CS315- Programming Languages

Project 1 Report

Programming Language: CodeAD

İlke Kaş 21803184 Section 1
Bilgehan Akcan 21802901 Section 1

Contents

BNF Description

Program

<program>::= start:<statements>end

<statements>::= <statement> | <statements><statement>

<statement>::= <if_stmt> | <nonif_stmt>

<nonif_stmt>::= <comment> | <expressions><semicolon> | <loop_stmt> |
<function_definitions> | <declaration_initialization_stmts> |
<element_assignment> | <assign_stmt> | <input_stmt> | <output_stmt> |
<function_call> | <return_stmt> | <break_stmt>| <primitive_system_statements>

Declarations and Initialization

<declaration_initialization_stmts>::= <ident_declaration> | <ident_initialization>
| <array_declaration> | <array_initilization>

<ident_declaration>::= <type_names><space><identifier><semicolon>

<ident_initialization>::= (double | int) <space><identifier><assign> (<arithmetic_expression> |
<identifier><LSB><int><RSB> | <function_call>) <semicolon> |
bool<space><identifier><assign> (<logical_expression> |
<identifier><LSB><int><RSB> | <function_call>) <semicolon> |
string<space><identifier><assign> (<string> |
<identifier><LSB><int><RSB> | <function_call>) <semicolon> |
char<space><identifier><assign> (<char>
| <identifier><LSB><int><RSB> | <function_call>) <semicolon> |
Map<space><identifier><assign> (<create_map> | <identifier> |
<function_call>) <semicolon> |

Time<space><identifier><assign> (<get_time> | <identifier> | <Time> |

<function_call>) <semicolon> |

Direction<space><identifier><assign> (<identifier> | <Direction> |

<function_call>) <semicolon>

<assign_stmt> ::= <identifier><assign> (<expressions> | <types> |

<identifier><LSB><int><RSB> | <function_call> | <assign_stmt>)

<semicolon> | <sub_assign>

<sub_assign> ::= <identifier><other_assign_op> (<arithmetic_expressions> | <identifier> |

<arit_const> | <identifier><LSB><int><RSB> | <function_call>) <semicolon>

Types

<types>::=<string> | <bool> | <arit_const> | <char> | <Time> | <Direction>

<arit_const>::= <double> | <int>

<Direction>::=North | East | West | South

<Time>::= <digits><digits><colon><digits><digits><colon><digits><digits>

<double>::= <number><dot><number> | <sign><number><dot><number>

<int>::= <number> | <sign><number>

<number>::= <zero> | <nonzero_decimal_num>

<nonzero_decimal_num>::= <nonzero_digits> | <decimal_num><digits>

<char> ::= '<char_body>'

<char_body>::= <char_all> | <digits> | <symbols>

<char_all>::= <uppercase_char> | <lowercase_char>

<string>::= "<string_body>"

<string_body>::= <char_body> | <string_body><char_body>

<bool>::= true | false

Variable Identifiers

<identifier>::= <char_all> | <identifier><digits> | <identifier><char_all> |

<identifier><underscore>

Operators

<other_assign_op>::= <mult_and_assign> | <plus_and_assign> | <minus_and_assign> |

<div_and_assign>

<increment_op>::= ++

<decrement_op>::= --

Expressions

- **Arithmetic**

<arithmetic_expression> ::= <arithmetic_expression> + <arit_term>

| <arithmetic_expression> - <arit_term> | <arit_term>

<arit_term> ::= <arit_term> *<arit_factor> | <arit_term> / <arit_factor>

| <arit_term> % <arit_factor> | <arit_factor>

<arit_factor> ::= <arit_exp> ** <arit_factor> | <arit_exp>

<arit_exp> ::= <LP><arithmetic_expression><RP> | <arit_oper>

<arit_oper> ::= <identifier><increment_op> | <identifier><decrement_op>

|<identifier> | <arit_const> | <Time>

- **Relational**

<relational_expression>::= <arithmetic_expression> < <arithmetic_expression>

| <arithmetic_expression> > <arithmetic_expression>

| <arithmetic_expression> <= <arithmetic_expression>

| <arithmetic_expression> >= <arithmetic_expression>

| <relat_term>

<relat_term>::= <arithmetic_expression>==<arithmetic_expression>

| <arithmetic_expression>!= <arithmetic_expression>

| <relat_factor>

<relat_factor>::= <LSB><relational_expression><RSB>

- **Logical**

<logical_expression>::= <logical_expression><or><logic_term> | <logic_term>

<logic_term>::= <logic_term><and><logic_factor> | <logic_factor>

<logic_factor>::= <not><logic_factor> | <logic_relation>

<logic_relation>::= <relational_expression> | <logic_exp>

<logic_exp>::= <LP><logical_expression><RP> | <logic_oper>

<logic_oper>::= <bool> | <identifier>

- **Combination**

<expressions>::= <arithmetic_expressions> | <logical_expressions> | <other_expressions>

