CS315 Programming Languages

Project 1 Report

Mars Language Design

Lara Fenercioğlu 21802536 Section 1

İlke Doğan 21702215 Section 1

Onuralp Avcı 21902364 Section 2

Table of contents

1.Introduction

In this report, our aim is to create a new readable, writable and reliable language for adventure games. Our language is called MARS because the game takes place there. We will show the BNF description of our language and explain what non terminals do. Also, we will introduce the reserved words that are special to MARS language.

2.BNF Description

Program

```
<program> ::= START_GAME <stmts> END_GAME

<stmts> ::= <stmt> | <stmt> <stmts> | <if_stmt> <stmts> |
<comment><stmts>

<stmt> ::= <declare> | <assign> | <declare_assign> | <expr> |
<loop> | <func_call> | <function_declare>  | <io_stmt> |
<special_func_call>

<comment> ::= #<string_exp>#
```

Statements

```
<declare> ::= <type><id>

<assign> ::= <id><assignment_op><expr> |
<id><assignment_op><func_call> |
<id><assignment_op><special_func_call>

<declare_assign> ::=  <constant_declare_assign> | <declare> =
<expr>

<constant_declare_assign> ::= const <declare> = <literal>

<expr> ::= <id> | <literal> | <arith_expr> | <logical_expr> |
<update_expr>

<loop> ::= <for_loop> | <while_loop>

<func_call> ::= <id>(<args>) | <id>()

<function_declare> ::= <non_void_funtion_declare> |
<void_function_declare>

<io_stmt> ::= <input_stmt> | <output_stmt>

<special_func_call> ::= CREATE_MAP(<int>)
                | ADD_ROOM(<id><float>,<float>, <int>)
                | ADD_DOOR(<id>,<float>,<float>)
```

```
| CREATE_PLAYER(<id>,<int>,<int>)
| ADD_TREASURE(<id>, <int>)
| ADD_MONSTER(<id>, <int>, <int>)
| MOVE(<id> , <singed_int>, <singed_int>)
| GET_ROOM_CONTENTS_MONSTER(<id>)
| GET_ROOM_CONTENTS_TREASURES(<id>)
| GET_PLAYER_WEALTH()
| GET_PLAYER_STRENGTH()
| GET_PLAYER_HEALTH()
| GET_CURRENT_ROOM()
| IS_PLAYER_DEAD()
| PICK_TREASURE()
| FIGHT_MONSTER(<id>)
| EAT_FOOD()
| USE_TOOLS(<id>)
| BUY(<id>)
| IF_QUIT()
```

Expressions

```
<expr> ::= <id> | <literal> | <arith_expr> | <logical_expr> |
<update_expr>

<arith_expr> ::= <arith_expr> + <low_term> | <arith_expr> -
<low_term> | <low_term>

<low_term> ::= <low_term> * <high_term> | <low_term> /
<high_term> | <high_term>

<high_term> ::= <low_term> ^ <factor> | <low_term> % <factor>
| <factor>

<factor> ::= ( <arith_expr> ) | <id> | <literal>

<update_expr> ::= <increment> | <decrement>

<increment> ::= <id>++ |  ++<id>

<decrement> ::= <id>-- | --<id>

<logical_expr> ::= <logical_expr> or <logic_term> |
<logic_term>
```

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

<logic_factor> ::= (<logical_expr>) | not <logical_expr> |
<id> | <literal>

<conditional_expr> ::= <id>

| <id><relational_op><id>

| <id><relational_op><signed_int>

| <signed_int><relational_op><id>
```

If Statements

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

<unmatched> ::= if (<conditional_expr>) [<stmts>] | if
(<conditional_expr>) [<matched>] else [<unmatched>]

<matched> ::= if (<conditional_expr>) [<matched>]
else[<matched>] | <non_if_stmt>

<non_if_stmt> ::= <stmt>;<non_if_stmt> | <stmt>;
```

Loops

```
<while_loop> ::= while (<conditional_expr>)[<stmts>]

<for_loop> ::= for
(<for_init>;<conditional_expr>;<for_update>) [<stmts>]

<for_init> ::=  <for_init>, <assign> | <for_init>, <declare> |
<for_init>, <declare_assign> | <declare_assign> | <assign> |
<declare> |

<for_update> ::= <assign> | <update_expr> |
```

IO

```
<input_stmt> ::= read <expr>

<output_stmt> ::= write <expr>
```

Functions

```
<non_void_funtion_declare> ::= <type>
<id>(<parameters>)[<stmts > return <return_stmt>;] | <type>
<id>()[<stmts> return <return_stmt>;]

<void_function_declare> ::= void <id>(<parameters>)[<stmts>] |
void <id>()[<stmts>]

<parameters> ::= <parameter> | <parameter> , <parameters>
```

