CS315 - Programming Languages

Project 2 Report

FLEX Language Design

GROUP 1

Alper Kandemir 21703062 Section 1

Lamia Başak Amaç 21601930 Section 1

BNF:

1.Program

<program>::= START <statements> STOP

<statements>::=<statement>;

|<statement>;<statements>

<statement>::=<assignment>

|<expression>

| <declaration>

|<loops>

|<functions>

|<if_stmt>

|<set_op>

|<comment>

| <input>

| <output>

| <read_line>

| <write>

| <console_read>

| <primitive_func>

|<del_set> <identifier>

|< increment >

| <decrement>

|<set_relations>

2.variable identifiers

<identifier> ::= <string>

| < string > <identifier>

| < string ><int>

|< string ><int> <identifier>

<declaration>::= <type_identifier> <identifier>

3.Assignment

<assignment>::= <identifier><assignment_operation><statement>

| <identifier><assignment_operation> <set_type> <set_list>

| <identifier><assignment_operation> <type_variable >

| <identifier><assignment_operation> <identifier>

<assignment_operation> :: = | -= | += | /= | %= |*= | **=

<expression>::= < type_variable > <relational> < type_variable >

| < type_variable> <arithmetic> < type_variable>

| < identifier> <relational> < identifier>

| < identifier> <arithmetic> < identifier>

|< identifier> <relational> < type_variable>

| < identifier> <arithmetic> < type_variable>

<relational_operation> :: > | <= | >= | < | == | < >

<arithmetic_operation>::= - | +| / | * | %| **

<increment> ::= < identifier >++

<decrement> ::= < identifier > --

4.Sets

< set_list > ::= <int_set>

|<string_set>

| <float_set>

| <char_set>

| <empty_set>

<int_set> ::=

{ int_set_var }

<string_set >::=

{ string_set_var }

<float_set >::=

{ float_set_var }

<char_set >::=

{ char_set_var}

<int_set_var >::=

<int>

| <int> , int_set_var;

<string_set_var >::=

< strings >

| < strings > , string_set_var

<float_set_var>::=

< float >

| < float >, float_set_var;

<char_set_var>::=

< char>

| < char>, char_set_var;

//set relations

<set_relations> ::= <set_list> <relation_operator> <set_list>

< relation_operator > ::= <subset>

| <not_belongs>

| <belongs>

|<supset>

|<proper_subset>

|<universal_set>

| <not_subset>

|<not_supset>

|<symmetric_diff>

<subset> ::= Ω

<not_belongs> ::= @’

<belongs> ::= @

<supset> ::= ¬

<proper_subset>::=ß

<not_subset>::= Ω’

<universal_set>::= α

<not_proper_subset>::= ß’

<symmetric_diff>::= Δ

//set operators

<set_opts> ::= <union>

| <intersec>

| <setdiff>

|<complement>

<set_op>::= <identifier> <set_opts> <identifier>

<union> ::= #

<intersec> :: $

<setdiff> ::= ~

<complement>::= ^

5.Boolean type

<boolean> ::= <true>

|<false>

<true> ::= TRUE |true | 1

<false> ::= FALSE |false | 0

6. Identifier type

<type_identifier>::= <int_type>

| <float_type>

| <string_type>

| <char_type>

| <boolean_type>

|<set_type>

|<void_type>

<set_type>::= intset| strset| boolset| floatset| charset

<del_set>::= del_set

<boolean_type>::= bool

<int_type>::= int

<float_type>::= fl
<string_type>::= str

<char_type>::= ch

<void_type>::=void

6.1 Variable Type

<type_variable>::= <int>

| <float>

| <string>

| <char>

| <boolean>

|<empty_set>

<int>::= <digit>

|<digit><int>

| <sign> <int>

<sign>::= +

|-

<float>::= .<int>

|<int>.<int>

<string> ::= ‘<chars>’

| ‘<chars> <string>’

<empty_set> ::= {}

<char> ::= “<chars>”

