CS 315

Project 2 Report

xkcd 297

Group Members:

Giray Akyol, 21902298, Section 3

Muhammed Can Küçükaslan, 21901779, Section 1

Selahattin Cem Öztürk, 21802856, Section 2

Contents

BNF Description of the Language **2**
- Statements 2
  - Conditional Statements 2
  - Non-conditional Statements 2
  - Declaration 3
  - Loops 3
  - Assignment 3
- Expression 3
  - Arithmetic 3
  - Boolean 4
  - Relation 4
- Variables 4
- Functions 5
- Primitive Functions 5

Explanations and Descriptions **6**
- Statements 6
  - Conditional Statements 6
  - Non-conditional Statements 6
- Expressions 7
- Terminals 10
- Nontrivial Tokens 11
  - Comments 11
  - Identifiers 11
  - Literals 11
  - Reserved Word or Keywords 12

Evaluation of the Language **14**
- Readability 14
- Writability 14
- Reliability 14

BNF Description of the Language

Statements

```
PROGRAM ::= FUNCTIONS
FUNCTIONS ::= MAIN_FUNCTION
    | FUNCTIONS FUNCTION_DEFINITION

MAIN_FUNCTION ::= FUNCTION TYPE MAIN LP ARGUMENTS RP LB STATEMENTS
RB
    | FUNCTION TYPE MAIN LP RP LB STATEMENTS RB

STATEMENTS ::= STATEMENT
    | STATEMENTS STATEMENT

STATEMENT ::= MATCHED
    | UNMATCHED
```

Conditional Statements

```
MATCHED ::= IF LP LOGICAL_EXPRESSION RP MATCHED ELSE MATCHED
    | NON_IF_STATEMENT
UNMATCHED ::= IF LP LOGICAL_EXPRESSION RP STATEMENT
    | IF LP LOGICAL_EXPRESSION RP MATCHED ELSE UNMATCHED
```

Non-conditional Statements

```
NON_IF_STATEMENT ::= ITERATIVE_STATEMENT
    | LB STATEMENTS RB
    | DECLARATIVE_STATEMENT SEMICOLON
    | ASSIGNMENT_STATEMENT SEMICOLON
    | FUNCTION_CALL SEMICOLON
    | RETURN_STATEMENT SEMICOLON
```

```
NORMAL_STATEMENT ::=
    | DECLARATIVE_STATEMENT
    | ASSIGNMENT_STATEMENT
    | FUNCTION_CALL
```

Declaration

```
DECLARATIVE_STATEMENT ::= TYPE VAR_NAME
```

Loops

```
ITERATIVE_STATEMENT ::= WHILE LP LOGICAL_EXPRESSION RP MATCHED
    | FOR LP NORMAL_STATEMENT SEMICOLON LOGICAL_EXPRESSION SEMICOLON
NORMAL_STATEMENT RP MATCHED
```

Assignment

```
ASSIGNMENT_STATEMENT ::= VAR_NAME ASSIGN_OP EXPRESSION
    | VAR_NAME ASSIGN_OP FUNCTION_CALL
    | DECLARATIVE_STATEMENT ASSIGN_OP EXPRESSION
    | DECLARATIVE_STATEMENT ASSIGN_OP FUNCTION_CALL
```

Expression

```
EXPRESSION ::= LOGICAL_EXPRESSION
```

Arithmetic

```
ARITHMETIC_EXPRESSION ::= ARITHMETIC_EXPRESSION ADDITIVE_OPERATOR
TERM  | TERM
TERM ::= TERM MULTIPLICATIVE_OPERATOR FACTOR
     | FACTOR
FACTOR ::= VALUE
    | LP EXPRESSION RP

ADDITIVE_OPERATOR ::= ADD | SUB
```

```
MULTIPLICATIVE_OPERATOR ::= MUL | DIV
```

