BILKENT UNIVERSITY CS 315 PROJECT REPORT

PART ONE

GRIFFIN LANGUAGE

Spring 2021-2022

Group Members:

Furkan Çalık 21802114 SEC-1

Berkay Çalmaz 21903245 SEC-1

The BNF Description of Griffin Language

1. Program:

<Program>:=<stmts>

<stmts> := <stmts> <stmt> <SC>

| <stmt> <SC>

| <stmt> <SC> <NL>

2. Statements

<stmt> := <dec_stmt> | <if_stmt> | <loop_stmt> | <func_stmt>

| <assign_stmt> | <return_stmt> | <stream_stmt>

| <set_dec_stmt> | <delete_stmt>

3. Declaration Statements

<dec_stmt> := <identifier> <COLON> <primitive_type> <ASSIGN_OP> <expression>

| <identifier> <COLON> <primitive_type>

<set_dec_stmt> := <identifier> <COLON> <set_identifier>

| <identifier> <COLON> <set_identifier>

<ASSIGN_OP> <set_expr>

| <identifier> <COLON><set_identifier><ASSIGN_OP><func_call>

4. Assignment Statements

<assign_stmt> := <identifier> <ASSIGN_OP> <expression>

<set_assign_stmt> := <identifier> <ASSIGN_OP> <set_expr_identifier>

5. Return Statements

<return_stmt> := | <return_identifier> <identifier>

| <return_identifier>

| <return_identifier> <expression>

| <return_identifier> <cond_expr>

| <return_identifier> <constant>

| <return_identifier> <set_constant>

| <return_identifer> <set_expr>

6. If Statements

<if_stmt> := <matched_if> | <unmatched_if>

<matched_if> := <if_identifier> <LP> <cond_expr> <RP> <LB> <matched_if> <RB>

else <LB> <matched>

| <if_block>

<unmatched_if> := <if_identifier> <LP> <cond_expr> <RP> <LB><if_block><RB>

| <if_identifier> <LP> <cond_expr> <RP> <LB> <matched_if>

<RB> <else_identifier> <LB> <unmatched_if> <RB>

<if_block> :=<dec_stmt> | <loop_stmt> | <func_stmt>

| <assign_stmt> | <return_stmt> | <stream_stmt>

| <set_dec_stmt> | <delete_stmt>

7. Loops

<loop_stmt> := <for_stmt> | <while_stmt>

<for_stmt> := <for_identifier> <LP> <dec_stmt> <SC> <cond_expr>

<SC> <arith_stmt> <RP> <LB><stmts> <RB>

<while_stmt> := <while_identifier> <LP> <cond_expr> <RP> <LB> <stmts> <RB>

8. Expressions

<expression> :=<arith_expr> | <func_call>

<cond_expr> := <set_cond_expr> | <logic_expr>| <arith_relational_expr> | <func_call> |

<equality_expr>

<equality_expr> := <equal_condition_elements>

| <equality_expr> <EQUALS> <equal_condition_elements>

<equal_condition_elements> := <arith_relational_expr> | <logic_expr>

| <arith_expr> | func_call | <boolean>

| <identifier> | <LP> <equality_expr> <RP>

<arith_relational_expr> := <arith_relational_elements>
<relational_op> <arith_relational_elements>

<arith_relational_elements> := <identifier> | <int> | <float> | <func_call>

<logic_expr> := <logic_expr_element>
| <logic_expr_element> <logic_op> <logic_expr>

<logic_expr_element> := <identifier> | <func_call> | <boolean>
| <LP> <logic_expr> <RP>

<set_expr> := <set_expr_identifier>
| <set_expr_identifier> <set_op><set_expr_identifier>
| <set_expr_identifier> <set_op> <set_expr>

<set_cond_expr> := <set_expr_identifiers> <dot> <set_cond_op>
<LP> <set_expr_identifier> <RP>

