CS315
Project 2 Report

Name of the Programming Language:
HoneyBadger

Group 34

Onurcan Ataç 22002194 CS315-02
İlker Özgen 21902719 CS315-02
Bora Yılmaz 22003359 CS315-02

Table of Contents

BNF Description of HoneyBadger

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

<stmt> ::= <declaration_stmt>
      | <assign_stmt>
      | <conditional_stmt>
      | <loop_stmt>
      | <break_stmt>
      | <func_def_stmt>
      | <func_call_stmt>
      | <comment>

<comment> ::= EOL_COMMENT

<declaration_stmt> ::= <data_type> <var_name> SC
      | <data_type> <assign_stmt>
      | <data_type> <logic_identifier> SC

<data_type> ::= INT_TYPE
     | FLOAT_TYPE
     | CHAR_TYPE
     | STRING_TYPE
     | BOOL_TYPE

<var_name> ::= IDENTIFIER

<assign_stmt> ::= <var_name> ASSIGN_OP <expression> SC
      | <var_name> ASSIGN_OP <func_call_stmt>
      | <logic_identifier> ASSIGN_OP <logic_expression> SC
      | <logic_identifier> ASSIGN_OP <func_call_stmt>
      | <var_name> ASSIGN_OP STRING SC
      | <var_name> ASSIGN_OP CHAR SC

<expression> ::= <arithmetic_expression>
      | <logic_expression>

<arithmetic_expression> ::= <arithmetic_expression> PLU_OP <arithmetic_term>
      | <arithmetic_expression> MIN_OP <arithmetic_term>
      | <arithmetic_term>

<arithmetic_term> ::= <arithmetic_term> DIV_OP <arithmetic_factor>
      | <arithmetic_term> MUL_OP <arithmetic_factor>
      | <arithmetic_factor>
```

```
<arithmetic_factor> ::= LP <arithmetic_expression> RP
      | INT
      | FLOAT
      | <var_name>


<logic_expression> ::= <logic_expression> OR_OP <logic_term>
      | <logic_term>

<logic_term> ::= <logic_term> AND_OP <logic_factor>
      | <logic_factor>

<logic_factor> ::= LP <logic_expression> RP
      | <comparison_expression>
      | <logic_val>
      | <logic_identifier>

<logic_identifier> ::= LOGIC_IDENTIFIER

<comparison_expression> ::= <arithmetic_expression>
<comparison_op> <arithmetic_expression>

<logic_val> ::= TRUE
      | FALSE

<comparison_op> ::= EQ
      | NEQ
      | LT
      | LTE
      | GT
      | GTE

<conditional_stmt> ::= <if_stmt>
      | <if_else_stmt>

<if_stmt> ::= IF LP <logic_expression> RP LCB
<non_func_def_stmt_list> RCB
      | IF LP <logic_expression> RP LCB RCB

<if_else_stmt> ::= <if_stmt> ELSE LCB <non_func_def_stmt_list> RCB
      | <if_stmt> ELSE LCB RCB

<non_func_def_stmt_list> ::= <non_func_def_stmt>
      | <non_func_def_stmt> <non_func_def_stmt_list>
```

```
<non_func_def_stmt> ::= <declaration_stmt>
      | <assign_stmt>
      | <conditional_stmt>
      | <loop_stmt>
      | <break_stmt>
      | <func_call_stmt>
      | <comment>

<loop_stmt> ::= <while_loop>

<while_loop> ::= WHILE LP <logic_expression> RP LCB
<non_func_def_stmt_list> RCB
      | WHILE LP <logic_expression> RP LCB RCB

<rtrn_stmt> ::= RETURN <expression> SC

<break_stmt> ::= BREAK SC

<func_def_stmt> ::= <non_void_func_def_stmt>
      | <void_func_def_stmt>

<non_void_func_def> ::= <data_type> <func_name> LP <param_list> RP
LCB <func_stmt_list> <rtrn_stmt> RCB
      | <data_type> <func_name> LP RP LCB <func_stmt_list>
<rtrn_stmt> RCB

