Bilkent University

Fall 2024-2025

CS-315 Project 2: flyScript

Team_47

Rayan HABCHI	22401201	Section 1
Emrehan Ates	22003465	Section 3
Edip Kerem Tayhan	21903708	Section 2

10 November 2024

Part A - Revised and Augmented Language Design

```
<start> ::= start <stmts> finish

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

<stmts> ::= <comment>
          | <matched>
          | <unmatched>
          | <return>


<matched> ::= <if_stmt>
            | <assignment>;
            | <loop_stmt>
            | <function_call>;
            | <drone_command>;
            | <io_statement>;

<unmatched> ::= <if_stmt_unmatched>

<if_stmt> ::= if( <logic_expr> ){ <stmts> } else { <stmts> }

<if_stmt_unmatched> ::= if( <logic_expr> ){ <stmts> }

<assignment> ::= <identifier> = <expression>
               | <array_assignment>;

<array_assignment> ::= <identifier>[<expression>] = {<var_list>}

<expression> ::= <expression> <sign> <term>
               | (<expression>)
               | -<term>
              | !<term>
               | <term>

<term> ::= <term> <op> <factor>
        | <factor>

<factor> ::= <var_list>
          | <function_call>
          | <array_access>

<array_access> ::= <identifier>[<expression>]

<function_def> ::= function <identifier>(<var_list>) {<stmts>}
```

```
                    | function <identifier>() {<stmts>}

<function_call> ::= <primitive_call>
                  | <identifier>(<expression>)
                  | <identifier>()

<primitive_call> ::=  input(<identifier>)
                  | output(<expression>)
                  | getTime()
                  | getCurrentHeading()
                  | getCurrentHeight()
                  | getCurrentAltitude()
                  | connectToUrl()

<drone_command> ::= climb( <double> )
                  | drop()
                  | moveForward( <double> )
                  | moveBackward( <double> )
                  | stopVertical()
                  | stopHorizontal()
                  | turnLeft(<degree>)
                  | turnRight(<degree>)
                  | sprayOn()
                  | sprayOff()

<return> ::= return <expression>;

<loop_stmt> ::= <for_loop>
              | <while_loop>

<for_loop> ::= for (<assignment>; <logic_expr>; <assignment>) {<stmts>}

<while_loop> ::= while (<logic_expr>) {<stmts>}

<logic_op> ::= <= | == | >= | < | >

<logic_expr> ::= <var> <logic_op> <var>
               | <logic_expr> && <logic_expr>
               | <logic_expr> || <logic_expr>

<var_list> ::= <var>
             | <var>, <var_list>

<var> ::= <identifier>
        | <number>
        | <double>
        | <str_const>
        | <degree>
```

```
<degree> ::= <number>d
           |  <float>d

<identifier> ::= <alphabetic>
               | <alphabetic><alphanumeric>

<double> ::= <number>
           | <number> "." <number>

<number> ::= <digit>
           | <digit><number>

<alphanumeric> ::= <valid_ident_char>
                 | <valid_ident_char><alphanumeric>

<valid_ident_char> ::= <alphabetic>
                     | <digit>

<str_const> ::= “<str_chars>”

<str_chars> ::= <valid_str_char>
              | <valid_str_char><str_chars>
              | “\n” <str_chars>

<valid_str_char> ::= <valid_ident_char>
                   | "!" | "#" | "%" | "½" | "'" | "(" | ")"
                   | "*" | "+" | "," | "-" | "." | "/" | "£"
                   | "^" | "€" | ";" | "<" | "=" | ">" | "?"
                   | "[" | "₺" | "]" | "{" | "|" | "}"
                   | "~" | "é" | "@" | "\" | "ß"

<comment> ::= /*<str_chars>*/

<op> ::= * | / | % | && | “||”

<sign> ::= "+" | "-"

<digits> ::= <digit>
           | <digit><digits>

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

<alphabetic> ::= "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" | "_"
```

Our language includes all the required elements, with the exception of a separate declaration rule. Since we designed our language to be dynamically typed, the assignment rule inherently covers declarations. This approach was chosen to keep the language straightforward and accessible for end users, who are likely technicians or engineers without extensive programming backgrounds.

We have made some adjustments to the lex file to improve the clarity and functionality of the lexical analysis for our language. These changes were aimed at enhancing token recognition and ensuring a smoother integration with the corresponding parsing processes.

