İhsan Doğramacı Bilkent Üniversitesi

Bilkent

1984

screw

Muhammad Ramish Saeed_21503436_Section-2
Yassine Gazzah_21801164_Section-2
Dorra El Mekki_21801163_Section-2

PART A :

The complete BNF description of ”screw ” is shown below. The language constructs will be explained after the BNF description.

BNF

```
<stmt_list> ::= <stmt>
              | <stmt> <stmt_list>

<stmt> ::= <expr_assign>
         | <function_call>
         | <if_stmt>
         | <loop_stmt>
         | <output>
         | <primitive_fct>
         | <comment>


<expr_assign> ::= <var_name> <assign_operator> <var_assign>


<var_name> ::= <lowercase> { ( <letter> | <digit> | <special_char>) }


<lowercase> ::= 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


<letter> ::= <uppercase> | <lowercase>


<uppercase> ::= 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


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


<special_char> ::= “_”  | “@”


<assign_operator> ::= = | %= | *= | += | -= | /=


<var_assign> ::= <expr>
               | <constant_name>
```

```
                   | <var_name>
                   | <input>
                   | <truth_val>
                   | <number>
                   | <function_call>

<expr> ::= <arithmetic_stmt>
           | <logical_stmt>
           | ( <var_name> | <expr> )  <arithmetic_operator> ( <var_name> | <expr> )
           | ( <var_name> | <expr> )  <logical_operator>  ( <var_name> | <expr> )


<arithmetic_stmt> ::=  ( <var_name> | <number> ) <arithmetic_operator> ( <var_name> | <number> )


<arithmetic_operator> ::= + | - | * | / | %


<number> ::=  <int> | <float>


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


<float> ::= <int> ‘.’ <int>


<logical_stmt> ::=  (<var_name> | <boolean> | empty) <logical_operator>  (<var_name> | <boolean>)


<logical_operator> ::= <not_operator> | <and_operator>  | <or_operator>  | <xor_operator>


<not_operator> ::= !


<or_operator> ::= |@


<and_operator> ::= &&


<xor_operator> ::= ^
```

```
<constant_name> ::= <uppercase> { <uppercase> | <digit> }


<input> ::= in ‘(‘ {<alphanumeric>}* ‘)’


<alphanumeric> ::= <letter> | <digit> | <special_char>


<truth_val> ::= True | False


<function_call> ::= <function_name> ‘( <params> ‘)’


<params> ::= <param>
           | <param> ‘,’  <params>


<param> ::= <var_name> | <number>


<function_def> ::= define <function_name> ‘(‘ <args> ‘) {’ <stmt_list> <return>‘}’


<function_name> ::= <uppercase> { ( <letter> | <digit> ) }


<args> ::= <arg>
         | <arg> ‘,’  <args>


<arg> ::= <var_name>


<return> ::= <var_assign>


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

<matched> ::= if  (  <boolean>  ) <matched> else <matched>
            |{ <stmt_list> }

<unmatched> ::= if(  <boolean> )<if_stmt>
           | if(  <boolean> )<matched> else <unmatched>
```

```
<boolean> ::= <constant_name>
              | <relational_stmt>
              |<logical_stmt>

<const_assign> ::= define <constant_name> <truth_val>

<relational_stmt> ::=  ( <var_name> | <int> ) <relational_operator> ( <var_name> | <int> )

<relational_operator> ::= < | >  | == | != | <= | >=

<loop_stmt> ::= ‘loop (‘ <boolean> ‘) {‘ <stmt_list> ‘}’

<output> ::= out ‘( ‘  {<var_name>}*  “{(<alphanumeric> | <space>)}*” {<var_name>}* ‘ )’

<space> ::= “ “

<primitive_fct> ::= <move>
                  | <turn>
                  | <grab_object>
                  | <release_object>
                  | <read_sensor_data>
                  | <send_to_master>
                  | <receive_from_master>

<move> ::= ‘move (‘ <steps_num> ‘)’

<steps_num> ::= <var_name>  | <int>

<turn> ::= ‘turn (‘ <degrees_num> ‘,’ <direction> ‘)’

<degrees_num> ::= <var_name> | <int>
```

```
<direction> ::=  left | right


<grab_object> ::= ‘grab (‘ <object_name> ‘)’


<object_name> ::= <var_name>


<release_object> ::= ‘release (‘  <object_name> ‘)’


<read_sensor_data> ::= ‘read_sensor_data (‘ <sensor_ID> ‘)’


<sensor_ID> ::= <var_name>  | <int>


<send_to_master>  ::= ‘send_to_master  (‘ <data> ‘,’ <master_ID> ‘)’

<master_ID> ::= <var_name> | <int>


<receive_from_master>  ::= ‘receive_from_master  (‘ <data> ‘,’ <master_ID> ‘)’


<data> ::= <number>
            |<var_name>
            | <direction>
            | <direction> ‘,’ <data>
            | <number> ‘,’ <data>


<comment> ::= ‘#’  {(<alphanumeric> | <special_char> | <space>}+
```

