GÖKHAN ÖZELOĞLU - 21627557

BURAK YILMAZ - 21627868

FİDAN SAMET – 21727666

BBM 301 – PROGRAMMING LANGUAGES

LEX – YACC PROJECT

1. DESCRIPTION OF THE TOKENS

We defined some tokens for built-in functions to perform basic GIS operations:

- BLTIN_SHOW_ON_MAP corresponds to **showonmap function**
- BLTIN_SEARCH_LOCATION corresponds to **searchlocation function**
- BLTIN_GET_ROAD_SPEED corresponds to **getroadspeed function**
- BLTIN_GET_LOCATION corresponds to **getlocation function**
- BLTIN_TARGET corresponds to **showtarget function**
- BLTIN_GET_CROSSROADS corresponds to **getcrossroads function**
- BLTIN_GET_ROADS corresponds to **getroads function**
- BLTIN_GET_CROSSROADS_NUM corresponds **getcrossroadsnum** to **function**
- BLTIN_GET_ROADS_NUM corresponds to **getroadsnum function**
- BLTIN_COLLOBORATE_USERS corresponds to **colloborateusers function**
- BLTIN_INSTRUCT_USER corresponds to **intructuser function**
- BLTIN_INCREASE_SCORE_OF_ROAD corresponds to **increasescoreofroad function**
- BLTIN_DECREASE_SCORE_OF_ROAD corresponds to **decreasescoreofroad function**
- BLTIN_GET_SCORE_OF_ROAD corresponds to **getscoreofroad function**
- BLTIN_SHOW_ROAD_ON_MAP corresponds to **showroadonmap function**
- BLTIN_SHOW_CROSSROAD_ON_MAP corresponds to **showcrossroadonmap function**
- BLTIN_ADD_CROSSROAD corresponds to **addcrossroad function**
- BLTIN_ADD_ROAD corresponds to **addroad function**
- BLTIN_PRINT corresponds to **print function**

We defined tokens for primitive data types:

- CHAR corresponds to **char primitive data type**
- INT corresponds to **int primitive data type**
- FLOAT corresponds to **float primitive data type**
- BOOL corresponds to **bool primitive data type**
- STR corresponds to **string primitive data type**
- VOID corresponds to **void primitive data type**
- TRUE corresponds to **true primitive data type**
- FALSE corresponds to **false primitive data type**

We defined tokens for GIS objects for ease of use:

- GPS corresponds to **gps GIS object**
- ROAD corresponds to **road GIS object**
- CROSSROAD corresponds to **crossroad GIS object**

- GRAPH corresponds to **graph GIS object**
- USER corresponds to **user GIS object**

We defined tokens for 3D objects:

- HOME corresponds to **home 3D object**
- HOSPITAL corresponds to **hospital 3D object**
- SCHOOL corresponds to **school 3D object**
- BRIDGE corresponds to **bridge 3D object**
- MALL corresponds to **mall 3D object**
- BUSSTOP corresponds to **busstop 3D object**
- HOTEL corresponds to **hotel 3D object**
- POSTOFFICE corresponds to **postoffice 3D object**

We defined tokens for arithmetic operators:

- AND_OP corresponds to **"&&" operator**
- OR_OP corresponds to **"||" operator**
- LESS_EQ_OP corresponds to **"<=" operator**
- GREATER_EQ_OP corresponds to **">=" operator**
- NOT_EQ_OP corresponds to **"!=" operator**
- EQUALITY_OP corresponds to **"==" operator**
- LESS_OP corresponds to **"<" operator**
- GREATER_OP corresponds to **">" operator**
- EQUAL_OP corresponds to **"=" operator**
- MULTIPLY_OP corresponds to **"*" operator**
- DIVIDE_OP corresponds to **"/" operator**
- SUB_OP corresponds to **"-" operator**
- ADD_OP corresponds to **"+" operator**
- MOD_OP corresponds to **"%" operator**
- POW_OP corresponds to **"^" operator**
- NOT_OP corresponds to **"!" operator**

We defined tokens for flow control:

- IF corresponds to **if flow controller**
- ELIF corresponds to **elif flow controller**
- ELSE corresponds to **else flow controller**
- WHILE corresponds to **while flow controller**
- FOR corresponds to **for flow controller**
- BREAK corresponds to **break flow controller**
- CONTINUE corresponds to **continue flow controller**
- RETURN corresponds to **return flow controller**

