BILKENT UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

CS315
Programming Languages

Prjct I – Report
Group 28

2F#

Deniz Dalkılıç
21601896
Section 2

Ahmet Ayrancıoğlu
21601206
Section 2

Kaan Gönç
21602670
Section 2

BNF Description

```
<program> ::= START <stmts> END

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

<stmt> ::= <non-block_stmt>? | <block_stmt>

<non_block_stmt> ::= <assignment_stmt> | <var_declaration> |  <return_stmt>
        | <function call statement>

<return_stmt> ::= return <expression>?

<assignment_stmt> ::= <left_hand_side> <assignment operator> <right_hand_side>

<left_hand_side> ::= <var_declaration> | <var_name>

<right_hand_side> ::= <expression>

<expression> ::= <conditional_expression> | <additive_expression>

<conditional_expression> ::= <or_expression>

<or_expression> ::= <and_expression>
        | <or_expression> || <and_expression>

<and _xpression> ::= <equality_expression>
        | <and_expression> &&  <equality_expression>

<equality_expression> ::= <relational_expression>
        | <relational_expression> <equality_operator> <relational_expression>
        | <additive_expression> <equality_operator> <additive_expression>

<relational_expression> ::= <additive_expression> <relational_operator>  <additive_expression>
```

```
            | <boolean_expression>

<boolean_expression> ::= <boolean> | <var_name> |  <function_call_stmt>
            | (<conditional_expression>)

<additive_expression> ::= <multiplication expression>
            | <additive-expression> <addition operator> <multiplication expression>

<multiplication_expression> ::= <primary_expression>
            | <multiplication_expression> <multipication_operator>
<primary_expression>

<primary_expression> ::= <var_name> | <primitive_type> |  <function_call_stmt>
            | (<additive_expression>)

<boolean> ::= right | wrong

<function_call_stmt> ::= <primitive_function_call> | <function_name> ( <parameter_input_list>? )

<function_name> ::= <upper_case_letter> <lower_case_word>

<primitive_function_call> ::=  <move_func> | <grab_func> | <relase_func> | <turn_func>
            | <read_func> | <send_func> | <recieve_func>

<move_func> ::= Move(<int>)

<grab_func> ::= Grab()

<release_func> ::= Release()

<turn_func> ::= Turn(<int>)

<read_func> ::=Read(<int>)

<send_func> ::= Send(<int>)
```

<receive_func> ::= Receive(<int>)

<multipication_operator> ::= * | / | %

<additive_operator> ::= + | -

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

<equality_operator> ::= =? | ~=?

<var_declaration> ::= <type name> <var_name> | <type_name> <var_name> = <expression>

<var_name> ::= <lower_case_word>

<parameter_input_list> ::= <expression> | <expression>, <parameter_input_list>

<block_stmt> ::= <if-then_stmt> | <if-then-else_stmt>| <loop_stmt>

<if_then_stmt> ::= if (<conditional_expression>) << <stmts> >>

<if_then_else_stmt> ::= if (<conditional_expression>) << <stmts> >> else << <stmts> >>

<loop_stmt> ::= <while_stmt> | <for_stmt>

<while_stmt> ::= while (<conditional_expression>) <body>

<for_stmt> ::= for (<var_initializer> ? <conditional_operation> ? <assignment_operation>) <body>

<function_declaration> ::= <function_header> < body>

<function_header> ::= <return_type> <function_declarator>

<return_type> ::= <type_name> | void

<function_declarator> ::= <function_name> (<parameter_list> ?)

<parameter_list> ::= <parameter> | <parameter_list>, <parameter>

<parameter> ::= <type_name> <var_name>

<body> ::= << <stmts>? >>

<primitive_type> ::= <int> |< float>

<type_name> ::= int | float | boolean

<float> ::= <int> . <natural_number>

<int> ::> <sign>? <natural_number>

<sign> ::= + | -

<natural_number> ::= <digit> | <digit> <natural_number>

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

<lower_case_word> ::> <lower_case_letter> <lower_case_word> | <lower_case_letter>

<upper_case_letter> ::= A | B | C | … | Z

<lower_case_letter> ::= a | b | c | … | z

<comment> ::= !! <comment_characters> !!

<comment_characters> ::= <comment_character> | <comment_characters><comment_character>

<comment_character> ::= <character>

<character> denotes Unicode character set.