Constructs description :

Explanation of language constructs are as follows:

<stmt_list> : This non-terminal is the representative of the statements that our language consists of. The statements of our language are the lists of the statements.

<**stmt**>: This is created to show the types of the statements that our language consists of. Therefore, the branching according to the statement types occurs after is terminal.

<expr_assign>: This defines the structure of an *assignment* expression.
It should be like this : <var_name> <assign_operator> <var_assign>
For example :
a%=2
the result will be 0 (can be divided by 2) or 1 (it can not).

<var_name> : This is created to begin variable names with lowercase letters and can also include digits.

<lowercase>: This is used to create small letters of the english alphabets.

<letter>: This defines what counts as a letter which are all the english alphabets and can be either uppercase or lowercase letters.

<uppercase>: This is used to create capital letters of the english alphabets.

<digit>: : This defines what counts as a digit, which are all the numbers from 0-9.

<assign_operator>: The %=, +=, -= and /= operators are *compound assignment operators*. They each access the value of a variable that is its left operand, perform a computation based on that value and the right operand, and then replace the original value of the variable with the result of the computation.

<var_assign> : This is used to assign a variable to an expression, a constant, an expression variable, input from user and boolean values.

<assign_operator>: The %=, +=, -= and /= operators are *compound assignment operators*. They each access the value of a variable that is its left operand, perform a computation based on that value and the right operand, and then replace the original value of the variable with the result of the computation.

<expr>: It can be an arithmetic statement or a logical statement
An arithmetic statement contains only arithmetic operators and operands.
A Logical statement is an Expression that uses conditions to return a true or false value.

<arithmetic_stmt>: Arithmetic Statements. The arithmetic statements are used for computations. individual operations are specified by the add, subtract, multiply, and divide statements. these operations can be combined symbolically in a formula, using the compute statement.

<arithmetic_operator>: An arithmetic operator is a mathematical function that takes two operands and performs a calculation on them.

Symbol	Symbol name	Meaning
+	Plus sign	addition
-	Minus sign	subtraction
*	Times sign	multiplication
/	Division slash	division
%	modulo	remainder calculation

<number>: Number is used to define signed (both positive and negative) integers and floats.

<int>: Int is used to define positive integers. It contains digits.

<float>: A number in which no fixed number of digits before and after the decimal point.

<logical_stmt>: A logical statement is a statement that, when true, allows us to take a known set of facts and infer (or assume) a new fact from them.

<logical_operator>: It defines four operators which are not_operator, and_operator, or_operator, xor_operator

<not_operator>: Its symbol is !=Example if (a != 0) : '*if a is not equal to zero*'.

<or_operator>: The logical **or_operator,** its symbol is |@, returns the boolean value true if either or both operands is true and returns false otherwise.

<and_operator>: The and_operator its symbol is &&, is a Boolean operator used to perform a logical conjunction on two expressions -- Expression 1 And Expression 2. AND operator returns a value of TRUE if both its operands are TRUE, and FALSE otherwise.

<xor_operator>: Exclusive or or exclusive disjunction **,**its symbol is ^, is a logical **operation** that outputs true only when inputs differ (one is true, the other is false).

<constant_name>: This is created to begin constant names with an uppercase letter and then we can use only uppercase letters or digits.

<input>: Standard input stream

<alphanumeric>: It contains letters or digits.

<truth_val>: It contains true or false value.

<function_call>: This is responsible for function callings, that has a identifier (which is its name actually) and function calling parameters inside.

<function_def>: It defines the **function's** name, return type, and parameters

<function_name>: This is created to begin function names with an uppercase letter and then we can use letters or digits.

<params>: It contains the function's parameters.

<param>: an argument of a function is a specific input in the function

<args>: It contains the function's arguments.

<arg>: It contains a variable name.

<if_stmt>: It takes an expression inside and does the corresponding statements in its statement part. Requires matched and unmatched parts to prevent the ambiguity

<matched>: If statement that is already have a matched else with itself.