We defined tokens for literals:

- INT_LITERAL corresponds to **int literal**
- FLOAT_LITERAL corresponds to **float literal**
- STR_LITERAL corresponds to **string literal**
- CHAR_LITERAL corresponds to **char literal**

We defined tokens for separators:

- COMMA corresponds to **"," separator**
- SEMICOLON corresponds to **";" separator**
- LEFT_PARANT corresponds to **"(" separator**
- RIGHT_PARANT corresponds to **")" separator**
- CURLY_OPEN corresponds to **"{" separator**
- CURLY_CLOSE corresponds to **"}" separator**
- SQR_OPEN coressponds to **“[” separator**
- SQR_CLOSE coressponds to **“]” seperator**
- DOLLAR_SIGN corresponds to **"$" as array significative**
- IDENT corresponds to **identifiers that declared by the programmer**
- FUNC corresponds to **func keyword to declare function**

2. BNF GRAMMAR

```
program : function

    | function program

    ;

function : FUNC return_type IDENT LEFT_PARANT parameter_list RIGHT_PARANT
block

    | error {error_msg("Missing 'func', not a valid function declaration!");}

    ;

return_type : data_type

    | data_type DOLLAR_SIGN

    | geo_types

    | geo_types DOLLAR_SIGN
    | three_d_objects

    | three_d_objects DOLLAR_SIGN

    | VOID

    ;

parameter_list : empty

    | VOID

    | data_type IDENT

    | data_type DOLLAR_SIGN IDENT

    | geo_types IDENT

    | geo_types DOLLAR_SIGN IDENT

    | three_d_objects IDENT

    | three_d_objects DOLLAR_SIGN IDENT

    | parameter_list COMMA data_type IDENT

    | parameter_list COMMA data_type DOLLAR_SIGN IDENT

    | parameter_list COMMA geo_types IDENT

    | parameter_list COMMA geo_types DOLLAR_SIGN IDENT

    | parameter_list COMMA three_d_objects IDENT

    | parameter_list COMMA three_d_objects DOLLAR_SIGN IDENT

    ;

argument_list : empty

    | IDENT

    | literal

    | argument_list COMMA IDENT

    | argument_list COMMA literal

    ;
```

```
block : CURLY_OPEN stmt_list CURLY_CLOSE
    | CURLY_OPEN empty CURLY_CLOSE
    ;
stmt_list : stmt
    | stmt_list stmt
    ;
stmt : declaration SEMICOLON
    | assignment SEMICOLON
    | function_call SEMICOLON
    | BREAK SEMICOLON
    | CONTINUE SEMICOLON
    | RETURN SEMICOLON
    | RETURN arithmetic_exp SEMICOLON
    | RETURN TRUE SEMICOLON
    | RETURN FALSE SEMICOLON
    | loop
    | if_stmt
    ;
declaration : data_type IDENT
    | data_type IDENT EQUAL_OP RHS
    | data_type DOLLAR_SIGN IDENT
    | data_type DOLLAR_SIGN IDENT EQUAL_OP CURLY_OPEN literal_list
CURLY_CLOSE
    | GPS IDENT EQUAL_OP tuple
    | GPS DOLLAR_SIGN IDENT EQUAL_OP CURLY_OPEN tuple_list CURLY_CLOSE
    | ROAD IDENT EQUAL_OP road
    | CROSSROAD IDENT EQUAL_OP cross_road

    | ROAD DOLLAR_SIGN IDENT EQUAL_OP CURLY_OPEN roads_list CURLY_CLOSE

    | CROSSROAD DOLLAR_SIGN IDENT EQUAL_OP CURLY_OPEN cross_roads_list CURLY_CLOSE

    | GRAPH IDENT EQUAL_OP CURLY_OPEN graph_arguments CURLY_CLOSE

    | USER IDENT EQUAL_OP LEFT_PARANT STR_LITERAL COMMA cross_road RIGHT_PARANT

    | USER IDENT EQUAL_OP LEFT_PARANT IDENT COMMA cross_road RIGHT_PARANT

