is designed to be passing by value if a parameter is a logic expression and pass by reference if a parameter is an array. The goal is to reduce memory consumption by avoiding the use of pass by value for arrays, which can be costly.

Operator evaluation order in expressions is determined by parentheses to ensure that the intended meaning of the logical expressions is preserved. Any compound expression that is made up of two simpler expressions joined together by a binary operator must be enclosed in parentheses. If a compound expression is the negation of a simpler expression, it doesn't have to be enclosed in parentheses. Also, atomic expressions do not require parentheses since there is no operation to indicate their precedence. To be more concrete here are some examples:

A	valid
(A)	invalid
!A	valid
(!A)	invalid
A && B	invalid
(A && B)	valid
!(A && B)	valid
!(A && B) => C	invalid
(!(A && B) => C)	valid

One might claim that introducing more precedence levels and associativity rules in logic operators such as between AND, OR might decrease the number of parentheses and increase writability. However, as a design choice, we keep our precedence levels limited. Therefore, we provided a rigorous syntax that allows people who study first-order logic from different backgrounds, which may have different precedence and associativity rules, to understand the same proposition. It is believed that this choice increases the maintainability of the language. Note that invalid expressions cause syntax errors to increase the reliability of the program. Even though it decreases writability, enforcing the use of parentheses for compound expressions can increase the readability and maintainability of the program in the long run, as it makes the code more explicit and easier to understand for other programmers. In addition, enforcing the use of parentheses can help catch potential mistakes and bugs early on in the development process, improving the overall reliability of the program.

D. Precedence, Ambiguity, and Conflicts

To define logical expressions unambiguously, our approach was to use parentheses rigorously, see part C for further explanation. The highest priority of logical operations is in parentheses and the second priority is in not operator. Other logic operators have the same precedence level and are defined as left-associative, but since we have parentheses for every two operands, we will not worry about associativity in binary logic expressions.

Initially in our rules, we faced 2 shift/reduce conflicts. One of them was caused by “logic_expr_list” and “logic_expr_terms” non-terminals. To resolve this conflict we used a precedence rule for the “comma”, claiming that it has higher precedence than the “fill_with”.

Additionally, the other conflict we came across was about a basic unambigiouty in our grammar. Where we had both the inclusive term and also the narrower one at the same BNF rule. This conflict was solved by deleting the narrowed rule.

Overall, these approaches have helped us to solve all our conflicts and end up with unambiguous language.

E. Non - Trivial Tokens

PROGRAM_BEGIN: Token to indicate starting to execute the program
PROGRAM_END: Token to indicate ending of the program.
IDENTIFIER : Token to represent identifiers.
FUNC_IDENTIFIER: Token to identify function identifiers
ARRAY_IDENTIFIER: Token to identify array identifiers
ARRAY_ELEMENT: Token to identify one element of the array
BOOLEAN : Token to represent boolean values.
ASSIGNMENT_OP: Token to represent assignment operator.
AND_OP: Token to represent logical AND operator.
OR_OP: Token to represent logical OR operator.
NOT_OP: Token to represent logical NOT operator.
EQL_OP: Token to represent logical EQUALITY operator.
NEQ_OP: Token to represent logical NOT EQUAL operator.
IMPLY_OP: Token to represent logical IMPLIES operator.
IFF_OP: Token to represent logical IFF operator.
VAR_TYPE: Token to represent variable declarations.
INPUT_SYMBOL: Token to take an input.
OUTPUT_SYMBOL: Token to print an output.
DOUBLE_COLUMN: Token to concatenate io expressions.
WHILE : Token to while loop.
BREAK: Token to break statement in loops.
RETURN : Token to return variables in functions.
FOR_EACH: Token to for_each loop.
FOR_EACH_SYM: Token to represent elements to iterate in for_each loop.
FILL_WITH : Token to partially fill an array.
IF: Token to indicate conditions in conditional statements.
ELSE: Token to indicate else in conditional statements.
ELSEIF: Token to indicate else if in conditional statements.
INTEGER : Token to represent array size.
COMMENT: Token to identify comments.

Logical operation tokens have been chosen with respect to commonly used notations in computer science and logic. Reserved words are defined in a very similar way in commonly used languages for readability and writability purposes. For detailed explanations see ***B.***