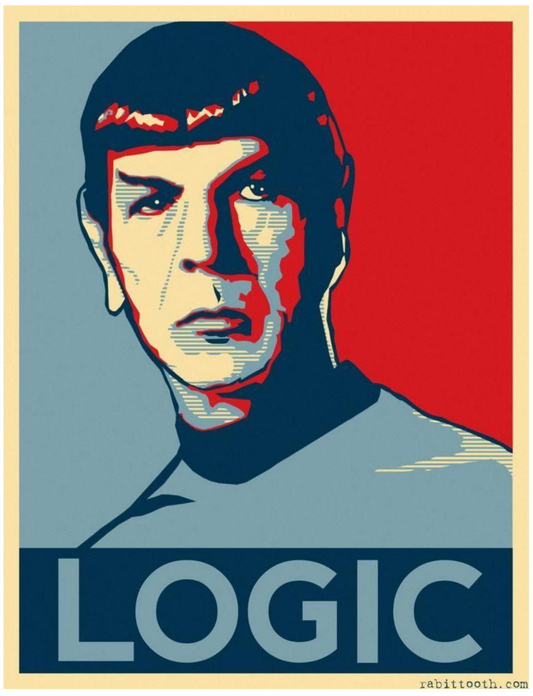
Group 2 - VolCAN Language



Alper Şahıstan – Section 1 - 21501207

E. Batuhan Kaynak – Section 1 - 21501178

Deniz Sipahioğlu – Section 3 - 21501526

In” Star Trek”, Vulcans are extremely logical beings as Mr. Spock says. Since this is a programming language for propositional calculus which is at core is logic, and since the main purpose of this language is to tell users what is true or false, we thought that it would be appropriate to name our language “VULCAN”. Yet we also though forming the language with parodic Turkish words was a good idea as well. So “VULCAN” turned into “VoLCAN”.

PART A)

The complete BNF description of VolCAN is shown below. The language constructs will be explained after the BNF description.

<program> → **tanum_başladu** <predicates> **tanum_bittu** <stmts>

| <stmts>

| <empty>

<predicates> → <predicates> <predicate> | <predicate>

<predicate> → <predicate_name> (<parameter list>) <body>

<body> → { <stmts> **dondür**<proposition>; }

| { <stmts> **dondür** <predicate instantiation>; }

<parameter list> → <empty>

|<parameter>

|<parameter>, <parameter list>

<parameter> →<atomic proposition>

<predicate_name> → <capital letter> <letters>

<stmts> → <stmt> ;

| <stmt> ; <stmts>

| $ <comment> $ <stmts>

| <stmt>; $ <comment> $

<stmt> → <matched> | <unmatched>

<matched> → **eğer** (<atomic proposition>) <matched> **değülse** <matched>

| <assignment statement>

| <loop statement>

|<print statement>

| <scan statement>

|<list initialization>

<unmatched> → **eğer** (<atomic proposition>) <stmts>

| **eğer** (<atomic proposition>) <matched> **değülse** <unmatched>

<assignment statement> → <var> = <proposition>

| <var> = <list initialization>

| <var> = <predicate instantiation>

| <list index> = <atomic proposition>

| <const> = <logic value>

<loop statement> → **turla** (<atomic proposition>) { <stmts> }

<list initialization> → **dizü[** **<**integer> **]**

<list index> → **<**var**>** **[** <integer> **]**

<print statement> → **bastur** <proposition>

| **bastur** <predicate instantiation>

<scan statement> → **tarattur** <var>

<predicate instantiation> → <predicate_name> (<parameter list>)

<var> → <letters>

<const> → <capital letters>

<letters> → <letter> | <letters> <letter>

<capital letters> → <capital letter>| <capital letters> <capital letter>

<letter> → a | b | c | … | z | _

<comment> → every possible ascii except '$'

<capital letter> → A |B |C|...| Z | _

<digit> → 0 | 1 | 2 | … | 9

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

<proposition> → <proposition> \<-> <imp_proposition>

| <imp_proposition>

<imp_proposition> → <imp_proposition> -> <or_proposition>

| <or_proposition>

<or_proposition> → <or_proposition> \| <and_proposition>

| <and_proposition>

<and_proposition> → <and_proposition> & <not_proposition>

| <not_proposition>

<not_proposition> → NOT <placeholder>

| <placeholder>

<placeholder> → (<proposition>)

| <atomic proposition>

<atomic proposition> → <var> | <logic value> | <const> | <list index>

<logic value> → **doğridur** | **yanluştur**

<empty> →

VoLCAN does not have a main method in the sense that languages like C or Java does. VoLCAN does not require any reserved words to mark beginning of code blocks. It does require reserved words to mark the beginning and ending of predicate declarations with **tanum_başladu**(uses "BEGN" token instead of "BEGIN" since BEGIN was used by the yacc at some point.) and **tanum_bittu** reserved words. The absence of keyword main method is inspired by languages like Perl and Python, we hope to improve the quality of the language by providing this feature. Small amounts of code can be written and executed without the unnecessary overhead of main method (e.g. as in Java).

