CS315 Programming Languages
Project-2: BOA
Group 35
Section: 3

Elif Gülşah Kaşdoğan / 21601183

Doğacan Kaynak / 21400682

Ece Çanga / 21600851

A. Revised and Augmented Language Design

< program > → **start** < functions > **stop** < statements >

| < statements >

| <empty>

< functions > → < functions > < function > | < function >

< function > → [< type >] < function name > (< parameter list >) **{** <body> **}**

<type> → **string** | **float | int** | **boolean** | **direction**

<function name> → <upper case><letters>

<parameter list> → <type> <variable>, <parameter list>

| <type> <variable>

| <empty>

<body> → <statements> **return** <function instantiation>

| <statements> **return** <element>

< statements > → < statement > **;**

| <statement> **;** <statements>

| **#** <comments> **#** < statements >

| <statement> **#** <comments> **#**

<statement> →<matched> | <unmatched>

<matched> → **if** (<logic_expr>) <matched> **else** <matched>

| <non-if statement>

<unmatched> → **if** (<logic_expr>) <stmt>

| **if** (<logic_expr>) <matched> **else** <unmatched>

<non-if statement> → <assignment>

| <loop>

| <list>

| <print>

| <scan>

|<primitive functions>

<primitive functions> → <move> | <grab> | <turn> | <release>| <sendData> | <readData>

<assignment> → <variable> = <logical expr>

| <variable> = <math expr>

| <variable> = <function instantiation>

|<list position>=<atom>

<variable> → <letters>| <constant>

<str> → **“**<letters>**”**

| **“**<empty>**”**

<letters> → <letter>

| <letter><letters>

<letter> → a|b|c|d…|z

<upper case> → A|B|C|D….|Z

<constant> →<upper case>
|<upper case><constant>

<int> → <digits>

<float> → <digits><dot><digits>

<digits> → <digit>

| <digit><digits>

<dot>→ **.**

<digit>→ 0|1|2…|9

<logic value>→true | false

<dir>→ **east** | **west** | **north** | **south** | **northeast** | **northwest** | **southeast** | **southwest**

<empty> →

<loop> → **while (**<element>**)** {<statements>}

<list> → **list[**<int>**]**

<list position> →<variable>**[**<int>**]**

<print> → **print(**<element>**)**

<scan> → **scan(**<variable>**)**

<move> → **Move** (<dir>,<float>)

<turn> → **Turn** (<float>)

< grab > → **Grab** (<variable>)

< release > → **Release** (<variable>)

< readData > → **ReadData** (<variable>)

< sendData > → **SendData** (<variable>)

<math expr>→ <expr> = <expr>
|<expr> + <expr>
|<expr> - <expr>
|<expr> * <expr>
|<expr> / <expr>

<function instantiation> → **function** <function name> (<parameter list>)

< element >→ **(**< logical expr >**)**

<logical expr>→< logical expr > **or** <and>

| <and>

<and> → <and> **and** <not>

| <not>

<not>→ **not** <element >

|<element>

<atom> → <str>

| <float>

| <int>

| < logic value >

| <dir>

<constant>→ <upper cases>

Our language does not require a main method, it requires start and stop before and after function definitions however you can directly type the statements without using these reserved keywords. We thought that it would increase writability to avoid main keyword like Python does, also it would be easier to learn (especially for starters) without remembering some keywords like public static void main(String[] args) which is used in Java as entry point of program.

We followed general math applications convention for precedence of logical and mathematical operators. Multiplication, division and modulo operator has precedence over addition and subtraction; logical operators have precedence not>and>or. This design is followed by most of the programming languages so we decided to follow convention at this point, we aimed to improve writability and readability by doing this.

We tried to avoid unnecessary reserved words to keep program simple and maintain writability. All reserved words clearly state the objective and we tried to follow general conventions while choosing these reserved words. With these specifications we tried to make sure that our program will have a good writability.

Everything is represented with a distinct name in our language because we tried to avoid aliasing to increase reliability. Run-time type checking is expensive thus compile time type checking is more desirable, also fixing errors will be less expensive when they are detected earlier. Because of these we defined types to allow type checking to make sure that errors will be caught during compile time rather than run time and we think that type checking will increase reliability. By providing high readability-writability we also

provide a more reliable programming language because if it is easy to write a program it will be most likely correct.