<other_expressions>::= <relational_expressions>

Loops

<loop_stmt> ::= <while_loop> | <for_loop>

<while_loop> ::= while<LP><logical_expression><RP><LCB><statements><RCB>

<for_loop> ::= for<LP><ident_initialization><logical_expression><semicolon>

<arithmetic_expression><RP> <LCB> <statements> <RCB>

<break_stmt> ::= break<semicolon>

Conditional Statements

<if_stmt> ::= <matched> | <unmatched>

<matched> ::= if <LP><logical_expression><RP><LCB><matched><RCB>else<LCB>

<matched><RCB> | <nonif_stmt>

<unmatched> ::= if<LP><logical_expression><RP><LCB><if_stmt><RCB>

| if<LP><logical_expression><RP><LCB><matched><RCB>

else<LCB><unmatched><RCB>

Comment

<comment> ::= <single_comment> | <multiline_comment>

<multiline_comment> ::= <multiline_comment_begins><string_body>

<multiline_comment_ends>

Arrays

<array_declaration> ::= <type_names><space><identifier><LSB><int><RSB><semicolon>

<array_initilization> ::= double<space><identifier><LSB><int><RSB><assign><LCB>

(<space> | <array_init_content_double>) <RCB><semicolon>

| int<space><identifier><LSB><int><RSB><assign><LCB> (<space |

<array_init_content_int>) <RCB><semicolon>

| bool<space><identifier><LSB><int><RSB><assign><LCB> (<space>

| <array_init_content_bool>) <RCB><semicolon>

| string<space><identifier><LSB><int><RSB><assign><LCB>

(<space> | <array_init_content_string>) <RCB><semicolon>

| char<space><identifier><LSB><int><RSB><assign><LCB> (<space>

| <array_init_content_char>) <RCB><semicolon>

<array_init_content_int> ::= <arithmetic_expression> | <int> | <identifier>

| <array_init_content_int><comma> (<arithmetic_expression> |
<int> | <identifier>)

<array_init_content_double> ::= <arithmetic_expression> | <double> | <identifier> |
<array_init_content_double><comma>
(<arithmetic_expression> | <double> | <identifier>)

<array_init_content_bool> ::= <logical_expression> | <bool> | <identifier>
| <array_init_content_bool><comma> (<logical_expression> |
<bool> | <identifier>)

<array_init_content_string> ::= <string> | <identifier> | <array_init_content_string> <comma>
(<string> | <identifier>)

<array_init_content_char> ::= <char> | <identifier> | <array_init_content_char> <comma>
(<char> | <identifier>)

<element_assignment>::= <identifier><LSB><int><RSB><assign> (<expressions>|<types>)
<semicolon>

Statements for input / output

<input_stmt>::=in<LP><identifier><RP><semicolon>

<output_stmt>::=out<LP><output_content><RP><semicolon>

<output_content>::= <identifier> | <expressions> | <types> | <output_content><comma>
(<types> | <identifier)

Function definitions and function calls

<function_definitions> ::= method<space><return_types><space><identifiers><LP>
[<parameters>] <RP><LCB><statements><RCB>

<parameters>::=<type_names><space><identifier> |

<type_names><space><identifier><LSB><RSB>

| <parameters><comma>

(<type_names><space><identifier>|<type_names><space><identifier><LSB><RSB>)

<arguments> ::= <identifier> | <expression> | <types>

| <arguments><comma>(<identifier> | <expression> | <types>)

<return_types>::= <type_names> | none | <type_names><LSB><RSB>

<function_call>::=<identifier><LP><arguments><RP><semicolon>

<return_stmt>::=return<space> (<expressions> | <identifier> | <types> |

<identifier><LSB><int><RSB> | none) <semicolon> |

return<space><function_call>

Constants

<lowercase_char> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

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

W | Y | Z

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

<zero> ::= 0

<digits> ::= <zero> | <nonzero_digits>

<LP>::= (

<RP>::=)

<LCB>::= {

<RCB>::= }

```
<LSB>::= [

<RSB>::= ]

<mult_op> ::= *

<div_op> ::= /

<mod_op> ::= %

<QM> ::= “

<question_mark> ::= ?

<column> ::= :

<comma>::= ,

<semicolon>::= ;

<underscore>::= _

<space> ::= “ ”

<not> ::= !

<assign> ::= =

<sign>::= + | -

<newline>::= \n

<dot>::= .

<or>::= |

<and>::=&

<single_comment>::= //

<multiline_comment_begins>::= /!

<multiline_comment_ends> ::= !/

<mult_and_assign>::= *=
```

<plus_and_assign>::= +=

<minus_and_assign>::= -=

<div_and_assign>::= /=

<equal_op> ::= ==

<not_equal_op> ::= !=

<less_than_op>::= <

<greater_than_op>::= >

<less_than_equal_op>::= <=

<greater_than_equal_op>::= >=

<symbols> ::= <LP> | <RP> | <LCB> | <RCB> | <LSB> | <RSB> | <comma> | <semicolon> |

<underscore> | <space> | <sign> | <newline> | <dot> | <equal> | <not> | <or> |

<and> | <mult_op> | <div_op> | <mod_op> | <less_than_op> |

<greater_than_op> | <QM> | <question_mark> | <column>

<type_names>::= int | double | bool | char | string | Map | Time | Direction

\System Functions

<primitive_system_statements>::= <create_map> | <add_chapter> | <connect_chapters>

| <add_item> | <get_pcontent> | <get_time>

| <is_connected> | <move>| <is_available>

<create_map>::=createMap(<arguments>);

<add_chapter>::= addChapter(<arguments>);

<connect_chapters>::= connectChapters(<arguments>);

<add_item>::= addItem(<arguments>);

<get_content>::=getContent(<arguments>);

<get_time>::= getTime();

<is_connected> ::= isConnected(<arguments>);

<use_tool>::= useTool(<arguments>);

<move>::= move(<arguments>);

<is_available>::= isAvailable(<arguments>);

Explanations of CodeAD Constructs

- **Program Structure**

1. **<program>::= start:<statements>end**

   This nonterminal is used to state that in CodeAD language, our program starts with a "start" reserved word followed by a colon and ends with an "end" reserved word. There are statements of the program between these two words.