<chars> ::= a|b|c|ç|d|e|f|g|ğ|h|ı|i|j|k|l|m|n|o|ö|p|r|s|ş|t|u|ü|v|y|z|!|#|£|$|%|&|_|

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

<new_line> ::= \n

7. Loops

<loops>::= <while>
|<for>
|<do_while>

<while>::= while (<expression>) { <statements> }end

<for>::= for (<assignment> :: <expression> :: <expression>) { <statements> }end

<do_while>::= do { <statements> } while (<expression>)end

8.I/O:

<input>::= to <identifier>

<output>::= display <identifier>
| display <type_variable>

<read_line> ::= read(<string>)

<write>::= write(<string> in <string>)
|write(<identifier> in <string>)

<console_read>::= console.read(<string>)

9.Conditional Statements

<if_stmt>::= <if_con>
|<switch_con>

<if_con>::= <matched>
|<unmatched>

<matched>::= if (<expression>) <matched> else <matched>
|{<statements>}

<unmatched>::= if (<expression>) <if_con>

|if (<expression>) <matched> else <unmatched>

<switch_con>::= switch (<identifier>) <switch_con>
|<case_stm>
|<case_stm><switch_con>

<case_stm>::= case<expression>:<statements> cut
|<default-case>

<default-case>::= default : <statements> cut

10.Primitive:

<primitive_func>::= <max>
|<min>
|<avg>
|<sum>
|<mul>
|< cardinality >
|< dot_product >
|< cross_product >
|<abs>
|<find>
|<findset>
| <flush>
| <venn>
| <insert>
| <del>
| <size>
|<change>

<max> ::= < identifier >.max()

<min> ::= < identifier >.min()

<avg> ::= < identifier >.avg()

<sum> ::= < identifier >.sum()

<mul> ::= < identifier >.mul()

<cardinality> ::= < identifier >.cardinality()

< dot_product > ::= < identifier >. dot_product(< identifier >)

< cross_product > ::= < identifier >.cross_product(< identifier >)

<abs>::= < identifier >.abs()

<find>::= < identifier >.find(< identifier >)

<findset>::= < identifier >.findset(< identifier >)

<flush>::= < identifier >.flush()

<venn>::= < identifier >.venn(< identifier >)

<insert>::= < identifier >.insert(< identifier >)

<del>::= < identifier >.del(< identifier >)

<size>::= < identifier >.size()

<change>::= < identifier >.change(< identifier >, <int>)

11.Fucntion definitions and calls

<functions>::= <func_defi>

|<func_call>

|<func_void>

<func_defi>::= < type_identifier > < identifier >(<parameters>){<statements> <return> < return_statements >}

| < type_identifier > < identifier >(){<statements> <return> < return_statements >}

<func_void>::= < type_identifier > < identifier >(<parameters>){<statements> }

| < type_identifier > < identifier >(){<statements> }

<func_call>::= < identifier >(<call_parameters>)

| < identifier >()

| < identifier >(< identifier >())

| < identifier >(< identifier >(<call_parameters>))

< return_statements >::= <expression>;

| < identifier >;

|<type_variable>;

<call_parameters>::= < identifier >

| < identifier >, <call_parameters>

<parameters>::= <parameter>

|<parameter>,<parameters>

<parameter> ::= < type_identifier > < identifier >

<return>::=RETURN

12.Comments

<comment>::= ^ ^ <text>

<text>::= <string>

| <string> <text>

Explanation of FLEX Language:

1. <program>::= START <statements> STOP

This non-terminal is the single program that may includes several statements. Program starts with START and finished with STOP.

2.<statements>::=<statement>;

|<statement>;<statements>

Non-terminal statements contain only one statement or several statements.

3. <statement>::=<assignment>

|<expression>

| <declaration>

|<loops>

|<functions>

|<if_stmt>

|<set_op>

|<comment>

| <input>

| <output>

| <read_line>

| <write>

| <console_read>

| <primitive_func>

|<del_set> <identifier>

|< increment >

| <decrement>

|<set_relations>

A non-terminal statement can be one of them. These form the main structure of FLEX language.

