Bilkent University

CS315

Project 2

Language Name: TD/FD

Section: 03

Group ID: 50

Doruk Çakmakçı [21502293]
Mehmet Oğuz Göçmen [21503128]

Instructor: Ertuğrul Kartal Tabak

Table of Contents

1. Revised and Augmented Language Design
   BNF form of TD/FD
2. LEX
   Lex Configuration Code
3. YACC
   YACC Configuration Code
4. Example Program

Part 1 – Revised and Augmented Language Design

The following sections will describe the revised and the augmented design of a context-free language represented in BNF form, for domain of propositional calculus, which is called TD/FD (the dollar sign acts as statement terminator in TD/FD. TD/FD stands for true$ / false$ due to the fact that every logical expression derives to either one of the two statements above). Explanations will be given for each subsection. Also, the comment syntax described below will be used to describe important nonterminal types individually. TD/FD syntax is similar to Java but it has a different application domain which is propositional calculus rather than general programming. Below, the BNF form of TD/FD is described in accordance with the tokens generated by scanner and also the tokens processed by the parser.

BNF form of TD/FD

1) Program Structure

A TD/FD program consists of groups of statements(or only one statement ending with $) enclosed by a main function (start) which is a special type of function without parameter list and the parser for TD/FD can only parse and comments or appropriate statements (written in accordance to the syntax of TD/FD). Programs are composed of different statement types which are declaration, initialization, expression, if, loop, predicate instantiation, list initialization, output to the user(exhale) and get input from the user(inhale) statements. `program`, `statements`, `inhale`, `exhale` , `listInit`, `listElementAssignment` nonterminals are the most important ones from our perception. `program` and `statements` nonterminals provide a way to combine all of the statements of TD/FD and generate a useful program with a purpose in real-life. `inhale` and `exhale` statements are important due to the fact that user is interacted with the program in scope of these statements. `listInit` and `listElementAssignment` nonterminals provide a data structure to hold the boolean values in chunks which improves the writability of the program.

Since comments are handled in the lexical analysis part(scanner), they are omitted for brevity. The BNF form of program structure is below:

```
<program>: MAIN LBRACE <statements> RBRACE;

<statements>:
               <statement> STMT_TERMINATOR
             | <statement> STMT_TERMINATOR <statements>;


<statement>:
              <declaration>
            | <init>
            | <expression>
            | <ifstmt>
            | <loop>
            | <predicateInstantiation>
            | <listInit>
            | <listElementAssignment>
            | <inhale>
            | <exhale>;
```

Refer below for the unexplained nonterminals.

2) <u>Relevant Keywords</u>

VAR_TYPE → `var` keyword to indicate variable
VAR_ID → consists of a string
CONST_TYPE → `const` keyword to indicate constant
CONST_ID → consist of a string inside '_' characters
CONST_CONTENT → consists of a string inside '`?`' characters
T_VAL → either true or false
CALL_KW → `call` keyword for predicate calling
PRINT_KW → `exhale` keyword for output
SCAN_KW → `inhale` keyword for input
LIST_KW → `list` keyword for input

3) <u>Variables and Constants</u>

TD/FD programmer is able to represent the needed data values in their code using 3 structures: lists, variables and constants. Variables can be declared and initialized separately however, constants must be initialized when they are used for the first time and their truth values can not be altered. Variables must be specified with VAR_TYPE token before their VAR_ID and they can only hold T_VAL. On the other hand constants must be

specified with CONST_TYPE followed by CONST_ID followed by CONST_CONTENT and the assigned T_VAL.

4) Declarations

TD/FD programmer can declare two things; variables and predicates. Constants can not be declared alone without initialising a default value since they hold their value unchanged during the entire program. Variables must be specified with `VAR_TYPE` token before their `VAR_ID` when declaring. `predicate` nonterminal will be given in "Predicate" section.

```
<declaration>:
                VAR_TYPE VAR_ID  //variable declaration
              | <predicate>;  //predicate declaration
```

5) Initializations

TD/FD programmer can initialize three things; constant initialization(`<constInit>`), variable initialization(`<varInit>`) and list initialization(`<listInit>`) .