<unmatched>: If statement that does not have a matched else with itself.

<boolean>: it can be logical expression or equality expression

<const_assign>: It defines a constant as false or true.

<relational_stmt>: This defines the comparison of two relational variable names.

<relational_operator>: This defines the operators greater than, less than, greater than or equal to, less than or equal to, equals to and not equals to.

<loop_stmt>: This denotes how to write loop statements, which is used to execute certain statements until a condition is met. We use the loop reserved word followed by a <boolean> within parenthesis, which is followed by statements within curly braces.

<output> : This is created to output alphanumerics to the console.

<primitive_fct>: This denotes the function used to program the robot for various actions such as move, turn, grab, release, read data from a sensor given the sensor ID, send and receive data from/to another robot or master.

<move>: This denotes the move step of robot to move by a defined number of steps.

<steps_num>: This denotes the parameter in the 'move' function which can either be a variable name or an integer.

<turn>: This denotes the turn function of robot's move which will be done by the reserved word turn provided with the parameter of the number of degrees to turn and the direction.

<degrees_num>: This denotes the parameter in the 'turn' function which can either be a variable name or an integer.

<direction>: This defines the left or right direction the robot can take.

<grab_object>: This denotes the function which uses the 'grab' reserved keyword to grab an object given as a parameter.

<object_name>: This defines the name of the object which can be any variable name.

<release_object>: This uses the 'release' keyword to release an object by the robot and has a parameter of object name.

<read_sensor_data>: This uses he read_sensor reserved keyword to allow robot to read data sent by signals with a sensor ID parameter.

<sensor_ID>: This denotes the parameter in the 'read_sensor_data' function which can either be a variable name or an integer.

<data>: This defines either a number, direction, direction and data together or number and data together.

<send_to_master>: This uses the 'send_to_master' reserved keyword to send data to robot with a parameter of data and master ID.

<master_ID>: This denotes the parameter in the 'send_to_master' function which can either be a variable name or an integer.

<receive_from_master>: This uses the 'receive_data_from_master' reserved keyword to receive data from robot with a parameter of data and master ID.

<comment>: This defines what counts as a comment, which are all ASCII characters except hash sign ('#'), the hash sign is reserved to start and end comments.

Reserved Words

The following are the reserved words for this language:

1. define → used to initialize constants and used in the definition of function.
2. return → used in the definition of function to return the result value.
3. if → conditional statement.
4. else → conditional statement.
5. elif → conditional statement.
6. loop → looping construct.
7. in → used to prompt user input.
8. out → used to print out to a console.
9. && → and operator.
10. |@ → or operator.
11. ! → not operator.
12. ^ → Xor operator.
13. >, >= → (greater) than and (greater than-equal to) operators.
14. <, <= → (less than) and (less than-equal to) operators.
15. == → equality operator.
16. + → addition operator.
17. - → subtraction operator.
18. * → multiplication operator.
19. / → division operator.
20. % → Modulus operator.
21. = → assignment operator.
22. %= → Modulus-assignment operator.
23. *= → multiplication-assignment operator.
24. += → addition-assignment operator.
25. -= → subtraction-assignment operator.
26. /= → division-assignment operator.
27. move → the primitive function for moving the robot 1 step (1 mm).
28. turn → the primitive function for turning the robot 1 degree.
29. grab_object → the primitive function for the robot to grab an object.

30. release_object → the primitive function for the robot to release a grabed object.
31. read_sensor_data → the primitive function for the robot to read data from sensors.
32. send_to_master → the primitive function for the robot to send data to a master.
33. receive_from_master → the primitive function for the robot to receive data from a master.
34. left, right → the directions that the robot can use, one of them at a time, when turning.
35. # → the symbol used to indicate the begin of a comment line.

Literals

Definition: literals are ; INTEGER, FLOAT, TRUTH_VAL.

Note on design

Our language uses polish notation in order to avoid issues with ambiguity, that result from associativity issues. Each line of comments start with a # symbol in our language. The use of a # makes the comments conspicuous and differentiate them from the code . For each # character, our program will print Comment. This makes them easier to locate for the programmer. The use of a # before comments also helps the lexical analyser to not mistake the comments as part of the code.

The string is represented by single quotation marks ‘ ’ and double quotation marks “ ” are used to represent variables.

Unlike C++ or Java, the programmer is forced to instantiate the constants with uppercase letters, variables with lowercase letter as a first character and functions with uppercase letter as a first character .

