Bilkent University

CS 315

Dice

Revised and Augmented Language Design, BNF and Example Program

<u>Group Members</u>

Turan Mert Duran (21601418), Sec 02

Mohammed S. Yaseen (21801331), Sec 02

Mohammad Elham Amin (21701543), Sec 02

<u>**Fall 2020-2021**</u>

BNF Description

Program

```
<Program>        ::= <statement_List>
```
Non-terminal token that shows our program consists of statements.

```
<statement_List> ::= <statement> <semicolon>
                   | <statement> <semicolon> <statement_list>
                   | <line_comment> <statement_list>
                   | <block_comment> <statement_list>
```
Non-terminal token that shows our statements can be one or more than one.

```
<statement>      ::= <declaration_statement>
                   | <assignment_statement>
                   | <loop_statement>
                   | <function_definition>
                   | <input_statement>
                   | <output_statement>
                   | <expression>
```

Non-terminal token that shows our statement can be declaration, assignment, function call, expression and input or output statements.

Statements

```
<declaration_statement> ::= <variable> <identifier>
             | <variable> <identifier> <assignment_operator> <expression>
```
Non-terminal token that shows how to declare a variable in 2 different ways.

```
<assignment_statement> ::= <identifier> <assignment_operator> <expression>
```
Non-terminal token that shows how assignment statement is constructed

```
<conditional_statement> ::= if <LP> <expression> <RP> <block>
                          | if <LP> <expression> <RP> <block> else
                            <block>
```
Non-terminal token that shows how conditional statement is constructed

----

Expressions

```
<expression>  ::= <arithmetic>
                | <relational>
                | <not_expression>
```

Non-terminal token that shows what is an expression

```
<arithmetic>  ::= <arithmetic> <add_op> <mult_div>
                | <arithmetic> <sub_op> <mult_div>
                | <mult_div>
```

Non-terminal token that shows how to create an arithmetic expression. It includes a <mult_div> token to prioritize the multiplication and division over addition and subtraction.

```
<mult_div>    ::= <mult_div> <mul_op> <in_paranthesis>
                | <mult_div> <div_op> <in_paranthesis>
                | <mult_div> <mod_op> <in_paranthesis>
                | <in_paranthesis>
```

Non-terminal token that shows how the multiplication and division are calculated. It includes a <in_paranthesis> token to prioritize the parentheses over all other operations.

```
<in_paranthesis> ::= <LP> <arithmetic> <RP>
                 | <number>
                 | <identifier>
                 | <function_call>
```

Non-terminal token that shows how to calculate the values in parenthesis. In parentheses there can be an arithmetic expression again or it can be just a number or an identifier.

```
<relational> ::= <in_paranthesis> <relational_operator> <in_paranthesis>
```

Non-terminal token that shows how to create a relational expression.

```
<relational_operator> ::= <LT>
                      | <LTE>
                      | <GT>
                      | <GTE>
                      | <and>
                      | <or>
                      | <equal_to>
                      | <not_equal_to>
```

Terminal token that shows the supported relational operators.

```
<not_expression> ::= <NOT_OP> <in_paranthesis>
                   | <NOT_OP> <LP> <relational> <RP>
```

Non-terminal token that shows that not is available in our programming language. { Ex: !(5>4) }

Function Declaration and Function Call

```
<function_definition> ::= <function_header> <function_body>
```
Non-terminal token that shows how functions are defined in the language. A function consists of a <function_header> and <function_body>

```
<function_header>     ::= function <function_signature>
```
Non-terminal token that shows how the function header is defined. Function header must start with the keyword ‘function’ followed by <function_signature>>

```
<function_signature>  ::= <identifier> <LP> <parameter_list> <RP>
                        | <identifier> <LP> <RP>
```
Non-terminal token that shows how <function_signature> is defined. Function signature consists of a name for the function followed by optional parameters inside parenthesis.

```
<parameter_list>      ::= <parameter_list> <comma> <identifier>
                        | <identifier>
```
This is a non-terminal token that shows how function parameters are defined. Function parameters are identifiers, as parameter names, separated by comma.

```
<function_body>       ::= <block>
                        | <semicolon>
```
Non-terminal token that shows function body is defined as a <block> or semicolon

```
<block>               ::= <LB> <statement_List> <RB>
                        | <LB> <RB>
```
Non-terminal that shows how a block of statements is defined. A block is defined as an optional <statement_list> inside curly braces.

```
<function_call>       ::= <function_name> <LP> <argument_list> <RP>
                        | <function_name> <LP> <RP>
```
Non-terminal that shows how function calls are defined. A function call is defined as the name of the function followed by <argument_list>, if any required, inside parenthesis.

```
<function_name>       ::= <identifier>
                        | <builtin_function_id>
```
Non-terminal shows that function name can be either an identifier or a builtin function name.

```
<argument_list>       ::= <expression>
                        | <expression> <comma> <argument_list>
```

Non-terminal shows that while calling functions, expressions can be called inside parentheses.