<set_expr_identifiers> := <identifier> | <set_constant> | <func_call>

<arith_expr> := <arith_expr_identifiers>
| <arith_expr_identifiers> <arith_op> < arith_expr>

<arith_expr_identifiers> := <identifier> | <int> | <float> | <LP> <arith_expr> <RP>

9. Variable Deletion

<delete_stmt> := <delete_identifier> <identifier>

10. Variables

<identifier> := <AT> <letter>
| <AT> <letter> <more>

<more> := <more> <letter>
| <more> <digit>
| <more> <under_score>

<set_constant> := <LB> <set_elements> <RB>

<set_elements> := <constant>
| <identifier>
| <constant> <COMMA> <set_elements>
| <identifier> <COMMA> <set_elements>

<constant> := <string> | <int> | <float> | <char> | <boolean>

11. Comments

<comment> := <comment_start> <char_list> <NL>

12. Operands and Built-In Functions

<set_op> := union | inter | diff | cross

<prim_set_func> := <getCardinality> | <getElement> | <addElement>
| <deleteElement> | <contains> | <isEmpty>

<arith_op> := <plus> | <minus> | <divide> | <mult> | <mod>

<logic_op> := <not_equal> | <and_relation> | <or_relation>

<relational_op> := <LT> | <GT> | <LTE> | <GTE> | <EQUALS>

<set_cond_op> := isSuperset | isSubset | isEqual | isEquivalent┼
| isOverlapping | isDisjoint

13. Types

<primitive_type> := string | int | float | char | boolean

<string> := <str_identifier> <char_list> <str_identifier>

<int> := <digit>
| <digit> <int>

<float> := <int> <dot> <int>

<char> := <char_identifier> <char_type_list> <char_identifier>

<char_list> := <char_type_list> | <letter> <char_list> | <digit> <char_list>
| <symbol> <char_list>

<char_type_list> := <digit> | <letter> | <symbol>

<boolean> := true | false

14. Function Declarations and Calls

<func_stmt> := <func_dec> | <func_call> | <prim_set_func_call>

<func_dec> := <func_identifier><identifier> <LP> <param_dec_list><RP>

<COLON><fuct_dec_primitive> <LB> <stmts> <RB>

<func_dec_primitive> := <primitive_type> | <set_identifier> | <void_identifier>

<param_dec_type> := <primitive_type> | <set_identifier> | <element_identifier>

<param_dec_list> := <empty> | <identifier> <COLON> <param_dec_type>

| <identifier><COLON> <param_dec_type> <COMMA>

<param_dec_type>

<constant_list> := <constant> | <set_constant>

<param_list> := <identifier> | <empty>

| <**constant_list**>

| <identifier> <COMMA> <param_list>

| <**constant_list**> <COMMA> <**constant_list**>

<func_call> := <func_call_identifier> <identifier> <LP> <param_list> <RP>

| <func_call_identifier> <prim_set_func> <LP> <param_list> <RP>

<prim_set_func_call> := <func_call_identifier> <identifier> <DOT>

<prim_set_func> <LP><param_list><RP>

| <func_call_identifier> <set_constant>

<DOT> <prim_set_func> <LP><param_list><RP>

15. Input-Output Formats

<stream_stmt> := <input_stmt> | <output_stmt>

<input_stmt> := <input_identifier> <LP> <input_from>

<RP> <input_stream> <identifier>

<input_from> := <identifier> | console | <file_constant>

<output_stmt> := <output_identifier> <LP> <output_to> <RP>

<output_stream> <output_from>

<output_to> := console | <identifier> | <file_constant>

<output_from> := <identifier> | <set_constant> | <constant>

<file_constant> := <file_identifier><LP> <string> <RP>

16. Terminals

<digit> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> := A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S

| T | U | V | W | Y | X | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n

| o | p | q | r | s | t | u | v | w | y | x | z

<if_identifier> := if

<for_identifier> := for

<return_identifier> := return

<char_identifier> := ‘

<string_identifier> := “

<else_identifier> := else

<input_identifier> := $>

<output_identifier> := <$

<func_call_identifier> := call

<element_identifier> := element

<void_identifier> := void

<plus> := +

<not_equal> := !=

<and_relation> := &&

<or_relation> := ||

<LT> := <

<GT> := >

<LTE> := <=

<GTE> := >=

<EQUALS> := ==

<func_identifier> := griffunc

<SC> := ;

