CS315

Project 2 Report

Fall 2022

Programming Language: *Quirk*

Team 41 Members:

Zeynep Doğa Dellal 22002572 Section 3

Kardelen Ceren 22003017 Section 3

Instructor: Karani Kardaş

Teaching Assistants:

Dilruba Sultan Haliloğlu, Hamza İslam, Onur Yıldırım

Contents

Complete BNF Description

1. Program Structure

<program> → <begin> <statements> <end>

<statements> → <statement> | <statement> <statements> | <comment> | <comment> <statements>

<statement> → <matched_statement> | <unmatched_statement>

2. Statements

<matched_statement> → if <LP> <control_expression> <RP> <LC> <matched_statement> <RC> else <LC> <matched_statement> <RC> | <non_if_statement>

<unmatched_statement> → if <LP> <control_expression> <RP> <LC> <matched_statement> <RC> | if <LP> <control_expression> <RP> <LC> <unmatched_statement> <RC> | if <LP> <control_expression> <RP> <LC> <matched_statement> <RC> else <LC> <unmatched_statement> <RC>

<non_if_statement> → <expression> <semicolon> | <loops> | <function_declaration>

<expression> → <assignment> | <control_expression> | <declaration> | <declaration_and_initialization> | <connection_to_URL> | <send_value> | <empty>

3. Condition Control Statements

<control_expression> → <logical_expression> | <not> <logical_expression> | <string_logical_expression> | <not> <string_logical_expression>

<logical_expression> → <logical_expression ><or><term1> | <term1>

<term1> → <term1><and><term2> | <term2>

<term2> → <term2><equal><term3> | <term2><not_equal><term3> | <term2><equal> <string> | <term2><not_equal> <string> | <term3>

<term3> → <term3><greater><term4> | <term3><less><term4> | <term3>< greater_or_equal ><term4> | <term3><less_or_equal><term4> | <term4>

<term4> → <term4><add_op><term5> | <term4><sub_op><term5> | <term5>

<term5> → <term5><mul_op><term6> | <term5><div_op><term6> | <term6>

<term6> → <term6><pow_op><term7> | <term7>

<term7> → <arithmetic_element> | <LP><logical_expression><RP>

<arithmetic_element> → <identifier> | <integer> | <float> | <function_call> |
<primitive_function_call> | <boolean>

<string_logical_expression> → <string> <equal> <string> | <string> <equal> <identifier> |
<string> <not_equal> <string> | <string> <not_equal> <identifier>

4. Expressions

<declaration> → <type> <identifier>

<assignment> → <identifier> <assign_op> <URL> | <identifier> <assign_op>
<control_expression> | <identifier> <assign_op> <receive_operation>

<declaration_and_initialization > → <type> <identifier> <assign_op> <URL> | <type>
<identifier> <assign_op> <control_expression> | <type> <identifier> <assign_op>
<receive_operation

<connection_to_URL> → connect <identifier> | connect <URL>

<send_value> → send <identifier> <identifier> | send <integer> <identifier> | send
<function_call> <identifier> | send <primitive_function_call> <identifier> | send
<identifier> <URL> | send <integer> <URL> | send <function_call> <URL> | send
<primitive_function_call> <URL>

<receive_operation> → receive <identifier> | receive <URL>

5. Loops

<loops> → <while_loop> | <for_loop>

<for_loop> → for <LP> <assignment><semicolon> <control_expression><semicolon>
<expression> <RP> <LC> <statements> <RC> | for <LP>
<declaration_and_initialization><semicolon> <control_expression><semicolon>
<expression> <RP> <LC> <statements> <RC>

<while_loop> → while <LP> <control_expression> <RP> <LC> <statements> <RC>

6. Functions

<function_declaration> → declare <identifier> <LP> <params> <RP> <LC> <statements> <return> <RC> | declare <identifier> <LP> <params> <RP> <LC> <statements> <RC>

