CS315- Programming Languages

PROJECT 1

Team 63

Language Name: *FTOF*

Mustafa Kağan Özsoy - 22002275 - Section 2

Süleyman Yağız Başaran - 22103782 - Section 3

BNF DESCRIPTION OF THE LANGUAGE

1. Program

<program> ::= fire <stmts>

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

<stmt> ::= <decl>

| <init> <stmt>

| <decl_n_init> <stmt>

| <math_exp> <stmt>

| <loop_stmt> <stmt>

| <input_stmt> <stmt>

| <output_stmt> <stmt>

| <ammo> <stmt>

| <ammo_assignElement> <stmt>

| <func_def> <stmt>

| <func_call> <stmt>

| <comment> <stmt>

| <conditional_stmt> <stmt>

| <return_stmt>

| <end>

| stop

<end> ::= <semicolon>

2. Variable Declaration

<var_name> ::= <var_name> <word> | <var_name> <digit> | <word>

<rhs_val> ::= <math_exp> | <func_call>

<decl> ::= <type_identifier> <var_name>

<init> ::= <var_name> <assign_op><rhs_val>

<decl_n_init> ::= <type_identifier> <var_name> <assign_op> <rhs_val>

3. Types & Constants

<type_identifier> ::= bool | int

<type> ::= <boolean_type> | <int_type>

<boolean_type> ::= <true> | <false>

<int_type> ::= <sign> <int_type> | <int_type> <digit> | <digit>

4. Operations

<math_exp> ::= <math_exp> + <term> | <math_exp> - <term> | <term>

<term> ::= <term> * <factor> | <term> / <factor> | <factor>

<factor> ::= <factor> % <expo> | <expo>

<expo> ::= <type> ^ <type> | <var_name> ^ <type> | <type> ^ <var_name> | <var_name> ^ <var_name> | <type> | <var_name>

5. Expressions

<logic_exp> ::= <or_exp>

<or_exp> ::= <or_exp> <or_op> <and_exp> | <and_exp>

<and_exp> ::= <and_exp> <and_op> <equality_exp> | <equality_exp>

<equality_exp> ::= <equality_exp> <equal_op> <relational_exp> |
<equality_exp> <not_equal_op> <relational_exp> | <relational_exp>

<relational_exp> ::= <relational_exp> <greater_op> <primary_exp> |
<relational_exp> <smaller_op> <primary_exp> | <relational_exp>
<greater_or_equal_op> <primary_exp> | <relational_exp>
<smaller_or_equal_op> <primary_exp> | <primary_exp>

<primary_exp> ::= <logic_type> | <left_p> <logic_exp> <right_p>

<logic_type> ::= <int_type> | <boolean_type> | <var_name>

6. Loop Statements

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

<while_loop> ::= while<left_p> <logic_exp> <right_p> <left_cb> <stmts>
<right_cb>

<for_loop> ::= for <left_p> <forloop_init> <end> <logic_exp> <end> <init>
<right_p> <left_cb> <stmts> <right_cb>

<forloop_init> ::= <decl_n_init> | <init>

7. Conditionals

<conditinal_stmt> ::=

if<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> |

if<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> elsif<left_p>
<logic_exp> <right_p> <left_cb> <stmts> <right_cb> |

if<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> elsif<left_p>
<logic_exp> <right_p> <left_cb> <stmts> <right_cb> else <left_cb> <stmts>
<right_cb> |

if<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> else <left_cb> <stmts> <right_cb>

8. Input/Output

<input_stmt> ::= input <left_p> <input_body> <right_p>

<input_body> ::= <var_name>

<output_stmt> ::= out <left_p> <output_body> <right_p>

<output_body> ::= <math_exp> | <func_call>

9.Arrays(Ammo)

<ammo_name> ::= <word>

<ammos> ::= <int_type>, <ammos> | <var_name>, <ammos> | <int_type> | <var_name> | <left_sq> <ammos> <right_sq>,<ammos> | <left_sq> <ammos> <right_sq>