<func_name> ::= IDENTIFIER

<void_func_def_stmt> ::= VOID <func_name> LP <param_list> RP LCB
<func_stmt_list> RCB
      | VOID <func_name> RP LP LCB <func_stmt_list> RCB

<param_list> ::= <param>
      | <param> COMMA <param_list>

<param> ::= <data_type> <var_name>

<func_stmt_list> ::= <func_stmt>
      | <func_stmt> <func_stmt_list>;

<func_stmt> ::= <declaration_stmt>
      | <assign_stmt>
      | <conditional_stmt>
      | <loop_stmt>
      | <break_stmt>
      | <func_call_stmt>
      | <comment>
```

```
<func_call_stmt> ::= <func_name> LP <input_list> RP SC
      | <func_name> LP RP SC
      | <prim_func_stmt>

<input_list> ::= <input>
      | <input> COMMA <input_list>

<input> ::= <expression>
      | STRING
      | <func_call_stmt>

<prim_func_stmt> ::= <read_temp>
      | <read_hum>
      | <read_air_pressure>
      | <read_air_quality>
      | <read_light>
      | <read_sound>
      | <read_timer>
      | <connect_url>
      | <send_int>
      | <receive_int>
      | <set_switch>
      | <get_switch>

<read_temp> ::= READ_TEMP LP RP SC
<read_hum> ::= READ_HUM LP RP SC
<read_air_pressure> ::= READ_AIR_PRESSURE LP RP SC
<read_air_quality> ::= READ_AIR_QUALITY LP RP SC
<read_light> ::= READ_LIGHT LP RP SC
<read_sound> ::= READ_SOUND LP RP SC
<read_timer> ::= READ_TIMER LP RP SC

<connect_url> ::= CONNECT_URL LP STRING RP SC
<send_int> ::= SEND_INT LP INT COMMA STRING RP SC
<receive_int> ::= RECEIVE_INT LP STRING RP SC

<set_switch> ::= <set_switch1>
      | <set_switch2>
      | <set_switch3>
      | <set_switch4>
      | <set_switch5>
      | <set_switch6>
      | <set_switch7>
      | <set_switch8>
      | <set_switch9>
      | <set_switch10>
```

```
<set_switch1> = SET_SWITCH1 LP <logic_expression> RP SC
<set_switch2> = SET_SWITCH2 LP <logic_expression> RP SC
<set_switch3> = SET_SWITCH3 LP <logic_expression> RP SC
<set_switch4> = SET_SWITCH4 LP <logic_expression> RP SC
<set_switch5> = SET_SWITCH5 LP <logic_expression> RP SC
<set_switch6> = SET_SWITCH6 LP <logic_expression> RP SC
<set_switch7> = SET_SWITCH7 LP <logic_expression> RP SC
<set_switch8> = SET_SWITCH8 LP <logic_expression> RP SC
<set_switch9> = SET_SWITCH9 LP <logic_expression> RP SC
<set_switch10> = SET_SWITCH10 LP <logic_expression> RP SC

<get_switch> ::= <get_switch1>
      | <get_switch2>
      | <get_switch3>
      | <get_switch4>
      | <get_switch5>
      | <get_switch6>
      | <get_switch7>
      | <get_switch8>
      | <get_switch9>
      | <get_switch10>