2. **<statements>::= <statement> | <statements><statement>**

   This nonterminal makes the consecutive statements valid by using left recursion. Thanks to left recursion, statements of the program are read in sequential order.

3. **<statement>::= <if_stmt> | <nonif_stmt>**

   This nonterminal states that statements can be if statements or statements other than if statements. Intended usage of this nonterminal is to make it easy to define matched conditional statements like this:

   <matched> ::= if <LP> <logical_expression> <RP> <LCB> <matched> <RCB> else <LCB> <matched> <RCB>| <nonif_stmt>

4. **<nonif_stmt>::= <comment> | <expressions><semicolon> | <loop_stmt> |
   <function_definitions> | <declaration_initialization_stmts> |
   <element_assignment> | <assign_stmt> | <input_stmt> | <output_stmt> |
   <function_call> | <return_stmt> | <brak_stmt> |
   <primitive_system_statements>**

   This nonterminal gathers all other statements and function calls, function definitions etc. under a single roof to connect them to the program.

- **Declarations and Initialization**

5. **declaration_initialization_stmts>::= <ident_declaration> | <ident_initialization>**

   | <array_declaration>|<array_initilization>

   This nonterminal gathers all other declarations and initializations including arrays, under a single nonterminal to make it easier to connect them to **<statement>** nonterminal.

6. **<ident_declaration>::= <type_names><space><identifier><semicolon>**

   This nonterminal is necessary to declare variables by specifying their types and names. Variables with any kind of types can be declared by using this nonterminal. However, there is a special case for **Time** variables. They are also declared fittingly to this nonterminal but their initialization is done automatically when it is declared.

7. **<ident_initialization>::= (double | int) <space><identifier><assign>**

   (<arithmetic_expression> | <identifier><LSB><int><RSB> |

   <function_call>) <semicolon> | bool<space><identifier><assign>

   (<logical_expression><identifier><LSB><int><RSB> |

   <function_call>) <semicolon> | string<space><identifier><assign>

   (<string> | <identifier><LSB><int><RSB> | <function_call>)

   <semicolon> | char<space><identifier><assign> (<char> |

   <identifier><LSB><int><RSB> | <function_call>) <semicolon>

   | Map<space><identifier><assign> (<create_map> | <identifier> |

   <function_call>) | Time<space><identifier><assign> (<get_time> |

   <identifier> | <Time> | <function_call>)

   | Direction<space><identifier><assign> (<identifier> |

   <Direction> | <function_call>)

   This nonterminal defines the form of the statements that makes both initalizations and declarations in detail. Variables whose types are **double** and **int** can be initialized with array elements with specific index, return values of the function calls, arithmetic expressions that can define standalone identifiers, integer and double literals. Variables that have **boolean** type can be initialized with array elements with specific index, return values of the function calls and the logical expressions which can make reference to relational expressions. **string** variables can be initialized with array elements, return values of function calls and string literals and similarly **char** variables can be initialized with array elements with specific index, return values of the function calls and character literals. In the same manner, return values of function calls and identifiers can be used in the initialization of variables with **Map, Time** and **Direction** types. **Map** variables can also be initialized with the return value of createMap() function. Similarly, **Time** variables can be initialized with getTime() function and time literals. Since CodeAD enables many possible combinations for initialization, it provides **orthogonality.**

8. **<assign_stmt> ::= <identifier><assign> (<expressions> | <types>|**

   <identifier><LSB><int><RSB> | <function_call> | <assign_stmt>)

   <semicolon> | <sub_assign>

   This nonterminal will define the form of assignment statements. In CodeAD, developers can assign expressions, literals, array elements and return values of the function calls to identifiers. Notice that this nonterminal is defined right recursively. That means in scenarios like this: a= b = c = 0, The operation order will be the same as (a = (b = (c=0))). Also, **<sub_assign>** is used to define precedence of other kinds of assignments.

9. **<sub_assign> ::= <identifier><other_assign_op> (<arithmetic_expressions> | <identifier> |**

   <arit_const> | <identifier><LSB><int><RSB> | <function_call>)

   <semicolon>

   This nonterminal will define "+=, -=, *= and /=" assignments. It has higher precedence than normal assignment statements. This is an example of **feature multiplicity.** Although it decreases the overall simplicity, hence readability, it increases the **expressivity** and writability.