<ammo_element>::= <var_name> <left_sq> <bullet> <right_sq>

<ammo_assignElement> ::= <ammo_element> <assign_op> <bullet>

<bullet> ::= <digit> | <var_name>

<ammo> ::= ammo <ammo_name> <assign_op> <left_sq> <ammos> <right_sq>

10.Functions (Missions)

<func_types> ::= <type_identifier> | void

<func_name> ::= <word>

<params> ::= <params>, <type_identifier> <var_name>

| <type_identifier> <var_name>

| <empty>

<call_params>::= <type>, <call_params> | <var_name>, <call_params> | <var_name> | <int_type> | <boolean_type> | <empty>

<func_def> ::= <func_types> mission <func_name> <left_p> <params> <right_p> <left_cb> <stmts> <right_cb>

<func_call> ::= mission <func_name> <left_p> <call_params> <right_p>

<return_stmt> ::= return <math_exp> <end>

11.Comments

<comment_block> ::= <word> | <digit> | <extras> | <space> | <new_line> | <comment_block> <word> | <comment_block> <digit> | <comment_block> <extras> | <comment_block> <space> | <comment_block> <new_line>

<comment> ::= <wave> <comment_block> <wave>

12.Symbols

<word> ::= <chars> | <chars> <digit>

<chars> ::= <letter> <chars> | <letter>

<letter> ::= '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'

<extras> ::= '(' | ')' | '{' | '}' | '[' | ']' | '.' | ',' | ';' | '=' | '"' | '`' | '+' | '-' | '#' | '&' | '|' | '!' | '_' | '£' | '^' | '*' | '$' | '%' | '/' | '\' | 'ß' | 'æ' | '~' | '€' | '@' | '<' | '>' | '½'

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

<true> ::= true | 1

<false> ::= false | 0

<new_line>::= \n

<semicolon> ::= ;

<assign_op> ::= =

<sign> ::= + | -

<and_op> ::= &&

<or_op> ::= ||

<equal_op> ::= ==

<not_equal_op> ::= !=

<smaller_op> ::= <

<greater_op> ::= >

<smaller_or_equal_op> ::= <=

<greater_or_equal_op> ::= >=

<left_p> ::= (

<right_p> ::=)

<left_sq> ::= [

<right_sq> ::=]

<left_cb> ::= {

<right_cb> ::= }

<empty> ::= “”

<wave> ::= ~

<space> ::= ` `

Detailed Explanation of BNF Description

1) Program Definition

- **<program> ::= fire <stmts>**

First non-terminal element which means language starts with “fire” key/reserved word and then comes non-terminal <stmts>.

- **<stmts> ::= <stmt> | <stmt> <stmts>**

From this part language starts to take a shape. This non-terminal demonstrates that it consists of a non-terminal program statement or again, statement with statements creating blocks. It means this language is left-recursive.

- **<stmt> ::= <decl>**

  | <init> <stmt>

  | <decl_n_init> <stmt>

  | <math_exp> <stmt>

  | <loop_stmt> <stmt>

  | <input_stmt> <stmt>

  | <output_stmt> <stmt>

  | <ammo> <stmt>

  | <ammo_assignElement> <stmt>

  | <func_def> <stmt>

  | <func_call> <stmt>

  | <comment> <tmt>

  | <conditional_stmt> <stmt>

  | <return_stmt>

  | <end>

  | stop

This non-terminal definition is basically the navigator of the program. From the stmt block, it can be navigated to any other executable statement in the program. To end the entire program, it should reach the “stop” keyword.

- **<end> ::= <semicolon>**

This non-terminal indicates the end of the written statement(s) with a “semicolon” character (;).

2. Variable Declaration

- **<var_name> ::= <var_name> <word> | <var_name> <digit> | <word>**

In this non-terminal declaration var_name -as known as identifiers in most languages-. Firstly, we don’t want to start with digits while declaring variables; thus, in this language, it starts with variable name and word, variable name and digit or at ending only a word. Thanks to this order, variables can be longer names.