<get_switch1> = GET_SWITCH1 LP RP SC
<get_switch2> = GET_SWITCH2 LP RP SC
<get_switch3> = GET_SWITCH3 LP RP SC
<get_switch4> = GET_SWITCH4 LP RP SC
<get_switch5> = GET_SWITCH5 LP RP SC
<get_switch6> = GET_SWITCH6 LP RP SC
<get_switch7> = GET_SWITCH7 LP RP SC
<get_switch8> = GET_SWITCH8 LP RP SC
<get_switch9> = GET_SWITCH9 LP RP SC
<get_switch10> = GET_SWITCH10 LP RP SC
```

Language Constructs

Non-Terminal Literals

<program>

It starts the language script. It contains a list of statements.

<stmt_list>

It is either a single statement or multiple statements. It is a left-recursive construct.

<stmt>

It is a single statement. It can be one of: declaration statement, assign statement, conditional statement, loop statement, break statement, function definition statement, function call statement, or comment statement.

<comment>

It contains a single line comment.

<declaration_stmt>

It is used to declare a variable. It can assign a value to the variable as well.

<data_type>

It contains various data types like int, float, string, char and bool.

<var_name>

It is an identifier to determine a variable name except boolean variables.

<assign_stmt>

It is used to assing a value to a variable.

<expression>

It can be either a logic expression or arithmetic expression.

<arithmetic_expression>

It is either addition expression, subtraction expression, or term.

<arithmetic_term>

It is either a multiplication expression, a division expression, or a factor.

<arithmetic_factor>

It can be either an arithmetic expression in parentheses, an int, a float, or a variable.

<logic_expression>

It can be a comparison expression, logic value, or a comparison of these with logic expressions.

<logic_term>

It is either a logic expression with AND operation, or a logic term.

<logic_factor>

It can be either an logic expression in parentheses, a comparison expression, a logic value, or a logic identifier.

<logic_identifier>

It is an identifier to determine boolean variables. It has to start with $.

<comparison_expression>

It contains two arithmetic expressions with a comparison operation between them.

<logic_val>

It is either true or false.

<comparison_op>

It can be either one of the operators <, >, <=, >=, ==, !=.

<conditional_stmt>

It is used to control logic mechanisms. It can be either an if block or an if-else block.

<if_stmt>

It is used as one of the logic control mechanisms. It contains a logic expression in parentheses and a statement block.

<if_else_stmt>

It is used as one of the logic control mechanisms. It contains a logic expression in parentheses and a statement block, and another else statement block.

<non_func_def_stmt_list>

It is similar to the statement list but it does not contain function definition statement.

<non_func_def_stmt>

It is similar to the statement but it cannot be a function definition statement.

<loop_stmt>

It is used to control the loop functionality. It contains the while-loop.

<while_loop>

It contains a logic expression in parentheses and a statement block.

<rtrn_stmt>

It is used to return a value in a function definition.

<break_stmt>

It is used to exit from a loop block.

<func_def_stmt>

It is used to define a function. It can be either a void function or a non-void function.

<non_void_func_def>

It is used to define a non-void function. It contains the return type, function name, parameter list (if exists) in parentheses, and a statement block with a return statement.

<func_name>

It is an identifier to determine the function name.

<void_func_def_stmt>

It is similar to the non-void function definition statement but it contains the void keyword instead of return type.

<param_list>

It is either a single parameter or multiple parameters.

<param>

It is a single parameter that is used in function definition statement.

<func_stmt_list>

It is similar to the statement list but it does not contain function definition statement.

<func_stmt>

It is similar to the statement but it cannot be function definition statement.

<func_call_stmt>

It is used to call a user-defined or a primitive function.

<input_list>

It is either a single input or multiple inputs.

<input>

It is a single input that is used in a function call statement as parameter value.

<prim_func_stmt>

It contains the primitive functions of the language.

<read_temp>

It is a primitive function. It returns the temperature from the sensor.

<read_hum>

It is a primitive function. It returns the humidity from the sensor.

<read_air_pressure>

It is a primitive function. It returns the air pressure from the sensor.

<read_air_quality>

It is a primitive function. It returns the air quality from the sensor.

<read_light>

It is a primitive function. It returns the light from the sensor.

<read_sound>

It is a primitive function. It returns the sound level from the sensor. It takes frequency as input.

<read_timer>

It is a primitive function. It returns the time from the timer.

<connect_url>

It is a mechanism to define a connection to the given URL. It takes a URL as input.