<parameter> ::= <declare>

<return_stmt> ::= <expr> | <func_call>

| <return_stmt><arithmetic_op><expr>

| <return_stmt><arithmetic_op><func_call>

<args> ::= <ids> | <literal> | <ids>,<args> |
<literal>,<args>

Types

<type> ::= int | float | char | bool | str | ptr

<literal> ::= <int> <signed_int> | <float> | <char> | <str> |
<bool> NULL

<char> ::= '<normal_chars>' | ''

<all_chars> ::= <normal_chars> | <special_chars>

<str> ::= "<string_exp>" | ""

<string_exp> :: <all_chars> | <space> |
<all_chars><string_exp> | <string_exp><space><string_exp>

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

<signed_int> ::= <positive_int> | <negative_int>

<positive_int> ::= +<int>

<negative_int> ::= -<int>

<float> ::= .<int> | <signed_int>.<int>

<ids> ::= <id> | <id>,<ids>

<id> ::= <normal_chars> | <normal_chars><id> | <id><digit>

Symbols

<normal_chars> ::=
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|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|_

<special_chars> ::= ! | @ | # | \$ | % | ^ | & | * | (|) | +
| = | / | * | - | ' | " | ; | '|' | { | } | [|]

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

<newline> ::= \n

<bool> ::= <true> | <false>

<true> ::= TRUE | 1

```
<false> ::= FALSE | 0
<space> ::= ' '
```

Operators

```
<arithmetic_op> ::= + | - | * | ^ | / | %
<assignment_op> ::= = | += | -= | *= | /= |^= | %=
<relational_op> ::= <= | >= | < | > | == | !=
```

3.Explanation

```
<program> ::= START_GAME <stmts> END_GAME
```

This non-terminal is the initial state which is made of statements. User has to write START_GAME in order to start the game and has to write END_GAME in order to finish the game.

```
<stmts> ::= <stmt> | <stmt> <stmts> | <if_stmt> <stmts> |
<comment> <stmts>
```

Statements are made of a combination of statements including a statement, if statement and comments.

```
<comment> ::= #<string_exp>\n
```

User has to type # in order to specify a comment followed by a new line.

```
<stmt> ::= <declare> | <assign> | <declare_assign> | <expr> |
<loop> | <func_call> | <function_declare>  | <io_stmt> |
<special_func_call>
```

A single statement can be many types of statements such as declaration statement, assign statement, both declaration and assign statements, an expression, loop statement, a function call or a function declaration, input output statement and finally special functions which are special to MARS Language.

```
<declare> ::= <type><id>
```

Declare is used for declaration of various types like int or str where its name is stored in the non terminal id.

```
<type> ::= int | float | char | bool | str | ptr
```

Type is used for showing what kind of a data is expected to be stored in the id coming after the type.

```
<id> ::= <normal_chars> | <normal_chars><id> | <id><digit>
```

Id is used as a label for storing the address of a value. It has to start with a char and it can also have digits at the end.

```
<assign> ::= <id><assignment_op><expr> |
<id><assignment_op><func_call> |
<id><assignment_op><special_func_call>
```

Assign is used for assigning a value to an id which is declared before. It can be assigned to an expression, to a return value of a function, or a return value of a special function which is specific for MARS language.

```
<assignment_op> ::= = | += | -= | *= | /= |^= | %=
```

These are all kinds of assignment operators. They can be used for simply assigning a value or they can also process the data before assigning it to the id.

```
<declare_assign> ::=  <constant_declare_assign> | <declare> =
<expr>
```

Declare assign is used for both declaring and assigning the value at the same time. Declaration can be a normal declaration or a constant declaration.

```
<constant_declare_assign> ::= const <declare> = <literal>
```

Constant declare assign is used for declaring a type and assigning it a value of literal which cannot be changed afterwards. User has to specify the word const in order to create a constant variable.

```
<literal> ::= <int> <signed_int> | <float> | <char> | <str> |
<bool> | NULL
```

Literal is basically any value which can be stored as 32-bit data in the CPU. It can be a number, a string, a float, a bool, or NULL.

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

Int consists of digits. It does not have a sign.