2.<identifier> ::= <string>

| < string > <identifier>

| < string ><int>

|< string ><int> <identifier>

Identifiers contains string only or string + int or string + int + string... However, cannot start with integers.

<declaration>::= <type_identifier> <identifier>

Non-terminal declaration. First write the type of the variable and then write the variable name.

3. <assignment>::= <identifier><assignment_operation><statement>

| <identifier><assignment_operation> <set_type> <set_list>

| <identifier><assignment_operation> <type_variable >

| <identifier><assignment_operation> <identifier>

Variables can be assigned to all context of statement or a set or integer, strings, all other types in type_identifier section.

<assignment_operation> :: = | -= | += | /= | %= |*= | **=

Different options for assigning the variables. % means modulo, * means multiplication, ** means power.

<expression>::= < type_variable > <relational> < type_variable >

| < type_variable> <arithmetic> < type_variable>

| < identifier> <relational> < identifier>

| < identifier> <arithmetic> < identifier>

|< identifier> <relational> < type_variable>

| < identifier> <arithmetic> < type_variable>

Variable or variable names can be used in expression. Then, relational or arithmetic operations can be used or expression can be just an integer, float, string... all types in <type_identifier>.

<relational_operation> :: > | <= | >= | < | == | < >

< > means not equals

<arithmetic_operation>::= - | +| / | * | %| **

<increment> ::= < identifier >++

Increase the variable

<decrement> ::= < identifier > --

Decrease the variable

4. <int_set > ::=

{ int_set_var }

Non-terminal int_set includes integers

<string_set > ::=

{ string_set_var }

Non-terminal string_set includes strings

<float_set > ::=

{ float_set_var }

Non-terminal float_set includes float numbers

<char_set > ::=

{ char_set_var}

Non-terminal char_set includes chars

<int_set_var > ::=

<int>

| <int> , int_set_var;

int_set_var can be one integer or several integers

<string_set_var > ::=

< strings >

| < strings > , string_set_var

string_set_var can be one string or several strings

<float_set_var > ::=

< float >

| < float >, float_set_var;

float_set_var can be one float number or several float numbers

<char_set_var > ::=

< char>

| < char>, char_set_var;

char_set_var can be one char or several chars

//set relations

<set_relations> ::= <set_list> <relation_operator> <set_list>

Non-terminal set_relations will have two non-terminal set_list and non-terminal relation operator

< relation_operator > ::= <subset>

| <not_belongs>

| <belongs>

|<supset>

|<proper_subset>

|<universal_set>

| <not_subset>

|<not_supset>

|<symmetric_diff>

Non-terminal relation operator can be one of them above and the non terminals' symbols are below

<subset> ::= Ω

Subset operation

<not_belongs> ::= @’

Not belongs to something operation

<belongs> ::= @

belongs to something operation

<supset> ::= ¬

Show super set operation

<proper_subset>::=ß

Show proper set operation

<not_subset>::= Ω’

Show not proper set operation

<universal_set>::= α

Show universal set

<not_proper_subset>::= ß’

Show not proper subset operation

<symmetric_diff>::= Δ

Symmetric difference operation

//set operators

<set_opts> ::= <union>

| <intersec>

| <setdiff>

|<complement>

Non teriminal set operators can be union, intersection, set difference and complement.

<set_op>::= <identifier> <set_opts> <identifier>

Non teriminal set operation shows relation between two identifiers

<union> ::= #

Operate union action

<intersec> :: $

Operate intersection action

<setdiff> ::= ~

Operate set difference action

<complement>::= ^

Operate complement action

5.<boolean> ::= <true>

|<false>

Non-terminal Boolean can be true or false

<true> ::= TRUE |true | 1

Non-terminal true has several types

<false> ::= FALSE |false | 0

Non-terminal false has several types

Boolean type includes true and false values that are not case sensitive. Also, true and false can be represented as 1 and 0.