<send_int>

It is a mechanism to send an integer value at a time, to a connection. It takes an int as input.

<receive_int>

It is a mechanism to receive an integer value at a time, from a connection. It returns the received int.

<set_switch>

It turns the switch on or off. It contains 10 separate functions for each switch.

<set_switch1>

Sets switch 1.

<set_switch2>

Sets switch 2.

<set_switch3>

Sets switch 3.

<set_switch4>

Sets switch 4.

<set_switch5>

Sets switch 5

<set_switch6>

Sets switch 6.

<set_switch7>

Sets switch 7.

<set_switch8>

Sets switch 8.

<set_switch9>

Sets switch 9.

<set_switch10>

Sets switch 10.

<get_switch>

It returns either 0 or 1 according to the switch's value. It contains 10 separate functions for each switch.

<get_switch1>

Gets the value of switch 1.

<get_switch2>

Gets the value of switch 2.

<get_switch3>

Gets the value of switch 3.

<get_switch4>

Gets the value of switch 4.

<get_switch5>

Gets the value of switch 5.

<get_switch6>

Gets the value of switch 6.

<get_switch7>

Gets the value of switch 7.

<get_switch8>

Gets the value of switch 8.

<get_switch9>

Gets the value of switch 9.

<get_switch10>

Gets the value of switch 10.

Terminals

=

Assignment operator. Used to assign a value to a variable.

+

Addition operator. Used to add variables and constants.

-

Subtraction operator. Used to subtract variables and constants.

*****

Multiplication operator. Used to multiply variables and constants.

/

Division operator. Used to divide variables and constants.

,

Comma. Used to separate parameters or inputs in a function definition or call.

()

Parentheses. Used to group expressions.

{}

Curly brackets. Used to reserve a block for statements.

<

Less than operator.

>

Greater than operator.

<=

Less than or equal to operator.

>=

Greater than or equal to operator.

==

Equal operator.

!=

Not equal operator.

&&

And operator. Used in logic expressions.

||

Or operator. Used in logic expressions.

//

Comment operator. Used to initiate a single line comment.

;

Semicolon. Used at the end of the statements.

Non-Trivial Tokens

Comments

Comments are initialized with “//” sequence and span until the line ends. Comments are in end-of-line format, so they can come after a statement or at the beginning of a line. They allow the programmers to note/specify their information and code descriptions. Comments enhance the readability and simplicity of HoneyBadger.

Identifiers

Identifiers are used as function or variable names. Identifiers can take lowercase letters (a-z), uppercase letters (A-Z), dollar sign ($) and the underscore (_) as the first character. For the remaining characters, identifiers also take digits (0-9) accompanied by the characters which could be taken as the first character. Identifiers enhance the readability and writability of HoneyBadger.

Literals

int

Type int is an integer number. It can be positive or negative. It does not contain decimals. It consists of integer digits.

float

Type float is a floating point number. It contains decimal digits additional to the integer part. Integer part and decimal part are separated with a comma (.). Integer part may be empty.

char

Type char represents exactly a single character or an escape sequence ((\) followed by a character). Chars are written between single quotation marks (‘’). It is separated by identifiers and strings by those single quotation marks. It cannot be empty. Escape sequences include: \0, \”, \’, \\, \n, \t.

string

Type string consists of consecutive characters which are between double quotation marks (“”). It’s separated by identifiers and chars by those double quotation marks.

bool

Type bool is simply true or false. The name of a boolean variable has to start with $.

Reserved Words

if

Used in conditional statements. Allows the programmer to create logic control mechanisms. Enhances the functionality of HoneyBadger.

else

Used in conditional statements. Allows the programmer to create logic control mechanisms. Enhances the functionality of HoneyBadger.

true

Used to indicate “true” value of a bool variable, a logic expression, or a comparison expression. Enhances the functionality of HoneyBadger.

false
Used to indicate “false” value of a bool variable, a logic expression, or a comparison expression. Enhances the functionality of HoneyBadger.

while
Used in while loop mechanisms to improve iterations. Enhances the functionality of HoneyBadger.

return
Used in non-void function definitions to return a value. Enhances the functionality and reliability of HoneyBadger.

break
Used in loops to exit the block. Enhances the functionality and reliability of HoneyBadger.

void
Used to specify that the function will not return a value. Enhances the functionality of HoneyBadger.

readTemp
Used as a primitive function to read the temperature from the sensor. Enhances the functionality of HoneyBadger for IoT devices.

readHumidity
Used as a primitive function to read the humidity from the sensor. Enhances the functionality of HoneyBadger for IoT devices.