- **Conditional Statements**

10. **<if_stmt> ::= <matched> | <unmatched>**

    This nonterminal combines two kinds of if statements **<matched>** and **<unmatched>.**

11. **<matched> ::= if<LP><logical_expression><RP><LCB><matched><RCB>else<LCB>**

    <matched><RCB> | <nonif_stmt>

    The intended use of this nonterminal is to separate if/else statements from if statements without else. Matched if statements consist of pairs of if and else. In other words, it contains equal numbers of if and else. Notice that it is defined as recursive. That means it enables nested if/else conditionals. However, matched nonterminals should also contain non-if statements. Otherwise, developers can't write any other statements such as declarations, expressions etc.

12. <unmatched> ::= if<LP><logical_expression><RP><LCB><if_stmt><RCB>

| if<LP><logical_expression><RP><LCB><matched><RCB>

else<LCB><unmatched><RCB>

The intended use of this nonterminal is to separate if statements without else from if/else pairs. Notice that it is defined as recursive. That means it enables nested unmatched if statements. In this terminal, the total number of if's are greater than else's.

- **Statements for input / output**

13. <input_stmt>::=in<LP><identifier><RP><semicolon>

This nonterminal states the form of input statements. In CodeAD, "in" is used to get input from the user. It reads all characters until the newline character. This statement does not return any type, literal or variable. Therefore, to hold the value of the input in variable, we write the variable name(identifier) between the parentheses. This statement enables interaction between developer and user of the program.

14. <output_stmt>::=out<LP><output_content><RP><semicolon>

This nonterminal states the form of output statements. In CodeAD, "out" is used to print the output contents which are defined below. Desired output elements are written between the parentheses. This statement also enables interaction between developer and user of the program.

15. <output_content>::= <identifier> | <types> | <expressions>

| <output_content><comma> (<types> | <identifier>)

This nonterminal helps the output statement by specifying the contents of the output statement's body. Developers can write identifiers, literals and expressions as output content. If there are multiple discrete contents to be printed, they are separated from each other by comma(,).

- **Expressions**

Arithmetic

16. <arithmetic_expression> ::= <arithmetic_expression> + <arit_term>

| <arithmetic_expression> - <arit_term>

| <arit_term>

This nonterminal is the root level of the arithmetic expressions. Therefore, it indicates the operations with lower precedence than other operators. So, saying that plus and minus operators have the lowest precedence would not be wrong. Besides that this nonterminal is defined as left recursive. In other words, operation order will be from left to right among the operators of this level. **<arit_term>** nonterminal is used to define higher precedences.

17. <arit_term> ::= <arit_term> * <arit_factor> | <arit_term> / <arit_factor>

| <arit_term> % <arit_factor> | <arit_factor>

The intended use of this nonterminal is to enable higher precedence for *, / and % operators than plus and minus operators. Also, this nonterminal is defined as left recursive. Again, operation order will be from left to right among the operators of this level. **<arit_factor>** nonterminal is used to define higher precedence than multiplication and division.

18. <arit_factor> ::= <arit_exp> ** <arit_factor> | <arit_exp>

This one is used for describing power operators whose precedence is higher than the operators of four operations. It is important to notice that this nonterminal is not defined as left recursive. Because power operation is a right recursive operation. As an operator, CodeAD uses ** symbol.

19. <arit_exp> ::= <LP><arithmetic_expression><RP> | <arit_oper>

The precedence of parentheses in arithmetic expressions are higher than any binary operators in arithmetic expressions. Notice that between the parentheses, there can be again any type of arithmetic expressions. Thanks to this backwarding, developers can determine expressions' precedence manually.

20. <arit_oper> ::=<identifier><increment_op> | <identifier><decrement_op>

| <identifier> | <arit_const> | <Time>

This nonterminal is the outermost level of the arithmetic expressions. Therefore, it has operators that have highest precedence. Unary increment (++) and decrement (--) operators are in this nonterminal. Besides that, it includes the smallest building blocks of arithmetic expressions. These can be identifiers, integer, double or Time literals.

- **Relational**

21. <relational_expression>::= <arithmetic_expression> < <arithmetic_expression>

| <arithmetic_expression> > <arithmetic_expression>

| <arithmetic_expression> <= <arithmetic_expression>

| <arithmetic_expression> >= <arithmetic_expression>

| <relat_term>

This nonterminal is the root level of the relational expressions. Therefore, it indicates the operations with lower precedence than other operators. It contains less than, greater than, less than or equal and greater than or equal operators. Relational operators compare the values of arithmetic expressions which can also be identifiers, integer, double and Time literals besides complex operations. **<relat_term>** terminal is used to define higher precedence.

22. <relat_term>::= <arithmetic_expression> == <arithmetic_expression>

| <arithmetic_expression> != <arithmetic_expression>

| <relat_factor>

The intended use of this nonterminal is to enable higher precedence for equality operators (== and !=) than other relational operators. Also, this nonterminal compares the values of arithmetic expressions. **<relat_factor>** nonterminal is used to define higher precedence than equality operators.

23. <relat_factor>::= <LSB><relational_expression><RSB>

The precedence of brackets in relational expressions is higher than any relational operators. Notice that between the brackets, there can be again any type of relational expressions. Thanks to this backwarding, developers can determine expressions' precedence manually. Notice that, normally parentheses are used instead of square brackets. The reason why CodeAD uses square brackets instead of parentheses is to avoid ambiguity. Logical expressions include relational expressions and they also have parentheses precedence. For example, (relational_expressions) is also a kind of logical expression. However, if CodeAD used parentheses instead of square brackets with this grammar structure, it would be unambiguous that which parentheses the grammar chooses. To avoid such an ambiguity, CodeAD uses square brackets to manage operator's precedence manually.