<COLON> := :

<input_identifier> := griffin

<NL> := \n

<delete_identifier> := delete

<AT> := @

<set_identifier> := set

<input_stream> := $>

<output_stream> := <$

<input_identifier> := griffin

<output_identifier> := griffout

<file_identifier> := griffile

<under_score> := _

<empty> :=

<comment_start> := $$

<getElement> := getElement

<addElement> := addElement

<deleteElement> := deleteElement

Explanations of the Griffin Language Grammar

1. **<Program>:=<stmts>**

   This non-terminal marks the start of the program. Our program starts with any of the statements included in <stmts>.

2. **<stmts> := <stmts> <stmt> <SC>**

   | <stmt> <SC>

   | <stmt> <SC> <NL>

   This non-terminal shows the structure of our statements in a recursive manner. A statements ends with a semicolon represented as SC, or a semicolon followed by a new-line character NL.

3. **<stmt> := <dec_stmt> | <if_stmt> | <loop_stmt> | <func_stmt>**

   | <assign_stmt> | <return_stmt> | <stream_stmt>

   | <set_dec_stmt> | <delete_stmt>

   This non-terminal is the declaration of all possible statements in Griffin Language.

4. **<dec_stmt> := <identifier> <COLON> <primitive_type>**

   <ASSIGN_OP> <expression>

   | <identifier> <COLON> <primitive_type>

   This non-terminal is the grammar of a declaration statement. In order to declare a variable, its name/identifier is followed by a colon and its type (e.g. string, Boolean etc.). You can both initialize the declaration with an expression or just state its type.

5. **<set_dec_stmt> := <identifier> <COLON> <set_identifier>**

   | <identifier> <COLON> <set_identifier>

   <ASSIGN_OP> <set_expr>

   | <identifier> <COLON>

   <set_identifier> <ASSIGN_OP> <func_call>

   This non-primitive is similar to a primitive-type variable declaration. Since a set is not considered as a primitive-type in Griffin language, it has to have a different declaration.

6. **<assign_stmt> := <identifier> <ASSIGN_OP> <expression>**

   <set_assign_stmt> := <identifier> <ASSIGN_OP>

   <set_expr_identifier>

   These non-primitives are explained together because they represent the

same operation of assigning a value to a primitive-type or a set. It is used to assign values to variables after declaration.

7. **<loop_stmt> := <for_stmt> | <while_stmt>**

   This non-terminal is used to group together possible loops in the language.

8. **<for_stmt> := <for_identifier> <COLON> <LP> <dec_stmt> <SC> <cond_expr> <SC> <arith_stmt><LB><stmts> <RB>**

   This non-terminal explains the grammar of "for loops" in the Griffin language. After stating the for identifier and a colon, it allows the user to initialize a variable, create a condition and give an arithmetic statement inside the parentheses. Then, any statement can be put inside the brackets. As long as the given condition is still satisfied, the loop will continue.

9. **<while_stmt> := <while_identifier> <COLON> <LP> <cond_expr> <RP> <LB> <stmts> <RB>**

   This non-terminal explains the grammar of "for loops" in the Griffin language. After stating the while-identifier and a colon, a conditional expression is created inside the brackets. Afterward, any statement could be put into the brackets. As long as the given condition is still satisfied, the loop will continue.

10. **<expression> :=<arith_expr> | <func_call>**

    This non-terminal explains the base of any expression which is used in declarations or assignments. An expression could either be an arithmetic expression which consists of numeric calculations, or a function call.