<params> → <type> <identifier> <comma> <params> | <type> <identifier> | <empty>

<function_call> → <identifier> <LP><call_params><RP>

<call_params> → <identifier> <comma> <call_params> | <identifier> | <empty>

<return> → return <string> | return <control_expression>

7. Primitive Functions

<primitive_function_call> → <time> | <temperature> | <humidity> | <air pressure> | <air quality> | <light> | <sound level> | <get_switch_state> | <change_switch_state> | <connect_to_internet> | <disconnect_from_internet>

<time> → get_time<LP><RP>

<temperature> → get_temperature <LP><RP>

<humidity> → get_humidity<LP><RP>

<air pressure> → get_air_pressure<LP><RP>

<air quality> → get_air_quality<LP><RP>

<light> → get_light<LP><RP>

<sound level> → get_sound_level<LP><float><RP> | get_sound_level<LP><identifier><RP>

<connect_to_internet> → connect_to_internet<LP><RP>

<disconnect_from_internet> → disconnect_from_internet<LP><RP>

<get_switch_state> → get_switch_state<LP><integer><RP> | get_switch_state<LP><identifier><RP>

<change_switch_state> → change_switch_state<LP><boolean><comma><integer><RP> | change_switch_state<LP><boolean><comma><identifier><RP> |

change_switch_state<LP><identifier><comma><integer ><RP> |

change_switch_state<LP><identifier><comma><identifier><RP>

8. Comments

<comment> → <long_comment> | <line_comment>

<long_comment> → <long_comment_indicator> <sentence> <long_comment_indicator>

<line_comment> → <line_comment_begin> <sentence> <new_line>

9. Symbols and Constants

<type> → int | float | boolean | string | URL
<begin> → begin
<end> → end
<LP> → (
<RP> →)
<LC> → {
<RC> → }
<dot> → .
<underscore> → _
<assign_op> → =
<comma> → ,
<semicolon> → ;
<new_line> → \n
<and> → &
<or> → |
<not> → !
<equal> → ==
<not_equal> → !=
<less_or_equal> → <=
<greater_or_equal> → >=
<greater> → >
<less> → <
<empty> → “”
<add_op> → +
<sub_op> → -
<div_op> → /
<mul_op> → *

<pow_op> → ^

<long_comment_indicator> → %%%

<line_comment_begin> → %%

<sign> → +|-

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

<character> → a | b | c | … | x | y | z | A | B | C | … | X | Y | Z

<symbol> → . | , | ; | : | (|) | { | } | [|] | / | \\ | = | < | > | \| | & | “ | ‘ | + | - | * | # | \n | ! | ? | _ | £ |
^ | % | | $ | @ | ~

<integer> → <digit> | <digit><integer> | <sign> <digit> | <sign> <digit> <integer>

<float> → <integer><dot><integer>

<string> → <quote><sentence><quote>

<boolean> → true | false

<scheme> → http | https | ftp | mailto | data | file | irc

<URL> → <single_quote><scheme>:<sentence><single_quote> |
<single_quote><scheme>://<sentence><single_quote>

<element> → <character> | <digit> | <symbol>

<sentence> → <sentence><element> | <element> | <empty>

<identifier> → <identifier><character> | <identifier><digit> | <identifier><underscore> |
<character>

Explanation of BNF Description

1. Program Structure

<program> → <begin> <statements> <end>

This is a non-terminal that indicates that in Quirk, the program begins with a begin statement, ends with an end statement, and consists of statements. This is the core of the program.

<statements> → <statement> | <statement> <statements> | <comment> | <comment> <statements>

This non-terminal indicates that the program consists of a statement, statements or comments. Additionally, it shows that statements and comments coexist.

<statement> → <matched_statement> | <unmatched_statement>

This non-terminal shows that the statements above can be either matched or unmatched, depending on whether the if statements are matched with an else statement. Details of matched and unmatched are written below.

2. Statements

<matched_statement> → if <LP> <control_expression> <RP> <LC> <matched_statement> <RC> else <LC> <matched_statement> <RC> | <non_if_statement>

Matched statement non-terminal is used for if statements that are matched with an else statement. This separation between matched and unmatched statements solves the issue of ambiguity in if or if-else statements, where sometimes it is unclear which “if” an “else” belongs to. In Quirk, an if statement requires parentheses for the control expression and curly brackets for the statements inside. A matched statement can also be a non-if statement, which is any statement without an “if” such as assignments, loops, function definitions, etc.