6. <type_identifier>::= <int_type>

| <float_type>

| <string_type>

| <char_type>

| <boolean_type>

|<set_type>

|<void_type>

Non-terminal type identifier is used to show the type of a identifier and they can be integer, float, string, character, boolean, set, void.

<set_type>::= intset| strset| boolset| floatset| charset

Non-terminal set type has integer set, string set, boolean set, float set, char set

<del_set>::= del_set

Special key word to destroy the set. Not for deleting an element

<boolean_type>::= bool

Reserved key identifies boolean type

<int_type>::= int

Reserved key identifies integer type

<float_type>::= fl

Reserved key identifies float type

<string_type>::= str

Reserved key identifies string type

<char_type>::= ch

Reserved key identifies character type

<void_type>::=void

Reserved key identifies void type

6.1<type_variable>::= <int>

| <float>

| <string>

| <char>

| <boolean>

|<empty_set>

Non-terminal type_variable has several option integer, float number, string, char, boolean, empty set.

<int>::= <digit>

|<digit><int>

| <sign> <int>

Integers can be positive or negative integers

<sign>::= +

|-

non terminal sign shows positivity or negativity

<float>::= .<int>

|<int>.<int>

float numbers's format can be imteger.integer or .integer

<string> ::= '<chars>'

| '<chars> <string>'

Strings are made of several chars and start with one ‘ symbol and end with one ’ symbol

<empty_set> ::= {}

This is empty set symbol made of right and left curly bracket.

<char> ::= “<chars>”

Characters are showed in “ ”

<chars> ::= a|b|c|ç|d|e|f|g|ğ|h|ı|i|j|k|l|m|n|o|ö|p|r|s|ş|t|u|ü|v|y|z|!|#|£|$|%|&|_|

All characters in the language

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

All digits in the language

<new_line> ::= \n

This the special character to identify new line.

7. <loops>::= <while>
|<for>
|<do_while>

Non terminal loops can be while loop, for loop or do while loop.

<while>::= while (<expression>) { <statements> }end

Non terminal while includes expression in the parenthesises the statement starts with curly left bracket and finishes with right curly bracket. While loop ends with the keyword “end”.

<for>::= for (<assignment> :: <expression> :: <expression>) { <statements> }end

In non-terminal for, “::” is used for separation and the loop ends with the keyword “end”.

<do_while>::= do { <statements> } while (<expression>)end

In Non terminal do while, statements are in the curl brackets and the expression in parentheses. the loop ends with the keyword “end”.

8.<input>::= to <identifier>

To get input from keyboard to a variable.

<output>::= display <identifier>

| display <type_variable>

To print variables to console also print integers, strings, chars also set.

<read_line> ::= read(<string>)

To read sets from file. File name should be string type like read(‘test.txt’).

<write>::= write(<string> in <string>)

|write(<identifier> in <string>)

To write file to file or write variables to a file. Example, write (‘test1.txt’ in ‘test2.txt’). This write tes1.txt to end of test2.txt. File name should be string type and indicate with “in” keyword. Example write(set1 in ‘test1.txt’).

<console_read>::= console.read(<string>)

To read from console that indicates with string.

9. <if_stmt>::= <if_con>
|<switch_con>

Non-terminal if statement can be if condition or swicth condition

<if_con>::= <matched>
|<unmatched>

Non-terminal If conditional has statements and it has two types marched and unmatched to prevent ambiguity in the language

<matched>::= if (<expression>) <matched> else <matched>

|{<statements>}

If the expression in the parenthesises is correct, non-terminal matched will be taken

<unmatched>::= if (<expression>) <if_con>
|if (<expression>) <matched> else <unmatched>

If the expression in the parenthesises is not correct non-terminal unmatched will be taken

<switch_con>::= switch (<identifier>) <switch_con>
|<case_stm>
|<case_stm><switch_con>

Non-terminal switch_con takes identifier and do the cases.