Input and Output Statement

<input_statement> ::= input <in_op> <expression> <LP>
Non-terminal that shows how the input statement is defined. Input statement is defined by an 'input' keyboard followed by expression.
{Ex: (input >> x)}

<output_statement> ::= print <out_op> <expression>
Non-terminal that shows how the output statement is defined. Output statement is defined by an 'output' keyboard followed by expression.
{Ex: (print << x)}

Loops

```
<loop_statement> ::= <while>
                   | <for>
```
This non-terminal shows two kinds of loops supported by the language.

<while> ::= while <LP> <expression> <RP> <block>

None-terminal token for the while loop. While loop requires <conditional_statements> to be satisfied for executing statements provided in <statement_list>.

<for> ::= for <LP> <declaration_statement> <semicolon> <expression> <semicolon> <assignment_statement> <RP> <block>

None-terminal token for for-loop. For loop requires an <declaration_statement> as loop variable initializer, a <expression> as the predicate, and an <assignment_statement> which is updated in every loop.

Comments

```
<line_comment>         ::= <line_comment_ident> <line_comment> <sentence>
                       ::= <line_comment_ident> <sentence>
```
A non-terminal token which defines line comments. It can be a single word or a sentence.

```
<block_comment>        ::= <block_comment_start> word <block_comment_end>
                         | <block_comment_start> <block_comment> <word>
                           <block_comment_end>
```

Another non-terminal token for describing multiline comments. A multiline comment can be a single line or multiple lines.

Numbers and Sentences

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

Non-terminal token that shows numbers.

```
<digit>                ::= 0 | <non_zero_digit>
```

Terminal statement that can be either zero or a no-zero digit, used for creating numbers.

```
<non_zero_digit>       ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This is a terminal statement. Used for creating numbers.

```
<sentence>             ::= <sentence><word>
                         | <sentence><digit>
                         | <word>
```

Non-terminal token that show how sentences are defined in the language. A sentence can be single word, a word with digits or more than one word and digit

```
<word>                 ::= <alphabet>|<digit>

                       | <word> <alphabet>|<digit>
```

Non-terminal that shows that words are defined as one or more alphabet with zero or more digits.

```
<identifier>           ::= <identifier> <alphabet>
                       | <identifier> <number>
                       | <alphabet>
```

Non-terminal token which is used for naming variables and functions. An identifier may consist of one or more alphabets and zero or more digits. Identifiers must start with a letter.

```
<alphanumeric>         ::= <alphabet> <alphanumeric>
                       | <digit> <alphanumeric>
                       | <digit>
                       | <alphabet>
```

Non-terminal that is a combination of alphabets and numbers.

Primitive Functions

```
<builtin_function_id>  ::= takeoff | land | flip_left | flip_right
                         | flip_front | flip_back | go | go_up | go_down
                         | go_forward | go_backward | go_left | go_right
                         | rotate_c | rotate_cc | video_on | video_off
                         | take_pic | emergency_stop | hover | set_speed
                         | set_wifi | get_altitude | get_temperature
                         | get_speed | get_acceleration | get_inclination
                         | get_time | get_battery | connect
```

Terminal tokens that are primitive builtin functions in the language.

takeoff A primitive void function that used for taking off the Tello

Land A primitive void function that used for landing the Tello

flip_left A primitive void function that used for flipping left the Tello

flip_right A primitive void function that used for flipping right the Tello

flip_front A primitive void function that used for flipping front the Tello

flip_back A primitive void function that used for flipping back the Tello

go A primitive void function that takes 4 parameter x,y,z coordinates and speed respectively and let the drone fly through coordinate with the given speed

go_up A primitive void function that used for increasing the altitude of the Tello according to the value that is given in parentheses and in between 20-500

go_down A primitive void function that used for decreasing the altitude of the Tello according to the value that is given in parentheses and in between 20-500

go_forward A primitive void function that makes Tello move forward according to the value that is given in parentheses and in between 20-500

go_backward A primitive void function that makes Tello move backward according to the value that is given in parentheses in between 20-500

go_left A primitive void function that makes Tello move left according to the value that is given in parentheses in between 20-500

go_right A primitive void function that makes Tello move right according to the value that is given in parentheses in between 20-500

rotate_c A primitive void function that rotates Tello clockwise according to the degree whose value is given in parentheses in between 1-360

rotate_cc A primitive void function that rotates Tello counter clockwise according to the degree whose value is given in parentheses in between 1-360

video_on A primitive void function that turns the video on of Tello

video_off A primitive void function that turns the video off of Tello

take_pic A primitive void function that allows Tello to take picture

emergency_stop A primitive void function that stops motor of the Tello immediately

hover A primitive void function that keeps Tello stable in the air

set_speed A primitive void function that sets speed of Tello to the given value that is given in parentheses

set_wifi A primitive void function that sets name and password Wifi of Tello

get_altitude A primitive function that returns altitude value of Tello

get_temperature A primitive function that returns temperature value of Tello

get_speed A primitive function that returns speed value of Tello

get_acceleration A primitive function that returns acceleration value of Tello

get_inclination A primitive function that returns inclination degree of Tello

get_time A primitive function that returns current time

get_battery A primitive function that returns battery situation in percentage value of Tello
connect A primitive function that connects Tello to the controlling computer

----

Terminals

```
<alphabet>             ::= 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
```
Terminal tokens that are used for creating <words>, <sentences> and <identifiers>.