<unmatched_statement> → if <LP> <control_expression> <RP> <LC> <matched_statement> <RC> | if <LP> <control_expression> <RP> <LC> <unmatched_statement> <RC> | if <LP> <control_expression> <RP> <LC> <matched_statement> <RC> else <LC> <unmatched_statement> <RC>

This non-terminal describes that an unmatched statement has more if statements than else statements, hence the name. They either have an “if” without an “else” or contain another unmatched statement within an “else” block. Unmatched statements, along with matched statements, help identify which if statement an else statement is connected to.

<non_if_statement> → <expression> <semicolon> | <loops> | <function_declaration>

This statement is used in every statement that is not if or if-else. Every single non-if action is perceived as this non-terminal in its core.

<expression> → <assignment> | <control_expression> | <declaration> |
<declaration_and_initialization> | <connection_to_URL> | <send_value> | <empty>

This non-terminal includes operations that are necessary to write a program for IoT. It has built-in functions that were added to support abstraction.

3. Conditional Control Statements

<control_expression> → <logical_expression> | <not> <logical_expression> |
<string_logical_expression> | <not> <string_logical_expression>

Control expression non-terminal indicates possible expression combinations that return a boolean value which can be used in if, while and for statements or arithmetic expressions that return integer or float values. It accounts for logical expressions' inverse versions as well.

<logical_expression> → <logical_expression ><or><term1> | <term1>
<term1> → <term1><and><term2> | <term2>
<term2> → <term2><equal><term3> | <term2><not_equal><term3> | <term2><equal>
<string> | <term2><not_equal> <string> | <term3>
<term3> → <term3><greater><term4> | <term3><less><term4> | <term3>< greater_or_equal
><term4> | <term3><less_or_equal><term4> | <term4>
<term4> → <term4><add_op><term5> | <term4><sub_op><term5> | <term5>
<term5> → <term5><mul_op><term6> | <term5><div_op><term6> | <term6>
<term6> → <term6><pow_op><term7> | <term7>
<term7> → <arithmetic_element> | <LP><logical_expression><RP>

The non-terminals above show how logical and arithmetic operators are used. Operator precedence increases with the term level, i.e., it is as follows from lowest to highest: or(||), and(&), equal(==)/not_equal(!=), greater(>)/less(<)/greater_or_equal(>=)/less_or_equal(<=), add(+)/subtract(-), multiply(*)/divide(/), power(^), parantheses(()). All logical and arithmetic operators are left associated, with the exception of "power" operator. This operation is right-associated, as is customary in mathematics (e.g. 5^2^3 = 5^8).

<arithmetic_element> → <identifier> | <integer> | <float> | <function_call> | <primitive_function_call> | <boolean>

This non-terminal indicates that arithmetic elements either can be identifiers, numbers, booleans, primitive functions' return variables or functions' return variables that conventionally store a number or a boolean. Arithmetic elements can be used with basic logical operations defined above.

<string_logical_expression> → <string> <equal> <string> | <string> <equal> <identifier> | <string> <not_equal> <string> | <string> <not_equal> <identifier>

This non-terminal shows that string literals and identifiers pointing (conventionally) to a string-type variable can only be compared with equal or not_equal operators. Other logical operators like "and" or "greater" have no meaning between strings in Quirk.

4. Expressions

<declaration> → <type> <identifier>

This non-terminal indicates how to declare a variable, which is by adding the identifier (the variable's name) next to the variable's type. Variables must be declared with their types before use.

<assignment> → <identifier> <assign_op> <URL> | <identifier> <assign_op> <control_expression> | <identifier> <assign_op> <receive_operation>

This non-terminal describes how to assign a value to a variable represented by its identifier. Another variable's value may be copied to the target variable (there are no reference assignments/aliases to reduce complexity). A variable may be assigned a literal value, the result of an arithmetic operation, a logical expression, or their combination through the control expression, or an integer received from a URL.