- **Logical**

24. <logical_expression>::= <logical_expression><or><logic_term> | <logic_term>

This nonterminal is the root level of the logical expressions. Therefore, it indicates the operations with lower precedence than other operators. So, saying that or operator has the lowest precedence would not be wrong. Besides that this nonterminal is defined as left recursive. In other words, operation order will be from left to right for or operator. **<logic_term>** nonterminal is used to define higher precedences. As an operator, CodeAD uses | symbol to indicate or operation. Logical expressions correspond to boolean values.

25. <logic_term>::=<logic_term><and><logic_factor> | <logic_factor>

The intended use of this nonterminal is to enable higher precedence for “and” operator than “or” operator (|). Also, this nonterminal is defined as left recursive. Again, operation order will be from left to right for “and” operators. **<logic_factor>** nonterminal is used to define higher precedence than “and” operator. As an operator, CodeAD uses & symbol to indicate “and” operation.

26. <logic_factor>::=<not><logic_factor> | <logic_relation>

This one is used for describing not operators whose precedence is higher than & and | operators. It is important to notice that this nonterminal is not defined as left recursive. It is defined as right recursive. That means the operation order will be from right to left among multiple “not” operators. As an operator, CodeAD use ! symbol to indicate “not”.

27. <**logic_relation>::=<relational_expression> | <logic_exp>**

This nonterminal is used to wire relational expressions into logical expressions. Because relational expressions also correspond to boolean values. Therefore, to make usage of them with logical operations such as |, &, ! operations valid, CodeAD uses this nonterminal and makes transitions. Since this nonterminal is further from the root, its precedence is higher than others. In other words, in a logical expression which includes both relational and logical operators, relational operators have higher precedence than logical ones.

28. <logic_exp>::= <LP><logical_expression><RP> | <logic_oper>

The precedence of parentheses in logic expressions is the highest. Notice that between the parentheses, there can be again any type of logical expressions. Thanks to this backwarding, developers can determine expressions’ precedence manually.

29. <logic_oper>::= <bool> | <identifier>

This nonterminal includes the smallest building blocks of logical expressions. These can be boolean literals or identifiers.

- **Combination**

30. <expressions>::= <arithmetic_operations> | <logical_expressions> | <other_expressions>

This nonterminal is used to gather all expressions under a single nonterminal named **<expressions>** to make it easier to connect them to **<statement>** nonterminal. Notice that this nonterminal uses **<other_expressions>** nonterminal to enable precedence.

31. <other_expressions>::= <relational_expressions>

This nonterminal is used to give higher precedence to relational expressions than others. The reason why CodeAD separates relational expressions is to avoid ambiguity. If this nonterminal did not exist, then relational expressions would have the same precedence with arithmetic and logical expressions. In this scenario, every time grammar encounters a relational expression, it can detect it in two ways: by directly choosing relational expressions or choosing logical expressions first, then making transition to relational expressions. Therefore, to solve this ambiguity, CodeAD takes advantage of this nonterminal.

● Loops

32. <loop_stmt> ::= <while_loop> | <for_loop>

This nonterminal states that there are two kinds of loop statements which are for loops and while loops and it aggregates these loop statements. We did not use do-while loop which is seen in some other languages because while loops can also carry out do-while loops' functions. Thanks to not using do-while loop, there are less reserved words to remember. Since this situation increases the simplicity, readability and writability increase too.

33. <while_loop> ::= while<LP><logical_expression><RP><LCB><statements><RCB>

This nonterminal states that while loop starts with "while" reserved word. Then, between left and right parenthesis, there is a logical expression that is evaluated as a condition and as long as this condition is true, the loop iterates. Then, between left and right curly braces, statements take part in the loop statement.

34. <for_loop> ::= for<LP><ident_initialization><logical_expression><semicolon>

<arithmetic_expression><RP><LCB><statements><RCB>

This nonterminal states that for loop starts with "for" reserved word. Then, between left and right parenthesis, there are, respectively, initialization with identifier declaration, logical expression and arithmetic expression separated by semicolons. After that, between left and right curly braces, statements take part in the loop statement.

35. <break_stmt> ::= break<semicolon>

This nonterminal states that the break statement consists of "break" reserved word followed by semicolon. When this statement is encountered, the loop is terminated even if its condition is still true.

● Function definitions and function calls

36. <function_definitions>::= method<space><return_types><space><identifiers><LP>

[<parameters>] <RP><LCB><statements><RCB>

This nonterminal states that functions in CodeAD are specified with “method” reserved word, return type of the function and identifier which is the name of the function separated by space character, respectively. Then, between the left and right parenthesis, zero or more parameters are specified. Therefore, a function without any parameter can be seen in the program. Finally, between left and right curly braces, statements take part in the function.

37. <parameters>::= <type_names><space><identifier>

| <type_names><space><identifier><LSB><RSB>

| <parameters><comma> (<type_names><space><identifier> |

<type_names><space><identifier><LSB><RSB>)

This nonterminal states that there can be zero or more number of arrays or any type of identifiers as parameters. While declaring this nonterminal, left recursion is used to specify multiple parameters possibility.