```
<variable>             ::= var
```
Terminal statement which is used for variable declarations.

```
<line_comment_ident>   ::= //
```
Terminal statement used for defining a line comment.

```
<block_comment_start>  ::= /*
```
Terminal statement used for defining start of multiline comment.

```
<block_comment_end>    ::= */
```
Terminal statement used for defining end of multiline comment.

```
<symbol>               ::= <LP>
                       | <RP>
                       | <semicolon>
                       | <underscore>
                       | <assignment_operator>
                       | <dot>
                       | <space>
                       | <LCB>
                       | <RCB>
                       | <string_ident>
                       | <char_ident>
```
Terminal statement that shows symbols defined by the language.

```
<LP>                   ::= (

<RP>                   ::= )

<LCB>                  ::= {

<RCB>                  ::= }

<semicolon>            ::= ;
```

```
<assignment_operator>  ::= =

<dot>                  ::= .

<space>                ::= " "

<string_ident>         ::= "\""

<char_ident>           ::= "\'"

<or>                   ::= ||

<and>                  ::= &&

<not>                  ::= !

<equal_to>             ::= ==

<not_equal_to>         ::= !=

<LT>                   ::= <

<GT>                   ::= >

<LTE>                  ::= <=

<GTE>                  ::= >=

<mul_op>               ::= *

<div_op>               ::= /

<add_op>               ::= +

<sub_op>               ::= -

<mod_op>               ::= %

<comma>                ::= ,

<in_op>                ::= >>

<out_op>               ::= <<

---------------------------------------------------------------------------
```

Reserved words

for a word to define for loops

while a word to define while loops

if a word to define if statements

else a word to define the else part of if statement

var a word used to define variables

function a word used to define functions

Nontrivial Tokens

Comments in Dice are defined in two types, line comment and multiple comment. Line comments are started by two forward slashes followed by any number of words or sentences. '/*' indicates the start of multiple lines and it's ended by '*/'. A multiple line can contain any number of anything. Comments are very important in terms of readability. A programmer needs to use comments for better documenting the code they've written. Therefore, it is very important in terms of readability.

Identifiers are defined as a combination of words and numbers. An identifier cannot start with a digit and it must have at least one letter. Starting with a letter has been a convention for all programming languages and is important for readability and reliability. As in any other language there are a bunch of reserved words too. Reserved words are very important in terms of analyzing the syntax.

Language Evaluation

Our new programming language is going to be used for a drone called Tello. Thus, we tried to create a new language that can be used for it and tried to make it simple to use and understandable for programmers. We tried to include all necessary functionality needed for drones. We added many important functions, according to the SDK of Tello, to be able to use almost all of these functionalities. Since the language is very similar to most of the popular programming languages as well as the heavy simplification (i.e.

not having types) that we did, it is very readable and writable. On the other hand, the functionality is limited and there is a built in support for the main drone functionality, that makes the language reliable as well.

Test Program

```
set_wifi(TelloWifiName, pass1234);
connect();
takeoff();

var userName;
print << PleaseEnterYourName;
input >> userName;
print << WelcomeTello;
print << userName;
video_on();
print << VideoIsOn;
print << PleaseEnterMaxAltitude;

var maxAltitude = 100;
// Max Altitude is 100 by default
input >> maxAltitude;
var a = 10;
// Funcltion definition
function increaseAcceleration(){
    var currentAltitude = get_altitude();
      if( currentAltitude > 100){
            var currentAccelaration = get_acceleration();
            currentAccelaration = currentAccelaration + 5;
            var speed = get_speed();
            speed = speed + currentAccelaration;
            set_speed(speed);
      };
};

//Function Call
increaseAcceleration();
//TODO CHANGE SMTHGS HERE
// Increase acceleration until reaching max Altitude
while ( get_altitude() != maxAltitude){
    increaseAcceleration();
};

// If - else test case
if (get_temperature() >= 40 ) {
    emergency_stop();
} else {
    flip_back();
};


while (get_time() < 12) {
      hover();
      if( get_battery() < 10){
                  land();
      };
};

//For Test
for ( var i = 4; i != 10; i = i + 1) {
      i = i + 1;
};

for( var distance = 0; distance != totalDistance; distance = distance + 1 )
{
      if( distance < 50){
            flip_left();
      }else{
            if( distance < 100){
                  flip_front();
            }else{
                  if( distance < 150){
                        flip_back();
                        take_pic();
                  }else{
                        flip_right();
                  };
            };
      };
      go_forward( speed);

      if( get_acceleration() < 5){
            speed =               speed +                1;
      };


      if( temperature           >            80 )      {
            land();
            emergency_stop();
      };
};

// variable declaration test
```

```
a = 5*2+5-10-(10/2);

//TODO CHECK MLINES COMMENTS
/*
Block Comment Test
Helo
*/

land();
video_off();
```