<declaration_and_initialization > → <type> <identifier> <assign_op> <URL> | <type> <identifier> <assign_op> <control_expression> | <type> <identifier> <assign_op> <receive_operation

This non-terminal combines the declaration of a new variable and its initialization in one line. Anything that can be assigned to a variable may also be given during the declaration-and-initialization process.

<connection_to_URL> → connect <identifier> | connect <URL>

This non-terminal describes the mechanism to connect the IoT device to a URL through the keyword “connect”. Either a URL-type variable or a URL literal may be connected with this mechanism.

<send_value> → send <identifier> <identifier> | send <integer> <identifier> | send <function_call> <identifier> | send <primitive_function_call> <identifier> | send <identifier> <URL> | send <integer> <URL> | send <function_call> <URL> | send <primitive_function_call> <URL>

This non-terminal shows how to send an integer literal, an integer variable, or an integer returned by a function to a URL via the keyword “send”. The first parameter after “send” is the integer, and the second is either a URL literal or a URL-type variable. The URL must be connected first.

<receive_operation> → receive <identifier> | receive <URL>

This non-terminal explains the mechanism to receive an integer from a URL through the keyword “receive” followed by the desired URL. The URL must be connected first. If the received integer is not stored in a variable, it is lost.

5. Loops

<loops> → <while_loop> | <for_loop>

Loops non-terminal represents while and for loops.

<for_loop> → for <LP> <assignment><semicolon> <control_expression><semicolon> <expression> <RP> <LC> <statements> <RC> | for <LP> <declaration_and_initialization><semicolon> <control_expression><semicolon> <expression> <RP> <LC> <statements> <RC>

This non-terminal displays the syntax of for loops in Quirk. As long as the control expression is true, actions between the curly brackets are being done. After all the statements between the curly brackets are executed, the expression between parentheses, which could be an arithmetic operation, is executed. The program gets out of the loop when the control expression is false.

<while_loop> → while <LP> <control_expression> <RP> <LC> <statements> <RC>

While loop non-terminal states the syntax of the while loop. As long as the control expression between the parentheses is true, the loop executes the statements between the curly brackets.

6. Functions

<function_declaration> → declare <identifier> <LP> <params> <RP> <LC> <statements> <return> <RC> | declare <identifier> <LP> <params> <RP> <LC> <statements> <RC>

This non-terminal shows the declaration (definition) of a function. After the keyword "declare", the function's identifier and then its parameters between parentheses are given. A function is essentially a block of code that can be called quickly and repeatedly by its identifier. Therefore, the function may contain any type of statement, as many as needed, between curly brackets. At the end, the function may return a variable if desired. The return type is not given in the declaration, which is easier for the programmer but increases the importance of documentation.

<params> → <type> <identifier> <comma> <params> | <type> <identifier> | <empty>

This non-terminal displays how the parameters in a function declaration should be structured. Each parameter's type and identifier are given between commas. Their identifiers may then be used inside the function. The parameters can be left empty if there are none.

<function_call> → <identifier> <LP><call_params><RP>

This non-terminal shows how to call a function. The function's identifier, along with its parameters between parentheses, is called. There are no default parameters, so every parameter must be given.

<call_params> → <identifier> <comma> <call_params> | <identifier> | <empty>

This non-terminal explains the structure of call parameters. Call parameters are similar to the parameters defined in the function declaration, except their data type is not given for convenience.

<return> → return <string> | return <control_expression>

This non-terminal indicates that the return statement of a function may return a string literal or through the control expression statement: a number literal, a boolean literal (true or false), an identifier, another function's return variable, or a logical expression such as "3>5", "true and false" (without quotation marks). If nothing is returned, the return statement may simply be omitted.

7. Primitive Functions

<primitive_function_call> → <time> | <temperature> | <humidity> | <air pressure> | <air quality> | <light> | <sound level> | <get_switch_state> | <change_switch_state> | <connect_to_internet> | <disconnect_from_internet>

This non-terminal lists which primitive functions exist in Quirk. These functions are directly linked to the IoT device's sensors and mechanisms. The programmer may simply call these functions without thinking about their implementation, which is an example of abstraction.