- **<rhs_val> ::= <math_expr> | <func_call>**

In this non-terminal the right hand side of a variable declaration is done. A variable can be assigned to another variable, integer, or an index of an ammo which comes from a mathematical expression or the variable of the return of the function call or ammo element.

- **<decl> ::= <type_identifier> <var_name>**

This non-terminal demonstrates declarations without assignment operator and it ends with a classic semicolon like most languages (eg. *int x; bool z;)*.

- **<init> ::= <var_name> <assign_op><rhs_val>**

This non-terminal demonstrates an initialization with assign operator but without the type declaration and ends with a classic semicolon (eg. *x = 7; z = false, x = x + 1;*) .

- **<decl_n_init> ::= <type_identifier> <var_name> <assign_op> <rhs_val>**

This non-terminal demonstrates both initialization and declaration. Starting with the type of the variable and then it's name, later on one can equalize it to any constant value, ends with a classic semicolon (eg. *int x = 5; int x = y + 3;*).

3. Types & Constants

Important note about declaring a negative integer!: While using the operator "-" (minus) the user needs to be careful. Because in *FTOF* language there shouldn't be any space between minus sign and integer constant if the user's aim is to declare a negative integer (eg. x = -5, -7). However, if the user wants to use minus sign as a mathematical expression they need to make space between constant integer and the operator (eg. 9 - 7).

- **<type_identifier> ::= bool | int**

This non-terminal demonstrates the data types of what *Ftof* language includes. "bool" reserved word represents the Boolean type, "int" reserved word represents the Integer type. These primitive data types exist to indicate the type of the variable.

- **<type> ::= <boolean_type> | <int_type>**

This non-terminal demonstrates and matches each data type with their statements.

- **<boolean_type> ::= <true> | <false>**

This "boolean" non-terminal states that boolean variables can get either true/1 or false/0.

- **<int_type> ::= <sign> <int_type> | <int_type> <digit> | <digit>**

This "integer" non-terminal states an integer. Integers can have signs so it is declared.

4. Operations

- **<math_exp> ::= <math_exp> + <term> | <math_exp> - <term> | <term>**

This non-terminal indicates the last step of the mathematical expression. To provide the precedence of the operators, math_exp branches out to terms. In this level, only addition and subtraction operations are done. To provide the applicability of multiple operations, it has a recursive structure. Since the program only supports integers, math_exp is used in a lot of area.

- **<term> ::= <term> * <factor> | <term> / <factor> | <factor>**

This non-terminal indicates the second-to-last step of a mathematical expression. Similar to math_exp it branches out to factors. In this level, only multiplication and division operations are done. To provide the applicability of multiple operations, it has a recursive structure.

- **<factor> ::= <factor> % <expo> | <expo>**

This non-terminal indicates the second step of a mathematical expression. Factors branches out to expos. In this level, only modulo operation is done. To provide the applicability of multiple operations, it has a recursive structure. In our language, modulo operation has higher precedence than multiplication and division. Because, in order to immediately determine the remaining ammunition in a magazine before calculating the potential for follow-up shots or reloads needed during a rapid-fire sequence. Therefore, it needs to have a step between exponential level and multiplication/division level.

- **<expo> ::= <type> ^ <type> | <var_name> ^ <type>**

  | <type> ^ <var_name> | <var_name> ^ <var_name> | <type> | <var_name>

This non-terminal indicates the first step of a mathematical expression. Since the exponential operation has the highest precedence, only this operation is done in this level, and gone to further operations. An expo can be exponential of integer or variable or boolean. It should be noted that since in the language true has integer value of 1 and false has integer value of 0, the result of using boolean types will also be equal to 1 or 0 according to exponential calculation.

It can be updated to only variables or integer constants in the future with deleting term non-terminal from the above non-terminal declaration.