Language Description

The boundaries of a program written in our 2F# language are defined with the reserved words “start” and “end”. Any code written without specifying these boundaries will not compile and give a syntax error. However, our language does not require a maın method unlike most languages as we are planning the robot to be controlled by sımple function calls allowing the users to write and execute very small programs. Moreover, to let the user easily identify the beginning and ending of the scopes such as method identifications and block-statements we use “<<” and “>>” which are metaphors of arrows facing each other. Method declarations have a similar structure with the Java programming language where we use a type and a function identifier followed by a parameter list to declare a function. Additionally, in our language we apply the universal mathematical rules so that the precedence of multiplication operators “*” and “/” are higher than the addition operators”+” and “-” . Also, the relational operators such as “<” , “>” , “<=” , “>=” have a higher precedence than the equality operators “=?” , “~=?” which also have a higher precedence than the conditional operators “&&” and “||” . This allows our language to be understood by anyone who has any knowledge ın mathematics. In terms of the types that our language supports, we provide only the types of int, float and boolean because these types are enough to accomplish the primitive tasks of the robot as well as performing new functions by using the different combinations of primitive functions. By simplifying the possible types in our language, we ensured that our language is not unnecessarily complicated. By considering our preferences in coding, we decided that all methods should start with a capital letter and followed by lowercase letters whereas a variable name can only be written with lowercase letters. Since our language is not an Object Oriented Language this increases the readability of the code written by distinguishing between variables and functions and creates a homogeneity in all programmes written in our language. Also, we restricted the usage of numbers in variable and function names as we think it will reduce the confusion in understanding the written code as well as leading the coder to define the variables and the functions in a more clear way.

All reserved words: for, while, if, else, return, void, int, float, boolean, Move, Turn, Grab, Release, Send, Receive, Read, start, end.

<program> : Represents the combination off all constructs used in a program. In order to define a program the lower constructs of this expression which are stmts (statements) should be used between the reserved words START and END so that different programs can be identified separately.

<stmts> ::= Determines how individual statements can be organized in our program. <stmts> can either be a single <stmt> or it can be yet another group of statements followed by a single statement. Which is left recursive allowing the users to write as many statements as they want.

<stmt> ::= This is an abstraction used in our language to represent all possible statements, which is the main non-recursive element in our language, that can be put together. We have two kind of <stmt>, non block statement and block statement.

<non-block_stmt> ::= This represents every statement that can be written in a single line which could be variable declaration, an assignment statement, a return statement or a function call statement.

<return_stmt> ::= Defines what can be returned inside a function using the reserved word **return**.

<assignment_stmt> ::= Defines how a value at the right hand side of the assignment operator can be copied to the variable at the left hand side of the assignment operator.

<left_hand_side> ::= Represents the all possible things that can be on the left hand side of the assignment operator. In our language it can be either a variable declaration or an existing variable represented by variable name.

<right hand side> ::= Represents the all possible things that can be on the right hand side of the assignment operator which are expressions.

<expression> ::= Represents all possible expressions that can be written. In our case it can either be an additive expression or a conditional expression.
<conditional expression> ::= Represents the type of expression which can result in booleans. Because we have operator precedence, our highest priority operation is at the lowest level inside the conditional

expression and our lowest priority operation is at the highest level inside the conditional expression which is or operation represented by or expression.

<or expression> ::= Defines what an “or expression” can be. An “or expression” can either be an “and expression” which is higher than an “or expression” in our order of precedence, or it can be an “or operation” between an “or expression” and an “and expression”. Which makes our “or expression” left recursive meaning that users can chain multiple “or expression” after each other infinitely.

<and expression> ::= Defines what an “and expression” can be. An “and expression” can either be an “equality expression” which is higher than an “and expression” in our order of precedence, or it can be an “and operation” between an “and expression” and an “equality expression”. Which makes our “and expression” left recursive meaning that users can chain multiple “and expression” after each other infinitely.

<equality expression> ::= Defines what an “equality expression” can be. An “equality expression” can either be a “relational expression” which is higher than an “equality expression” in our order of precedence, or it can be an “equality operation” between two “relational expression” or it can be an “equality operation” between two “additive expression”. We separated equality expression into two parts because we did not want relational expressions to be comparable with additive expressions.

<relational expression> ::= Defines what a relational expression is. A relational expression could be simply a boolean expression, or it can be a relation operation between two additive expressions.

<boolean expression> ::= A boolean expression is simply all the things that can hold a boolean value such as variables and function calls.

<additive expression> ::= Defines what a additive expression could be. Additive expression could be a multiplication expression if there is no operation because multiplication comes before addition in order of operation in our language. It could also be used to represent an addition operation between an additive expression and a multiplication expression using the addition operator. This makes additive expressions

left recursive meaning that users can write multiple additive expression after each
other.

