CS 315 Project Report

CS 315 - Spring 2018

Project 2
Group No 4

Group members:

Alemdar Salmoor	21500430	Section 2
Kasymbek Tashbaev	21603136	Section 2
Alina Zhumasheva	21503347	Section 2

TBFL

Overview of TBFL:

TBFL(The Best Friendly Language) is a simple language designed for operations on sets. It provides users to declare a new data type ‘set’, as well as other data types, such ‘num’, ‘char’ and ‘boolean’. For example,

set @X = { 1, 5, 6}; is declaration and initialization of ‘set’ data type.

User can do set operations and some mathematical operations on the sets: union(<), intersection(>), difference(-), cross product(*). For example,

*set @Y = @X - { 2, 5, 6}; /** @Y = { 1 } ***

Also it provides user to write comments on code, define functions, use for-loop and if-else statement and some other features:

```
/** for-loop **\
for ( num @n = 0; @n << 10; @n ++) do
        /** if statement **\
        if( @X->size << 10) do
                @X < @n;
        end
end
```

BNF (modified):

****New: added set_relations (subset and superset); some rules are changed to have no ambiguity****

<program> → do <stmt_list> end

<stmt_list> → <stmt>
| <stmt_list> <stmt>

<stmt> → <matched_stmt>
| unmatched_stmt

<matched_stmt> → if <condition_stmts> do <matched_stmt> else <matched_stmt> end
| <non_alternative_stmt>

<unmatched_stmt> → if <condition_stmts> do <stmt_list> end
| if <condition_stmts> do <matched_stmt> else <unmatched_stmt> end

<condition_stmts> → <condition_stmt>
| <condition_stmts> <logical_operator> <condition_stmt>

<condition_stmt> → <boolean>
| <variable>

<non_alternative_stmt> → <assignment>
| <variable_declaration>
| <for_loop>
| <function_stmt>
| <io_assignment>
| <comment>
| \n
| ;

<set_relation> → <superset>
| <subset>

<powerset> → <powerset_sign> <set>
| <powerset_sign> <powerset>

<partition> → <partition_sign> <set>
| <partition_sign> <partition>

<complement> → <set> <complement_sign>
| <complement> <complement_sign>

<superset> → <set> <superset_sign> <set>
| <superset> <superset_sign> <set>

<subset> → <set> <subset_sign> <set>
| <set> <subset_sign> <subset>

<for_loop> →
for (<assignment> <condition_stmts> ; <assignment>) do <stmt_list> end
| for (; <condition_stmts> ; <assignment>) do <stmt_list> end
| for (<assignment> ; <condition_stmts> ;) do <stmt_list> end
| for (; <condition_stmts> ;) do <stmt_list> end

<variable_declaration> → <parameter>
| <variable_type> <assignment>

<variable_type> → boolean | set | num | char

<variable> → <variable_identifier><id>

<variable_identifier> → @

<assignment> → <variable> <assignment_operator> <value>

<assignment_operator> → = | += | -= | <= | >= | ++ | --

<value> → <set_element>
| <boolean>
| <math_operation>
| <function_call>
| <set_relation>
| <powerset>
| <partition>
| <complement>

<math_operation> → <num> <math_operator> <num>
| <math_operation> <math_operator> <num>

<set_operation> → <set_val> <set_operator> <set_val>

<set_val> → { <set_elements> }
| <variable>

<function_call> → <variable> <call_sign> <default_function>
| <function_name> (<parameter_list>)

<function_stmt> →
function <function_name> (<parameter_list>) do <stmt_list> return <value> ;
end
| function <function_name> (<parameter_list>) do <stmt_list> end
| function <function_name> (<parameter_list>) do <stmt_list> return <value> ;
end
| function <function_name> ( ) do <stmt_list> end

<parameter_list> →
<parameter>
| <parameter_list> , <parameter>

<parameter> →
<variable_type> <variable>

<io_assignment> →
<input_assignment>
| <output_assignment>

<io_assignment> → <input_assignment> | <output_assignment>

<input_assignment> → <variable> =< tbin

<output_assignment> → tbout => <output_vals>

<output_vals> → <output_vals> => <output_val>
| <output_val>

<output_val> → <value>
| “ <string> “

<boolean> → <boolean_literal>
| <comp_val> <comparison_operator> <comp_val>

<comp_val> →<set_element>
| <math_operation>
| <function_call>
| <set_relation>
| <powerset>
| <partition>
| <complement>