11. **<cond_expr> := <cond_expr> := <set_cond_expr> | <logic_expr> | <arith_relational_expr> | <func_call> | <equality_expr>**

    This non-terminal contains basic conditional expressions in Griffin Language. The conditional expression is used to manage for and while loop lifecycle which can either continue or end the loop. The conditional expression can be set conditional expression, logic expression, arithmetic conditional expression, function call and equality expression.

12. **<equality_expr> := <contidition_elements> | <equality_expr> <EQUALS> <conditional_elements>**

    This non-terminal expression contains the main equality expression. Equality expression includes recursion. Equality expression checks if a condition expression at RHS equals a boolean expression at LHS with the notification "==".

13. **<equal_condition_elements> := <arith_relational_expr> | <logic_expr> | <arith_expr> | func_call | <boolean> | <identifier> | <LP> <equality_expr> <RP>**

This non-terminal expression includes the elements of the equals condition expression that can be arithmetic relational expression, logical expression, set expression, arithmetic expression, function call, boolean, identifier and itself equality expression. To use equality expression as an equality condition element, the equality expression can be put between left and right parenthesis.

14. **<arith_relational_expr> := <arith_relational_elements>**
    <relational_op> <arith_relational_elements>

    This expression includes arithmetic relation expressions such as greater than, less than etc. It cannot be recursive since the return is boolean and it cannot be compared with int, identifier, function call and float.

15. **<arith_relational_elements> := <identifier> | <int> | <float> | <func_call>**

    This expression includes the elements of the arithmetic relational expressions. The element can be identifier, int, float, function call and arithmetic relational expression.

16. **<logic_expr> := <logic_expr_element>**
    | <logic_expr_element> <logic_op> <logic_expr>

    This expression defines the syntax of logical relational expressions such as AND, OR etc. It can be recursive since it can take multiple booleans with logic operators between them

17. **<logic_expr_element> := <identifier> | <func_call> | <boolean> | <LP> <logic_expr> <RP>**

    This expression includes the elements of the logical expression. The elements of the logical expression can be identifier, function, boolean and logical expression. However, to use logical expression, the logical expression at RHS must be put between left and right parenthesis.

18. **<set_expr> := <set_expr_identifier>**
    | <set_expr_identifier> <set_op><set_expr_identifier>
    | <set_expr_identifier> <set_op> <set_expr>

    This expression explains the creation between sets by using the set operations such as union, diff, inter etc. The expression can be recursive so that it can be defined like A union B union C or A union (B union C).

19. **<set_cond_expr> := <set_expr_identifiers> <dot> <set_cond_op> <LP> <set_expr_identifier> <RP>**

    This non-terminal expression is used to recognize the conditional expression between two sets. The return type is boolean so that it can be used at the equal expression part. It can interact between other boolean type conditional expressions.

20. **<set_expr_identifiers> := <identifier> | <set_constant> | <func_call>**

This non-terminal expression includes the elements of the set conditional expression. The elements can be identifier, set constant, function call.

21. **<arith_expr> := <arith_expr_identifiers>**
    | <arith_expr_identifiers> <arith_op> < arith_expr>

    This non-terminal expression includes the expression for the arithmetic calculations. The arithmetic operators can be minus, plus, multiply, divider and mod.

22. **<arith_expr_identifiers> := <identifier> | <int> | <float> | <LP> <arith_expr> <RP>**

    This expression includes the elements of the arithmetic expression. The elements can be identifier, integer, float and arithmetic expression. To use arithmetic operations expression, the element must be used at between left and right parenthesis.

23. **<identifier> := <AT> <letter>**
    | <AT> <letter> <more>

    This non-terminal identifier is used to recognize variable and function names. An Identifier starts with the '@' sign and continues with a letter. Afterwards, it recursively constructs a name with 'more' which will be explained next.

24. **<more> := <more> <letter>**
    | <more> <digit>

    This non-terminal is used in identifiers to construct after an '@' sign and a letter. Then, it can include either a letter or a digit.