< program > : In our programming language functions can only be defined at start and can be followed by statements. User has to indicate that function definition expected with **start** and end of the function definitions should be stated by **stop**. User can write statements without declaring functions. We thought that defining functions only above or below would increase readability because whoever reads the code does not have to look for statements between functions, he/she will know where to look to find statements or functions.

< functions > : Functions can be declared one after another. Implementation will be capsulated by using curly brace and we think that it will increase readability because one can clearly see the beginning and end of a function. Functions will be declared by using return type, function name, parameter list order respectively. Parameters will be given to function in type-parameter order; user can see the expected parameter type from function definition.

Function can receive no parameters. Seeing return type and type of parameters in parameter list makes debugging easier if there is a problem related to variable types it makes this language reliable in that sense.

< function >: Function consists of return type, parameter list, function name and function body. Function can form functions either standalone or together with other functions.

<type> : This non-terminal represents types this language supports. We choose convenient data types and added direction as a data type since it will be highly important for robot.

<function name> : This non-terminal represents function name which can be any combination of letters from English alphabet. We left the choice to users for function names and we expect that developers will come up with a convention for function names. First letter of function name is uppercase and rest is lower case.

<parameter list> : This non-terminal represents parameter list for functions. It allows right recursion, it expects parameters and comma between parameters, parameter list can be empty.

<body>: This non-terminal represents function body. It has statements and expects a function instantiation or an element as return type.

<parameter>: Non-terminal for parameter,

<statement>: A statement can be either matched or unmatched in such sense we can form a logically appropriate if/else structure.

<matched>: matched if else clause exists, this type of structure together with unmatched part allows us to use nested if/else clauses. A non-if statement is also matched.

<unmatched>: first part of unmatched represents if statement without an else, nested statements are still available in this case ie. you can use multiple if else statements even it includes if without an else. Second part is where we match an else statement with the proper if so that code will be logically correct if user constructs a standard if/else structure.

<comments>: This non-terminal represents single or multiple lines of comments, it allows right recursion.

<comment>: Non-terminal representing comment statements. Comments start and end with

character. Multi-line comments are not supported in our language.

<non-if statement>: Non-terminal representing statements assignments, loops, list, primitive functions and print, scan functions.

<primitive functions>: Non-terminal defined for primitive functions of a robot.

<assignment>: Non-terminal which allows assignments between variables and expressions and function instantiations. It also allows assignments to a list index.

<variable>: This non-terminal defines variables which is represented by letters, variable name should be all uppercase if variable is constant.

<str>: This non-terminal defines what counts as letters or a letter that selected from Turkish alphabet in any form.

<digit>: This non-terminal represents the numbers from 0 to 9.

<letters>: This non-terminal allow left recursive definition of **<letter>**.

<letter>: This non-terminal represent lower case and upper case letters.

<int>: This non-terminal defines what counts as an integer. Integer can consist of a single digit or multiple digits.

<float>: This non-terminal defines floating-point numbers.

<logic value>: This non-terminal represents logical ture/false.

<dir> This non-terminal defines directions which are south, west, north and including intermediate directions.

<loop> This non-terminal explains how to write a loop statement. We decided to use **while**

which is followed by **<element>** and a while body consisting of statements.

<list initialization>: This non-terminal is used when initializing an array. It uses **list** reserved word followed by an integer in square brackets.

<list position> This non-terminal is used when retrieving an element from an array. It is called after **list** reserved word which is integer in square brackets.

<print>: This non-terminal is a primitive function named print which displays element taken as function parameter.

<scan>: This non-terminal represent primitive function scan. Function has a parameter list and a function body. Followed primitive functions have the same structure as this.

<move>: Primitive function move. Takes direction to move and amount of movement as parameters.

<turn> : Primitive function turn. Takes degree of turning as parameter.

< grab > : Primitive function grab. Takes variable name as reference to object to grab.

< release >: Primitive function release.

< readData >: Primitive function readData

< sendData >: This non-terminal represent primitive function sendData.

<math expr> : This non-terminal is constructed to obtain a precedence relation between operators, it allows left recursion. With this type of structure multiplication and division will be lower in all parse trees obtained because they will be derived first. In our first project we tried to obtain precedence among operators using non-terminals however this lead conflicts in yacc thus we defined association of operators in yacc file externally.

<function instantiation>: This non-terminal represents function instantiation ie how the functions will be called by user in this language. We added **function** reserved word in front of the instantiation, we though that it might increase readability. It is easier to identify functions used for whoever reads code.