<set> → { <set_elements> }
| <set_operation>
| <variable>

<logical_operator> → && | ||

<set_elements> → <set_elements> , <set_element>
| <set_element>

<set_element> → <num>
| <set>
| <char>

<variable_type> → boolean | set | num | char

TOKENS:

<comparison_operator> → == | << | >> | <=< | >=> | !=
<logical_operator> → && | ||
<default_function> → size, at[num n], remove[num n]
<char> → ‘ <letter> ’ | ‘ <special_characters> ’ | ‘ <math_operator> ’ | ‘ <set_operator> ’
| ‘ <digit> ’ | ‘ ’ | ‘ other single characters ’
<num> → - <positive_num> | <positive_num>
<positive_num> → <non zero digit>
| <non zero digit> <digits>
<digits> → <digit> | <digit> <digit>
<digit> → 0 | <non_zero_digit>
<non_zero_digit> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<boolean_literal> → true | false
<id> → <name_string> except <keywords>
<name_string> → <name_string> <letters> | <name_string> <num> | <letters>
<string> → <char><string> | <string> <string> |<char> | ε(ellipsis) | \t | \n
<letters> → <letter><letters> | <letter>
<letter> → a to z | A to Z

<special_characters> → , | . | ! | ? | ~ | @ | # | $ | other symbols
<keywords> → do | end | num | boolean | set | char| true | false | if | else | for | function | tbin
|tbout | return
<union> → <
<intersection> → >
<set_difference> → -
<cross_product> → *
<powerset_sign> → %
<complement_sign> → ^
<partition_sign> → #
<math_operator> → + | - | / | *

<subset_sign> → /<
<superset_sign> → />

Nonterminals explanation

Below is the explanation of nonterminals of TBFL programming language.

<program> start at keyword ‘do’, consist of <stmt_list> and ends at keyword ‘end’.

<stmt_list> consist of many <stmt> that in total do the all work of the program.

<stmt> → <matched_stmt>
| unmatched_stmt The statement consist of matched and unmatched, that expand to if/else statements.

<matched_stmt> → if <condition_stmts> do <matched_stmt> else <matched_stmt> end
| <non_alternative_stmt>

Where in case of matched statements, according to the condition statements, expands further to matched statements or non_alternative_stmt. Where If/else statements check whether the condition is either true or false and will do corresponding statements that between ‘do’ and ‘end’ keywords.

<unmatched_stmt> → if <condition_stmts> do <stmt_list> end
| if <condition_stmts> do <matched_stmt> else <unmatched_stmt> end

<condition_stmts> → <condition_stmt> Compare
| <condition_stmts> <logical_operator> <condition_stmt> statements define to

<condition_stmt> → <boolean>
| <variable>

boolean
variable.

<non_alternative_stmt> → <assignment>
| <variable_declaration>
| <for_loop>
| <function_stmt>
| <io_assignment>
| <comment>
| \n
| ;

These statements expand to the declaration, assignment, for loop, function statements, input-output statements, or comments.

<set_operator> → <union>
| <intersection>
| <set_difference>
| <cross_product>

<set_operator> does the following set operation on two or more sets:
<intersection> operator takes out elements which are in both A and B sets in another set.
Union operator gives the set of elements which are in A, in B, or in both A and B. Set_difference operator (A - B), provides with the set of elements which are only in A but not in B. Cross_product operator gives the The Cartesian product of the sets.

<set_relation> → <superset>
| <subset>

<set_relation> further expands to <superset>, or <subset>. Set is the superset of the other set if it includes it. And vice versa for subset. Set relations return boolean.