25. **<matched_if> := <if_identifier> <LP> <cond_expr> <RP> <LB>**
    <matched_if> <RB>
    else <LB> <matched_if>
    | <if_block>

    This non-terminal is used to remove the ambiguity in dangling-else problems in terms of braces. An "else" is matched with the closest preceding unmatched if statement. A matched-if statement, can either be an if-block or a matched if-else statement. This provides the necessary recursive if statements if needed.

26. **<unmatched_if> := <if_identifier> <LP> <cond_expr>**
    <RP> <LB><if_block><RB>
    | <if_identifier> <LP> <cond_expr> <RP> <LB> <matched_if>
    <RB> <else_identifier> <LB> <unmatched_if> <RB>

    This non-terminal is used to structure unmatched if-else statements. An unmatched-if statement either consists of a single if statement with an if-block inside or an if-else statement where the if statement is matched and else statement is

unmatched.

27. **<if_stmt> := <matched_if> | <unmatched_if>**

    This statement is used to create an unambiguous if-else structure. In the language, an if-statement consists of either a matched-if or an unmatched-if. Every matched-if structure matches an unmatched-if so that there is no dangling else.

28. **<if_block> :=<dec_stmt> | <loop_stmt> | <func_stmt>**
    | <assign_stmt> | <return_stmt> | <stream_stmt>
    | <set_dec_stmt> |<set_manipulat_stmt>| <delete_stmt>

    This non-terminal is used to help create any kind of statements in if or else structures. This is different than the "stmts" structure because it includes the if statements, which would corrupt the matched-unmatched balance.

29. **<return_stmt> := | <return_identifier> <identifier>**
    | <return_identifier>
    | <return_identifier> <expression>
    | <return_identifier> <cond_expr>
    | <return_identifier> <constant>
    | <return_identifier> <set_constant>
    | <return_identifer> <set_expr>

    This non-terminal is used to define the reserved word 'return' in functions. A function can return a variable with identifier, return empty for void functions, return expressions (simple arithmetic expression or a function call), return booleans by conditional expressions, return constants for either primitive types or for a set and finally return set expressions (e.g. union of sets).

30. **<delete_stmt> := <delete_identifier> <identifier>**

    This non-terminal is used to define how a variable memory is deallocated through the reserved word "delete" which is defined as delete-identifier.

31. **<set_constant> := <LB> <set_elements> <RB>**

    This non-terminal is used to define a set without declaring an identifier before hand. As an example, this is useful in function returns, parameter declarations etc. It includes set elements inside the brackets.

32. **<set_elements> := <constant>**
    | <constant> <COMMA> <set_elements>

    This non-terminal is used to structure the elements of a set. A set can include any kind of primitive-type variables. By recursion, it differentiates the elements with a comma.

33. **<constant> := <string> | <int> | <float> | <char> | <boolean>**

This non-terminal is used to define any kind of primitive types without declaring an identifier for it beforehand. As an example, this is useful in function returns, parameter declarations etc.

34. **<comment> := <comment_start> <char_list> <NL>**

    This non-terminal is the structure for comments in the language. In order to create a single-line comment, the comment-start terminal ($$) is used and afterwards, any possible characters in our language can be included until a newline character.

35. **<primitive_type> := string | int | float | char | boolean**

    This terminal explains the structure of every possible types expect sets.

36. **<string> := <str_identifier> <char_list> <str_identifier>**

    This non-terminal is the structure for the string type. A string is created by all possible characters in the language between the strings.

37. **<int> := <digit>**
    | <digit> <int>

    This non-terminal is the structure for the integer type. An integer is either a single digit or recursively any amount of integers.

38. **<float> := <int> <dot> <int>**

    This non-terminal is the structure for the float type. A float is defined by an integer followed by a dot and another integer.

39. **<char> := <char_identifier> <char_type_list> <char_identifier>**

    This non-terminal is the structure for the char type. A char is defined by the char-type-list (digit, symbol or letter) inside the char identifiers which is ‘’.