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

Digit contains all 10 digits.

```
<signed_int> ::= <positive_int> | <negative_int>
```

Signed int is either a positive integer or a negative integer which has a sign.

```
<positive_int> ::= +<int>
```

Positive int is an int with a + sign in front of it.

```
<negative_int> ::= -<int>
```

Negative int is an int with a - sign in front of it.

```
<float> ::= .<int> | <signed_int>.<int>
```

Float is used for float representation. It accepts both numbers starting with a '.' sign and the numbers which have a '.' in between digits.

```
<char> ::= '<normal_chars>' | ''
```

Char is a single character which can be a space or one of the normal chars.

```
<normal_chars> ::=
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|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|_
```

Normal chars basically include all letters and the underscore sign.

```
<str> ::= "<string_exp>" | ""
```

In order to specify a string user must type "" between a string expression or can leave it empty.

```
<string_exp> :: <all_chars> | <space> |
<all_chars><string_exp> | <string_exp><space><string_exp>
```

A string expression can be formed with chars, space, starting with char and continuing string expression or string expression having spaces between. This will be used to create string values.

```
<all_chars> ::= <normal_chars> | <special_chars>
```

This non terminal will include all normal chars and special chars.

```
<special_chars> ::= ! | @ | # | \$ | % | ^ | & | * | ( | ) | +
| = | / | * | - | ' | " | ; | '|' | { | } | [ | ]
```

Special chars will include chars that are not letters and _.

```
<space> ::= ' '
```

Space is just an indicator of a space in the program.

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

Bool non terminal will include true and false.

```
<true> ::= TRUE | 1
```

True will give TRUE or 1 as a result.

```
<false> ::= FALSE | 0
```

False indicates the value FALSE or 0.

```
<expr> ::= <id> | <literal> | <arith_expr> | <logical_expr> |
<update_expr>
```

Expr is used for various expressions like id, literal, arithmetic expression like addition or multiplication, logical expression like smaller or bigger than, and for update expression like increment or decrement.

```
<arith_expr> ::= <arith_expr> + <low_term> | <arith_expr> -
<low_term> | <low_term>
```

Arith_expr will consider each arithmetic operation based on its precedence. Right now it includes addition and subtraction but if the user wants to perform other high precedence operations non terminal will choose low_term non terminal.

```
<low_term> ::= <low_term> * <high_term> | <low_term> /
<high_term> | <high_term>
```

Low_term will consider multiplication and division. If a user wants to use higher precedence operations, a high_term non terminal will be chosen.

```
<high_term> ::= <low_term> ^ <factor> | <low_term> % <factor>
| <factor>
```

High_term will consider power and mod operations and if the user wants to use higher precedence operations, factor non terminal will be called.

```
<factor> ::= ( <arith_expr> ) | <id> | <literal>
```

Factor will include the highest precedence non terminals among all other arithmetic operations such as parentheses, id, and a single value.

```
<logical_expr> ::= <logical_expr> or <logic_term> |
<logic_term>
```

Logical expressions are used for using or operation. It executed after “not” and “and” operators

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

Logical terms are used for using and operation. It executed after “not” operator

```
<logic_factor> ::= (<logical_expr>) | not <logical_expr> |
<id> | <literal>
```

Logical factor is used for not operator, id or literal. They are considered before “or” and “and” operations.

```
<update_expr> ::= <increment> | <decrement>
```

Update_expr will be called when the user wants to increment or decrement a value by one.

```
<increment> ::= <id>++ |  ++<id>
```

Increment will basically increment a value by one.

```
<decrement> ::= <id>-- | --<id>
```

Decrement will decrement a value by one.

```
<loop> ::= <for_loop> | <while_loop>
```

In MARS language, there will be only two types of loops. For loop and while loop.

```
<for_loop> ::= for
(<for_init>;<conditional_expr>;<for_update>) [<stmts>]
```

This non terminal will start with a reserved word “for” and will continue other related non terminals which will determine loop’s variable, condition and update.

```
<for_init> ::=  <for_init>, <assign> | <for_init>, <declare> |
<for_init>, <declare_assign> | <declare_assign> | <assign> |
<declare> |
```

Not every for loop has the same initial statement so in MARS language, users can have many variables assigned, declared or both at the same time in this non terminal. This for loop also may not have an initial value so we give an option to leave this for_init as blank.

```
<conditional_expr> ::= <id>

| <id><relational_op><id>

| <id><relational_op><signed_int>

| <signed_int><relational_op><id>
```

Every loop has a condition and it will be determined by the user whether it will be just an id, or a relational operator type of condition.

```
<relational_op> ::= <= | >= | < | > | == | !=
```

This non terminal included all relational operands that may be used in conditional expressions.

```
<for_update> ::= <assign> | <update_expr> |
```

For loops may have an update part as well which may be an assign operation or an update expression. For loops may not also have an update part so if this is the case, for_update will be blank.

```
<while_loop> ::= while (<conditional_expr>)[<stmts>]
```