    | three_d_objects IDENT EQUAL_OP LEFT_PARANT STR_LITERAL COMMA cross_road RIGHT_PARANT

    | three_d_objects IDENT EQUAL_OP LEFT_PARANT IDENT COMMA cross_road RIGHT_PARANT

    | three_d_objects DOLLAR_SIGN IDENT EQUAL_OP CURLY_OPEN three_d_arguments CURLY_CLOSE

    ;

three_d_arguments : LEFT_PARANT IDENT COMMA cross_road RIGHT_PARANT

    | LEFT_PARANT STR_LITERAL COMMA cross_road RIGHT_PARANT

    | IDENT

    | empty

    | three_d_arguments COMMA LEFT_PARANT STR_LITERAL COMMA cross_road RIGHT_PARANT

    | three_d_arguments COMMA LEFT_PARANT IDENT COMMA cross_road RIGHT_PARANT

    | three_d_arguments COMMA IDENT

    ;

graph_arguments : empty

    | IDENT COMMA IDENT
    | IDENT COMMA LEFT_PARANT roads_list RIGHT_PARANT

    | LEFT_PARANT cross_roads_list RIGHT_PARANT COMMA IDENT

    | LEFT_PARANT cross_roads_list RIGHT_PARANT COMMA LEFT_PARANT
roads_list RIGHT_PARANT

    ;

road : LEFT_PARANT tuple COMMA tuple RIGHT_PARANT

    | LEFT_PARANT tuple COMMA IDENT RIGHT_PARANT

    | LEFT_PARANT IDENT COMMA tuple RIGHT_PARANT

    | LEFT_PARANT IDENT COMMA IDENT RIGHT_PARANT

    ;

cross_road : tuple

    | IDENT

    ;

roads_list : empty

    | road

    | IDENT

    | roads_list COMMA road

    | roads_list COMMA IDENT

    ;

cross_roads_list : cross_road

    | cross_roads_list COMMA cross_road

    ;

tuple : LEFT_PARANT long_lat_param COMMA long_lat_param RIGHT_PARANT

    | empty

    ;

literal_list : empty
```

```
    | literal

    | literal_list COMMA literal

    ;

tuple_list : tuple

    | tuple_list COMMA tuple

    ;

arithmetic_exp : term

    | arithmetic_exp ADD_OP term

    | arithmetic_exp SUB_OP term

    ;

term : term MULTIPLY_OP literal

    | term DIVIDE_OP literal

            { if ($3) $$ = $1 / $3;

                    else {error_msg("Divide by zero!");}

            }

    | term POW_OP literal

    | term MOD_OP literal

    | term MULTIPLY_OP IDENT

    | term DIVIDE_OP IDENT

    | term POW_OP IDENT

    | term MOD_OP IDENT

    | factor

    ;

factor : LEFT_PARANT arithmetic_exp RIGHT_PARANT

    | literal
    | IDENT

    ;

RHS : arithmetic_exp

    | function_call

    | bool_exp

    | IDENT SQR_OPEN INT_LITERAL SQR_CLOSE

    | IDENT SQR_OPEN IDENT SQR_CLOSE

    | error{error_msg("Not a valid expression!");}

    ;

function_call : IDENT LEFT_PARANT argument_list RIGHT_PARANT

    | BLTIN_SHOW_ON_MAP LEFT_PARANT long_lat_param COMMA long_lat_param
RIGHT_PARANT

    | BLTIN_SEARCH_LOCATION LEFT_PARANT IDENT RIGHT_PARANT

    | BLTIN_SEARCH_LOCATION LEFT_PARANT STR_LITERAL RIGHT_PARANT

    | BLTIN_GET_ROAD_SPEED LEFT_PARANT road_param RIGHT_PARANT

    | BLTIN_GET_LOCATION LEFT_PARANT user_param RIGHT_PARANT

    | BLTIN_TARGET LEFT_PARANT STR_LITERAL RIGHT_PARANT

    | BLTIN_TARGET LEFT_PARANT IDENT RIGHT_PARANT

    | BLTIN_GET_ROADS LEFT_PARANT cross_road RIGHT_PARANT

    | BLTIN_GET_CROSSROADS LEFT_PARANT road_param RIGHT_PARANT

    | BLTIN_GET_ROADS_NUM LEFT_PARANT cross_road RIGHT_PARANT