5. Expressions

- **<logic_exp> ::= <or_exp>**

This non-terminal includes all expressions in the language. At start, it evaluates to the logical *or* operation, which hierarchically can be evaluated into other expressions. The *or* expression is at the top of the hierarchy since it is the operation with the least precedence in our language. By this implementation, hierarchical design pattern achieves correct operator precedence.Therefore, the expressions entered by the users are evaluated according to appropriate logical and mathematical rules.

- **<or_exp> ::= <or_exp> <or_op> <and_exp> | <and_exp>**

This non-terminal indicates the last step of logical expressions. In our language, or operation is executed last. To provide the precedence of logical expressions, or expressions branches to *and* expression. In this level, only or expressions are done. To provide nested logical expressions, it has a recursive structure.

- **<and_exp> ::= <and_exp> <and_op> <equality_exp> | <equality_exp>**

This non-terminal indicates the second-to-last step of logical expressions. In the language, *and* operation has a higher precedence than *or* operation. Therefore, it has a separate level. To provide further precedence of logical expressions, and expression branches to *equality* expression. In this level, only and operations are done. To provide multiple logical expressions, it has a recursive structure.

- **<equality_exp> ::= <equality_exp> <equal_op> <relational_exp> | <equality_exp> <not_equal_op> <relational_exp> | <relational_op>**

This non-terminal demonstrates the third-to-last step of logical expressions. It can be seen that in this language equality expressions (!= and ==) have a higher precedence degree than *and* operation. To provide further precedence of logical expressions, *equality* expression branches to *relational* expression. In this level, only *equality* operations are done. To provide multiple logical expressions, it has a recursive structure.

- **<relational_exp> ::= <relational_exp> <greater_op> <primary_exp> | <relational_exp> <smaller_op> <primary_exp> | <relational_exp> <greater_or_equal_op> <primary_exp> | <relational_exp> <smaller_or_equal_op> <primary_exp> | <primary_exp>**

This non-terminal indicates the second step of logical expressions. In the language, *relational* expressions are done first, after the *primary* expressions. In this level, relational expressions are done such as greater than, smaller than etc. To provide further precedence, it branches to *primary* expressions. Also, to provide multiple logical expressions, it has a recursive structure.

- **<primary_exp> ::= <logic_type> | <left_p> <logic_exp> <right_p>**

This non-terminal demonstrates the first step of logical expressions. This *primary* expression has the highest precedence level. It is the last step for any mathematical or logical operations made in the language.Therefore, it evaluates to an expression that is parenthesized or a single term.

- **<logic_type> ::= <int_type> | <boolean_type> | <var_name>**

This non-terminal matches which type is selected while doing the logic expression. It can only choose from variables, ints or booleans.

6. Loop Statements

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

This non-terminal is used to generalize the loop types of the *Ftof* language. The language has two types of loops: for loop and a while loop.

- **<while_loop> ::= while<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb>**

This non-terminal indicates the structure of the while loop. A while loop can be created using the “while” reserved word, and a logical expression between parentheses. Finally, any kind and number of statements can be given between the curly braces, within the scope of the while loop.

(eg. *while(true){output(1);}*)

- **<for_loop> ::= for <left_p> <forloop_init> <end> <logic_exp> <end> <init> <right_p> <left_cb> <stmts> <right_cb>**

This non-terminal indicates the structure of the for loop. A for loop can be created using the "for" reserved word, an initialization for the loop, a logical expression to check the continuity of the loop, and mathematical expression between parentheses. After that, any kind and number of statements can be given between curly braces, within the scope of the for loop.

(eg. *for(int i = 10; i > 0; i = i - 1){output(true)}*)

- **<forloop_init> ::= <decl_n_init> | <init>**

This non-terminal indicates the declaration of a variable for using in the for loop. It can be either declaration and initialization of a variable or only a declaration.

7. Conditionals

- **<conditinal_stmt> ::=**

  if<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> |

  if<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> elsif<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> |

  if<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> elsif<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> else <left_cb> <stmts> <right_cb> |

  if<left_p> <logic_exp> <right_p> <left_cb> <stmts> <right_cb> else <left_cb> <stmts> <right_cb>

This non-terminal statement indicates the conditional blocks. A conditional block can be a single if block, if-elsif block, if-else block, or a complete if-elsif-else block. Then, anything can be written in the scope of the conditional block.