<powerset> → <powerset_sign> <set>
| <powerset_sign> <powerset>

<partition> → <partition_sign> <set>
| <partition_sign> <partition>

<complement> → <set> <complement_sign>
| <complement> <complement_sign>

To use <power_set> on the set <power_sign> should be written before the set, the same syntax goes for <partition> and <complement>, with <partition_sign> and <complement_sign> respectively, however in <complement> syntax, sign goes after the <set>, not before.
<power_sign> → %
<complement_sign> → ^
<partition_sign> → #

<for_loop> →

for (<assignment> <condition_stmts> ; <assignment>) do <stmt_list> end
| for (; <condition_stmts> ; <assignment>) do <stmt_list> end
| for (<assignment> ; <condition_stmts> ;) do <stmt_list> end
| for (; <condition_stmts> ;) do <stmt_list> end

Statement in a for loop, that should be executed if it matches with conditions, should be between ‘do’ and ‘end’ keywords.

<variable_declaration> → <parameter>
in
| <variable_type> <assignment>

<variable_type> → boolean | set | num | char
<variable> → <variable_identifier><id>
<variable_identifier> → @

<assignment> → <variable> <assignment_operator> <value>
<assignment_operator> → = | += | -= | <= | >= | ++ | --

Variable declaration in our language is a simple process, similar to most of the languages. However, we have @ as our variable identifier. When set is declared it is automatically empty set. For example, set @X; /**@X = {} **\ .

<value> → <set_element>
| <boolean>
| <math_operation>
| <function_call>
| <set_relation>
| <powerset>
| <partition>
| <complement>

User can assign value to an already defined variable. Besides values, result of called functions and operations can be assigned. .

<math_operation> → <num> <math_operator> <num>
| <math_operation> <math_operator> <num>

Math operation is addition/subtraction/multiplication/ division of two num values.

<set> → { <set_elements> }
| <set_operation> | <variable>

Sets in our language can be sequence of set elements, which can be numbers, characters and other sets. All elements should be inside the braces. There are two types of operations: <math_operation> and <set_operation>, that was explained above.

<set_elements> → <set_elements>, <set_element>
| <set_element>

<set_element> →| <num> | <set> | <char>

<set_operation> → <set_val> <set_operator> <set_val>

<set_val> → { <set_elements> }
| <variable>

<comment> → /** <string> **\

Comments are strings that do not affect the work of the program. In our language they should should be taken insine /** and **\ .

<boolean> → <boolean literal>
| <comp_val> <comparison_operator> <comp_val>

Boolean variable can either be declared as <boolean_literal> that is true or false, or check the relation of two comparison values and then be assigned true or false accordingly. Where comp_val is set element, math operation, function call, set relation, powerset, partition, complement.

<comp_val> →<set_element>
| <math_operation>
| <function_call>
| <set_relation>
| <powerset>
| <partition>
| <complement>

<function_call> → <variable> <call_sign> <default_function>
| <function_name> (<parameter_list>)

<function_stmt> →
function <function_name> (<parameter_list>) do <stmt_list> return <value> ;
end
| function <function_name> (<parameter_list>) do <stmt_list> end

| function <function_name> (<parameter_list>) do <stmt_list> return <value> ;
end
| function <function_name> ( ) do <stmt_list> end

<parameter_list> →
<parameter>
| <parameter_list> , <parameter>

<parameter> →
<variable_type> <variable>

<io_assignment> →
<input_assignment>
| <output_assignment>

<io_assignment> → <input_assignment> | <output_assignment>

<input_assignment> → <variable> =< tbin

<output_assignment> → tbout => <output_vals>

<output_vals> → <output_vals> => <output_val>
| <output_val>

<output_val> → <value>
| “ <string> “

TBFL allow user to define routine code as function to ease usage of language.There are some default function for sets, that you call as *set_name -> function[]*. Also users can define their own function as *function(parameters) do … end.* Functions called by *variable = function(...)*

<io_assignment> → <input_assignment> | <output_assignment>
<input_assignment> → <variable> =< tbin
<output_assignment> → tbout => <output>
<output> → <output> + <output>
| <value>
| “ <string> “

Input/output stream in TBFL is a simple process that has quite similar syntax to that of C group languages. However, we decided to make tbin (the best in) to be standard input stream, and tbout (the best out) to be standard output stream of our language, as the name of our language is TBFL.

Nontrivial tokens explanation

1. Comments

   Comments in our language are defined as /** <string> **\ . A comment, that can be a sequence of strings that should be enclosed in double stars and slashes.