According to the feedback we added functionalities :

- Else ifs
- Multiline comments
- Function definition without parameters
- Function call with expressions as args
- Boolean operations and logical NOT

Structure of the language

Here's a general description of the structure and key nonterminals of the language based on the provided BNF and lex rules:

1. **Program Structure**:
    - The program structure is defined by the `<start>` and `<stmts>` nonterminals.
    - The program starts with the `start` token, followed by a sequence of statements `<stmts>`, and ends with the `finish` token.
    - The `<stmts>` nonterminal can contain various types of statements, including `<empty>`, `<comment>`, `<matched>`, `<unmatched>`, and `<return>`.
2. **Statements**:
    - `<matched>` statements include `if-else` blocks, assignments, loops, function calls, drone commands, and I/O statements.
    - `<unmatched>` statements are `if` statements without an `else` block.
    - `<return>` statements allow returning values from functions.
3. **Expressions and Variables**:
    - `<expression>` represents arithmetic expressions with various operators (`+`, `-`, `*`, `/`, `%`).
    - `<term>` and `<factor>` define the building blocks of expressions, including variables, function calls, and array accesses.
    - `<var>` can represent various types of values, such as identifiers, numbers, doubles, strings, and degrees.
    - `<array_assignment>` and `<array_access>` handle array-related operations.
4. **Functions**:
    - `<function_def>` defines user-defined functions with a list of parameters.
    - `<function_call>` invokes both user-defined and built-in functions.
5. **Loops**:
    - `<for_loop>` and `<while_loop>` define the two types of loops in the language.
6. **Logic and Comparison**:
    - `<logic_expr>` handles boolean expressions with logical operators (`&&`, `||`) and comparison operators (`<=`, `==`, `>=`, `<`, `>`).
7. **Primitives and I/O**:
    - The language provides a set of built-in primitive functions for drone control and I/O operations, such as `climb`, `drop`, `input`, `output`, etc.
8. **Identifiers and Constants**:
    - `<identifier>` represents variable and function names, following standard naming conventions.
    - Various types of constants are supported, including `<number>`, `<double>`, `<degree>`, and `<str_const>`.
9. **Comments**:

    - `<comment>` handles multi-line comments using the `/*` and `*/` delimiters.
10. **URLs**:
    - The language supports both secured (`https://`) and unsecured (`http://`) URLs, which are recognized as `<securedUrl>` and `<unsecuredUrl>` tokens, respectively.

The key aspects of the language's syntax and semantics are captured by these nonterminals. By understanding how they are defined and used in the grammar, someone reading your report should be able to comprehend and parse programs written in this language.

The language has a few notable rules and features:

1. **Precedence and Associativity**: The precedence and associativity of operators in expressions are defined by the grammar rules. For example, the `*`, `/`, and `%` operators have higher precedence than `+` and `-`.
2. **Array Assignments and Accesses**: The language allows assigning values to array elements using the `<array_assignment>` rule, and accessing array elements using the `<array_access>` rule.
3. **Matched and Unmatched `if` Statements**: The language distinguishes between `<matched>` and `<unmatched>` `if` statements, where the former have a corresponding `else` block, and the latter do not.
4. **Built-in Primitive Functions**: The language provides a set of built-in primitive functions for drone control and I/O operations, which are recognized using the `<primitive_call>` rule.
5. **URL Support**: The language can recognize and parse both secured and unsecured URLs, which are treated as distinct token types (`<securedUrl>` and `<unsecuredUrl>`).

Conflicts Unresolved and How to Run the Program

There are no conflicts unresolved left in the yacc program. Any example program can be executed by following the steps:
1. Make
2. ./parser < CS315_24F_Team_47_1.txt
3. ./parser < CS315_24F_Team_47_1_syntax_error.txt
4. ./parser < CS315_24F_Team_47_2.txt
5. ./parser < CS315_24F_Team_47_2_syntax_error.txt
6. ./parser < CS315_24F_Team_47_3.txt
7. ./parser < CS315_24F_Team_47_3_syntax_error.txt
8. ./parser < CS315_24F_Team_47_4.txt
9. ./parser < CS315_24F_Team_47_4_syntax_error.txt

in the linux shell. (Assuming .txt files are in the same directory with lex and yacc files)