8. Input/Output

- **<input_stmt> ::= input <left_p> <input_body> <right_p>**

This non-terminal indicates the structure of an input statement. An input statement can be used by starting with an “input” reserved word. An input will be taken after its call in the input body.

- **<input_body> ::= <var_name>**

This non-terminal is used to read the values coming from input. An input can be taken as any type(int, bool) and it assigns the taken value to the given var_name.

- **<output_stmt> ::= out <left_p> <output_body> <right_p>**

This non-terminal indicates the structure of an output statement to log anything to the console. An output statement can be used by starting with “output” reserved word. The function will print the data given on the output body.

- **<output_body> ::= <math_exp> | <func_call>**

This non-terminal is used to print the values given to output. An output can be taken from math_exp, so it can be a variable, constant integer value or a specific location of an ammo. It can also be used to print out the value of a function.

9. Arrays (Ammos)

- **<ammo_name> ::= <word>**

This non-terminal indicates the name of the ammo, mostly known as arrays in other languages. It is used to extend the parse tree for prioritizing arrays.

- **<ammos> ::= <int_type>, <ammos> | <var_name>, <ammos> | <int_type> | <var_name> | <left_sq> <ammos> <right_sq>,<ammos> | <left_sq> <ammos> <right_sq>**

This non-terminal takes the values of the ammo. Any number of integer items, variables or any ammos can be added into ammos. It allows ammos to have more than one dimension. Keep in mind that in this language empty ammos cannot be initialized because technological guns would always be loaded.

- **<ammo_element>::= <var_name> <left_sq> <bullet> <right_sq>**

This non-terminal indicates the specific index (bullet) of an ammo (array). It can be used to assign a value to a specific index or get the value of a specific index (eg. *x[3], arr[y]*).

- **<ammo_assignElement> ::= <ammo_element> <assign_op> <bullet>**

This non-terminal indicates the assignment operation of an array with known index to a specific value (eg. *x[a] = 5, arr[3] = y, arr1[x] = y, arr2[3] = 12*).

- **<bullet> ::= <math_exp>**

This non-terminal is basically the index or value definition of accessing and assigning array elements. Any mathematical expression can be given for index or it can be simply a variable or a constant integer value.

- **<ammo> ::= ammo <ammo_name> <assign_op> <left_sq> <ammos> <right_sq>**

This non-terminal is used to initialize an ammo. Ammo is a data structure that holds multiple integer values. An ammo initialization begins with “ammo” reserved word along with ammo name on the left hand side. After assignment operator, ammo is initialized with the given ammos.

*(*eg. *ammo ammoName = [3,5,a,b,15,22,887]; ammo ammoName = [[]]; ammo ammoName = [[1,2]],[a,b],7])*

10. Functions (Missions)

- <**func_types> ::= <type_identifier> | void**

This non-terminal is used to specify the return type of the function. It can be either given types(int, bool) or void. In this language we also allow users to work with functions that have void return. This can cause reliability issues in some cases such as adding a constant with a function that has void return type. However, in our language the benefits of having a void return type outweighs its issues. They are useful because it indicates that a function performs an action without producing a result, simplifies understanding of the code's intent and enhances readability. Users should be careful of void functions to avoid facing any issues.

- **<func_name> ::= <word>**

This non-terminal indicates the name of the function. The name of the function can only be a word. It is used to extend the parse tree for prioritizing functions.

- **<params> ::= <params>, <type_identifier> <var_name> | <type_identifier> <var_name> | <empty>**

This non-terminal indicates the parameters it takes to define a function. Because of its recursive structure, any number of parameters can be assigned. The function can also take no parameter, in that case, it is empty.

- **<call_params>::= <type>, <call_params> | <var_name>, <call_params> | <var_name> | <int_type> | <boolean_type> | <empty>**

This non-terminal indicates the parameters it takes to call a function after its definition. Due to its recursive structure, any number of parameters can be added or it can be empty while calling a function.