<time> → get_time<LP><RP>

This non-terminal shows how to call the get_time primitive function. This function returns the number of seconds elapsed since midnight Coordinated Universal Time as a float. The returned value is not very meaningful; it is conventionally kept for future reference to check the number of seconds passed during the execution of code blocks.

<temperature> → get_temperature <LP><RP>

This non-terminal shows how to call the get_temperature primitive function, which returns the temperature the IoT device's sensors captured as an integer.

<humidity> → get_humidity<LP><RP>

This non-terminal shows how to call the get_humidity primitive function, which returns the humidity the IoT device's sensors captured as an integer between 0 and 100 (included) to show the percentage.

<air pressure> → get_air_pressure<LP><RP>

This non-terminal shows how to call the get_air_pressure primitive function, which returns the air pressure the IoT device's sensors captured as an integer.

<air quality> → get_air_quality<LP><RP>

This non-terminal shows how to call the get_air_quality primitive function, which returns the air quality health index the IoT device's sensors calculated as an integer between 0 and 10, where 0 is the safest air quality.

<light> → get_light<LP><RP>

This non-terminal shows how to call the get_light primitive function, which returns the light levels the IoT device's sensors captured as an integer.

<sound level> → get_sound_level<LP><float><RP> | get_sound_level<LP><identifier><RP>
This non-terminal shows how to call the get_sound_level primitive function, which returns the sound level the IoT device's sensors captured as an integer. It takes a float literal or a float type variable's identifier as the frequency.

<connect_to_internet> → connect_to_internet<LP><RP>
This non-terminal shows how to connect the IoT device to the internet. This primitive function does not return anything and should be called before connecting to a URL.

<disconnect_from_internet> → disconnect_from_internet<LP><RP>
This non-terminal shows how to disconnect the IoT device from the internet. The programmer may decide to disconnect the device as the program ends, not to drain the device's battery.

<get_switch_state> → get_switch_state<LP><integer><RP> |
get_switch_state<LP><identifier><RP>
This non-terminal shows how to get the IoT device's switches' state with the primitive function get_switch_state. This function takes a number between 1-10 (included) and returns true if the switch with that number is on or false if the switch is off. The switches start from 1 instead of 0, as Quirk assumes the device's operators started counting from 1.

<change_switch_state> → change_switch_state<LP><boolean><comma><integer><RP> |
change_switch_state<LP><boolean><comma><identifier><RP> |
change_switch_state<LP><identifier><comma><integer><RP> |
change_switch_state<LP><identifier><comma><identifier><RP>
This non-terminal defines the primitive function change_switch_state, which takes a boolean value for determining whether to turn the switch on (true) or off (false), and a number between 1-10 (included) to identify the switch.

8. Comments

<comment> → <long_comment> | <line_comment>

This non-terminal indicates that Quirk has comments which are natural language sentences written to explain the code, that are ignored by the compiler. There are two types of comments: A long comment may contain multiple lines, whereas a line comment can only be one line.

<long_comment> → <long_comment_indicator> <sentence> <long_comment_indicator>
This non-terminal shows that long comments are zero or more sentences separated by <long_comment_indicator>s, i.e., “%%%”. The sentences may contain letters, digits, and symbols; however, three %s would end the comment. Long comments may be used to describe programs, document functions, or explain long code blocks.

<line_comment> → <line_comment_begin> <sentence> <new_line>
This non-terminal indicates that line comments are one-line-long sentences that start with <line_comment_begin>, i.e., “%%”. They end automatically when the line ends; percent signs are not needed. Line comments may be used to explain single lines of code or to easily “comment out” one line of code for debugging purposes.

Non-Trivial Tokens

1. Comments

There are two types of comments in Quirk: Line comments that start with “%%” which can also be placed next to a line of code (except the if statement), and long comments that span multiple lines between two %%% symbols. This feature increases readability as programmers may explain their code and also increases writability as programmers may put comments as reminders next to faulty code or code that may be improved in the future.

