CS315
PROGRAMMING LANGUAGES

PROJECT 1
TEAM 2

HIndie (Head Indie Developer) LANGUAGE

Eylül Çağlar 21703949
İsmet Alp Eren 21703786
Kerem Alemdar 21702133

BNF

Program:

<program> ::= xox <code_block> oxo

Variables:

<digit> ::= [0-9]

<letter> ::= [a-z] | [A-Z]

<sign> ::= + | -

<semicolon> ::= ;

<lessThan> ::= <

<greaterThan> ::= >

<leftBracket> ::= {

<rightBracket> ::= }

<leftParenthesis> ::= (

<rightParenthesis> ::=)

<leftSquareBracket> ::= [

<rightSquareBracket> ::=]

<dot> ::= .

<star> ::= *

<symbol> ::= ! | ^ | + | % | & | / | (|) | = | * | - | £ | # | $ | ½ | { | [|] | } | _ | \| | ; | < | > | : | .

<newLine> ::= /n

Assignment:

<assign> ::= <var> = <expr>

<declaration> ::= <primitive_objects_types> <var> | <primitive_objects_types> <assign>

<array_assign> ::= <array_primitive_objects> [expr] = <expr><semicolon>

Statements:

<codes> ::= <code> | <codes><code>

<code_block> ::= <leftBracket><codes><rightBracket>

<code> ::= <expr><semicolon>| <condition_smt> | <loops> | <function_call><semicolon> | <assign><semicolon> | <comment_block> | <print><semicolon> | <array_declaration_with_a_list><semicolon> | <array_declaration><semicolon>

Expressions:

<expr> ::= <expr> <op> <primitive_objects> | <primitive_objects> | <var> | <int><op><op> | <op><op><int> | <ask>

<var> ::= <letter> | <var> <var_name_things>

<var_name_things> ::= <letter> | <digit>

<primitive_objects> ::= <int> | <double> | <boolean> | <string> | <char> | <array_element>

<primitive_object_types> ::= “int” | “double” | “boolean” | “string” | “char”

Integer:

<int> ::= <singed_int> | <unsigned_int>

<singed_int> ::= <sign> <unsigned_int> | <unsigned_int>

<unsigned_int> ::= <digit> | <digit><unsigned_int>

Double:

<double> ::= <signed_double> | <unsigned_double>

<singed_double> ::= <signed_int> <dot> <unsigned_int> | <signed_int>

<unsinged_double> ::= <unsigned_int> <dot> <unsigned_int> | <unsigned_int>

String:

<string> ::=<double_quotation_mark> <sentence> <double_quotation_mark>

<sentence> ::= < word> | <sentence><word>

<word> ::= <word><string_things>| <string_things>

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

<double_quotation_mark> ::= <left_double_quotation_mark> | <right_double_quotation_mark>

<left_double_quotation_mark> ::= “

<right_double_quotation_mark> ::= ”

Boolean:

<boolean> ::= <true> | <false> | <var> <comparison_op> <expr>

<true> ::= 1 | “true”

<false> ::= 0 | “false”

Character:

<char> ::=<single_quotation_mark> <string_things> <single_quotation_mark>

<single_quotation_mark> ::= <left_single_quotation_mark> | <right_single_quotation_mark>

<left_single_quotation_mark> ::= ‘

<right_single_quotation_mark> ::= ’

Array:

<array_primitive_object> ::= <primitive_object> <leftSquareBracket><rightSquareBracket>

<array_element> ::= <primitive_object> <leftSquareBracket> <expr> <rightSquareBracket>

<array_declaration_with_a_list> ::= <primitive_object_types> <primitive_object> <leftSquareBracket><rightSquareBracket> = <int_list>

<array_declaration> ::= <array_primitive_objects> <var><leftSquareBracket><expr><rightSquareBracket>

<primitive_objects_list> ::= <primitive_objects_list><comma><primitive_objects_list> | <primitive_objects_list>

Loops:

<loops> ::= <for> | <while> | <for> | <for_each> | <do_while>

<for> ::= for <leftParenthesis><assign> ; <logic_exprs>; <expr><rightParenthesis> <code_block> | for <leftParenthesis><var> : <array><rightParenthesis> <code_block>

<while> ::= while <leftParenthesis><logic_exprs><rightParenthesis> <code_block>

<do_while> ::= do <code_block> while <leftParenthesis><logic_exprs><rightParenthesis>

Conditional Statements:

<condition_stmt> ::= <if_stmt> | <if_stmt> <else_stmt> | <if_stmt> <else_if_stmts> <else_stmt> | <if_stmt> <else_if_stmts>

<else_if_stmts> ::= <else_if_stmts><else_if_stmt> | <else_if_stmt>

<else_if_stmt> ::= else if <leftParenthesis><logic_exprs><rightParenthesis> <code_block>

<else_stmt> ::= else <code_block>

<if_stmt> ::= if <leftParenthesis><logic_exprs><rightParenthesis>
<code_block>

<or_and> ::= | | &

<comparison_op> ::= != | < | > | == | <= | >=

<logic_exprs> ::= <boolean> | <logic_exprs> <or_and> <boolean>

<switch_step> ::= case int: <code_block>

<switch_case> ::= switch <leftParenthesis> <expr><rightParenthesis>
<leftBrace> <switch_step> <rightBrace>

Statements For Input / Output:

<print> ::= print <leftParenthesis><print_things><rightParenthesis>

<print_things> ::= <primitive_object> | <var> | <print_things> <plus>
<primitive_object> | <print_things> <plus> <var>

<ask> ::= ask <leftParenthesis><rightParenthesis>

Function Definitions and Function Calls:

<function_call> ::= <function_name><leftParenthesis><rightParenthesis> |
<function_name><leftParenthesis> <primitive_object_list> <rightParenthesis>

<function_declaration> ::= <function_name><leftParenthesis> <parameters>
<rightParenthesis> <code_block>

<parameter> ::= <primitive_object_types><var>

<parameters> ::= <parameters> <comma> <parameter> | <parameter>

<primitive_object_list> ::= <primitive_objects> | <primitive_object_list>
<comma> <primitive_objects>

Comments:

<comment_block> ::= // <sentence> | /* <sentence> */

Explanation

Program:

- **<program> ::= xox <code_block> oxo**

Our program starts with xox and ends with oxo command.

Variables:

- **<digit> ::= [0-9]**

This terminal defines digits in this language.

- **<letter> ::= [a-z] | [A-Z]**

This terminal defines letters in this language.

- **<sign> ::= + | -**

Sign terminal is a positive and negative sign.

- **<semicolon> ::= ;**

This terminal is for semicolons.

- **<leftBracket> ::= {**

This terminal is for the left bracket.

- **<rightBracket> ::= }**

This terminal is for the right bracket.

- **<leftParenthesis> ::= (**

This terminal is for left parenthesis.

- **<rightParenthesis> ::=)**

This terminal is for right parenthesis.

- **<leftSquareBracket> ::= [**

This terminal is for the left square bracket.

- **<rightSquareBracket> ::=]**

This terminal is for the right square bracket.

- **<dot> ::= .**

This terminal is for the "." .

- **<symbol> ::= ! | ^ | + | % | & | / | (|) | = | * | - | £ | # | $ | ½ | { | [|] | } | _ | \| | ; | < | > | : | .**

This terminal is for every symbol.

Assignment:

- **<assign> ::= <var> = <expr>**

This non-terminal is for assigning values to variables.

- **<declaration> ::= <primitive_objects_types> <var> | <primitive_objects_types> <assign>**

This non-terminal is for declaration of variables. It could either be declared with or without value.

- **<array_assign> ::= <array_primitive_objects> [expr] = <expr>;**

This non-terminal is for assigning value on the corresponding index of the array.

Statements:

- **<codes> ::= <code> | <codes><code>**

This non-terminal is for the code body. Codes body composed of one or more code. This is a recursive call.

- **<code_block> ::= <leftBracket><codes><rightBracket>**

This non-terminal is for code blocks. Code block is any code or codes inside of the left bracket and the right bracket.

- **<code> ::= <expr><semicolon>| <condition_smt> | <loops> | <function_call><semicolon> | <assign><semicolon> | <comment_block> | <print><semicolon> | <array_declaration_with_a_list><semicolon> | <array_declaration><semicolon>**

This non-terminal is for any particular meaningful line which is ended and a complete line of code.

Expressions:

- **<expr> ::= <expr> <op> <primitive_objects> | <primitive_objects> | <var> | <int><op><op> | <op><op><int> | <ask>**

This non-terminal is for mathematical expressions, variables, taken input and some expressions peculiar to our language. Basically it includes everything which represents a value in our language.

- **<var> ::= <letter> | <var> <var_name_things>**

This non-terminal represents the name of the variable declared.