Constant initialization can occur in 4 different ways. In each initialization the starting tokens are same which are `CONST_TYPE` and `CONST_ID`. These tokens indicate the initialization of a constant. User has the option to define constant as a string by putting the token `CONST_CONTENT` after `CONST_TYPE` and `CONST_ID`. After these tokens with the help of assignment operator, values are assigned to given constant. As indicated 4 different ways include assigning truth value to constant, assigning a compound logical operation(`<expression>`) to constant, assigning a list element(`<listElement>`) to constant and assigning predicate instantiation(`<predicateInstantiation>`) to constant.

Variable initialization is consisted of both first time initializations(`<varFirstInit>`) and assignment(`<varAssign>`).

- In each first time initializations(`<varFirstInit>`) the starting tokens are same which are `VAR_TYPE` and `VAR_ID`. These tokens indicate the initialization of a variable. After these tokens with the help of assignment operator(`ASSIGN_OP`), values are assigned to given variable.

- In each assignment(`<varFirstInit>`) the starting token is same which is `VAR_ID`. This token indicates the assignment to a variable. After this token with the help of assignment operator(`ASSIGN_OP`), values are assigned to given variable.

In both `<varFirstInit>` and `<varAssign>`, 4 different ways include assigning truth value(`T_VAL`) to constant, assigning a compound logical operation(`<expression>`) to constant, assigning a list element(`<listElement>`) to constant and assigning predicate

instantiation(<predicateInstantiation>) to constant. <expression>, <listElement> and predicateInstantiation are explained in later sections.

Crucial thing to remember is in both constant and variable initializing a compound logical statement every logical element should include parenthesis at the beginning and at the end, this is done to remove some conflicts from the language.

List initialization(<listInit>) has 2 different initializations. One is creating an empty list and the other is creating a list with given variables. In both initializations strating tokens are same which are list keyword(LIST_KW) which is list followed by list's id(VAR_ID). After these with the help of assignment operator(ASSIGN_OP) the contents of list is set. This is done either just giving left and right braces(LBRACE and RBRACE) or putting list's elements(<listParameterList>) inside these braces. <listParameterList> is explained in list section.

```
<init>: <constInit> | <varInit> | <listInit>;

<constInit>:
        CONST_TYPE  CONST_ID CONST_CONTENT ASSIGN_OP T_VAL
      | CONST_TYPE  CONST_ID ASSIGN_OP T_VAL
      | CONST_TYPE  CONST_ID CONST_CONTENT ASSIGN_OP <expression>
      | CONST_TYPE CONST_ID ASSIGN_OP <expression>
      | CONST_TYPE  CONST_ID CONST_CONTENT ASSIGN_OP <listElement>
      | CONST_TYPE  CONST_ID  ASSIGN_OP <listElement>
      | CONST_TYPE  CONST_ID CONST_CONTENT ASSIGN_OP
                                          <predicateInstantiation>
      | CONST_TYPE CONST_ID ASSIGN_OP <predicateInstantiation>;



<varInit:   <varFirstInit> | <varAssign>

<varFirstInit>:
              VAR_TYPE VAR_ID ASSIGN_OP T_VAL
           |  VAR_TYPE VAR_ID ASSIGN_OP <expression>
           |  VAR_TYPE VAR_ID ASSIGN_OP <listElement>
           |  VAR_TYPE VAR_ID ASSIGN_OP <predicateInstantiation>

<varAssign>:
             VAR_ID ASSIGN_OP T_VAL
           |  VAR_ID ASSIGN_OP <expression>
           |  VAR_ID ASSIGN_OP <listElement>
           |  VAR_ID ASSIGN_OP <predicateInstantiation>;

<listInit>:
       LIST_KW VAR_ID ASSIGN_OP LBRACE <listParameterList> RBRACE
     | LIST_KW VAR_ID ASSIGN_OP LBRACE  RBRACE;
```

6) Predicates

TD/FD programmer can write predicates and instantiate these predicates . Following nonterminals are explained from simplest nonterminal to complex nonterminal.