readAirPressure
Used as a primitive function to read the air pressure from the sensor. Enhances the functionality of HoneyBadger for IoT devices.

readAirQuality
Used as a primitive function to read the air quality from the sensor. Enhances the functionality of HoneyBadger for IoT devices.

readLight
Used as a primitive function to read the light from the sensor. Enhances the functionality of HoneyBadger for IoT devices.

readSound
Used as a primitive function to read the sound level from the sensor. Enhances the functionality of HoneyBadger for IoT devices.

readTimer
Used as a primitive function to read the time from the timer. Enhances the functionality of HoneyBadger for IoT devices.

setSwitch (includes the setSwitch functions from 1 to 10)
Used as a primitive function in order to turn the switches on/off. Enhances the functionality of HoneyBadger for IoT devices.

getSwitch (includes the getSwitch functions from 1 to 10)
Used as a primitive function in order to get values from different input switches. The function returns the value of the specified switch as 0 or 1. Enhances the functionality of HoneyBadger for IoT devices.

connectURL
Used as a primitive function in order to define a connection to a given URL string. Enhances the functionality of HoneyBadger for IoT devices.

sendInt
Used as a primitive function in order to send an integer value from the URL connection. It takes an integer to be sent and a specific URL address to which the integer will be sent as inputs. Enhances the functionality of HoneyBadger for IoT devices.

receiveInt
Used as a primitive function in order to receive an integer value from the URL connection. It takes a specific URL address as an input from which the integer will be received. Enhances the functionality of HoneyBadger for IoT devices.

The Rules Adopted by HoneyBadger

- For arithmetic operations, the regular mathematical precedence rules are applied.
- For logic expressions, the precedence is handled. AND operations are performed before OR. Additionally, logic expressions with parentheses are performed before others.
- For if-else statements, curly brackets are used to avoid ambiguity.
- For loop statements, curly brackets are used to avoid ambiguity.
- For function definition statements, curly brackets are used to avoid ambiguity.

Concern on Language Evaluation Criteria

Writability and Readability

We included various features to increase the writability and readability of HoneyBadger. There are comments which allow the programmers to write understandable and clean code, which enhances readability. We used identifiers to increase writability and readability.

The mechanisms such as if-else structures and loops are similar to most popular programming languages in the world, which makes it even easier to write and read HoneyBadger for even starter programmers. Declaration statements require specifying the data type, which increases readability. Similarly, functions require a return type during declaration.

Reliability

It's apparent that there is a trade-off between writability, readability and reliability. For example, we specified data types while declaring variables in our language, which increased reliability while decreasing writability to some extent. However, we visioned HoneyBadger as a middle point, a language that is both relatively easy to write and read, while also being relatively reliable. Additionally, since functions require a return type during declaration, HoneyBadger became more reliable.

Functionality

We used many mechanisms to maintain HoneyBadger's functionality. We added conditional statements, loops, keywords like break, and functions. We also included several primitive functions to increase the functionality of HoneyBadger for IoT devices.

Simplicity

In terms of simplicity, we did not use any complex mechanism blocks. We made the primitive functions easy to understand and use. We also included comments which enhanced the simplicity of the language.

Updates After Feedback on Project 1

- Previously, there was an ambiguity regarding sending and receiving integers from URL connections.There may be multiple URL connections and it wasn't certain that which one will be used for those functions. We fixed the issue by setting a specific URL address string as an input in those functions.
- We didn't have the functions for setting switches. Functions to set each switch separately are added in this version.
- Our previous functions for reading and setting switches had the number of the switch as an integer input. If the user provided any integer lower than 1 or bigger than 10, that caused a problem since the switch didn't exist. Thus, we implemented separate reading and setting functions dedicated to each switch in order to solve that problem.
- In the function that allows the user to get sound level from the sensor, an input for frequency was provided. Since sensors do not take inputs, that input was removed.
- Logical expression precedence problem is handled. The AND (&&) operation is now performed before OR (||) operation.