<case_stm>::= case<expression>:<statements> cut
|<default-case>

Non-terminal case statement evaluates the expression and compare with identifier in the statements and evaluates the expression and compare with identifier in the statements. Proper statement will be executed.

<default-case>::= default : <statements> cut

Non-terminal default case start with default key word and end with cut and, it executes the default case in no case scenario.

10. <primitive_func>::= <max>

|<min>

|<avg>

|<sum>

|<mul>

|< cardinality >

|< dot_product >

|< cross_product >

|<abs>

|<find>

|<findset>

| <flush>

| <venn>

| <insert>

| <del>

| <size>

|<change>

Non-terminal primitive function has several operations to increase ability of the language

<max> ::= < identifier >.max()

Non terminal max works like in object-oriented languages and find the maximum element in the set

<min> ::= < identifier >.min()

Non terminal min works like in object-oriented languages and find the minimum element in the set

<avg> ::= < identifier >.avg()

Non terminal avg works like in object-oriented languages and find the average of the elements in the set

<sum> ::= < identifier >.sum()

Non terminal sum works like in object-oriented languages and find the sum of the elements in the set

<mul> ::= < identifier >.mul()

Non terminal mul works like in object-oriented languages and find the multiplication of the elements in the set

<cardinality> ::= < identifier >.cardinality()

Non terminal cardinality works like in object-oriented languages and find the number of the elements in the set

< dot_product > ::= < identifier >. dot_product(< identifier >)

Non terminal dot product works like in object-oriented languages and find the dot product of two sets

< cross_product > ::= < identifier >.cross_product(< identifier >)

Non terminal cross product works like in object-oriented languages and find the cross product of two sets

<abs>::= < identifier >.abs()

Non terminal abs works like in object-oriented languages and find the absolute values of the elements in a set

<find>::= < identifier >.find(< identifier >)

Non terminal find works like in object-oriented languages and search an element in the set.

<findset>::= < identifier >.findset(< identifier >)

Non terminal find set works like in object-oriented languages and search a set in the set.

<flush>::= < identifier >.flush()

Non terminal flush works like in object-oriented languages and create an empty set from the identifier.

<venn>::= < identifier >.venn(< identifier >)

Non terminal venn works like in object-oriented languages and create venn schema from the two sets.

<insert>::= < identifier >.insert(< identifier >)

Non terminal insert works like in object-oriented languages and insert an element end of the set.

<del>::= < identifier >.del(< identifier >)

Non terminal del works like in object-oriented languages and delete an element from the set.

<size>::= < identifier >.size()

Non terminal del works like in object-oriented languages and gives the size of the set.

<change>::= < identifier >.change(< identifier >, <int>)

Non terminal change works like in object-oriented languages and change an element from the set according to given index. Set1.change(value, 0) puts the value into index zero in the set and initial value at the index zero will go to value's old position.

11 <functions>::= <func_defi>

|<func_call>

|<func_void>

Non terminal functions can be function definition, functions call or void function definition.

<func_defi>::= < type_identifier > < identifier >(<parameters>){<statements> <return> < return_statements >}

| < type_identifier > < identifier >(){<statements> <return><return_statements >}

Function is defined as return type, function name, parameters, statements, return statement.

<func_void>::= < type_identifier > < identifier >(<parameters>){<statements> }

| < type_identifier > < identifier >(){<statements> }

Void is the return type here. Void function is defined as function name, parameters, statements.

<func_call>::= < identifier >(<call_parameters>)

| < identifier >()

| < identifier >(< identifier >())

| < identifier >(< identifier >(<call_parameters>))

Functions can call with or without parameters also a function can call another function.

< return_statements >::= <expression>;

| < identifier >;

|<type_variable>;

Non terminal return statement can be expression or identifier or type variable like integer,string etc.

<call_parameters>::= < identifier >

| < identifier >, <call_parameters>

Non terminal call parameters are parameters that can be one or more in function call another function call relation.

<parameters>::= <parameter>

|<parameter>,<parameters>

Non terminal parameters are parameters that can be one or more in function definitions.