2. Identifiers

   In our language, we used @ as an variable identifier. When interpreter will see @ in variable declaration statement, it will understand that after @ goes a name for a variable

   .

3. Literals

   String literals in our language are enclosed in double quotes “”. Character literals are enclosed in single quotes.

4. Reserved words

   We have some default functions for the set: at this function gets an element from the set -> size this function is used to return size of the set -> remove this function is used to remove certain element from the set.
   To declare a function, user should write reserved word *function* before the function name -> do - end are used to signify the start and end of the programs, functions, if/else/for/elseif/ statements.
   boolean, char, true, false, if, else,return are the same as in C, Java programming languages num is integer data type in TBFL
   tbin - is the TBFL specific standard input stream
   tbout - is the TBFL specific standard output stream
   set - is the TBFL specific data type

LEX (modified):

****New: added set_relations (subset and superset); some tokens were removed or changed; added code to count line number and to report error****

```
%{
int line_num = 1;
%}
lb                      \{
```

rb	\}
lp	\(
rp	\)
or	|
and	\&
not	\!
logical_or	"||"
logical_and	"&&"
variable	@[A-Za-z][A-Za-z0-9]*
assign_value	=|\+=|\-=|\<=|\>=|\+\+|\-\-
equals	==
not_eq	\!=
greater	\>\>
greater_eq	\>=\>
less	\<\<
less_eq	\<=\<
do	do
end	end
comment	\/**.****
newline	\n
for	for\(
tbin	tbin
tbout	tbout
in	=\<
out	=\>
true	true
false	false
string	\".*\"
return	return
function_def	function
char_val	\'.*\'
num_val	[+-]?[0-9]+
num	num
boolean	boolean
char	char
set	set
union	\<
intersection	\>
difference	\-
cross_product	*
addition	\+
division	\/
if	if

```
else                else
function_name       [A-Za-z][A-Za-z0-9]*\(
df_at               at\[[0-9]*\]
df_size             size
df_remove           remove\[[0-9]*\]
semicol             \;
powerset_sign       \%
complement_sign     \^
partition_sign      \#
subset_sign         \/\<
superset_sign       \/\>
comma               \,
call_sign           \-\>

%%
[ \t]               {}
{lb}                return LB;
{rb}                return RB;
{lp}                return LP;
{rp}                return RP;
{or}                return OR;
{and}               return AND;
{not}               return NOT;
{variable}          return VARIABLE;
{assign_value}      return VALUE_ASSIGN;
{equals}            return EQUALS;
{comment}           return COMMENT;
{newline}           {line_num++;
                    return NEWLINE;}
{true}              return TRUE;
{false}             return FALSE;
{string}            return STRING;
{return}            return RETURN;
{do}                return DO;
{end}               return END;
{not_eq}            return NOT_EQUAL;
{greater}           return GREATER;
{greater_eq}        return GREATER_OR_EQUAL;
{less}              return LESS;
{less_eq}           return LESS_OR_EQUAL;
{function_def}      return FUNCTION_DEFINE;
{num}               return NUMBER;
{boolean}           return BOOLEAN;
```

```
{num_val}           return NUMERIC_VALUE;
{set}               return SET;
{for}               return FORLP;
{tbin}              return THE_BEST_IN;
{tbout}             return THE_BEST_OUT;
{in}                return INPUT;
{out}               return OUTPUT;
{union}             return UNION;
{intersection}      return INTERSECTION;
{difference}        return DIFFERENCE;
{cross_product}     return CROSS_PRODUCT;
{addition}          return ADDITION;
{division}          return DIVISION;
{char}              return CHAR;
{if}                return IF;
{else}              return ELSE;
{function_name}     return FUNCTIONLP;
{char_val}          return CHARACTER_VALUE;
{df_size}           return SIZE;
{df_at}             return AT;
{df_remove}         return REMOVE;
{logical_or}        return LOGICAL_OR;
{logical_and}       return LOGICAL_AND;
{semicol}           return SEMICOL;
{powerset_sign}     return POWERSET_SIGN;
{complement_sign}   return COMPLEMENT_SIGN;
{partition_sign}    return PARTITION_SIGN;
{subset_sign}       return SUBSET_SIGN;
{superset_sign}     return SUPERSET_SIGN;
{comma}             return COMMA;
{call_sign}         return CALL_SIGN;
.                   return -1;
%%
yywrap() { return 1; }
```