Boolean

```
LOGICAL_EXPRESSION ::= LOGICAL_EXPRESSION OR AND_EXPRESSION
     | AND_EXPRESSION
AND_EXPRESSION ::= AND_EXPRESSION AND BOOLEAN_FACTOR
     | BOOLEAN_FACTOR
BOOLEAN_FACTOR ::= NOT BOOLEAN_FACTOR
    | RELATIONAL_EXPRESSION
    | ARITHMETIC_EXPRESSION
```

Relation

```
RELATIONAL_EXPRESSION ::= ARITHMETIC_EXPRESSION RELATIONAL_OPERATOR
ARITHMETIC_EXPRESSION

RELATIONAL_OPERATOR ::= LT
    | GT
    | GE
    | LE
    | EQ
    | NEQ
```

Variables

```
TYPE ::= PRIMITIVE_TYPE | VOID

PRIMITIVE_TYPE ::= INT | FLOAT | BOOL | STR

VALUE ::= VAR_NAME | CONSTANT

CONSTANT ::= BOOLEAN_VAL | FLOAT_NUMBER | INTEGER | STRING

BOOLEAN_VAL ::= TRUE | FALSE
```

Functions

```
ARGUMENTS ::= TYPE VAR_NAME
    | ARGUMENTS COMMA TYPE VAR_NAME
FUNCTION_DEFINITION ::= FUNCTION TYPE VAR_NAME LP ARGUMENTS RP LB
    STATEMENTS RB
    | FUNCTION TYPE VAR_NAME LP RP LB STATEMENTS RB
PARAMETERS ::= VALUE
    | PARAMETERS COMMA VALUE
FUNCTION_CALL ::= VAR_NAME LP RP
    | VAR_NAME LP PARAMETERS RP
    | PRIMITIVE_FUNCTION_CALL
RETURN_STATEMENT ::= RETURN EXPRESSION
    | RETURN FUNCTION_CALL
```

Primitive Functions

```
PRIMITIVE_FUNCTION_CALL ::= GET_HEADINGS LP RP
    | GET_ALTITUDE LP RP
    | GET_TEMPERATURE LP RP
    | SET_VERTICALCLIMB LP ARITHMETIC_EXPRESSION RP
    | SET_HORIZONTAL LP ARITHMETIC_EXPRESSION RP
    | SET_HEADING LP LOGICAL_EXPRESSION RP
    | SET_SPRAY LP LOGICAL_EXPRESSION RP
    | CONNECT LP EXPRESSION COMMA EXPRESSION RP
    | SLEEP LP ARITHMETIC_EXPRESSION RP
    | SCAN_NEXT LP RP
    | PRINT_OUT LP EXPRESSION RP
```

Explanations and Descriptions

Statements

PROGRAM:

Each program is composed of function definitions with a single main function at the beginning.

FUNCTIONS : Are starts with the main function and follows with 0 or more functions definitions.

MAIN_FUNCTION : Is the beginning of execution in our language. It takes any number of arguments (e.g. for command line arguments) and it is a collection of statements in a block (between an { and a }). It must be named “main”.

STATEMENTS : Is a linear collection one or more “statement”s, has the left recursive property.

STATEMENT : Is the smallest valid command in our language, it is made out of matched and unmatched statements which are for matched if statements and unmatched if statements respectively

Conditional Statements

MATCHED :

Used for indicating an if statement that has a matching else, or non if statements.

UNMATCHED:

Used for indicating an if statement that does not have a matching else.

Non-conditional Statements

NON_IF_STATEMENT:

Used for indicating iteration, declaration, grouping statements in blocks, assignment, return statements as well as primitive and function calls. Which all must end with a semicolon.

NORMAL_STATEMENT: Is either a variable declaration statement or a variable assignment statement or a (primitive) function call, or nothing. It is only used in the for loops.

ITERATIVE_STATEMENT:

Could be a for or a while loop that contains a logical expression and followed by a matched statement inside the parentheses, or could be a for loop that has normal statement followed by logical expression and normal statement inside the parenthesis.