    | BLTIN_GET_CROSSROADS_NUM LEFT_PARANT road_param RIGHT_PARANT

    | BLTIN_COLLOBORATE_USERS LEFT_PARANT user_param RIGHT_PARANT

    | BLTIN_INSTRUCT_USER LEFT_PARANT user_param COMMA destination
RIGHT_PARANT

    | BLTIN_INCREASE_SCORE_OF_ROAD LEFT_PARANT road_param
RIGHT_PARANT
```

```
    | BLTIN_DECREASE_SCORE_OF_ROAD LEFT_PARANT road_param
RIGHT_PARANT

    | BLTIN_GET_SCORE_OF_ROAD LEFT_PARANT road_param RIGHT_PARANT

    | BLTIN_SHOW_ROAD_ON_MAP LEFT_PARANT road_param RIGHT_PARANT

    | BLTIN_SHOW_CROSSROAD_ON_MAP LEFT_PARANT cross_road
RIGHT_PARANT

    | BLTIN_ADD_CROSSROAD LEFT_PARANT IDENT COMMA cross_road
RIGHT_PARANT

    | BLTIN_ADD_ROAD LEFT_PARANT IDENT COMMA road_param
RIGHT_PARANT

    | BLTIN_PRINT LEFT_PARANT argument_list RIGHT_PARANT

    ;

destination : LEFT_PARANT STR_LITERAL COMMA cross_road RIGHT_PARANT

    | FLOAT_LITERAL

    | IDENT

    | tuple

    ;

user_param : LEFT_PARANT STR_LITERAL COMMA cross_road RIGHT_PARANT

    | LEFT_PARANT IDENT COMMA cross_road RIGHT_PARANT

    | IDENT

    ;

road_param : road

    | IDENT

    ;

long_lat_param : FLOAT_LITERAL

    | IDENT

    ;
```

```
literal : SUB_OP INT_LITERAL
    | ADD_OP INT_LITERAL
    | SUB_OP FLOAT_LITERAL
    | ADD_OP FLOAT_LITERAL
    | INT_LITERAL
    | FLOAT_LITERAL
    | STR_LITERAL
    | CHAR_LITERAL
    ;
assignment_op : EQUAL_OP
    | MULTIPLY_OP
    | DIVIDE_OP
    | SUB_OP
    | ADD_OP
    | MOD_OP
    | POW_OP
    ;
assignment : LHS assignment_op RHS
    ;
LHS : IDENT
    | error{error_msg("Invalid identifier declaration!");}
    ;
loop : while
    | for
    ;
```

```
while : WHILE LEFT_PARANT bool_exp RIGHT_PARANT block

    | WHILE LEFT_PARANT IDENT RIGHT_PARANT block

    ;

for : FOR LEFT_PARANT for_stmt RIGHT_PARANT block

    ;

for_stmt : for_init SEMICOLON bool_exp SEMICOLON assignment

    | for_init SEMICOLON IDENT SEMICOLON assignment

    ;

for_init : declaration

    | assignment

    ;

if_stmt : IF LEFT_PARANT bool_exp RIGHT_PARANT block

    | IF LEFT_PARANT IDENT RIGHT_PARANT block

    | IF LEFT_PARANT function_call RIGHT_PARANT block

    | if_stmt ELIF LEFT_PARANT bool_exp RIGHT_PARANT block

    | if_stmt ELIF LEFT_PARANT IDENT RIGHT_PARANT block

    | if_stmt ELSE block

    ;

bool_exp : comparison

    | NOT_OP IDENT

    | TRUE

    | FALSE

    ;

comparison : IDENT relational_op compared

    | bool_exp logic_op compared
    | IDENT logic_op compared

    | function_call relational_op compared

    ;

compared : IDENT

    | FALSE

    | TRUE

    | literal

    ;

relational_op : LESS_EQ_OP

    | GREATER_EQ_OP

    | NOT_EQ_OP

    | EQUALITY_OP

    | LESS_OP

    | GREATER_OP

    ;

logic_op : AND_OP

    | OR_OP

    ;

data_type : CHAR

    | INT

    | FLOAT

    | BOOL

    | STR

    ;

three_d_objects : HOME
```

```
    | HOSPITAL

    | SCHOOL

    | BRIDGE

    | MALL

    | BUSSTOP

    | HOTEL

    | POSTOFFICE

    ;

geo_types : GPS

    | ROAD

    | CROSSROAD

    | USER

    ;

empty :

    ;
```