Paramater of a predicate(<parameter>) is simply given variable's id(VAR_ID).

Parameter list(<parameterList>) of predicate is either one parameter(parameter) or more than one parameters(parameter COMMA parameterList) separated with comma character(COMMA). To define more than one parameter, right recursion is used.

Predicate parameter list(<predicateParameterList>) is simply parameter list which include parenthesis in the beginning and at the end of parameterList.

Predicate name(VAR_ID) is simply it's id. VAR_ID token is used here since predicate name is also a string.

Predicate prototype(<predicatePrototype>) is the combination of <predicateName> and <predicateParameterList> nonterminals.

Predicate body(<predicateBody>) can have 6 different representations. Every one of them include braces in the beginning and at the end of body. In body there can be either only a return statement which includes return token(RETURN) and return value or statements followed by return statement. Return values can be a truth value(T_VAL), a compound logical operation(<expression>) or a predicate instantiation(<predicateInstantiation>).

Finally predicate(predicate) is consisted of prototype of predicate (<predicatePrototype>) followed by it's body(<predicateBody>).

To instantiate these predicates, user should first use call keyword(CALL_KW) which is call followed by defined predicate's name(<predicateName>) and predicate's parameter list(<predicateParameterList>).

```
<parameter>:  VAR_ID;

<parameterList>:
                    <parameter>
               | <parameter> COMMA <parameterList>;

<predicateParameterList>: LP <parameterList> RP;

<predicateName>: VAR_ID ;

<predicatePrototype>: <predicateName> <predicateParameterList>;
```

```
predicateBody:
    LBRACE RETURN T_VAL STMT_TERMINATOR RBRACE
  | LBRACE RETURN <expression> STMT_TERMINATOR RBRACE
  | LBRACE RETURN <predicateInstantiation> STMT_TERMINATOR RBRACE
  | LBRACE <statements> RETURN T_VAL STMT_TERMINATOR RBRACE
  | LBRACE <statements> RETURN <expression> STMT_TERMINATOR RBRACE
  | LBRACE <statements> RETURN <predicateInstantiation>
                               STMT_TERMINATOR RBRACE;


<predicate>: <predicatePrototype> <predicateBody>;

<predicateInstantiation>:
                      CALL_KW <predicateName>
                <predicateParameterList>;
```

7) Lists

TD/FD programmer can define lists inside this language. Following nonterminals are explained from simplest nonterminal to complex nonterminal. Initialization of lists is given in “Initializations” section.

List’s parameter(`<listparameter>`) can be 3 things; T_VAL, VAR_ID, CONST_ID.

List’s parameter list can either be one parameter(`<listParameter>`) or more than one parameters(`<listParameter> COMMA <listParameterList>`) separated with comma character(`COMMA`). To define more than one parameter, right recursion is used.

To access an element of list `<listElement>` nonterminal is used. List’s id(`VAR_ID`) followed by left and right square brackets(`LSQUARE RSQUARE`) which inside holds a index gives the wanted element from list.

To put an element inside a list `<listElementAssignment>` nonterminal is given. User should simply write in the form `<listElement>` followed by assignment operator(`ASSIGN_OP`) to a `T_VAL`.

```
<listParameter>:     T_VAL
                   | VAR_ID
                   | CONST_ID;

<listParameterList>:
                        <listParameter>
                     |  <listParameter> COMMA <listParameterList>;

<listElement>: VAR_ID LSQUARE NUMERIC RSQUARE;

<listElementAssignment>: <listElement> ASSIGN_OP T_VAL;
```

8) Expressions

TD/FD programmer can write compund logical operations(<expression>).