While loops consist of a reserved word “while”, a conditional expression followed by square brackets that is outside of the statements.

```
<func_call> ::= <id>(<args>) | <id>()
```

In MARS language, there is no specific token to specify a function call but instead the user has to write an id of the function followed by parentheses. There can be arguments in parentheses or not depending on the function declaration.

```
<args> ::= <ids> | <literal> | <ids>,<args>  |
<literal>,<args>
```

Args can be a bunch of identifiers, values, or both.

```
<function_declare> ::= <non_void_funtion_declare> |
<void_function_declare>
```

If the user wants to declare a function, s/he must decide whether the function will return something or it will be just void.

```
<non_void_funtion_declare> ::= <type>
<id>(<parameters>)[<stmts > return <return_stmt>;] | <type>
<id>()[<stmts> return <return_stmt>;]
```

If the function returns something this non terminal will work. A non void function must have a return type at the beginning, name of that function. Depending on the user, the function may have parameters or not. But it will have square brackets outside a bunch of statements. Also there will be a return statement.

```
<parameters> ::= <parameter> | <parameter> , <parameters>
```

If a function has parameters part in its declaration then this non terminal will be called. There can be one parameter or more than one parameter.

```
<parameter> ::= <declare>
```

Parameter will be a declaration.

```
<return_stmt> ::= <expr> | <func_call>

| <return_stmt><arithmetic_op><expr>

| <return_stmt><arithmetic_op><func_call>
```

If a function is non void then this non terminal will return values. This return may be an expression, function call, or an arithmetic operation.

```
<void_function_declare> ::= void <id>(<parameters>)[<stmts>] |
void <id>()[<stmts>]
```

Void function declaration will only be different than non void declaration by not having a return statement. It will also have a token called “void” to determine the function as a void.

```
<io_stmt> ::= <input_stmt> | <output_stmt>
```

Users can write and read by using io_stmt non terminal.

```
<input_stmt> ::= read <expr>
```

If a user wants to read an expression, s/he just needs to specify the word “read”.

```
<output_stmt> ::= write <expr>
```

If a user wants to write an expression, s/he just needs to specify the word “write”.

```
<special_func_call> ::= CREATE_MAP(<int>)
                      | ADD_ROOM(<id><float>,<float>, <int>)
                      | ADD_DOOR(<id>,<float>,<float>)
                      | CREATE_PLAYER(<id>,<int>,<int>)
                      | ADD_TREASURE(<id>, <int>)
                      | ADD_MONSTER(<id>, <int>, <int>)
                      | MOVE(<id> , <singed_int>, <singed_int>)
                      | GET_ROOM_CONTENTS_MONSTER(<id>)
                      | GET_ROOM_CONTENTS_TREASURES(<id>)
                      | GET_PLAYER_WEALTH()
                      | GET_PLAYER_STRENGTH()
                      | GET_PLAYER_HEALTH()
                      | GET_CURRENT_ROOM()
                      | IS_PLAYER_DEAD()
                      | PICK_TREASURE()
                      | FIGHT_MONSTER(<id>)
                      | EAT_FOOD()
                      | USE_TOOLS(<id>)
                      | BUY(<id>)
                      | IF_QUIT()
```

Special function calls are very important for MARS programming language because they make the game development process much easier. CREATE_MAP(int roomNumber) function creates a map which has a room capacity of roomNumber. ADD_ROOM(ptr room, float x_coordinate, float y_coordinate, int size) function adds a room to a specific map. It also sets the size and the position of the rooms relative to the center coordinate of the map. ADD_DOOR(ptr room, float x_coordinate, float y_coordinate) function is used for adding doors to the room objects. It is also possible to set the position of the door relative to the coordinates of the center of the room. CREATE_PLAYER(ptr room, int health, int strength) function creates the player in the room specified and it also assigns initial health and strength values to the player. ADD_TREASURE(ptr room, int value) function is used for adding treasures to a room specified. It is also possible to set the value of the treasure. ADD_MONSTER(ptr room, int health. int strength) function is used for adding monsters to a room specified. It is also possible to set health and strength values for the monster object. MOVE(ptr object, int x_change, int y_change) function is used to modify the current