<parameter> ::= < type_identifier > < identifier >

Non terminal parameter has its type and name

<return>::=RETURN

Reserved key for return keyword

12.<comment>::= ^ ^ <text>

Non terminal comment starts with ^^ symbol and it takes text.

<text>::= <string>

| <string> <text>

Non terminal text is typical comment text consist of strings.

Reserved Tokens:

Start

Stop

Set: Reserved for stating set data type.

del_set: Reserved for destroying the set data type.

bool: Reserved for stating Boolean data type.

int: Reserved for stating integer data type.

fl: Reserved for stating floating points number data type.

str: Reserved for stating string data type.

num: Reserved for stating special number data type.

ch: Reserved for stating character data type.

while: Reserved for stating while loops.

end: Reserved for loops.

for: Reserved for stating for loops.

do: Reserved for stating do while loops.

display: Reserved for calling display function.

to: Reserved for taking input from keyboard.

read: Reserved for calling read function.

in: Reserved for using in write function.

console.read: Reserved for console readings.

if: Reserved for stating if statements.

else: Reserved for stating else statements.

switch: Reserved for stating switch statements.

case: Reserved for stating cases.

cut: Reserved for switch-case statements.

default: Reserved for switch-case statements.

max: Reserved for calling max primitive function.

min: Reserved for calling min primitive function.

avg: Reserved for calling avg primitive function.

sum: Reserved for calling sum primitive function.

mul: Reserved for calling mul primitive function.

cardinality: Reserved for calling cardinality primitive function.

dot_product: Reserved for calling dot_product primitive function.

cross_product: Reserved for calling cross_product primitive function.

abs: Reserved for calling abs primitive function.

find: Reserved for calling find primitive function.

findset: Reserved for calling findset primitive function.

flush: Reserved for calling flush primitive function.

venn: Reserved for calling venn primitive function.

insert: Reserved for calling insert primitive function.

del: Reserved for calling del primitive function.

size: Reserved for calling size primitive function.

change: Reserved for calling change primitive function.

True: Reserved for when Boolean is true.

true: Reserved for when Boolean is true.

False: Reserved for when Boolean is false.

false: Reserved for when Boolean is false.

void: Reserved for void function

return: Reserved for return key word.

intset: Reserved for stating integer set data type.

strset: Reserved for stating string data type.

boolset: Reserved for stating boolean set data type.

floatset: Reserved for stating float number set data type.

charset: Reserved for stating character set data type.

Language Evaluation

Readability:

The Flex language can be traced easily while reading the program. It has understandable variable names, understandable function ends and so on. While loop, for loop, do-while loop statements have a proper ending as “end”. Moreover, switch case has also other ending as “cut”. Our language has “for, while, if” statements for understandability. The reason why Flex uses that idea is that programmers use different languages which have various syntax properties. We didn't want to create a different program because we want users to adapt to Flex program without a prior knowledge of the language. We want to increase the readability without using any extraordinary tokens or symbols to use. If someone is not really familiar with these kinds of languages, even they can easily learn our language because of its convenience.

Writability:

User can experience flexibility with the Flex language while they are writing codes. Also, our language is created to easily adapt and be used fast without a misunderstanding. Moreover, it is created with a different assigning order. It can be used with right associativity when assigning or expressions. We think that Flex language can be used all around the world. Also, we expanded it by adding Turkish letters in our language. This can increase the desire to use our program language. Inversely, Flex language has similar structures with popular programs all around the world.

Reliability:

The Flex language offers users reliability. Through both readability and writability, tokens are created for increasing reliability. For instance, we designed the while loop with “while (some expression)”, and after writing the calculation that the user wants, we created a token “end” to understand that user comes to ending the while loop. It increases understandability and it becomes clear that users can use our language easily. Another example, we have implemented tokens as “to” and “display”. These are used for input/output operations that increase reliability in giving a more accurate usability without a misunderstanding. Finally, in Flex language, the user can write variables, functions etc. that are very understandable by the user and also shows the expected outputs.