40. **<char_list> := <char_type_list> | <letter> <char_list> | <digit> <char_list>**
    | <symbol> <char_type_list>

    This non-terminal is used inside strings to specify what can be inside strings, which is any possible characters in the language. It can be a single character with char-type-list or recursively any amount of digits, letters, or symbols.

41. **<char_type_list> := <digit> | <letter> | <symbol>**

    This non-terminal is used to specify the possible single character types. It is used in char-list to create single characters.

42. **<set_op> := union | inter | diff | cross**

    This terminal defines the 4 operations related to sets. This operators will be explained in their own sections.

43. **<prim_set_func> := <getCardinality> | <getElement> | <addElement>**
    | <deleteElement> | <contains>

This non-terminal consists of multiple built-in set functions. Example usage is (set Identifier or set constant).prim_set_func(parameters). This usage increases both writability with included built in functions and readability with regular-language-like syntax.

44. **<arith_op> := <plus> | <minus> | <divide> | <mult> | <mod>**

    This non-terminal defines the operators to be used between the types of integer and floats.

45. **<logic_op> := <not_equal> | <and_relation> | <or_relation>**

    This non-terminal defines the operators to be used between boolean types which could be an identifier or a function call.

46. **<relational_op> := <LT> | <GT> | <LTE> | <GTE> | <EQUALS>**

    This non-terminal defines the relational operators between integers or floats and returns a boolean.

47. **<set_cond_op> := isSuperset | isSubset | isEqual | isEquivalent | isOverlapping | isDisjoint**

    This terminal shows the varying built-in functions in our language.

48. **<func_stmt> := <func_dec> | <func_call>**

    This non-terminal expression explains the function statement in Griffin Language. The function statement can be either a function declaration or a function call.

49. **<func_dec> := <func_identifier> <identifier> <LP> <param_dec_list><RP><COLON> <fuct_dec_primitive> <LB> <stmts> <RB>**

    This non-terminal expression demonstrates the declaration statement of a function. To declare a function, the function identifier which is “griffunc” must be used. After the function identifier, the function name must be started with “@” and followed with a letter. Then, the parameter list must be added. The parameter list will be explained later. Then, the statements can be written at the curly brackets after function signature.

50. **<func_dec_primitive> := <primitive_type> | <set_identifier> | <void_identifier>**

    This expression includes the return type of the function which is indicated at the declaration of the function. The function return type can be primitive type, set and void which returns nothing.

51. **<param_dec_type> := <primitive_type> | <set_identifier>| <element_identifier>**

    This expression includes the variable types of the elements of the parameter list. An element is any type of variable that can be a member of a set.

52. **<param_dec_list> := <empty> |<identifier> <COLON> <param_dec_type> | <identifier><COLON> <param_dec_type> <COMMA> <param_dec_type>**

This expression includes the structure of the parameter list. The param declaration list might be empty, only one parameter or includes more than one parameter. Each parameter is separated by comma and the type of the param is written after the identifier by separated between them with colon.

53. **<constant_list> := <constant> | <set_constant>**

The constant list includes all possible constant created variables. The purpose is to decrease the complexity of the param_list grammer.

54. **<param_list> := <empty> | <identifier>**
| <constant_list>
| <identifier> <COMMA> <param_list>
| <constant_list> <COMMA> <param_list>

This non-terminal includes the function parameter list structure. Parameters can be empty, an identifier, a constant or recursively multiple of them separated with a comma.

55. **<func_call> := <func_call_identifier> <identifier> <LP> <param_list> <RP>**
| <func_call_identifier> <prim_set_func> <LP> <param_list> <RP>

This non-terminal explains the syntax of the function call. Function call includes function call identifier which is “call”. After and identifier, the parameter list between parentheses. This structure increases readability with the “call” syntax since it is obvious where the function is called.