DECLARATION_STATEMENT:

Indicates the variable declaration. It will have the variable's type followed by the name of it.

ASSIGNMENT_STATEMENT :

It is used for assigning a value, function call, declarative statement, or expression to a variable.

RETURN_STATEMENT:

It is used for returning an expression, or function call. Return reserved word will be followed by an expression or a function call.

Expressions

EXPRESION : Is a logical expression, which can turn into relational or arithmetic expressions.

ARITHMETIC_EXPRESSION : Arithmetic expression is either followed by an addition/subtraction operator followed by a term, or just a term , it has the left recursive property. Term is a term followed by multiplication/division followed by a factor or just a factor, this also has the left recursive property. And factor is either a value or an ordinary expression in parenthesis (for precedence). If integers and doubles are used together, the integers are implicitly casted to double. If aforementioned types used with string they are implicitly casted to string and all operations become string concatenation.

TERM:

Term is either followed by a multiplicator operator and a factor, or just a factor. It has a left recursive property.

FACTOR :

Factor is either an expression inside of a parenthesis, or a value.

ADDITIVE_OPERATOR:

It is either ADD or SUB tokens.

MULTIPLICATIVE_OPERATOR:

It is either MUL or DIV tokens.

RELATIONAL_EXPRESSION :

Is an arithmetic expression followed by a relational operator followed by another arithmetic expression.

RELATIONAL_OPERATOR:

Less than (<), greater than (>), less or equal (<=), greater or equal (>=), equality (==) and inequality (!=) have their normal meanings as in mathematics for float and int types. Equality and inequality return true or false if and only if the operands correspond to the same values.

LOGICAL_EXPRESSION:

Used to represent boolean expressions. such as and, or, not. Not operator has the top precedence of all 3 followed by and and lastly or. It is also left associative. RELATIONAL_EXPRESSION is casted to corresponding boolean value, other types are casted false if they correspond to 0, 0.0, empty string etc.

LOGICAL_EXPRESSION :

it is either an or expression, or relational expression.

OR_EXPRESSION:

It has left recursive property, also it is either an and expression, or or expression followed by token OR, followed by an and expression.

AND_EXPRESSION:

It is either a boolean factor, or and expression followed by tokend AND, followed by a boolean factor. It also has a left recursive property.

BOOLEAN_FACTOR:

Could be NOT token that is followed by a boolean factor, or arithmetic expression, or relational expression. It has a right recursive property so that it can be preceeded by arbitrary number of NOT tokens.

VALUE:

Value represents either the use of a declared variable or a constant.

VAR_NAME:

It is the variable name, starts with a letter and may or may not be followed by an arbitrary number of alphanumeric characters.

TYPE: Is either a primitive type or the VOID token used in defining functions.

PRIMITIVE_TYPE: is either a INT, FLOAT, BOOL or a STR token. Used for defining variables with primitive types.

CONSTANT: Is either a boolean value (below not a token), a token such as FLOAT_NUMBER, INTEGER, STRING.

BOOLEAN_VAL: Is either TRUE for true or FALSE for falsity.

Functions

ARGUMENTS:

Used for identifying the arguments id the function declaration. It has a left recursive property. Used in defining functions.

FUNCTION_DEFINITION:

It is FUNCTION token followed by return type, function name, optionally the function arguments inside the parentheses and a block of statements. A function may or may not have arguments. A function that does not return a value should have VOID as return type.

PARAMETERS:

param is used for indicating the function call arguments. It is either a param followed by a value or just a value. It has a left recursive property.

FUNCTION_CALL:

Used for calling a function. Function's name will be either a primitive function call, or a function name followed by empty parentheses or parenthesis with the appropriate parameters in it.

PRIMITIVE_FUNCTION_CALL:

It is used for indicating the functions that are special to this programming language.

RETURN_STATEMENT:

It is used for return statements in the functions. It is a return terminal that is followed by an expression, or a function call.