3. ADDITIONAL FUNCTIONALITY

- **showonmap(longitude, latitude)**

  In this function, we take 2 parameters: longitude as a float and a latitude as also a float. It shows the given location(gps data) on the map.

- **searchlocation(address)**

  In this function, we take one parameter: address as a string. It searches, finds, shows and returns the gps type value that we defined in our language on the map. That gps value holds the longitude and latitude value of the given address' location.

- **getroadspeed(road)**

  In this function, we take one parameter: road as a road type that we defined in our language. It calculates the average speed of the vehicles in given road according to density of the vehicles and traffic jam. After that it increases or decreases the score of given road according to the calculated average speed. Finally it returns the calculated average speed of the vehicles in given road.

- **getlocation(user)**

  In this function, we take one parameter: user as a user type that we defined in our language. It returns a gps type value that we defined in our language. That gps value holds the longitude and latitude value of the given user’s current location.

- **showtarget(address)**

  In this function, we take one parameter: address as a string. It searches, finds and shows the gps type value that we defined in our language on the map. That gps value holds the longitude and latitude value of the given address’ location.

- **getcrossroads(road)**

  In this function, we take one parameter: road as a road type that we defined in our language. It returns a crossroad array that includes the crossroads of given road on the map.

- **getroads(crossroad)**

  In this function, we take one parameter: crossroad as a crossroad type that we defined in our language. It returns a road array that includes the roads of given crossroad on the map.

- **getcrossroadsnum(road)**

  In this function, we take one parameter: road as a road type that we defined in our language. It returns the number of crossroads of given road on the map.

- **getroadsnum(crossroad)**

  In this function, we take one parameter: crossroad as a crossroad type that we defined in our language. It returns the number of roads of given crossroad on the map.

- **colloborateusers(user)**

  In this function, we take one parameter: user as a user type that we defined in our language. It uses mobile devices to collaborate with other users who use same navigation program and similar roads for instant location information of given user.

- **intructuser(user, crossroad)**

  In this function, we take 2 parameters: user as a user type that we defined in our language and crossroad as a crossroad type that we also defined in our language. It gets the user's current location and guides to user’s convenient roads in terms of closeness to target, density of the vehicles, average speed of these vehicles and traffic jam according to the score for each road with respect to road conditions. It does that by using functions colloborateusers and getroadspeed.

- **increasescoreofroad(road)**

  In this function, we take one parameter: road as a road type that we defined in our language. It increases the score of the given road by 1 for convenience.

- **decreasescoreofroad(road)**

  In this function, we take one parameter: road as a road type that we defined in our language. It decreases the score of the given road by 1 for convenience.

- **getscoreofroad(road)**

  In this function, we take one parameter: road as a road type that we defined in our language. It returns the current score of the given road for convenience.

- **showroadonmap(road)**

  In this function, we take one parameter: road as a road type that we defined in our language. It shows the given road on the map.

- **showcrossroadonmap(crossroad)**

  In this function, we take one parameter: crossroad as a crossroad type that we defined in our language. It shows the given crossroad on the map.

- **addcrossroad(graph, crossroad)**

  In this function, we take 2 parameters: graph as a graph type that we defined in our language and crossroad as a crossroad type that we also defined in our language. It adds the given crossroad to given graph.

- **addroad(graph, road)**

  In this function, we take 2 parameters: graph as a graph type that we defined in our language and road as a road type that we also defined in our language. It adds the given road to given graph.

- **print(value)**

  In this function, we take one parameter: which can be empty, a identifier or a literal of any primitive data types.

4. SHORT TUTORIAL

Our programming language, Fibugo, provides some Global Positioning System(GPS) structures whereas it also satisfies classic programming languages fundamentals like loop, variables, functions, control statements. We are going to explain details of our languages in this section.

Fibugo has some different data types. These are **char, string, int, float** and **bool.**

In addition, fibugo has some complex data types such as **3D objects, road, gps, crossroad, graph** and **user.**

char: This data type gets character. It only accepts one letter or digit. You can define a variable in char data type with different ways. Here is the sample char declarations you can use:

Sample: char c = ‘a’;

char ch;

ch = ‘b’;

string: This data type is similar to Java string data type. You can define a variable in char data type in different ways. Here is the sample string declarations you can use:

Sample: string s = "this is string";

string s;

s = "this is string";

int : This data type accepts integers. You can define a variable in char data type in different ways. Here is the sample int declarations you can use:

Sample: int t = 5 + 4;

int ii = 5;

int i;

i=-10;

float : The float data type is a single-precision floating point. You can define a variable in char data type in different ways. Here is the sample float declarations you can use:

Sample: float f; f=1.3;

float ff = 2.3;

bool: The bool data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions.