56. **<prim_set_func_call> := <func_call_identifier> <identifier> <DOT>**
<prim_set_func> <LP><param_list><RP>
| <func_call_identifier> <set_constant>
<DOT> <prim_set_func> <LP><param_list><RP>

This non-terminal explains how the built-in primitive set functions are called. After the function-call identifier “call”, either an identifier or a set constant is given, then a dot with parameters list inside the parentheses. This improves the writability of functions since it gives a variety of functions for the set elements.

57. **<getElement> := getElement**

This terminal is the built-in function getElement. Inside the parameters list, it takes an integer and returns the set element at that index. If the integer is larger than the size of the set, an error will be returned, which improves the reliability.

58. **<addElement> := addElement**

This terminal is the built-in function addElement. If the function is only given the element to be added, it pushes the element to the end of the set. However if both an element

and an index is given, it adds the element to the given index. This usage increases writabiliy yet reduces readability.

59. **<deleteElement> := deleteElement**

    This terminal is the built-in function deleteElement that deletes a given element inside the parameter list. If the given parameter does not exist in the set or cannot be an element of the set, it returns an error. This improves the reliability of the language.

60. **<isEmpty> := isEmpty**

    This terminal is the built-in function for sets which returns true if the set is empty, false if not.

61. **<contains> := contains**

    This terminal is the built-in function contains which returns a boolean on whether or not a set includes an element.

62. **<stream_stmt> := <input_stmt> | <output_stmt>**

    This expression includes the statement of the stream operations. The stream operations can be input or output operations.

63. **<input_stmt> := <input_identifier> <LP> <input_from> <RP> <input_stream> <identifier>**

    This statement includes the input statement grammar. The input statement starts with an input identifier which is “griffin”. The source is defined inside the parentheses as either a file address or the reserved word console. After the input identifier “$>”, the variable that will take the input.

64. **<input_from> := <identifier> | console | <file_constant>**

    This expression shows what the input source might be. It might be coming from an identifier, the console or a file.

65. **<output_stmt> := <output_identifier> <LP> <output_to> <RP> <output_stream> <output_from>**

    This expression defines an output stream statement grammar. The output stream statement should be started with an output identifier that is “griffout”. After words the target should be declared between parentheses. Then, the output stream operator should be written as “<$”. Finally, the source should be identified.

66. **<output_to> := console | <identifier> | <file_constant>**

    This expression shows what the output target might be. It might be a console, identifier or file constant.

67. **<output_from> := <identifier> | <set_constant> | <constant>**

    This expression shows what the output source migTht be. It might be an identifier, set constant or other constant types such as string, int, float etc..

68. **<file_constant> := <file_identifier><LP> <string> <RP>**

    This expression shows the grammar of the file constant. The file constant includes a file identifier and a string as a file name. The file identifier is “griffile”. Then the file name should be declared as a string between parentheses.

69. **<symbol> := <LP> | <RP> | <FN_BEGIN> | <SC>**

    | <DOT> | <COLON> | <LB> | <RB> | <COMMA> | <ASSIGN_OP>

    This non-terminal defines the possible non-letter characters in the language.

Reserved Words and Terminals

1. **<digit> := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

   This terminal is the base structure for integers and floats in the language.

2. **<letter> := A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S**

   | T | U | V | W | Y | X | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n

   | o | p | q | r | s | t | u | v | w | y | x | z

   This terminal is the base structure for strings and characters in the language.

3. **<if_identifier> := if**

   This terminal is the reserved word for the if statements.

4. **<else_identifier> := else**

   This terminal is the reserved word for the else statements.

5. **<for_identifier> := for**

   This terminal is the reserved word for the for loops.

6. **<return_identifier> := return**

   This terminal is the reserved word for the function returns.

7. **<char_identifier> := ‘**

   This terminal is the reserved word for the start and end of characters.

8. **<string_identifier> := “**

   This terminal is the reserved word for the start and end of strings.