Terminals

LT: < smaller than operator, used for comparing two ARITHMETIC_EXPRESSIONs, returns true when the right operand is strictly bigger.

GT: > greater than operator, used for comparing two ARITHMETIC_EXPRESSIONs, returns true when the left operand is strictly bigger.

LE: <= smaller or equal to operator, used for comparing two ARITHMETIC_EXPRESSIONs, returns true when the right operand is bigger or equal.

GE: >= greater than or equal to operator , used for comparing two ARITHMETIC_EXPRESSIONs, returns true when the right operand is smaller or equal.

EQ: == equivalence comparison operator , used for comparing two ARITHMETIC_EXPRESSIONs, returns true if their internal representations are equal.

NEQ: != converse of the EQ, used for comparing two ARITHMETIC_EXPRESSIONs. Returns true if their internal representations are not equal.

ASSIGN_OP: = assignment operator

AND: && logical and operator

OR logical or operator

NOT logical not operator

ADD: + addition operator

SUB: - subtraction operator

DIV: / division operator

MUL * multiplication operator

LP and RP: () parentheses

LB and RB: {} curly brackets

LSB and RSB [] square brackets

Nontrivial Tokens

Comments

Comments are identified by the surrounding elements /* at the beginning and */ at the end of the comment. The comment can contain any ASCII character, including the new line characters. Comments are ignored during parsing and compilation. Comments are matched on a shortest match possible so "/* example */ */" will not match the red token and so this will be a syntax error. The programmer can put comments anywhere he wants as long as it doesn't cut a token, this increases the readability of code greatly because it enables another programmer to read the intent of code in text.

Identifiers

We chose to only allow identifiers of any length with the 2 constraints, they must start with a letter and it can only have alphanumeric characters. We did it this way because we wanted to force the programmers to use camelCase with the intention that uniformity of variable names will increase readability.

Literals

floating point number: Standard IEEE 754 floating point number, can be compared, added, subtracted, multiplied or divided. We added it because navigating in 3D space may not be practically possible with only integer numbers. If compared, added, subtracted, multiplied or divided with an integer, the integer is automatically transformed into a floating point number and then the operation is done.

integer: Standard two's complement integer numbers can be compared, added, subtracted, multiplied or divided. We added it because the unpredictability of floating point numbers in comparisons would have hampered reliability. If compared, added, subtracted, multiplied or divided with an integer, the integer is automatically transformed into a floating point number and then the operation is done.

boolean: Standard "true" and "false" as used in discrete mathematics. It is added to improve the practicality of control flow statements like if, if-else, while and for loops.

string: A linear sequence of zero or more CHARs. At declaration it should be surrounded by quotation marks as it is in many other C-family languages, thus increasing readability. It is frequently needed in communicating with users. If an arithmetic operation (+, -, *, /) has a string as one of its operands then if the other operand is a string then the two will be appended, else if the other operand is integer or float then number the will be converted to a string with base 10 and then the resulting string will be appended, if its a boolean then if the value true then the string "true" and if false the string "false" will be appended.