YACC:

```
%{
   #include <stdio.h>
   int yylex(void);
   void yyerror(char *);
%}
%start program
```

```
%token LB RB LP RP SEMICOL
%token REMOVE AT SIZE
%token CHARACTER_VALUE NUMERIC_VALUE
%token VARIABLE
%token VALUE_ASSIGN
%token AND OR LESS GREATER LESS_OR_EQUAL GREATER_OR_EQUAL
LOGICAL_AND LOGICAL_OR
%token FUNCTIONLP FUNCTION_DEFINE
%token NOT
%token NUMBER BOOLEAN SET CHAR STRING
%token FORLP
%token ELSE
%token RETURN IF
%token NEWLINE
%token OUTPUT
%token DO END
%token INPUT OUTPUT
%token TRUE FALSE
%token COMMENT
%token ADDITION DIFFERENCE
%token CROSS_PRODUCT DIVISION
%token INTERSECTION UNION
%token THE_BEST_OUT THE_BEST_IN
%token EQUALS NOT_EQUAL
%token POWERSET_SIGN PARTITION_SIGN
%token COMPLEMENT_SIGN
%token SUBSET_SIGN
%token SUPERSET_SIGN
%token COMMA
%token CALL_SIGN

%left AND OR LESS GREATER LESS_OR_EQUAL GREATER_OR_EQUAL LOGICAL_AND
LOGICAL_OR
%nonassoc ELSE
%left ADDITION DIFFERENCE
%left CROSS_PRODUCT DIVISION
%left INTERSECTION UNION
%left THE_BEST_OUT THE_BEST_IN
%nonassoc EQUALS NOT_EQUAL
%right POWERSET_SIGN PARTITION_SIGN
%left COMPLEMENT_SIGN
%right SUBSET_SIGN
%left SUPERSET_SIGN
```

```
%%

program:
DO stmt_list END {  printf("\nValid syntax!\n");
                                        return 0;}
;

stmt_list:
stmt
| stmt_list stmt
;

stmt:
matched_stmt
| unmatched_stmt
;

matched_stmt:
IF condition_stmts DO matched_stmt ELSE matched_stmt END
| non_alternative_stmt
;

unmatched_stmt:
IF condition_stmts DO stmt_list END
| IF condition_stmts DO matched_stmt ELSE unmatched_stmt END
;

condition_stmts:
condition_stmt
| condition_stmts logical_operator condition_stmt
;

condition_stmt:
boolean
| VARIABLE
;

non_alternative_stmt:
assignment
| variable_declaration
| for_loop
| function_stmt
| io_assignment
```

```
| COMMENT
| NEWLINE
| SEMICOL
;

set_relation :
superset
| subset
;

powerset:
POWERSET_SIGN set
| POWERSET_SIGN powerset
;

partition:
PARTITION_SIGN set
| PARTITION_SIGN partition
;

complement:
set COMPLEMENT_SIGN
| complement COMPLEMENT_SIGN
;

superset:
set SUPERSET_SIGN set
| superset SUPERSET_SIGN set
;

subset:
set SUBSET_SIGN set
| set SUBSET_SIGN subset
;

for_loop:
FORLP assignment SEMICOL condition_stmts SEMICOL assignment RP DO stmt_list END
| FORLP SEMICOL condition_stmts SEMICOL assignment RP DO stmt_list END
| FORLP assignment SEMICOL condition_stmts SEMICOL RP DO stmt_list END
| FORLP SEMICOL condition_stmts SEMICOL RP DO stmt_list END
;

variable_declaration:
```

```
parameter
| variable_type assignment
;

assignment:
VARIABLE VALUE_ASSIGN value

value:
set_element
| boolean
| math_operation
| function_call
| set_relation
| powerset
| partition
| complement
;

math_operation:
NUMERIC_VALUE math_operator NUMERIC_VALUE
| math_operation math_operator NUMERIC_VALUE
;


set_operation:
set_val set_operator set_val
;

set_val:
LB set_elements RB
| VARIABLE
;

function_call:
VARIABLE CALL_SIGN default_function
| FUNCTIONLP parameter_list RP
;

function_stmt:
FUNCTION_DEFINE FUNCTIONLP parameter_list RP DO stmt_list RETURN value SEMICOL
END
| FUNCTION_DEFINE FUNCTIONLP parameter_list RP DO stmt_list END
| FUNCTION_DEFINE FUNCTIONLP RP DO stmt_list RETURN value SEMICOL END
```

```
| FUNCTION_DEFINE FUNCTIONLP RP DO stmt_list END
;


parameter_list:
parameter
| parameter_list COMMA parameter
;

parameter:
variable_type VARIABLE
;

io_assignment:
input_assignment
| output_assignment
;

input_assignment:
VARIABLE INPUT THE_BEST_IN
;

output_assignment:
THE_BEST_OUT OUTPUT output_vals
;

output_vals:
output_vals OUTPUT output_val
| output_val
;

output_val:
value
| STRING
;

boolean:
boolean_literal
| comp_val comparison_operator comp_val
;

comp_val:
set_element
```

```
| math_operation
| function_call
| set_relation
| powerset
| partition
| complement
;

set:
LB set_elements RB
| set_operation
| VARIABLE
;

logical_operator:
LOGICAL_AND
| LOGICAL_OR
;

set_elements:
set_elements COMMA set_element
| set_element
;

set_element:
NUMERIC_VALUE
| set
| CHARACTER_VALUE
;

variable_type:
BOOLEAN
| SET
| NUMBER
| CHAR
;

default_function:
SIZE
| AT
| REMOVE
;
```

```
boolean_literal:
TRUE
| FALSE
;

math_operator:
ADDITION
| DIFFERENCE
| DIVISION
| CROSS_PRODUCT
;

set_operator:
UNION
| INTERSECTION
| DIFFERENCE
| CROSS_PRODUCT
;

comparison_operator:
EQUALS
| LESS
| GREATER
| LESS_OR_EQUAL
| GREATER_OR_EQUAL
| NOT_EQUAL
;

%%
#include "lex.yy.c"
extern int line_num;

void yyerror( char *s )
{
        fprintf(stderr,"\nSyntax Error in line: %d\n",line_num);
}

int main(void) {
   yyparse();
   return 0;
}
```