< element >: Non-terminal element can be a logical expression and it is connected to hierarchy of logical operators.

<logical expr>: Logical expression is a representative non-terminal of all types of logical operations. It stands for “or” operation in fact. Since or operation is the lowest in precedence hierarchy it should be derived last. This structure is similar to what we did in

arithmetic operator precedence. We followed general logical operator precedence used by most of the languages.

<and>: Non-terminal representing logical and operator. And has precedence over or operator and not operator has precedence over and.

<not>: This non-terminal represents logical operation not which has the highest precedence among logical operators in our language. Every element can be used with not.

<atom>: Non-terminal which represents variables that can change values over runtime, we decided to call them atoms because they are basically the smallest part of our program in a high-level manner.

Lex Description File

```
integer [0-9]+
letters [a-z]+
capital [A-Z]+
float [0-9]+\.[0-9]+
uppercase [A-Z]
%{
int lineCounter = 1;
%}
%%
start return(PROGRAM_START);
stop return(PROGRAM_STOP);
string return(TYPE_STRING);
float return(TYPE_FLOAT);
int return(TYPE_INT);
boolean return(TYPE_BOOL);
direction return(TYPE_DIRECTION);
return return(RETURN);
if return(IF);
else return(ELSE);
true return(TRUE);
false return(FALSE);
```

```
east return(DIR_EAST);
west return(DIR_WEST);
north return(DIR_NORTH);
south return(DIR_SOUTH);
northeast return(DIR_NORTHEAST);
northwest return(DIR_NORTHWEST);
souteast return(DIR_SOUTHEAST);
southwest return(DIR_SOUTHWEST);
list return(LIST_INIT);
while return(WHILE_LOOP);
function return(FUNC_BEGN);
or return(OR);
and return(AND);
not return(NOT);
move return(MOVE_FUNC);
turn return(TURN_FUNC);
grab return(GRAB_FUNC);
release return(RELEASE_FUNC);
readData return(READ_FUNC);
sendData return(SEND_FUNC);
print return(PRINT_STATEMENT);
scan return(SCAN_STATEMENT);
{letters} return(VARIABLE);
{integer} return(INTEGER);
, return(COMMA);
\( return(LP);
\) return(RP);
\{ return(LC);
\} return(RC);
\[ return(LSQ);
\] return(RSQ);
\; return(SEMI_COL);
```

```
\= return(ASGN_OP);
#.+\# return(COMMENT);
\" return(QUOTE);
{capital} return(CONSTANT);
[A-Z]{letters} return(FUNC_NAME);
\+ return(PLUS);
\- return(MINUS);
\/ return(DIVISION);
\* return(MULTIPLICATION);
\% return(MODULO);
\n {lineCounter++;}
.;
%%
int yywrap(){return 1;}
```