38. <arguments> ::= <identifier> | <expression> | <types>

| <arguments><comma> (<identifier> | <expression> | <types>)

During calling a function, we use arguments to replace the parameters of the corresponding function definition. These arguments can be any type of identifier, expression or constant value. While declaring this nonterminal, left recursion is used to specify that multiple arguments might be needed.

39. <return_types>::= <type_names> | none | <type_names><LSB><RSB>

This nonterminal states that a function can return any variable (int, double, bool, char, string) and array. It can also return nothing which is specified by the “none” reserved word. The return type is specified in the function signature.

40. <function_call>::=<identifier><LP><arguments><RP><semicolon>

This nonterminal states that function call starts with the name of the function as an identifier. Then, between left and right parenthesis, we put arguments that will replace the parameters of the function.The call is finished by a semicolon.

41. <return_stmt>::= return<space> (<expressions> | <identifier> | <types> |

<identifier><LSB><int><RSB> | none) <semicolon> |

return<space><function_call>

This nonterminal states that in accordance with the return type specified in the function signature, function returns an expression, identifier, constant value, array or a function call. We use space character between the "return" reserved word and the returned type and semicolon to end the return statement.

- **Arrays**

42. <array_declaration>::= <type_names><space><identifier><LSB><int><RSB><semicolon>

This nonterminal states that array can be declared by using any data type, space character, name of the array, left square brackets, an integer value as the size of the array, right square brackets and semicolon, respectively. Since arrays can contain any type of data, any data type can be used while declaring the array, except Time, Map and Direction which are special to our CodeAD language. In CodeAd language, array size must be specified in the array declaration and it must be a fixed integer value. This approach decreases writability.

43. <array_initilization> ::= double<space><identifier><LSB><int><RSB><assign>
<LCB> (space | <array_init_content_double> <RCB><semicolon>

|int<space><identifier><LSB><int><RSB><assign><LCB> (space | <array_init_content_double> <RCB><semicolon>

|bool<space><identifier><LSB><int><RSB><assign><LCB>(space | <array_init_content_double> <RCB><semicolon>

|string<space><identifier><LSB><int><RSB><assign><LCB>(space | <array_init_content_double> <RCB><semicolon>

|char<space><identifier><LSB><int><RSB><assign><LCB>(space | <array_init_content_double> <RCB><semicolon>

This nonterminal states that during the initialization of an array, we can equalize the array declaration without semicolon to arithmetic expression (a + 2), logical expression (a < 2), data type(int, double,...) or identifiers (a) between left and right curly braces in accordance with the array data type declared in the beginning of the statement. This enables the programmer to initialize array elements when the array is declared. If it is not initialized, all array's elements are initialized to "none" reserved word.

44. <element_assignment>::=<identifier><LSB><int><RSB><assign> (<expressions> |

<types>) <semicolon>

This nonterminal states that array elements can be assigned to a value or an expression somewhere in the code. This is achieved by writing the name of the array, specific index of the array between left and right square brackets, assignment operator, the value that the array index is assigned and semicolon, respectively.

45. **<array_init_content_int> ::= <arithmetic_expression> | <int> | <identifier> |**

<array_init_content_int><comma> (<arithmetic_expression>

| <int> | <identifier>)

This nonterminal states that while initializing the array elements between the left and right curly braces in the declaration statement, arithmetic expressions, integer values or identifiers can be used. This nonterminal enables us to have 0 or more initialization value between the left and curly braces by using left recursion.

46. **<array_init_content_double> ::= <arithmetic_expression> | <double> | <identifier> |**

<array_init_content_double><comma>

(<arithmetic_expression> | <double> | <identifier>)

This nonterminal states that while initializing the array elements between the left and right curly braces in the declaration statement, arithmetic expressions, double values or identifiers can be used. This nonterminal enables us to have 0 or more initialization value between the left and curly braces by using left recursion.

47. **<array_init_content_bool> ::= <logical_expression> | <bool> | <identifier> |**

<array_init_content_bool><comma> (<logical_expression>

| <bool> | <identifier>)

This nonterminal states that while initializing the array elements between the left and right curly braces in the declaration statement, logical expressions, boolean values or identifiers can be used. This nonterminal enables us to have 0 or more initialization value between the left and curly braces by using left recursion.

48. **<array_init_content_string> ::= <string> | <identifier> |<array_init_content_string>**

<comma> (<string> | <identifier>)

This nonterminal states that while initializing the array elements between the left and right curly braces in the declaration statement, string values or identifiers can be used. This nonterminal enables us to have 0 or more initialization value between the left and curly braces by using left recursion.

49. **<array_init_content_char> ::= <char> | <identifier> | <array_init_content_char>**

<comma> (<char> | <identifier>)

This nonterminal states that while initializing the array elements between the left and right curly braces in the declaration statement, char values or identifiers can be used. This nonterminal enables us to have 0 or more initialization value between the left and curly braces by using left recursion.

Definition of Nontrivial Tokens

Comments

Tokens:

- **SINGLE_COMMENT:** // single-line comment
- **MULTILINE_COMMENT:**

  /!

  multiline comment

  !/

Definitions:

CodeAD has two comment types: **single-line comment** and **multiline** comment. Every character that is written after // is treated as a single-line comment till the end of that line. In addition to this comment type, multiline comments are used between the symbols **/!** and **!/.** In other words, every character except the exclamation mark that is written between **/!** and **!/** symbols is treated as a multiline comment. Having more than one comment type will increase the writability of programs with CodeAD.