coordinates of any object in the map. User can move the object to any direction with the parameters but the function still checks if there is any obstacle in that direction. If there is not any obstacle then the position of the object is updated. GET_ROOM_CONTENTS_MOSNTER(ptr room) returns the number of monsters in a specific room. GET_ROOM_CONTENTS_TREASURES(ptr room) returns the total value of treasures in a specific room. GET_PLAYER_WEALTH() returns the wealth of the player if the player is created before. GET_PLAYER_STRENGTH() returns the strength of the player if the player is created before. GET_PLAYER_HEALTH() returns the health of the player if the player is created before. GET_CURRENT_ROOM() returns the pointer of the room object where the player is currently in. IS_PLAYER_DEAD() returns true if player is dead and returns false if player is alive. PICK_TREASURE() function picks the treasure if the player is near the treasure. FIGHT_MONSTER(ptr monster) makes the player attack a monster specified if that monster is near to the player. EAT_FOOD() function makes the player eat food if there is food in inventory and this increases the player's health. USE_TOOL(ptr tool) picks the tool specified if it is in inventory. BUY(ptr tool) is for buying a tool specified. IF_QUIT() returns true if the player quits the game.

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

If statement is either a matched if statement or unmatched if statement.

```
<unmatched> ::= if (<conditional_expr>) [<stmts>] | if
(<conditional_expr>) [<matched>] else [<unmatched>]
```

Unmatched if statement is an if statement which either does not have an else statement or it has matched content inside the if statement and unmatched content inside the else statement.

```
<matched> ::= if (<conditional_expr>) [<matched>]
else[<matched>] | <non_if_stmt>
```

Matched if statement is an if statement which also has an else statement with matched content in it or it is a non if statement.

```
<non_if_stmt> ::= <stmt>;<non_if_stmt> | <stmt>;
```

Non if statement only includes statements followed by other non if statements or just a statement.

Reserved Words

`START_GAME` used to start the game

`END_GAME` used to finish the game

`TRUE` true value of a boolean

`FALSE` false value of a boolean

`if` used for if statements

`else` used for else statements

void used for void function declarations

read used for receiving inputs

write used for giving outputs

return used to state the return statements

const used to declare constant identifiers

for used for stating for loops

while used for while loops

CREATE_MAP used to create a map for the game

ADD_ROOM used to add a room to the map

ADD_DOOR used to add a door to a room

CREATE_PLAYER used to create a new player

ADD_TREASURE used to add a treasure into a room

ADD_MONSTER used to add a monster into a room

MOVE used to move the player

GET_ROOM_CONTENTS used to get what is in the room currently

GET_PLAYER_WEALTH used to get player's current wealth

GET_PLAYER_STRENGTH used to get player's current strength

GET_PLAYER_HEALTH used to get player'S current health

IS_PLAYER_DEAD used to get whether the player is dead or not

PICK_TREASURE used to pick a treasure from a room

FIGHT_MONSTER used to fight a monster

EAT_FOOD used to eat food

USE_TOOLS used to use a tool

BUY used to buy something

IF_QUIT used to get whether the player wants to quit or not

and used for logical and operator

or used for logical or operator

not used for logical not operator

int used to indicate the type as integer

float used to indicate the type as float

`bool` used to indicate the type as boolean

`char` used to indicate the type as char

`str` used to indicate the type as string

`ptr` used to indicate the type as pointer which is special to MARS language

`NULL` used to indicate the variable as a null value

4.Evaluation

a.Readability

One of our main goals while designing the MARS programming language was to prioritize readability. Although a normal person with no coding experience would have some hard time understanding, a person with coding experience won't face any major issues reading MARS. There are some reserved tokens and special functions that no other programming language has but they are easy to keep in mind so if the user knows these specialities of the program, s/he can easily understand what is happening in the code. Although the future multiplicity a++, a = a + 1, ++a and a += 1 may reduce the readability, they are still easy to read expressions and are easy to understand at first sight. MARS language does not allow the user to overload an operator so it is good for readability. Also, newly added special functions are readable as well because their functionalities are implicitly indicated in their names. So, users won't have difficulty using these functions.

b.Writability

MARS programming language is designed specifically for adventure game developments. For this reason, it is strongly optimized for writability in order to make coding with MARS easier. For instance, the special functions defined in MARS language like CREATE_MAP() or CREATE_PLAYER() can be easily learned and used by the programmers. Additionally, there are multiple ways to write the same expressions such as a = a + 1 and a += 1. As a result, the writability of the language increases again.

c.Reliability

In MARS language there is no type checking feature yet. There may be some errors related with the syntax so while compiling there should be a detector to determine whether there is an error or not. In our language there is also no exception handling. Lack of these features might reduce reliability but there is also a fact that readability

and writability also affects the reliability. So the easier a program is to write, the more likely it is to be correct. Also, programs that are difficult to read are difficult both to write and to modify thus if we can increase readability of the language we can also increase its reliability.