B. Parser

Yacc description file:

```
%token PROGRAM_START PROGRAM_STOP TYPE_STRING TYPE_FLOAT TYPE_INT
TYPE_BOOL TYPE_DIRECTION
RETURN IF ELSE TRUE FALSE DIR_EAST DIR_WEST DIR_SOUTH DIR_NORTH DIR_NORTHEAST
DIR_NORTHWEST DIR_SOUTHEAST DIR_SOUTHWEST LIST_INIT WHILE_LOOP
SCAN_STATEMENT FUNC_BEGN
OR AND NOT MOVE_FUNC TURN_FUNC GRAB_FUNC RELEASE_FUNC READ_FUNC
SEND_FUNC VARIABLE
INTEGER COMMA LP RP LC RC LSQ RSQ SEMI_COL ASGN_OP COMMENT QUOTE
PRINT_STATEMENT
SCAN_STATEMENT VARIABLE INTEGER CONSTANT FUNC_NAME FLOAT PLUS MINUS
DIVISION MULTIPLICATION
MODULO SINGLE_CAPITAL
%right '='
%left '+' '-'
%left '*''/''%'
%%
start: program
;
program: PROGRAM_START functions PROGRAM_STOP statements
|statements
|/*EMPTY*/
;
functions: function|functions function
;
function: LSQ type RSQ function_name LP  parameter_list RP  LC body RC
;
type: TYPE_STRING|TYPE_FLOAT|TYPE_INT|TYPE_BOOL|TYPE_DIRECTION
;
```

```
function_name: FUNC_NAME
;
parameter_list: /*EMPTY*/
|type variable COMMA parameter_list
|type variable
;
body: statements RETURN function_instantiation
|statements RETURN element
;
statements:statement SEMI_COL
|statement SEMI_COL statements
|COMMENT statements
|statement COMMENT
;
statement: matched | unmatched
;
matched: IF LP logic_expr RP matched ELSE matched
|non_if
;
unmatched: IF LP logic_expr RP statement
| IF LP logic_expr RP matched ELSE unmatched
;
non_if: assignment
|loop
|list
|print
|scan
|primitive_func
;
primitive_func: move|grab|turn|release|sendData|readData
;
```

```
assignment: variable ASGN_OP logic_expr
|variable ASGN_OP math_expr
|variable ASGN_OP function_instantiation
|list_position ASGN_OP atom
;
variable: VARIABLE | CONSTANT
;
int: INTEGER
;
float: FLOAT
;
logic_value: TRUE|FALSE
;
dir: DIR_EAST| DIR_WEST| DIR_SOUTH|DIR_NORTH|DIR_NORTHEAST
|DIR_NORTHWEST |DIR_SOUTHEAST |DIR_SOUTHWEST
;
loop: WHILE_LOOP LP element RP LC statements RC
;
list: LIST_INIT LSQ int RSQ
;
list_position: variable LSQ int RSQ
;
print: PRINT_STATEMENT LP element RP
;
scan: SCAN_STATEMENT LP variable RP
;
move: MOVE_FUNC LP dir COMMA float
;
turn: TURN_FUNC LP float RP
;
grab: GRAB_FUNC LP variable RP
;
```

```
release: RELEASE_FUNC LP variable RP
;
readData: READ_FUNC LP variable RP
;
sendData: SEND_FUNC LP variable RP
;

function_instantiation: FUNC_BEGN LP parameter_list RP
;
math_expr:  expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
;
expr: float|int
;

logic_expr: logic_expr OR and
| and
;
and: and AND not
| not
;
not: NOT element
| element
;
element: LP logic_expr RP
;
atom: float|int|logic_value|dir
;
%%
#include "lex.yy.c"
```

```
extern int lineCounter;
int main(){
  yyparse();
  printf("Valid Input");
  return 0;
}
int yyerror(const char *s){fprintf(stderr, "%s in line %d\n", s, lineCounter); }
```

C. Test Program:

```
#functions
start
[string]Robottostring() {
        string str = "Robot's location x: " + x + " y: " + y;
return str;
}
[string]Returnstring(){
        string str = "";
        return str;
}
[boolean]Setlocationx(float f){
        locationX = f;
        return locationX == f;
}
[boolean]Setlocationy(float f){
        locationY = f;
        return locationY == f;
}
[float]Getlocationx() {
        return locationX;
}
[float]Getlocationy() {
        return locationY;
}
[dir]Perceiveenvironment(float d) {
```

```
        float x = robot.Getlocationx();
        float y = robot.Getlocationy();
        dir robotsDirection;
        if (f == 90) {
                dir = north;
        } else if (f == 180) {
                dir = south;
        } else if (f == 0) {
                dir = west;
        } else if (f == 270) {
                dir = east;
        } else if (f>90 and f > 180) {
                dir = northwest;
        } else if (f > 0 and f < 90) {
                dir = northeast;
        } else if (f < 270 and f > 180) {
                dir = southwest;
        } else {
                dir = southeast;
        }
        return dir;
}
# main
[int]Main() {
Robot robo;
string welcoming = “Welcome!”;
```

```
        Print(welcoming)
        robo.Grab(box)
        Print("Enter distance to carry out")
        int distance;
Scan(distance)
robo.SendData(distance)
Print("Enter degrees  for rotation")
float r;
Scan(r)
robo.SendData(Perceiveenvironment(r))
Print("Tricky question…What is the value of PI?")
int answer;
Scan(answer)
float PI = 3.14;
robo.Release(box)
if(answer != PI)
        Print("loser")
else
        Print("Everybody knows that")

        robo.Turn(r)
        robo.Move(east,distance)
        int count = 5;
        Print(robo.Robottostring())
        while (count > 0) {
                Print(robotList[count-1].Robottostring())
```

```
                    count = count – 1;
}
return 0;
}
stop

#statements
robotList = list[5];
float locationX;
float locationY;
dir defaultDir;
```