Identifiers

Tokens:

- **IDENTIFIER:** Tokens that indicate identifiers

Definitions:

CodeAD uses identifiers to give names to arrays, methods and variables. Like many other languages CodeAD has also naming conventions for identifiers:

- **Reserved words** can't be used as identifier names. This makes programs more readable.
- Identifier's names can only begin with lowercase letters, uppercase letters and underscores.
- Identifier's names can't begin with any digits or any other characters different from letters in the English alphabet.
- Except the first character of it, identifier's names can include letters, digits, underscores and $ sign.
- Spaces are not allowed and there is no upper limit for the length of identifier's names.

Literals

Tokens:

- **INT_LITERAL:** Tokens that corresponds to any integer value
- **DOUBLE_LITERAL:** Tokens that corresponds to any double value
- **CHAR_LITERAL:** Tokens that corresponds to any character value
- **STRING_LITERAL:** Tokens that corresponds to any string value
- **TIME_LITERAL:** Tokens that corresponds to any time value

Definitions:

CodeAD has 4 types of literals that represent the fixed-values in programs: **INT_LITERAL**s can correspond to any kind of integer. They represent negative integers with - sign. However, for positive values + sign is optional. **DOUBLE_LITERAL**s can correspond to any number with fractional components. Their sign representation is the same as integers. **CHAR_LITERAL**s can correspond to any character except newline and **STRING_LITERAL**s can correspond to any strings. However, quotation marks can't be used in string literals. **TIME_LITERAL**s corresponds to time types with format 00:01:12 that actually defines hours:minutes:seconds that elapsed. It is useful for adventure game developers to keep time.

Reserved Words

1.Basic words

Tokens:

- **INT_TYPENAME:** Tokens for **int** data types.
- **BOOL_TYPENAME:** Tokens for **bool** data types.
- **STRING_TYPENAME:** Tokens for **string** data types.
- **CHAR_TYPENAME:** Tokens for **char** data types.
- **DOUBLE_TYPENAME:** Tokens for **double** data types.
- **MAP_TYPENAME:** Tokens for **Map** data types.
- **DIRECT_TYPENAME:** Tokens for **Direction** data types.
- **TIME_TYPENAME:** Tokens for **Time** data types.
- **FOR:** Tokens for **for**
- **WHILE:** Tokens for **while**
- **BREAK:** Tokens for **break** reserved words
- **IF:** Tokens for **if** reserved words
- **ELSE:** Tokens for **else** reserved words
- **INPUT:** Tokens for **in** reserved words
- **OUTPUT:** Tokens for **out** reserved words
- **METHOD:**Tokens for **method** reserved words
- **RETURN:** Tokens for **return** reserved words
- **NONE:** Tokens for **none** reserved words
- **START:** Tokens for **start** reserved words
- **END:** Tokens for **end** reserved words

- **BOOL_TRUE:** Tokens for **true** value of boolean types
- **BOOL_FALSE:**Tokens for **false** value of boolean types
- **DIREC_NORTH:** Tokens for **North** values of Direction types
- **DIREC_SOUTH:** Tokens for **South** values of Direction types
- **DIREC_EAST:** Tokens for **East** values of Direction types
- **DIREC_WEST:** Tokens for **West** values of Direction types

Definitions:

CodeAD has also several reserved words that should not be used out of their purpose or naming identifiers. These are similar to other programming languages. For example, words such as **int, bool, string, char** and **double** are used to describe data types. In addition to these, CodeAD has another type called **Map, Time** and **Direction.** These types provide convenience to the adventure game developers while developing their game world's and conditions. Besides, **for** and **while** words describe loop types that can be used by developers and the **break** word enables to terminate the loop statement which it is written in. **if** and **else** words are used to describe conditionals; **in** and **out** are used to get input and print output respectively. Developers use **method** to define any function, **return** to finish the execution of the method and return a value. Also, they can use **none** to describe no return type in function definitions. Besides that, **none** can be used for array elements which are not initialized or are empty. Additionally, **start** and **end** words are used to start and end programs. Even though using them instead of curly braces is opposite to simplicity, it enhances readability. **true** and **false** are reserved words for boolean types. Finally **North, West, South** and **North** are reserved words for Direction types**.** Number of reserved words are not too much and this enhances the simplicity of CodeAD and makes it more readable and writable.