This not helps the readability of the program but more importantly it is the only way for the lexical analyser to differentiate between a constant, a variable and a function, as there is not a reserved word like 'const'in the language.
The use of reserved words like 'loop', 'in' and 'out' makes the purpose of their use fairly intuitive, unlike words like 'while' , ‘print’, ‘puts’ , 'system.in' etc.

PART B :

Lex Description.

```
%option main
%%
= printf("ASSIGN_VAR ");
define printf("ASSIGN_DEF ");
\+= printf("ASSIGN_ADD ");
-= printf("ASSIGN_SUB ");
\*= printf("ASSIGN_MULT ");
```

```
\/= printf("ASSIGN_DIV ");
\%= printf("ASSIGN_MOD ");
\+ printf("ADD ");
- printf("SUB ");
\* printf("MULT ");
\/ printf("DIV ");
\% printf("MOD ");
&& printf("AND ");
\|@ printf("OR ");
! printf("NOT ");
\^ printf("XOR ");
> printf("GT ");
\< printf("LT ");
== printf("EQ ");
>= printf("GT-EQ ");
\<= printf("LT-EQ ");
if printf("IF ");
else printf("ELSE ");
elif printf("ELSE_IF ");
loop printf("WHILE_LOOP ");
in printf("INPUT ");
out printf("OUTPUT ");
return printf("RETURN ");
\( printf("LP ");
\) printf(" RP ");
(True|False) printf("TRUTH_VAL ");
(left|right) printf("DIRECTION ");
[0-9]+.[0-9]+ printf("FLOAT ");
[0-9]+ printf("INTEGER ");
#([a-z]|[A-Z]|[0-9]|" ")*  printf("COMMENT ");
move printf("MOVE ");
turn printf("TURN ");
grap_object printf("GRAP ");
release_object printf("RELEASE ");
send_to_master printf("SEND_MASTER ");
receive_from_master printf("RECEIVE_MASTER ");
[a-z]([A-Za-z]|[0-9]|"_")* printf("VAR ");
[A-Z]([A-Z]|[0-9]|"_")* printf("CONSTANT ");
[A-Z]([A-Za-z]|[0-9]|"_")* printf("FUNCTION ");
'([A-Za-z]|[0-9]|” “)*' printf("STRING ");
```

PART C :

Example Program

```
#this is a comment
# this program is an example program to test the language constructs

# this is a definition of a constant with ‘True’ as value
define CONST1 True

# this is a definition of a constant with ‘False’ as value
define CONST2 False

#this is a definition of a function
define FunctionTest1 ( arg1 , arg2 ) {

#the following are assignment and operator statements
result = arg1
result += arg2
result -= 2
result *= 3
result /= 2
result %= 4

return result }

#the following are also assignment statements
varX = 5
varY = 8.4

#this is a call of the defined function
FunctionTest1 ( varX , 16)


#the following is an if statement
if ( varX < varY )
   { out (varY “is greater than “ varX) }
else { out (varX  “is greater than” varY)}


if (varX==5) {out(varY “is equal to 5”)}
if (varY!=5) {out(varY “is different from 5”)}


#the following is a loop  statement
varI=0
loop ( varI>=0 && varI<=10  )
```

```
{
varI +=1
out ("iteration n=" varI)
 }

# this is an example of using the Logic operators
varAndLogic = CONST1 && CONST2
varOrLogic = CONST1 |@ varAndLogic
varNotLogic = ! varOrLogic

### the following are an examples of using the primitive functions

## move function
steps = 6
# the robot going to move (value_of_steps) = 6 mm (6 steps)
move (steps)
# the robot going to move 9 mm (9 steps)
move (9)

## turn function
degrees = 8
# the robot going to turn (value_of_degrees) =  8
turn (degrees)
# the robot going to move 12 degrees
turn (12)

## grap function
objectName1 = 3
grap ( objectName1 )

## release function
objectName2 = 8
grap ( objectName2 )

## read_sensor_data function
sensorID=50
read_sensor_data ( sensorID )
read_sensor_data ( 100 )

## receive_from_master and send _to_master functions
masterID1 = 5189
data1= 3.15
receive_from_master ( data1 , masterID1 )
receive_from_master ( right , 501)

data2 =  right
varID = 3078
masterID2=varID
receive_from_master ( data2 , masterID2 )
send_to_master ( data2 , masterID2)

receive_from_master ( data1, data2 , masterID2 )
send_to_master ( data1 , left , masterID2)
```