Reserved Word or Keywords

if:	It is the reserved word used for if statements. Enables structured control flow which increases understandability of code.
else:	It is the reserved word used for the alternate part of if-else statements. Enables structured control flow which increases understandability of code.
while:	It is the reserved word used for while loop statements. Enables structured control flow which increases understandability of code.
for:	It is the reserved word used for for loop statements. Enables structured control flow which increases understandability of code.
void:	It is the reserved word used for declaring functions that return “nothing”. Adding return types makes it clear where a function can be used.
int:	It is the reserved word used in declaring integer variables or parameters.
float :	It is the reserved word used in declaring floating point number variables or parameters.
bool:	It is the reserved word used in declaring boolean variables or parameters.
str:	It is the reserved word used in declaring a sequence of 0 or more ASCII characters, variables or parameters.
true:	It is the reserved word used for the meaning of true (T or 1) in boolean algebra. Similarity with discrete mathematics increases readability.
false:	It is the reserved word used for the meaning of false (F or 0) in boolean algebra. Similarity with discrete mathematics increases readability.
function:	It is the reserved word used for declaring functions, and highlights the definitions for readability.
return:	It is the reserved word used for returning the result of the function. Similarity with the C family of languages increases readability.
getHeading:	It is the primitive function used for obtaining the direction the drone is facing in degrees. Using a common prefix and camelCase like in Java increases writability.
getAltitude:	It is the primitive function used for obtaining the altitude of the drone. Using a common prefix and camelCase like in Java increases writability.
getTempature:	It is the primitive function used for obtaining the temperature of the environment surrounding the drone. Using a common prefix and camelCase like in Java increases writability.
setVerticalClimb:	It is the primitive function used for controlling whether the drone has a vertical speed of 0.1m/s (positive values), -0.1m/s (negative values) or 0m/s (“0”, zero value). Using a common prefix and camelCase like in Java increases writability.
setHorizontal:	It is the primitive function used for controlling whether the drone has a horizontal speed in direction of the heading with 1 m/s forward (positive values), -1m/s (negative values) or 0m/s (“0”, zero value. Using a common prefix and camelCase like in Java increases writability.

setHeading:	It is the primitive function used for turning the heading direction of the drone 1 degree counter-clockwise (true) or clockwise (false). Using a common prefix and camelCase like in Java increases writability.
setSpray:	It is the primitive function used for turning the spray nozzle on (true) or off (false). Using a common prefix and camelCase like in Java increases writability.
connect:	It is the primitive function used for connecting to a Wi-Fi access point, its first parameter in a string containing the name of the network and the second parameter is the secret key/password for authenticating with that network. Returns true for successful connection and false otherwise.
sleep:	It is the primitive function that stalls the program execution for the given time in milliseconds.
scanNext:	It is the primitive function that reads input. Using a common prefix and camelCase like in Java increases writability.
printOut:	It is the primitive function that prints outputs. Using a common prefix and camelCase like in Java increases writability.
main:	It is the main function that will be called on program execution.

Evaluation of the Language

Readability

The language has an increased readability especially since it has a similar structure with the C-family programming languages. This readability is reinforced by having similar types, and naming conventions. Its precedence and associativity rules follow the mathematical standards which are intuitive expectations of programmers. Its types are also a factor that increases readability, though it may have small inconvenience for writability. However, it is not always perfectly readable. For example, some type names are shortened to increase writability, but it may be a small inconvenience, even though they can be easily understandable (e.g. *str* for *string*, *bool* for *boolean*).

Writability

To increase writability in our language design, instead of using double and float as floating point numbers we only used float type. Same design choice made for integers also. Instead of using long and int separately, we merged them in int type since writability increases when there is less variable type. In that way common convention issues will not occur that much. Also common loop structures for, and while added to the language. Also we use English words while we are declaring reserved words such as if, else, and while. Thus, our language has become more writable for a variety of users. Also mathematical expressions and relational operations' precedence, and associativity is identical to real life usage of them, therefore users of the language could write them in the way that they wrote on paper. There is a downside in our language; we do not allow array declarations so it decreases writability since the user has to declare a lot of variables.

Reliability

In order to improve our languages reliability we chose the motto of *principle of least surprise* (POLS). POLS can be translated as “the language should fit into the mental model of the programmer”. As an example all of our language’s binary operators are left recursive and so the programmer does not have to think extra when using arithmetic expressions inside a logical expression. We also chose to honor precedence rules of algebra for addition, subtraction, multiplication and division which we hope won’t clash with the programmers knowledge in mathematics. We have also added types to our language in the hope that when programmers are cooperating they can use type annotations on variables aided with comments to make the meaning of the code more clear. We chose not to add arrays or custom data structures because we think in our application running on an resource constrained system like a drone (which has to preserve weight and battery power) they do not have much use but more importantly they may significantly increase the number of outOfMemory and invalidIndex errors.