2.System Functions

Tokens:

- **CREATE_MAP_FUNCT:** Token for **createMap** reserved words
- **ADD_CHP_FUNCT:** Token for **addChapter** reserved words
- **CONNECT_CHPS_FUNCT:** Token for **connectChapters** reserved words
- **ADD_ITEM_FUNCT:** Token for **addItem** reserved words
- **GET_TIME_FUNCT:** Token for **getTime** reserved words
- **MOVE_FUNCT:** Token for **getContent** reserved words
- **IS_AVAILABLE:**Token for **isAvailable** reserved words
- **GET_CONTENT_FUNCT:** Token for **getContent**reserved words
- **IS_CONNECTED_FUNCT:** Token for **isConnected** reserved words
- **USE_TOOL :** Token for **useTool** reserved words

Definitions:

CodeAD has primary system functions whose aim is to provide great convenience to adventure game developers to create their games by using them. Firstly, **createMap** function will get an argument as the number of chapters wanted by the developer. Then, it creates a **Map** variable that contains the chapters with the wanted number and returns that **Map** variable. Besides that, **addChapter** function is used to add chapters to the map variable that is taken as the argument of this function. In addition to these, **connectChapters** function takes three arguments: two of them are the index of chapters that are wanted to be connected and the third one is direction. Then, the function connects these chapters. In this way, the relation between the chapters would be established. It connects the second chapter whose index is written later to the first chapter according to the value of the third argument. **addItem** function takes two arguments: one of them is the name of the item that is wanted to be added, other one is the chapter index in which the item is added. **getTime** function returns the elapsed time from the point of the declaration of its Time argument. Additionally, **move** function takes the movement direction as an argument. It moves to one of the connected chapters according to the Direction argument and returns the index of one of these chapters. **isAvailable** function checks the availability of the Direction which is taken as an argument by checking the connected chapters. **getContent** function will return the content of the chapter whose index is given as argument. It returns an array. Finally, **useTool** function takes the tool name as argument and removes it from the item list of the player. The number and contents of these functions seem to decrease the simplicity of CodeAD, but actually it offers common grounds for adventure game developers and helps them to develop algorithms by using abstraction.

Operators

Operators	**Tokens**	**Operators**	**Tokens**
(	LP	++	INCREEMENT_OP
)	RP	--	DECREMENT_OP
{	LCB	+	PLUS_OP
}	RCB	-	MINUS_OP
[	LSB	\n	NL
]	RSB	|	OR_OP
*	MULT_OP	&	AND_OP
/	DIV_OP	*=	MULT_AND_ASSIGN
%	MOD_OP	+=	PLUS_AND_ASSIGN
**	POWER_OP	-=	MINUS_AND_ASSIGN
Operators	**Tokens**	**Operators**	**Tokens**
“	QM	/=	DIV_AND_ASSIGN
?	QUESTION_MARK		
:	COLUMN	==	EQUAL_OP

,	COMMA	!=	NOT_EQUAL_OP
;	SEMICOLON	<	LESS_THAN_OP
_	UNDERSCORE	>	GREATER_THAN_OP
!	NOT	<=	LESS_THAN_EQUAL_OP
=	ASSIGN_OP	>=	GREATER_THAN_EQUAL_OP
.	DOT		

Evaluation of CodeAd

- **Readability**

Feature multiplicity: In CodeAD, there are more than one way to accomplish a particular operation. For example, incrementation by one operation can be done by using these statements:

```
i = i + 1;

i++;

i += 1;
```

This causes degradation in readability by decreasing simplicity of the language.

Orthogonality: In CodeAD, functions can return any type including arrays. This provides orthogonality in the language and that means less exceptions. On the other hand, array elements can be any type except Time,Map and Direction. This is opposed to orthogonality.

Operator Overloading: In CodeAD, the user is not allowed to overload any operator and it increases overall simplicity. Therefore, readability also increases. However, in the language, there are some operation symbols that are used in different ways. For example, [] symbol is used to specify the array index and it can be used to indicate the precedence of logical expressions e.g [a < b] .

Syntax Design: CodeAD has used matching pairs of reserved words called **start** and **end** to form a group as a main function body.This makes syntax clearer by using a distinct syntax but on the other hand conflicts with simplicity.

Data Types: In CodeAD, there are a total of 8 data types which are int, double, bool, string, char, Direction, Time and Map. There is not any type which can be thought as redundant. There are Direction, Time, Map data types as well as other basic data types that are common in most of the programming languages. Since our programming language is particularly for creating adventure games, data types to hold time, direction and map features are essential. Therefore, it can be stated there are adequate predefined data types in CodeAD and it increases readability.

● Writability

Simplicity and Orthogonality: CodeAD does not have excessive numbers of primitive constructs or set of rules to combine these constructions. All of its constructions are designed to aim for the use of language. Thus, they are necessary. When using these constructions and combinations of rules, developers do not have difficulty.

Expressivity: As mentioned previously, in CodeAD, there are more than one way to accomplish a particular operation such as incrementation by one operation. Although this reduces the readability, it increases the expressivity of language and makes it more writable.

Support for Abstraction: CodeAD enables developers to write functions. Functions are great examples for abstractions since they allow details to be ignored while using them. Therefore, it enhances the writability of the language.

● Reliability

Aliasing: Since CodeAD is not an object-oriented programming language, there is no need for references or pointers.Therefore, it does not have an aliasing problem. This makes CodeAD more reliable.

Type Checking: CodeAD does not have a formal type checking system. However, during the initialization of variables, each variable can be initialized with a literal whose type is the same as the variable's type. For example,

```
int x = "string";
```

is not a valid statement for CodeAD.

Writability and Readability: The more readability and writability increase, the more reliability increases.

Memory Management: Since our programming language does not use pointers as C++ does, the possibility of having a memory leak is decreased and the programs written in CodeAD language need less memory. Also, while declaring an array, an integer value is fixed as the size of the array. Therefore, it leads programmers to use less memory in our programming language than in any array size extendable programming language.