- **<var_name_things> ::= <letter> | <digit>**

This represents what a var name can include.

- **<primitive_objects> ::= <int> | <double> | <boolean> | <string> | <char> | <array_element>**

Primitive objects are integers, floats, boolean, characters and combination of characters, symbols, digits which is string.

- **<primitive_object_types> ::= “int” | “double” | “boolean” | “string” | “char”**

Primitive object types are lists of the names of our primitive objects.

Integer:

- **<int> ::= <singed_int> | <unsigned_int>**

int keyword represents integer numbers. Integer could be signed or unsigned integer. Unsigned integers are considered as positive.

- **<singed_int> ::= <sign> <unsigned_int> | <unsigned_int>**

Signed int represents integers which have signs before them.

- **<unsigned_int> ::= <digit> | <digit><unsigned_int>**

Unsigned int represents integers which have no sign.

Double:

- **<double> ::= <signed_double> | <unsigned_double>**

Floats are represented as double in our language. Double could be signed or unsigned double. Unsigned doubles are considered as positive.

- **<singed_double> ::= <signed_int> <dot> <unsigned_int> | <signed_int>**

Signed integers and floats are represented as signed doubles. Combination of signed integer and unsigned integer with a dot between them represents double.

- **<unsinged_double> ::= <unsigned_int> <dot> <unsigned_int> | <unsigned_int>**

Unsigned double is for unsigned integers and unsigned floats. Both the combination of unsigned integer dot unsigned integer and only unsigned integers are unsigned double.

String:

- **<string> ::=<double_quotation_mark> <sentence> <double_quotation_mark>**

Strings must have double quotation marks before and after they are written. If we have any sentence inside of double quotation marks it is string.

- **<sentence> ::= < word> | <sentence><word>**

Sentences are formed by one or more words coming together.

- **<word> ::= <word><string_things>| <string_things>**

Words are formed by adjoining string things.

- **<string_things> ::= <letter> | <symbol> | <digit>**

String things are a combination of letters, symbols and digits.

- **<double_quotation_mark> ::= <left_double_quotation_mark> | <right_double_quotation_mark>**

double_quotation_mark is represents double quotation marks

- **<left_double_quotation_mark> ::= “**

left_double_quotation_mark is represents double quotation marks’ left hand side

- **<right_double_quotation_mark> ::= ”**

right_double_quotation_mark is represents double quotation marks’ right hand side

Boolean:

- **<boolean> ::= <true> | <false> | <var> <comparison_op> <expr>**

Boolean is for true or false statements. Comparisons and equality expressions in mathematics also represent true or false statements.

- **<true> ::= 1 | “true”**

This term refers to the result of the comparison is correct.

- **<false> ::= 0 | “false”**

This term refers to the result of the comparison is false.

Character:

- **<char> ::=<single_quotation_mark> <string_things> <single_quotation_mark>**

Characters between single quotation marks are considered as char in our language.

- **<single_quotation_mark> ::= <left_single_quotation_mark> | <right_single_quotation_mark>**

- **<left_single_quotation_mark> ::= ‘**

left_single_quotation_mark is pointing to the beginning of the char.

- **<right_single_quotation_mark> ::= ’**

right_single_quotation_mark is pointing to the end of the char.

Array:

- **<array_primitive_object> ::= <primitive_object> <leftSquareBracket><rightSquareBracket>**

Array_primitive_object is a list of primitive objects. It is represented as a primitive object with left and right square brackets in its right.

- **<array_element> ::= <primitive_object> <leftSquareBracket> <expr> <rightSquareBracket>**

Array_elements are primitive objects which are stored in that array. It refers to the value of the corresponding index.

- **<array_declaration_with_a_list> ::= <primitive_object_types> <primitive_object> <leftSquareBracket><rightSquareBracket> = <int_list>**

This non-terminal is for array declaration with a specified list.

- **<array_declaration> ::= <array_primitive_objects> <var><leftSquareBracket><expr><rightSquareBracket>**

This non-terminal is for array declaration.

- **<primitive_objects_list> ::= <primitive_objects_list><comma><primitive_objects_list> | <primitive_objects_list>**

Sequence of primitive objects with comma between them.

Loops:

- **<loops> ::= <for> | <while> | <for> | <for_each> | <do_while>**

This non-terminal is a combination of all loops in our language.