While evaluating the compound logical operations, operator precedence and associavity of operators. From lowest to higher precedence operators are; if and only if(IFF), implies(IMPLIES), or(OR), xor(XOR), and(AND), not(NOT), parenthesis(LP RP). These operators have different associtivities. IMPLIES operator has right associativity, NOT operator is unary and others have left associativity. According to these precedence levels and associativity types, following BNF is organised.

```
<expression>: <iffExpression>;

<iffExpression> : <iffExpression> IFF <impliesExpression>
                | <impliesExpression>;

<impliesExpression>: <orExpression> IMPLIES <impliesExpression>
                   | <orExpression;

<orExpression>: <orExpression> OR <xorExpression>
              | <xorExpression>;

<xorExpression>: <xorExpression> XOR <andExpression>
               | <andExpression>;

<andExpression> : <andExpression> AND <notExpression>
                | <notExpression>;

<notExpression> : NOT <paranthesisExpression>
                | <paranthesisExpression>;

<paranthesisExpression>: LP <iffExpression> RP
                       | LP <id> RP
                       | LP <predicateInstantiation> RP
                       | LP T_VAL RP;

<id>: VAR_ID | CONST_ID;
```

9) Decision(Selection) Structures

TD/FD programmer can use decision(selection) structures. There are 2 decision structures in TD/FD; if(<noElse>) and if/else(<yesElse>).

The structure of if/else statements are similar to Java. User should write if(IF) keyword followed by parenthesis. Inside parenthesis, there should be a condition(<condition>) to decide for entering inside if body. Otherwise, if there is an else(ELSE) keyword followed by its body, it should enter inside that body.

```
<ifstmt>: <noElse>
        | <yesElse>;

<noElse>: IF LP condition RP <noElse> ELSE <noElse>
      | LBRACE <statements> RBRACE ;

<yesElse>: IF LP <condition> RP LBRACE <statements> RBRACE
       | IF LP <condition> RP <noElse> ELSE <yesElse>;
```

10) Loop Structure

TD/FD programmer can also use loop structures. There are 2 loop structures in TD/FD; `for` and `while`.

To write a `for` loop, coder should first write `for(FOR)` keyword. After that inside paranthesis one should give a variable initialization(`<varInit>`)followed with a condition(`<condition>`) to check every time execution came to end of `for` loop. After these, inside braces user can write any number of statements(`statements`). `varInit` and `<condition>` has to be followed by a statement terminator(`STMT_TERMINATOR`). Structure is similar to those for loops in Java.

To write a `while` loop, coder should first write `while(WHILE)` keyword. After that inside parenthesis one should give a condition(`<condition>`) to check every time execution came to end of while loop. After these, inside braces user can write any number of statements(`<statements>`). Structure is similar to those while loops in Java.

```
<loop>: <for>
      | <while>;

<for>:
   FOR LP <varInit> STMT_TERMINATOR <condition> STMT_TERMINATOR RP
     LBRACE <statements> RBRACE;

<while>: WHILE LP <condition> RP LBRACE <statements> RBRACE;
```

11) Conditions

TD/FD programmer can define conditions for equality check.

Conditions(`<condition>`) are consisted of a comparison between variables and constants. In BNF expression is used so compound logical operations(`<expression>`) can also be compared as well as a variable or a constant. This is enabled by `<id>` in `<expression>`.

```
<condition>: VAR_ID EQUAL_OP <expression>
           | CONST_ID EQUAL_OP <expression>;
```

12) I/O Statements

TD/FD programmer can also use I/O statements. Languages has 2 I/O choices for now. One of them enables user to give input to program and the other one enables user to give output from program to possible users.

Statements that start with `inhale(SCAN_KW)` keyword enables user to give input. There are 4 ways to get inputs. All of them starts with `inhale(SCAN_KW)` keyword followed by parenthesis. Inside parenthesis coder should specify if input will be a constant or not. Coder also can specify this constant with it's content. Other than that user can enable to `inhale` a list element or a variable. Structure of `inhale` is given below.