Precedence of logical operators in VoLCAN is as follows:

negation (~) >> and (&) >> or (|) >> implies (->) >> if-and-only-if (<->)

Our reasoning in these selections come from mathematics and general programming habits. We chose negation to have the most precedence since it is a unary operator and mostly has the highest precedence in maths applications. We chose and to have higher precedence since it is mostly implemented this way in contemporary languages. We want to improve writability and readability by doing this.

<program>: This non-terminal is an abstraction of the main program flow. In VoLCAN, we can write our programs with the help of predicates, but this is not required. If we want to have predicates, they have to be declared between **tanum_başladu** and **tanum_bittu** reserved words. Then the program statements themselves can come after the **tanum_bittu** reserved word. If we do not need/want any predicate declarations, we can start writing our program statements without using **tanum_başladu** and **tanum_bittu** reserved words.

<predicates>: This non-terminal allows left recursive definitions of <predicate>.

<predicate>: This non-terminal specifies how a logic function (predicate) can be declared.

<body>: This non-terminal is where the actual predicate statements are written. After writing the necessary statements, we can either return (using **dondür** <body>'s reserved word) the calculated value of a proposition or call a predicate and return the return value of the callee.

<parameter list>: This non terminal denotes what are accepted as parameters to predicate calls. We can have predicates that receive no parameters, a single parameter, or multiple parameters separated by a comma (',').

<**parameter>:** This non-terminal denotes what is accepted as a parameter to a predicate call.

<predicate_name>: This non-terminal defines what counts as a predicate name, which is used to differentiate between predicate declarations/calls and other language constructs that use string literals to build (such as variables and constants).

<stmts>: This non-terminal designates how individual <stmt> non-terminals can be put together. We can either put a single <stmt> and end this statement with a semicolon (';'). Or, we can write more than one statement by putting semicolons in between each statement. We can also put comments before, after or in between statements.

<stmt>: This non-terminal is an abstraction to all kind of statements we can write in our program. We use <matched> and <unmatched> non-terminals to match **eğer** and **değülse** reserved words.

<matched>: This non-terminal contains all types of statements we can write in VoLCAN, with the exception of conditional statements with unequal amounts of **eğer** and **değülse**.

<unmatched>: This non-terminal contains a conditional statement with only one **eğer**, or a recursive rule for creating additional conditional statements.

<assignment statement>: This non-terminal specifies the ways to bind a value at right hand side of the assignment operator to a variable/constant at the left hand side of the assignment operator. We can make a variable hold the evaluated value of a proposition, the list initialization, the return value of a predicate call, assign a logic value to a constant or assign a list index to an atomic proposition.

<loop statement>: This non-terminal denotes how to write loop statements, which is used to execute certain statements until a condition is met. We use the **turla** reserved word followed by an <atomic proposition> within parenthesis, which is followed by statements within curly braces. After the last curly brace, there should be a semi colon.

<print statement>: This non-terminal denotes how to write print statements, to print values to the console. We use **bastur** reserved word, followed by a proposition to print logic value of resulted by the evaluation of a proposition. Alternatively, we can call a predicate after **bastur** to print the return value of the predicate instantiation.

<scan statement>: This non-terminal denotes how to write scan statements, which are used to read values from the user into the program at runtime. We use the **tarattur** reserved word, followed by a variable to bind the value we have read by the user to the variable.

<predicate instantiation>: This non-terminal specifies what counts as a predicate call within the program.

<proposition>: This non-terminal specifies the **if and only if (<->)** proposition in propositional calculus. It is left recursive, and it has the lowest precedence among the propositions.

<imp_proposition>: This non-terminal specifies the **implies (**->) proposition in propositional calculus. It is left recursive, and has precedence over the **if and only if** proposition.

<or_proposition>: This non-terminal specifies the **or (|)** proposition in propositional calculus. It is left recursive, and has precedence over the **implies** proposition.

<and_proposition>: This non-terminal specifies the **and (&)** proposition in propositional calculus. It is left recursive, and has precedence over the **or** proposition.

