CS315 Programming Languages

Project 2 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. Finally, we will present the reason for the conflicts in our program.

2.BNF Description

Program

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

<stmts> ::= <stmt> | <stmt> <stmts>

<stmt>: <if_stmt> | <non_if_stmt> | <comment>
```

If Statements

```
<if_stmt> ::= if(<logical_or_expr>)[<stmts>];
     | if(<logical_or_expr>)[<stmts>]else[<stmts>];

<non_if_stmt> ::= <declare> ;| <assign> ;| <declare_assign> ;
| <loop>; | <func_call> ;| <function_declare>  ;| <io_stmt> ;|
<special_func_call> ;| <update_stmt>;
```

Statements

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

<assign> ::= <id><assignment_op><logical_or_expr>

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

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

<update_stmt> ::= <increment> | <decrement>

<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> ::= CREATE_MAP(<int>)
```

| ADD_ROOM(<int>,<int>, <int>)

| ADD_DOOR(<id>,<int>)

| CREATE_PLAYER(<id>,<int>,<int>)

| ADD_MINE(<id>, <int>, <int>)

| ADD_ALIEN(<id>, <int>, <int>)

| MOVE(<id>, <singed_int>, <singed_int>)

| MOVE_NORTH()

| MOVE_SOUTH()

| MOVE_EAST()

| MOVE_WEST()

| MOVE_NORTHWEST()

| MOVE_NORTHEAST()

| MOVE_SOUTHWEST()

| MOVE_SOUTHEAST()

| ATTACK_NORTH()

| ATTACK_SOUTH()

| ATTACK_EAST()

| ATTACK_WEST()

| ATTACK_NORTHWEST()

| ATTACK_NORTHEAST()

| ATTACK_SOUTHWEST()

| ATTACK_SOUTHEAST()

| PICK_NORTH()

| PICK_SOUTH()

| PICK_EAST()

| PICK_WEST()

| PICK_NORTHWEST()

| PICK_NORTHEAST()

| PICK_SOUTHWEST()

| PICK_SOUTHEAST()

| MINE_NORTH()

```
| MINE_SOUTH()
| MINE_EAST()
| MINE_WEST()
| MINE_NORTHWEST()
| MINE_NORTHEAST()
| MINE_SOUTHWEST()
| MINE_SOUTHEAST()
| GET_ROOM_CONTENTS()
| GET_PLAYER_WEALTH()
| GET_PLAYER_STRENGTH()
| GET_PLAYER_HEALTH()
| IS_PLAYER_DEAD()
| FIGHT_ALIEN()
| EAT_FOOD()
| USE_TOOLS(<id>)
| BUY(<id>)
| IF_QUIT()
```

Expressions

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

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

<logical_or_expr> ::= <logical_and_expr> |<logical_or_expr> or
<logical_and_expr>

<logical_and_expr> ::= <logical_xor_expr> | <logical_and_expr>
and <logic_xor_expr>

<logic_xor_expr> ::= <logic_not_expr> | <logic_xor_expr> xor
<logic_not_expr>

<logic_not_expr> ::= <relational_expr> | not <logic_not_expr>
| <relational_expr> not <logical_not_expr>

<relational_expr>::= <arithmetic_expr> | <arithmetic_expr>
<relational_op> <arithmetic_expr>
```

```
<arithmetic_expr>::= <arithmetic_factor> | <arithmetic_expr> +
<arithmetic_factor> | <arithmetic_expr> - <arithmetic_factor>

<arithmetic_factor>::= <arithmetic_term> | <arithmetic_factor>
* <arithmetic_term> | <arithmetic_factor> / <arithmetic_term>

<arithmetic_term>::= <primary_expr> | <arithmetic_term> ^
<primary_expr> | <arithmetic_term> % <primary_expr>

<primary_expr>: <id>|  <literal> | <func_call> |
<special_func_call> | ( <logical_or_expr>)
```

Loops

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

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

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

<for_update> ::= <update_stmt> |
```

IO

```
<input_stmt> ::= read <logical_or_expr>

<output_stmt> ::= write <logical_or_expr>
```

Functions

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

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

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

<parameter> ::= <declare>

<return_stmt> ::= <logical_or_expr>

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

Types

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

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

<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> | <comment>
```

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>: <if_stmt> | <non_if_stmt> | <comment>;
```

A statement can be either if statement, a non if statement or a comment.

```
<non_if_stmt> ::= <declare> ;| <assign> ;| <declare_assign> ;
| <loop>; | <func_call> ;| <function_declare>  ;| <io_stmt> ;|
<special_func_call> ;| <update_stmt>;
```

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><logical_or_expr>
```

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> =
<logical_or_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>
```

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 or bool.

```
<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.