- **<for> ::= for <leftParenthesis><assign> ; <logic_exprs>; <expr><rightParenthesis> <code_block> | for <leftParenthesis><var> : <array><rightParenthesis> <code_block>**

This non-terminal is for loops. For loops could be in 2 ways. First one is starting with an assignment and after it takes a logic expression which is equal to mathematical equations in our language. By doing the given expression until the logic expression becomes true it loops the code block inside of it.

- **<while> ::= while <leftParenthesis><logic_exprs><rightParenthesis> <code_block>**

Until the given logic expression is true, execute the given code block.

- **<do_while> ::= do <code_block> while <leftParenthesis><logic_exprs><rightParenthesis>**

It executes the given code block inside of do and while until the corresponding logic expression is turned to false. Alternative syntax of while.

Conditional Statements:

- **<condition_stmt> ::= <if_stmt> | <if_stmt> <else_stmt> | <if_stmt> <else_if_stmts> <else_stmt> | <if_stmt> <else_if_stmts>**

It shows the combination of if else syntaxes. There could be if inside of if and else if inside of if.

- **<else_if_stmts> ::= <else_if_stmts><else_if_stmt> | <else_if_stmt>**

It represents a sequence of else ifs.

- **<else_if_stmt> ::= else if <leftParenthesis><logic_exprs><rightParenthesis> <code_block>**

Else if statement only comes after if statement to create new conditions.

- **<else_stmt> ::= else <code_block>**

Else part of if statements. If the given logic expression in if statement is not true then the else part will be executed. Else with a code block is a complete else statement in our language and it is represented as else_stmt.

- **<if_stmt> ::= if <leftParenthesis><logic_exprs><rightParenthesis> <code_block>**

If the given logic expression is true then the corresponding code block will be executed.

- **<or_and> ::= | | &**

Or and and operators are represented as double and or double or symbols.

- **<comparison_op> ::= != | < | > | ?= | <= | >=**

Mathematical comparison operators are named as comparison expr in our language.

    - **< ::=**

This terminal is for comparisons. Indicates if the left hand side is less than the right hand side or not.

    - **> ::=**

This terminal is for comparisons. Indicates if the left hand side is greater than right hand side or not.

    - **?=**

Checks whether the left hand side and right hand side is equal.

    - **!=**

Not equal operator.

    - **<=**

Less than or equal.

    - **>=**

More than or equal.

- **<logic_exprs> ::= <boolean> | <logic_exprs> <or_and> <boolean>**

Booleans and combination of booleans with and or or operators are logic expressions that represent either true or false.

- **<switch_step> ::= case int: <code_block>**

Switch conditional statement is combination of switch steps.

- **<switch_case> ::= switch <leftParenthesis> <expr><rightParenthesis> <leftBrace> <switch_step> <rightBrace>**

It takes an expression and the value of that expression will be compared to given cases. In each switch step an integer is given which will be compared with the given expression’s value. If the integer in switch step and expression is the same then the code block next to that case will be executed. If no case matches then nothing will happen.

Statements For Input / Output:

- **<print> ::= print <leftParenthesis><print_things><rigthParenthesis>**

It prints the given content between the left and right parenthesis.

- **<print_things> ::= <primitive_object> | <var> | <print_things> <plus> <primitive_object> | <print_things> <plus> <var>**

Print things represent the printable objects in our language.

- **<ask> ::= ask <leftParenthesis><rightParenthesis>**

Ask if you can take input from the user.

Function Definitions and Function Calls:

- **<function_call> ::= <var><leftParenthesis><rightParenthesis> | <function_name><leftParenthesis> <primitive_object_list> <rightParenthesis>**

Functions are called by function name followed by left and right parenthesis. It contains parameters between parentheses if needed.

- **<function_declaration_answer> ::= <primitive_object_types><var><leftParenthesis> <parameters> <rightParenthesis> <code_block>answer<primitive_objects>**

Function declaration includes function name following parameters inside parentheses and it follows code blocks which explains what function does, it returns a primitive object.

- **<function_declaration_free> ::= free <var><leftParenthesis> <parameters> <rightParenthesis> <code_block>**

Function declaration includes function name following parameters inside parentheses and it follows code blocks which explains what function does. It does not return anything.

- **<parameter> ::= <primitive_object_types><var>**

Parameter represents a variable with its primitive object type.

- **<parameters> ::= <parameters> <comma> <parameter> | <parameter>**

It is a list of parameters with commas between them.