<not_proposition>: This non-terminal specifies the **not** (**~**) proposition in propositional calculus. It is left recursive, and has precedence over the **and** proposition.

<placeholder>: This non-terminal is used to avoid ambiguity in the grammar. It also designates the precedence of parenthesis and negation over other logical operators. This is also used as the base case of the <proposition> recursion, leading to an <atomic proposition>.

<var>: This non-terminal specifies what counts as a variable, which is defined by <letters>.

<const>: This non-terminal specifies what counts as a constant, which is defined by <capital letters>.

<letters>: This non-terminal allows left recursive definitions of <letter>.

<capital letters>: This non-terminal allows left recursive definitions of <capital letter>.

<letter>: This non-terminal defines what counts as a letter. Which are all the letters in the Turkish alphabet in lowercase form, plus the underscore character.

<capital letter>: This non-terminal defines what counts as a capital letter, which are all the letters in the English alphabet in upper case form.

<comment>: This non-terminal defines what counts as a comment, which are all ASCII characters except dollar sign (‘$’), the dollar sign is reserved to start and end comments.

<atomic proposition>: This non-terminal denotes what counts as an atomic proposition, which can be variables that change values as runtime, constants that have a logic value that cannot be changed, the logic values themselves or list indexes.

<logic value>: This non-terminal defines what counts as a logic value. We have two logic values, **doğri** and **yanluş**, that correspond to true and false respectively.

<empty>: This non-terminal is used in the BNF description to make an empty space (‘ ’) more clear.

<list initialization>: This non-terminal is used when initializing an array. It uses the **dizü** reserved word followed by an integer(the index) in square brackets.

<list index>: This non-terminal is used when getting an element from an array. It is used by calling the name of the **dizü** variable, followed by an integer(the index) in square brackets.

<digit>: This non-terminal defines what counts as a digit, which are all the numbers from 0-9.

<integer>: This non-terminal defines what counts as an integer, which is all the numbers that are a single digit, or a digit that is continued with integers.

In VoLCAN, we took some of our reserved words form “laz language”, and Turkish. We thought that it would be enjoyable to write for Turkish (or “laz”) users, more than the mainstream syntaxes. Another aim of ours was to help programmers of VoLCAN remember the conventions easier by making the language syntax parodic.We hope to provide an understandable syntax and reserved words to ease the process of learning a language, and to make coding feel more natural.

Our language does not have a lot of reserved words, and the reserved words do what they mean to do (at least for the target user base, i.e. laz people that love logic). The language is not high on orthogonality, and this helps the user achieve the intended use of reserved words, without accidentally writing unwanted statements. So with the combination of low reserved word space and clear objectives assigned to all those reserved words, we argue that VoLCAN has good writability.

In terms of readability, in our language, we wanted to turn some programming conventions into actual requirements. We hope to eliminate the confusion caused by the people that do not follow conventions or good practices by forcing some of them thusly we might argue that our language is reliable. For example: in VoLCAN, constants have to be uppercase letters, but can have an underscore for more readability. This way, whenever a programmer encounters some word with all uppercase letters, he or she will know that it is a constant that cannot be changed. The same goes for variable naming and predicate naming. We hope to increase readability of our language with these additions.

We wanted VoLCAN to stand out from other programming languages yet we also wanted to keep the traditional conventions. We are aware that this is an assignment and we hope to make it fun for everyone to read and work on that's why preferred these set of comical reserved words.

PART B)

The revised lex description file is given below:

```
letters [a-zgüsöç_]+

integer [0-9]+

%{

int lineCounter = 1;

%}

%%

eger return(IF);

degülse return(ELSE);

tanum_basladu return(BEGN);
```

```
tanum_bittu return(END);

turla return(LOOP);

[ \t] ;

, return(COMMA);

\( return(LP);

\) return(RP);

\{ return(LC);

\} return(RC);

\[ return(LSQ);

\] return(RSQ);

\; return(SEMI_COL);

\| return(OR_OP);

& return(AND_OP);

-> return (IMP_OP);

\<-> return(IFF_OP);

~ return(NOT_OP);

tarattur return(SCAN);

bastur return(PRINT);

dondür return(RETURN);

dizü return(ARRAY_INIT);

doğridur|yanluştur  return(LOGIC_VAL);

\= return(ASSIGNMENT_OP);

[A-ZIGÜSÖÇ_]+ return(CONST);

{integer} return(INT);

[A-ZIGÜSÖÇ]{letters} return(PREDICATE_NAME);

{letters} return(VAR);

$.+\$ return(COMMENT);
```

```
\n {lineCounter++;}

. ;

%%

int yywrap() {return 1;}
```