<multiplication expression> ::= Defines what a multiplication expression could be. Multiplication expression could be simply a primary expression if there is no operation or it could be used to represent a multiplication operation between a multiplication expression and a primary expression using the multiplication operator. This makes multiplication operation left recursive meaning that multiple multiplication expression can be written after each other.

<primary_expression> ::= Defines what a primary expression can be. A primary expression represents the smallest expression in our language. It can be anything that can hold primitive values such as variables, primitive types or function calls.

<boolean> ::= It is used to represent a boolean. A boolean can only hold 2 values, it can either be right or it can be wrong. This abstraction helps to increase the readability of the language.

<function_call_stmt> ::= Function call statement is used to call functions. It can either be used to call a primitive function or a function that is declared by the user.

<function_name> ::= Defines how to write a function name. Functions have to be named according to the rule which is it should start with a uppercase letter and should be followed by lower case word.

<primitive_function_call> ::= This is used to represent primitive functions which is predefined.

<multipication_operator> ::= Defines the multiplication operators of the language, which are "*", "/", "%" for multiplication, division and finding the remainder after the division respectively.

<additive_operator> ::= Defines the additive operator which are + and -, for addition and subtraction respectively.

<relational_operator> ::= Defines the relational operator which is used to explain the relations expressions have with each other.

<equality_operator> ::= Defines the equality operator which is used to check equality between expressions. The operators are “=?” and “~=?” . One returns the equality whereas the other returns the opposite outcome of the equality respectively.

<var_declaration> ::= Defines how a variable is declared.

<var_name> ::= Represents variables using its name which can be written in all lower case letters.

<parameter input list> ::= This represents the parameters that a we can pass into a function. It can be a single expression or it can be a single expression followed by yet another parameter input list with by commas in between.

<block_stmt> ::= This represents statement that require or can require other statements to function, such as loops and if statements. These statements have initial statements followed by body statement which could have multiple statements.

<if-then-stmt> :: Defines how to make an if statement which includes a conditional check by using a <conditional operation> between parentheses. If the given condition is met, the body of the id statement is executed, otherwise it is ignored.

<if-then-else_stmt> ::= Defines how to make an if statement that is matched by and else statement. This statement is similar to a regular if statement in that if the condition is met the body of the if statement is executed. However, if the condition given is not met, the body of the if statement is ignored and the body of the else statement is executed.

<loop_stmt> ::= Defines what a loop statement is. A loop statement can be either a while statement or a for statement.

<while_stmt> ::= Defines how to write a while statement which is used to execute written statements inside the body until a certain condition is met. In our language **while** reserved word is used followed by a conditional statement inside brackets to define the terminating condition. There is a also the body of the while loop, which holds the statement, after the conditional statement.

<for_stmt> ::= Defines how to write a for statement which is used to execute written statements inside the body certain number of times specified by using initializing a function, modifying its value each and checking for the termination condition in each iteration . In our language **for** reserved word is used followed by a var initialization, conditional operation and assignment operation separated by semicolons inside brackets to define the terminating condition. There is a also the body of the for loop, which holds the statement, after the conditional statement.

<function_declaration> ::= Defines how to declare a function to be used later in the program by calling the function. Itis done by first specifying the function header followed by the body of the function.

<function_header> ::= It is used to specify what the function will return and it also contains the declaration of the function.

<return_type> ::= Defines what a function can return when it is called. The result can either be types represented by type names or it can be **void** to represent nothing.

<function_declarator> ::= Defines how to declare a function by defining the functions name and its parameters.

<parameter_list> ::= Left recursive definition of parameter to identify a sequence of parameters that will be used in a function declaration.

<parameter> :: Denotes what is accepted as a input to a function declaration. A parameter is defined by giving the parameter a type and a name.

<body> ::= Represent the entirety of a functions or a block statement’s body. Which can be either empty meaning <<>>, or it could have <stmts> inside the arrows to program functionality.

<primitive_type> ::= Denotes the type of variables such as int or float which is used in mathematical expressions.

<type_name> ::= This represents the name of the available types in our language. They are used for declaring variables and functions. These are all reserved words which are **int, float, boolean.**

<float> ::= Represents floating point number.

<int> ::= Represents integers.

<sign> ::= This is used to make floats and integers positive and negative.

<natural number> ::= Represent natural numbers.

<digit> ::= Represents digits.

<lower_case_word> ::= Represents a word made with lower case letters.

<upper_case_letters> ::= Represents uppercase letters.

<lower_case_letter> ::= Represents lower case letters.

<comment> ::= Represents comments that can be made in the programme. Comments can have every possible characters in them with the condition that they are in between double “!”.