```
<logical_or_expr> ::= <logical_and_expr> |<logical_or_expr> or
<logical_and_expr>
```

Starting with or logical operator because it is in the lowest precedence in between logical operators.

```
<logical_and_expr> ::= <logical_xor_expr> | <logical_and_expr>
and <logic_xor_expr>
```

After the or operation and operation will come because its precedence is higher.

```
<logic_xor_expr> ::= <logic_not_expr> | <logic_xor_expr> xor
<logic_not_expr>
```

After the and operation xor operation will come because its precedence is higher. We added this new.

```
<logic_not_expr> ::= <relational_expr> | not <logic_not_expr>
| <relational_expr> not <logical_not_expr>
```

After the xor operation not operation will come because its precedence is higher.

```
<relational_expr>::= <arithmetic_expr> | <arithmetic_expr>
<relational_op> <arithmetic_expr>
```

After finishing the logical operations, then we move on to relational operations like less than, greater than because their precedence is higher than logical expressions.

```
<arithmetic_expr>::= <arithmetic_factor> | <arithmetic_expr> +
<arithmetic_factor> | <arithmetic_expr> - <arithmetic_factor>
```

After the relational expressions, addition and subtraction operations will come because their precedence is higher.

```
<arithmetic_factor>::= <arithmetic_term> | <arithmetic_factor>
* <arithmetic_term> | <arithmetic_factor> / <arithmetic_term>
```

After the addition and subtraction operation, multiplication and division will come because their precedence is higher.

```
<arithmetic_term>::= <primary_expr> | <arithmetic_term> ^
<primary_expr> | <arithmetic_term> % <primary_expr>
```

After the multiplication and division operation, power and mod will come because their precedence is higher.

```
<primary_expr>: <id>|  <literal> | <func_call> |
<special_func_call> | ( <logical_or_expr>)
```

After the necessary expression, identifier, literal, function call, special function call, and expressions with parentheses will come because they're the highest precedence expressions.

```
<update_stmt> ::= <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>;<logical_or_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.

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

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

```
<for_update> ::= <update_stmt> |
```

For loops may have an update part as well which may be 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 (<logical_or_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> func
<id>(<parameters>)[<stmts > return <return_stmt>;] | <type>
func <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, func token and 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> ::= <logical_or_expr>
```

If a function is non void then this non terminal will return values. This return may be a logical expression.

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

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

```
<output_stmt> ::= write <logical_or_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(<int>,<int>,<int>)
                | ADD_DOOR(<id>,<int>)
                | CREATE_PLAYER(<id>,<int>,<int>)
                | ADD_MINE(<id>, <int>, <int>)
                | ADD_ALIEN(<id>, <int>, <int>)
                | MOVE(<id>, <int>, <int>)
                | MOVE_NORTH()
                | MOVE_SOUTH()
                | MOVE_EAST()
                | MOVE_WEST()
                | MOVE_NORTHWEST()
                | MOVE_NORTHEAST()
                | MOVE_SOUTHWEST()
                | MOVE_SOUTHEAST()
                | ATTACK_NORTH()
```