The yacc description file is given below:

```
%token IF ELSE BEGN END LOOP COMMA LP RP LC RC SEMI_COL OR_OP AND_OP
IMP_OP IFF_OP NOT_OP SCAN PRINT COMMENT RETURN LOGIC_VAL
ASSIGNMENT_OP VAR PREDICATE_NAME CONST ARRAY_INIT LSQ RSQ INT
%%
start:          program
program:    BEGN predicates END stmts
            | COMMENT BEGN predicates END stmts
            | stmts
            | /*EMPTY*/
;

predicates: predicate |predicates predicate
;

predicate:  PREDICATE_NAME LP parameter_list RP body
;

body:   LC stmts RETURN proposition SEMI_COL RC
        | LC RETURN proposition SEMI_COL RC
        | LC stmts RETURN predicate_instantiation SEMI_COL RC
;


parameter_list:  /*EMPTY*/
            | parameter
            | parameter COMMA parameter_list
;

parameter:       atomic_proposition
;

stmts:  stmt
        | stmt stmts
        | COMMENT stmts
        | stmt COMMENT
```

```
;

stmt:  matched | unmatched
;

matched:          IF LP atomic_proposition RP LC matched RC ELSE LC matched RC
            | other SEMI_COL
;
other:   assignment_statement
            | loop_statement
            | print_statement
            | scan_statement
            | list_initialization
;

unmatched: IF LP atomic_proposition RP LC stmts SEMI_COL RC
        | IF LP atomic_proposition RP LC matched RC ELSE LC unmatched RC
;

assignment_statement: VAR ASSIGNMENT_OP proposition
        | VAR ASSIGNMENT_OP list_initialization
        | VAR ASSIGNMENT_OP predicate_instantiation
        | array_index ASSIGNMENT_OP atomic_proposition
        | CONST ASSIGNMENT_OP LOGIC_VAL
;

loop_statement: LOOP LP atomic_proposition RP LC stmts RC
;

list_initialization: ARRAY_INIT LSQ INT RSQ
        | LSQ parameter_list RSQ
;



array_index:      VAR LSQ INT RSQ
;

print_statement: PRINT proposition
            | PRINT predicate_instantiation
;

scan_statement: SCAN VAR
;

predicate_instantiation: PREDICATE_NAME LP parameter_list RP
```

```
;


proposition: proposition IFF_OP imp_proposition
        | imp_proposition
;

imp_proposition: imp_proposition IMP_OP or_proposition
        | or_proposition
;
or_proposition: or_proposition OR_OP and_proposition
        | and_proposition
;

and_proposition: and_proposition AND_OP not_proposition
        | not_proposition
;

not_proposition: NOT_OP placeholder
        | placeholder
;

placeholder: LP proposition RP
        | atomic_proposition
;

atomic_proposition: VAR | LOGIC_VAL | CONST | array_index
;


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

int main(){

  yyparse();

  printf("Input program accepted");

  return 0;

}
int yyerror(char *s){ fprintf ( stderr, "%s in line %d\n", s, lineCounter);}
```

PART C)

The example program is given below:

```
$tanumlarimizu ha buraya yazayruz$
tanum_basladu
Mutlu_miyum(mutliluk)
{
      dondür mutliluk | dogridur;
}
Tarlaya_gidecek_miyum(hava_güneslidur, fadime_evde_yokidur){
   gidecegum = hava_güneslidur & fadime_evde_yokidur;

   eger(gidecegum)
   {
      bastur gidecegum;
   }
   degülse
   {
      gidecegum = gidecegum | dogridur;
   }
   dondür gidecegum;
}
tanum_bittu

$Ha burada çalistirilacak kodu yazayruz (main)$
tarattur hava_güneslidur;
tarattur fadime_evde_yokidur;

gidecegum = Tarlaya_gidecek_miyum(hava_güneslidur, fadime_evde_yokidur);

LAZ_DOGDUM_LAZ_KALACAGUM = dogridur;
mutlu_bir_lazum = dogridur;

bastur LAZ_DOGDUM_LAZ_KALACAGUM <-> gidecegum -> mutlu_bir_lazum;

haftanun_günleru = dizü[7];

fadime_benu_oldurecek = dogridur;

turla(mutlu_bir_lazum)
{
   haftanun_günleru[6] = dogridur;
   turla(fadime_benu_oldurecek)
   {
      yarduma_ihtiyacum_var = fadime_benu_oldurecek;
      bastur yarduma_ihtiyacum_var;
      mutlu_bir_lazum = ~mutlu_bir_lazum;
   };
};
```