Example Programs

```
do
	/**calculator for sets**\

	char @repeat = 'Y';

	for( ; @repeat == 'Y'; ) do
		tbout => "Enter the first set as {num or char, ...}";
		set @A;
		@A =< tbin;
		tbout => "Enter an operation (< for union, > for intersection, - for difference, * for
cross product)";
		char @op;
		@op =< tbin;
		tbout => "Enter the second set as {num or char, ...}";
		set @B;
		@Y =< tbin;

		set @C;

		if @op == '<' do
			@C = @A < @B;
		end
		if @op == '>' do
			@C = @A > @B;
		end
		if @op == '-' do
			@C = @A - @B;
		end
		if @op == '*' do
			@C = @A * @B;
		end

		tbout => "The result of " => @A => @op => @B => " is " => @C;

		tbout => "Do you want to use calculator? Enter Y or N"
		@repeat =< tbin;

	end
```

```
        /** Simple operations on sets and other data types **\
        set @X;
        @X = { 1, 5, 6};
        set @Y;
        @Y = @X - { 2, 5, 6};                   /** @Y = { 1 } **\
        set @Z;
        @Z = { 1, 5, 6};
        @Z -= { 2, 5, 6};                       /** @Y = { 1 } **\

        if @Z == @Y do
                @Z <= { 2, 3};
        end

        set @PowerSet =  %@Z;                   /** @Z = { 1, 2, 3 } **\

        /** @PowerSet = { {1, 2, 3}, {1, 2}, {1, 3}, {2, 3}, {1}, {2}, {3}, {}  } **\

        num @powSetSize;
        @powSetSize = @PowerSet -> size;
        @PowerSet <= { @X, @Y};
        @powSetSize += 2;
        /** {1, 5, 6} and {1} added to @PowerSet **\

end
```