```
| ATTACK_SOUTH()
| ATTACK_EAST()
| ATTACK_WEST()
| ATTACK_NORTHWEST()
| ATTACK_NORTHEAST()
| ATTACK_SOUTHWEST()
| ATTACK_SOUTHEAST()
| PICK_NORTH()
| PICK_SOUTH()
| PICK_EAST()
| PICK_WEST()
| PICK_NORTHWEST()
| PICK_NORTHEAST()
| PICK_SOUTHWEST()
| PICK_SOUTHEAST()
| MINE_NORTH()
| MINE_SOUTH()
| MINE_EAST()
| MINE_WEST()
| MINE_NORTHWEST()
| MINE_NORTHEAST()
| MINE_SOUTHWEST()
| MINE_SOUTHEAST()
| GET_ROOM_CONTENTS()
| GET_PLAYER_WEALTH()
| GET_PLAYER_STRENGTH()
| GET_PLAYER_HEALTH()
| IS_PLAYER_DEAD()
| FIGHT_ALIEN()
| 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(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_MINE(ptr room, int value, int amount)function is used for adding treasures to a room specified. It is also possible to set the value of the mine and amount. ADD_ALIEN(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. Choosing a way based on the coordinate values with the tokens MOVE_NORTH(), MOVE_SOUTH(), MOVE_EAST(), MOVE_WEST() are created for directions known as north, south, east, west, respectively. Additionally, to get more accurate results while specifying character's moving direction, MOVE_NORTHWEST(), MOVE_NORTHEAST(), MOVE_SOUTHWEST(), MOVE_SOUTHEAST() functions are created. Based on the adventure on Mars, main character will attach to the aliens by using the functions ATTACK_NORTH(), ATTACK_SOUTH(), ATTACK_EAST(), ATTACK_WEST(), ATTACK_NORTHWEST(), ATTACK_NORTHEAST(), ATTACK_SOUTHWEST() so as to find the attach direction. As mentioned above, created mines will be found by the main character. However, at the first stage, after finding the mine place, main character will dig the soil by using the direction functions that are mainly MINE_NORTH(), MINE_SOUTH(), MINE_EAST(), MINE_WEST(),and additively, MINE_NORTHWEST(), MINE_NORTHEAST(), MINE_SOUTHWEST(), MINE_SOUTHEAST(). At the second stage, PICK_NORTH(), PICK_SOUTH(), PICK_EAST(), PICK_WEST(), PICK_NORTHWEST(), PICK_NORTHEAST(), PICK_SOUTHWEST(), PICK_SOUTHEAST() will be used to pick the direction that will be chosen by the player. GET_ROOM_CONTENTS() returns what is in the specific room currently. 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> : IF LP logical_or_xpr RP LSB stmts RSB;
      | IF LP logical_or_expr RP LSB stmts RSB ELSE LSB stmts RSB;
```

Users will have to type if token in order to create an if statement. Then there will be a logical expression to decide whether to enter the if or not. Users can both only type if statement or if and else statement. These statements will accept other if statements and also other statements in the program in their body.

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

elseif used for else if 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

func used for function declaration

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_MINE used to add a treasure into a room

ADD_ALIEN used to add a monster into a room

MOVE used to move the player

MOVE_NORTH move north

MOVE_SOUTH move south

MOVE_EAST move east

MOVE_WEST move west

MOVE_NORTHWEST move northwest

MOVE_NORTHEAST move northeast

MOVE_SOUTHWEST move southwest

MOVE_SOUTHEAST move southeast

ATTACK_NORTH attack north

ATTACK_SOUTH attack south

ATTACK_EAST attack east

ATTACK_WEST attack west

ATTACK_NORTHWEST attack northwest

ATTACK_NORTHEAST attack northeast

ATTACK_SOUTHWEST attack southwest

ATTACK_SOUTHEAST attack southeast

PICK_NORTH pick north

PICK_SOUTH pick south

PICK_EAST pick east

PICK_WEST pick west

PICK_NORTHWEST pick northwest

PICK_NORTHEAST pick northeast

PICK_SOUTHWEST pick southwest

PICK_SOUTHEAST pick southeast

MINE_NORTH mine north

MINE_SOUTH mine south

MINE_EAST mine east

MINE_WEST mine west

MINE_NORTHWEST mine northwest

MINE_NORTHEAST mine northeast

MINE_SOUTHWEST mine southwest

MINE_SOUTHEAST mine southeast

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_MINE used to pick a treasure from a room

FIGHT_ALIEN 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

xor used for logical xor 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

4.Conflicts

At the end of our final progress, we have left with zero conflicts. We have followed the precedence and associativity rules and also resolved ambiguous problems to come up with zero conflicts.