2. Identifiers

Identifiers are required to start with a letter and may contain letters, digits, and underscores. The convention is to use all lowercase characters and separate words with underscores. The lack of symbols except for the underscore in identifiers increases the ease of reading and simplifies the program.

3. Literals

Numbers are grouped into integers and floating-point numbers which have a decimal point. For simplicity, other numerical data types like byte, short, and long are not featured. String literals are a string of characters between quotation marks. URL literals are a string of characters between single quotation marks that have to start with a valid URL scheme. This distinction between strings and URLs increases readability; however, since only a certain number of valid URL schemes are added, reliability is decreased.

4. Reserved words

begin: Token reserved to indicate the start of the program.

end: Token reserved to indicate the end of the program.

if: Token reserved to identify where an “if” conditional block starts

else: Token reserved to identify where an “else” conditional block starts

while: Token reserved for the start of while loops.

for: Token reserved for the start of for loops.

int: Token reserved for integer numbers.

float: Token reserved for floating numbers.

boolean: Token reserved for boolean type data.

string: Token reserved for a string of characters. Note that there isn’t a character data type for simplicity.

URL: Token reserved for URL type data. This is a data type specific to Quirk and eases the connection to URLs for IoT devices.

true: Token reserved for the boolean true.

false: Token reserved for the boolean false.

declare: Token reserved for declaring (defining) functions.

return: Token reserved to identify the value or the variable a function returns.

connect: Token reserved to connect to a given URL.

send: Token reserved to send an integer value to a given URL, one at a time.

receive: Token reserved to receive an integer from a given URL, one at a time.

get_time: Token reserved for the primitive function that returns the time in seconds.

get_temperature: Token reserved for the primitive function that returns the temperature read from the IoT device's sensors.

get_humidity: Token reserved for the primitive function that returns the humidity read from the IoT device's sensors.

get_air_pressure: Token reserved for the primitive function that returns the air pressure read from the IoT device's sensors.

get_air_quality: Token reserved for the primitive function that returns the air quality level read from the IoT device's sensors.

get_light: Token reserved for the primitive function that returns the light level read from the IoT device's sensors.

get_sound_level: Token reserved for the primitive function that returns the sound level read from the IoT device's sensors, depending on the given frequency.

connect_to_internet: Token reserved for the primitive function that connects the IoT device to the internet.

disconnect_from_internet: Token reserved for the primitive function that disconnects the IoT device from the internet.

get_switch_state: Token reserved for the primitive function that returns the state of the switch with the given number. True for on, false for off.

change_switch_state: Token reserved for the primitive function that changes the state of the switch with the given number, according to the parameters.

Evaluation

1. Readability

a. Simplicity

Quirk is a very simple programming language primarily focused on IoT devices with sensors and switches that can connect to the Internet. People with a basic understanding of imperative languages may understand Quirk's code intuitively. Additionally, there is minimal feature multiplicity and no operator overloading.

b. Orthogonality

The language is not completely orthogonal: Certain combinations are not allowed, like dividing a string by a string, subtracting a URL from another URL, or sending an integer to an integer instead of an URL.

c. Data Structures

In addition to conventional int, float, string, and boolean primitive types, Quirk offers a URL type that contains valid URLs. The added URL type makes it easier to distinguish URLs with different features, such as being connected to, sending, and receiving integers.

2. Writability

a. Simplicity and Orthogonality

Quirk is a language that is designed specifically for IoT, which is why there are few constructs and a small number of primitives. People who are familiar with well-known programming languages can easily understand and write Quirk.

b. Support for abstraction

Quirk partially supports abstraction. In Quirk, the user can hide the details of a subprogram to improve readability and avoid implementing the same set of code several times in the program. Quirk hides the details for writability.

c. Expressivity

Quirk is a convenient programming language. It supports usages such as for loops, using the variable a subprogram returns directly in expressions.

3. Reliability

Quirk does not support type checking, exception handling, and aliasing. Not supporting aliasing increases reliability. Quirk supports both arithmetic and logical operator precedence. The simplicity, ease of writability, and readability of Quirk improve reliability.