Sample: bool b = true;

bool bb;

bb = false;

road: The road data type is a complex data type .Basically it defines that there is a connection between two tuples which each tuple represents a coordinate on the map

and user can think as a road. Here is the sample road declaration you can use:

Sample: road r2 = ((3.4, 5.7), (3.4, 5.7));

crossroad: The crossroad data type is a complex data type. Basically, it defines that there is tuple which represents a coordinate on the map and user can think crossroad as a vertex. In addition, you can assign gps data type to the crossroad. Here is the sample crossroad declaration you can use:

Sample: gps g = (3.4, 5.7);

crossroad c = g;

crossroad c1 = (3.4, as);

gps: The gps data type is a complex data type. It is very similar to crossroad.It takes longitude,latitude as an floating point number or integer.That data type shows a location on the map

Sample: gps g = (3.4, 5.7);

array: User can define arrays by simply putting dollar sign in front of a variable name.User can define every complex and primitve data types arrays. Here is the sample array declarations you can use:

Sample: road $r4 = {r2}; //r2 is simple road.

road $r3 = {((3.4, 5.7), (3.4, 5.7)), ((3.4, 5.7), (3.4, 5.7))};

road $r5 = {((3.4, 5.7), (3.4, 5.7)), r2};

crossroad $kk = {(3.4, 5.7), c, c1}; //c and c1 are simple crossroads

graph: The graph data type is complex data type.It consists of roads and crossroads.User can define graph in several ways.

Sample: graph g = {c2, r3}; //c2 is a crossroad and r3 is a road

graph g = {((3.4, 5.7), (3.4, 5.7)), (((3.4, 5.7), (3.4, 5.7)), ((3.4, 5.7), (3.4,5.7)))};

graph g = {((3.4, 5.7), (3.4, 5.7)), a};

user: The user data type is also a complex data type. It takes a string which is username and a location where user currently locates. Here is the sample user declaration you can use:

Sample: user ali = (“ali”, (3.7, 5.4));

LOOPS

for loop :

The for statement provides a compact way to iterate over a range of values. User often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows: . Here is the sample for declaration you can use:

```
for (initialization; termination; increment) {

        statement(s)

}
```

Sample:
```
for (i = 0 ; i < 10; i = i+1) {

        int v = 12;   }
```

while loop : The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```
while (expression) {

        statement(s)

}
```

Sample:
```
while (i < 12) {

i = i+1;    }
```

if-elif-else statements: Fibugo uses the usual flow control statements known from other languages.The most well-known statement type is the if statement.There can be zero or more elif parts, and the else part is optional. The keyword ‘elif’ is short for ‘else if’.

Sample:
```
if (i < 12) {

        i=10;
```

```
} else {

        i = 12;
```

LOGICAL OPERATORS

Fibugo have 3 different logical operators. These are **"&&","||"** and "**!**"**.** Logical operators can be used for conditional statements. It provides that combining statements and returning **true** or **false.**

- **&&** : This is **AND** operator. If both statements are true, it returns **true.**
- **||** : This is **OR** operator. If either of the statements are true, it returns **true.**
- **!:** This is **NOT** operator. It changes the boolean value to inverse. For instance, if statement is **true,** we apply **NOT** operator and it converts to **false.**

Sample:

```
bool b1 = true;

bool b2 = true;

if (b1 == true && b2 == true) {

        // This if block will be executed

}

b1 = false;

b2 = true;

if (b1 == true || b2 == true) {

        // This if block will be executed

        // b1 is false but b2 is true, so one of the conditions are true

}

bool b1 = true;

if ( !b1 ) {

        // This block will not be executed
} else {
        // This block will be executed
}
```

3D OBJECTS:

Fibugo have 8 different 3D objects. These are defined for our language's purpose. Our language provides some features which are specified for GPS structure, so these 3D objects give some flexibility to programmer while building new programs. These 3D objects are: **home, hospital, school, bridge, mall, busstop, hotel, postoffice.** You can define these objects in several ways. We are going to explain how you can define these objects.

home: home is used for defining home places on GPS. You can define 4 different ways.You can find different definitions below. It takes 2 parameters to create **home** object. The first parameter represents ***name,*** so it takes only **string** variable. The second parameter represents ***crossroad***, and it takes tuple. Tuple should takes 2 different **float** values. These are *longitude and latitude.* You can also give identifiers as a parameter. Be careful not using reserved words ,like int, float, road, while passing parameters to declaration.

Examples:

```
home h = ("Ali", (12.3, 43.5));

home h = ("Ali", coordinates);

home h = (name, (12.3, 43.5));

home h = (name, coordinates);

home h = (name, (x, y));       // x and y are identifier which in float type
```

hospital: hospital is used for defining hospital places on GPS. You can define 4 different ways. You can find different definitions below. It takes 2 parameters to create **hospital** object. The first parameter represents ***name of the hospital***, so it takes only **string** variable. The second parameter represents ***crossroad,*** and it takes tuple. Tuple should takes 2 different **float** values. These are *longitude* and *latitude.* You can also give identifiers as a parameter. Be careful not using reserved word like int, float, road while passing parameters to declaration.

Examples:

```
hospital h = ("Hacettepe", (32.45, 12.56));
```

hospital h = ("Hacettepe", coordinates);

hospital h = (hospitalName, (32.45, 12.56));

hospital h = (hospitalName, (x, y));

school: : school is used for defining school places on GPS. You can define 4 different ways. You can find different definitions below. It takes 2 parameters to create **school** object. The first parameter represents ***name of the school***, so it takes only **string** variable. The second parameter represents ***crossroad,*** and it takes tuple. Tuple should takes 2 different **float** values. These are *longitude* and *latitude.* You can also give identifiers as a parameter. Be careful not using reserved word like int, float, road while passing parameters to declaration.

Examples:

school s = ("Beytepe School", (132.45, 412.56));

school s = ("Beytepe School", coordinates);

school s = (schoolName, (132.45, 412.56));

school s = (schoolName, (x, y));

bridge: : bridge is used for defining bridges on GPS. You can define 4 different ways. You can find different definitions below. It takes 2 parameters to create **bridge** object. The first parameter represents ***name of the bridge***, so it takes only **string** variable. The second parameter represents ***crossroad,*** and it takes tuple. Tuple should takes 2 different **float** values. These are *longitude* and *latitude.* You can also give identifiers as a parameter. Be careful not using reserved word like int, float or road while passing parameters to declaration.

Examples:

bridge b = ("Golden Gate", (132.45, 412.56));

bridge b = ("Golden Gate", coordinates);

bridge b = (bridgeName, (132.45, 412.56));

bridge b = (bridgeName, (x, y));

mall: : mall is used for defining malls on GPS. You can define 4 different ways. You can find different definitions below. It takes 2 parameters to create **mall** object. The first parameter represents ***name of the mall***, so it takes only **string** variable. The second parameter represents ***crossroad,*** and it takes tuple. Tuple should takes 2 different **float** values. These are *longitude* and *latitude.* You can also give identifiers as a parameter. Be careful not using reserved word like int, float or road while passing parameters to declaration.

Examples:

mall m = ("Ankamall", (132.45, 412.56));

mall m = ("Ankamall", coordinates);

mall m = (mallName, (132.45, 412.56));

mall m = (mallName, (x, y));

busstop: : mall is used for defining busstops on GPS. You can define 4 different ways. You can find different definitions below. It takes 2 parameters to create **busstop** object. The first parameter represents ***name of the busstop***, so it takes only **string** variable. The second parameter represents ***crossroad,*** and it takes tuple. Tuple should takes 2 different **float** values. These are *longitude* and *latitude.* You can also give identifiers as a parameter. Be careful not using reserved word like int, float or road while passing parameters to declaration.

Examples:

bussstop bs = ("BeytepeBusStop", (132.45, 412.56));

bussstop bs = ("Ankamall", coordinates);

bussstop bs = (busstopName, (132.45, 412.56));

bussstop bs = (busstopName, (x, y));

hotel: : mall is used for defining hotel on GPS. You can define 4 different ways. You can find different definitions below. It takes 2 parameters to create **hotel** object. The first parameter represents ***name of the hotel***, so it takes only **string** variable. The second parameter represents ***crossroad,*** and it takes tuple. Tuple should takes 2 different **float** values. These are *longitude* and *latitude.* You can also give identifiers as a parameter. Be careful not using reserved word like int, float or road while passing parameters to declaration.

Examples:

hotel h = ("RixosHotel", (132.45, 412.56));

hotel h = ("RixosHotel", coordinates);

hotel h = (hotelName, (132.45, 412.56));

hotel h = (hotelName, (x, y));

postoffice: : mall is used for defining postoffice on GPS. You can define 4 different ways. You can find different definitions below. It takes 2 parameters to create **postoffice** object. The first parameter represents ***name of the postoffice***, so it takes only **string** variable. The second parameter represents ***crossroad,*** and it takes tuple. Tuple should takes 2 different **float** values. These are *longitude* and *latitude.* You can also give identifiers as a parameter. Be careful not using reserved word like int, float or road while passing parameters to declaration.

Examples:

postoffice po = ("BilkentPTT", (132.45, 412.56));

postoffice po = ("BilkentPTT", coordinates);

postoffice po = (postOfficeName, (132.45, 412.56));

postoffice po = (postOfficeName, (x, y));

User-Defined Functions: FiBugo provides flexibility about defining functions which is defined by user as on the other languages. You can define your own functions by following some of rules. Firstly, you have to start your function with **func** keyword. FiBuGo understands that this is a user-defined function. After that you should specify your function's data type. You can define your function with **char, int, float, bool** and **string** types. Also, if you want to define the function without return type, you should give **void** as a return type in your function. There is a restriction on defining function names. You should not use underscore on names. You can use camelCase convention in FiBuGo. Functions can take one or more parameters or they are not take any parameter. You can define a function with zero parameter. There are some function definitions below:

Examples:

```
func int intFunc(int a) {

        return 12;

}

func float floatFunc(float a) {

        return 12.3;

}
```

```
        func bool boolFunc(bool a) {
                return true;
        }
        func char charFunc(char c) {
                return c;
        }
        func void voidFunc() {
                print("Nothing.");
        }
        func string stringFunc(string s) {
                return "This is a string";
        }
```

Arithmetic Operators: FiBuGo allows you to make arithmetic operations such as *addition, subtraction, division, multiplication, power* and *modulus.* You can do these operations easily. Addition operator is +, subtraction is **-**, division is /**,** multiplication is *, power is ^ and modulus is **%.** Also, you can do operations both identifiers and literals. Here is the examples.

```
int a = 12 + 3;         // Addition
int a = 12 + i;         // Addition
int a = 13 - 9;         // Subtraction
int a = 13 - i;         // Subtraction
float a = 32.1 * 4.1;   // Multiplication
float a = 32.1 * i;     // Multiplication
int a = 32 / 4;         // Division
int a = 32 / i;         // Division
int a = 2 ^ 5;          // Power
int a = 2 ^ i;          // Power

int a = 34 % 7;         // Modulus

int a = 34 % i;         // Modulus
```

ERROR SITUATIONS: FiBuGo, handles some error situations and returns the error messages to the programmer.

1. **Divide by zero:** If you try to divide a number by zero, it returns **Divide by zero!** error message and prints out the console. Although FiBuGo catches the divide by zero error, it only returns this error for denominator is zero. If there is a identifier, it cannot be catched by Yacc, because it is compiler's duty.

   Example:
   ```
   int a = 12 / 0;          // This is error

   int a = 0;

   int b = 12 / a;          // This cannot be catched by Yacc
   ```

2. **Missing 'func', not a valid function declaration! :** This error would be handled if you do not start your function with **func** keyword. You have to start with **func.**

   Example:
   ```
   int tmpFunc(int a, int b) {          // Wrong function declaration

       a = a + b;

   }

   func int tmpFunc(int a, int b) {     // Correct function declaration

       a = a + b;

   }
   ```

3. **Not a valid expression! :** If you miss quotes or parentheses in your definition, you will get this error. Be careful about missing declarations.

   Example:
   ```
   int a = 12              // This will be occurred error

   int a = 12;             // True

   print("wrong print);         // Wrong. Missing quote

   print("wrong print";         // Wrong. Missing parantheses
   ```

4. **Invalid identifier declaration!:** If you use reserved word as a variable name, you will get error.

   Example: int float = 12;   // Wrong. float is reserved word.

5. CHALLENGES OF DEVELOPING A PARSER

-User cannot use increment or decrement operators in our languages such as i++,i--,i+=1.