9. **<input_identifier> := $>**

   This terminal is the reserved word for the input streams.

10. **<output_identifier> := <$**

    This terminal is the reserved word for the output streams.

11. **<func_call_identifier> := call**

    This terminal is reserved in order to identify function calls.

12. **<plus> := +**

This terminal is reserved to arithmetic plus operation.

13. **<not_equal> := !=**

    This terminal is reserved for arithmetic comparison.

14. **<and_relation> := &&**

    This terminal is reserved for 2 or more boolean comparisons which return the "and" of all booleans.

15. **<or_relation> := ||**

    This terminal is reserved for 2 or more boolean comparisons which return the "or" of all booleans.

16. **<LT> := <**

    This terminal is reserved for arithmetic less than comparisons which returns a boolean.

17. **<GT> := >**

    This terminal is reserved for arithmetic less than comparisons which returns a boolean.

18. **<LTE> := <=**

    This terminal is reserved for arithmetic "less than or equal" comparisons which returns a boolean.

19. **<GTE> := >=**

    This terminal is reserved for arithmetic "greater than or equal" comparisons which returns a boolean.

20. **<EQUALS> := ==**

    This terminal is used to determine equality in conditional expressions.

21. **<func_identifier> := griffunc**

    This terminal is a reserved word that is used at the start of a function declaration.

22. **<SC> := ;**

    This terminal marks the end of every statement.

23. **<COLON> := :**

    This terminal is used in variable or function declarations.

24. **<input_identifier> := griffin**

    This terminal is used to declare an input stream.

25. **<output_identifier> := griffout**

    This terminal is used to declare an output stream.

26. **<input_stream> := $>**

This terminal is used to create continuous input streams. It is used between inputs.

27. **<output_stream> := <$**

    This terminal is used to create continuous output streams. It is used between outputs.

28. **<NL> := \n**

    This terminal show the endline character.

29. **<delete_identifier> := delete**

    This terminal is a reserved word that is used for memory deallocations of variables.

30. **<AT> := @**

    This terminal is used at the start of identifiers to distinguish them.

31. **<set_identifier> := set**

    This terminal is used when declaring a set.

32. **<file_identifier> := griffile**

    This terminal is used when defining a file.

33. **<under_score> := _**

    This terminal is another possible symbol in the language.

34. **<empty> :=**

    This terminal represents the empty sections in the language. It is declared so that the BNF representation is more clear.

35. **<comment_start> := $$**

    This terminal is used to declare the start of a single line comment that ends with a newline character.

Evaluation of the Griffin Language

1. Readability

Since the Griffin Language is designed for sets, other operations are kept limited. For example, in arithmetic expressions, no recursive operations are allowed. The language uses brackets and parentheses in multiple places which could create hardship in reading with bad indentation.

The language has orthogonal features. For example, any type of a variable or a constant can be passed in functions and can be returned. The only exception is that set variables are different then primitive types and has different syntaxes, which is inevitable since sets cannot have sets as elements. Overall, this makes our language easy to learn and read.

2. Writability

Writability is increased with merging variable types other than sets to primitive_types. Moreover, set expressions have mostly similar syntax with other variable types, which lets the user learn and write easier. Identifiers for variables and functions have the same syntax which is also another plus. For and While loops are included which increases the expressivity. All built in functions are written similar to regular language constructs which makes writing more intuitive. However, not all functions can be used for every type which is the trade-off of having high readability.

3. Reliability

Our language is built with variable type declarations that are also stated in parameter declarations and function return types. This feature could be used to detect wrong variable uses in compile time, which is a huge increase in reliability. It does not have any aliasing which is beneficial to the reliability. The recursive grammars are either defined as left-recursive or right-recursive so that ambiguity is prevented. However, it was not the main priority and there could be exceptions for the sake of readability. Moreover, exception handling possibilities were not considered and could be hard to do.

Most of the aspect of reliability is not considered while constructing the grammar of the Griffin language.