- **<func_def> ::= <func_types> mission <func_name> <left_p> <params> <right_p> <left_cb> <stmts> <right_cb>**

This non-terminal is used to define a function. In order to define a function, the function type must be given with the “mission” reserved word. Then the function name is written. Between parentheses, parameters are given. Then between curly brackets any statement can be given. A return statement should be used for boolean and int type functions, reached from stmts.

(*eg. int mission missionName(int a){output(b); return a;}*)

- **<func_call> ::= mission <func_name> <left_p> <call_params > <right_p>**

This non-terminal is used to call a function after its definition. To call a function; “mission” keyword should be used along with the function name and variables or constant values between parantheses.

(*eg. mission missionName(7, x);*)

- **<return_stmt> ::= return <math_exp> <end>**

This non-terminal is used for return statements in functions. The return statement has a “return” reserved word, then a math_exp, therefore a function can return bool, int or a variable, or a mathematical expression itself. And it ends with a semicolon.

11. Comments

- **<comment> ::= <wave> <comment_block> <wave>**

This non-terminal element is used to add comment blocks between two tilde signs(~). Everything between tilde signs is considered as comments and does not affect the code.

- **<comment_block> ::= <word> | <digit> | <extras> | <space> | <new_line> | <comment_block> <word> | <comment_block> <digit> | <comment_block> <extras> | <comment_block> <space> | <comment_block> <new_line>**

This non-terminal is used to define the contents of comments. It can be composed of words, digits, extras or their combination . It can be as long as one wants thanks to its recursive structure. It can be used for both single line or multiple line comments as long as the comments are in between two tilde (~ ~) signs.

12. Symbols

- **<word> ::= <chars> | <chars><digit>**

This non-terminal is used to define the words, basically. A word can include a char along with digits.

- **<chars> ::= <letter><chars> | <letter>**

This non-terminal is used to form a char body with multiple letters.

Descriptions of *Ftof* Non-Trivial Tokens

- **fire:** Token reserved for indicating the start of the program.
- **stop:** Token reserved for indicating the end of the program.
- **bool:** Token reserved for boolean data type.
- **int:** Token reserved for integer type.
- **while:** Token reserved for detection of a while loop.
- **for:** Token reserved for detection of a for loop.
- **if:** Token reserved for conditional if-else statements.
- **elsif:** Token reserved for conditional if-else if-else statements.
- **else:** Token reserved for conditional if-else statements.
- **out:** Token reserved for printing contents of set to the console.
- **input:** Token reserved for reading from an input stream.
- **ammo:** Token reserved for declaring a data structure to hold multiple integer values.
- **mission:** Token reserved for function declaration.
- **return:** Token to return values within a function.
- **void:** Token reserved for returning nothing in functions.
- **true:** Token reserved for boolean value true.
- **false:** Token reserved for boolean value false.

Evaluation

Readability:

Clear Structure: The *Ftof* language provides a structured and hierarchical format, which means that programs written in this language will likely have a clear and consistent structure, enhancing readability.

Descriptive Names: Constructs like "mission" for functions or "ammo" for arrays, though unconventional, are descriptive and can make the purpose of certain constructs immediately apparent to those familiar with the language's theme.

Explicit Endings: The use of semicolons and defined end statements for various constructs ensures that blocks of code are explicitly terminated, reducing ambiguity.

Writability:

Inclusion of Common Constructs: The *Ftof* includes common programming constructs like loops, conditionals, and functions. This ensures that users don't have to "reinvent the wheel" for basic tasks.

Simple Data Types: With only a few data types, the programmer doesn't have to juggle many complex type-related issues, making it easier to write straightforward programs.

Reliability:

Structured Grammar: The *Ftof* with the BNF provides clear rules, reducing the chances of syntactical errors.

Explicit Type Definitions: The *Ftof* requires explicit type definitions, which can prevent type-related errors.