- **<primitive_object_list> ::= <primitive_objects> | <primitive_object_list> <comma> <primitive_objects>**

It is a list of primitive objects with commas between them.

Comments:

- **<comment_block> ::= <comment_op> <sentence> | <comment_opening> <sentence> <comment_close>**

Comment_block is for lines of strings which are not a part of the code itself and are represented with two backslashes in the beginning or between one backslash star and star backslash.

- **<comment_op> ::= <lessThan><lessThan>**

Starting of the comment line.

Primitive Functions:

- **door_open()**

If a door is found then this function opens the door.

- **door_close()**

If an open door and player wants to close the door after opening it, this function closes the door.

- **chest_open()**

If the player finds and wants to open an unlocked chest this function opens the chest and loots the items.

- **buy_food()**

This function buys food from food merchants if the player has enough money.

- **buy_tools(string)**

This function buys tools from tool merchants if the player has enough money.

- **eat_food(string)**

This function the player eats its food if the player has low power.

- **open_map()**

This function opens the scroll of the map to see current location of the player and other stuff.

- **change_weapon(string** weaponName**)**

This function changes the equipped weapon with one of the owned weapons inside the inventory.

- **print_strength()**

This function shows the current strength of the player with equipped weapons.

- **fight()**

This function starts the fight with the nearest monster. If there are more than one monsters that have the same distance to the player then ask the player to choose one of them.

- **change_wealth(string)**

This function updates the wealth rate according to the action.

- **print_wealth()**

This function shows the current wealth of the player.

- **print_own_status()**

This function shows the current statutes of the player.

- **equip()**

This function equips the weapon that is found from chests or drops from monsters. The difference between this function and change functions is, change functions let the player change items by looking at the inventory.

- **move_up()**

This function changes the location of the character to (y = y - 1)th location.

- **move_down()**

This function changes the location of the character to (y = y + 1)th location.

- **move_left()**

This function changes the location of the character to (x = x - 1)th location.

- **move_right()**

This function changes the location of the character to (x = x + 1)th location.

- **equipment_list()**

This function shows the current equipment list of the player.

- **break_wall(int)**

This function allows the player to break the wall if the player has enough strength. Breaking walls will run out of strength.

Non-Trivial Tokens

- xox: Token to start the program.
- oxo: Token to terminate the program.
- answer: Token to return values within a function.
- free: Token is a function type which returns nothing.
- if: Token reserved for conditional if statements.
- else: Token reserved for conditional else statements.
- else if: Token reserved for conditional else if statements.
- while: Token reserved for detection of a while loop.
- for: Token reserved for detection of a for loop.
- print: Token reserved for printing contents of set to the console.
- ask: Token reserved for reading from an input stream.
- int: Token reserved for integer variable.
- double: Token reserved for double variable.
- boolean: Token reserved for boolean variable.
- string: Token reserved for string variable.
- char: Token reserved for char variable.
- switch: Token reserved for start conditional switch cases.
- case: Token reserved to indicate one case of switch.

Readability:

The main purpose in our language is maintaining english as the base language. Programmers or game developers could even understand our additional functions which are designed for adventure game development by only looking at their name. Our language syntax is very close to modern programming languages and contains many constructs from most used programming languages to ease the learning process of new structures of our language. Our logic expressions are almost the same as mathematical expressions and simplicity of conditional statements and structure of the overall coding are increasing readability. Language is created avoiding the use of too many complex words. It has multiple data types which also increase the readability by defining purposes to those data types. When the usage of a variable could not be understood it’s data type also gives information about the purpose of that variable so its not reliant on practice of naming variables.

Writability:

To make our language more writable we implement it simple and compatible with other languages, thanks to that users can code with this new programming language without wasting time to learn it. Yet, while implementing our language simple we avoid conflicts and useless cases. Since this language includes methods for adventure games it makes writing an adventure game from scratch easier than other languages.

Reliability:

In this programming language it is considered that lots of programmers are used to some reserved words. These reserved words are taken into consideration. Thus, possible confusions are blocked. This quality increases the rate of reliability of the language. Since it is simple to read and easy to write it fulfils the need of using time effectively.

In addition, this language includes lots of methods which can be modified and used in every type of adventure game. While creating adventure games, every basic function which could be inside of it is engraved into language itself to ease the programmers job. Those functions are also modifiable that support adaptations for their games.

The language also meets the needs of programmers keyboard dominance since it does not include characters which are hard to press while writing fast.

To conclude, this language is a reliable language for users.