Statements that start with `exhale(PRINT_KW)` keyword enables user to retrieve output. There are 2 ways to get outputs. All of them starts with `exhale(PRINT_KW)` keyword followed by parenthesis. Inside parenthesis coder should give an `expression` or a string followed by newline character(`STRINGNL`) to command external user. Structure of `exhale` is given below.

```
inhale:
       SCAN_KW LP VAR_ID COMMA T_VAL RP
      |SCAN_KW LP CONST_TYPE COMMA CONST_ID COMMA T_VAL RP
      |SCAN_KW LP CONST_TYPE COMMA CONST_ID COMMA CONST_CONTENT
                                                COMMA  T_VAL RP
      |SCAN_KW LP <listElement> COMMA T_VAL RP;

<exhale>:
       PRINT_KW LP <expression> RP
    | PRINT_KW LP STRINGNL RP;
```

Part 2: Lex

Lex Configuration Code

```
%{
      #include <stdio.h>
      #include "y.tab.h"
    void yyerror(char *);
%}
CALL_KW               call
T_VAL            (false|true)
LETTER           [A-Za-z]
DIGIT                 [0-9]
NUMERIC               [0-9]+
ALPHANUMERIC          ({LETTER}|{DIGIT})
stringTerminator      \*
NEWLINE               \n
STMT_TERMINATOR       \$
NOT                   \!
AND                   &&
XOR                   \^
OR                    \|\|
STRING                {LETTER}[{LETTER}{DIGIT} ]*
STRINGNL              {LETTER}[{LETTER}{DIGIT} ]*{NEWLINE}
COMMA                 \,
DOUBLEQUOTE           \"
CONST_CONTENT         \?([A-Za-z ]*)\?
HASHTAG               \#
SLASH                 \/
COMMENT               {HASHTAG}([^\n]*)(\n)
PRINT_KW              exhale
SCAN_KW               inhale
LIST_KW               list
CONST_TYPE            const
VAR_TYPE              var
RETURN                return
IF                    if
ELSE                  else
FOR                   for
WHILE                 while
MAIN                  start
%option yylineno
%%
{COMMA}               return COMMA;
{CALL_KW}        return CALL_KW;
{PRINT_KW}       return PRINT_KW;
{SCAN_KW}        return SCAN_KW;
{LIST_KW}        return LIST_KW;
```

```
{NOT}            return NOT;
{AND}            return AND;
{XOR}            return XOR;
{OR}             return OR;
{CONST_TYPE}     return CONST_TYPE ;
{VAR_TYPE}       return VAR_TYPE;
{RETURN}         return RETURN;
{IF}             return IF;
{ELSE}           return ELSE;
{FOR}            return FOR;
{WHILE}          return WHILE;
{MAIN}           return MAIN;
{T_VAL}          return T_VAL;
\=\=             return EQUAL_OP;
\=               return ASSIGN_OP;
\(               return LP;
\)               return RP;
\{               return LBRACE;
\}               return RBRACE;
\[               return LSQUARE;
\]               return RSQUARE;
\<\=\>           return IFF;
\=\>             return IMPLIES;
{STMT_TERMINATOR}                return STMT_TERMINATOR;
{NUMERIC}                        return NUMERIC;
\_{LETTER}{ALPHANUMERIC}*\_      return CONST_ID;
{LETTER}{ALPHANUMERIC}*          return VAR_ID;
{CONST_CONTENT}                  return CONST_CONTENT;
{STRING}         return STRING;
{STRINGNL}       return STRINGNL;
{COMMENT}  ; // skip comments
[ \t]*     ;
\n {}
%%
int yywrap(void)
{
  return 1;
}
```

Part 3: Yacc

Yacc Configuration Code

```
%{
      //#define YYDEBUG 1
      #include "stdio.h"
    void yyerror(char *);
    extern int yylineno;
    #include "y.tab.h"
      int yylex(void);
%}

// can be deleted if error exists ^^^

// Declare the tokens
%token MAIN
%token CALL_KW PRINT_KW SCAN_KW LIST_KW//keyword related
%token NOT AND XOR OR IFF IMPLIES //logical connectives
%token CONST_TYPE VAR_TYPE CONST_ID VAR_ID CONST_CONTENT
//constant-variable
%token RETURN //return
%token IF ELSE //condition staments
%token FOR WHILE //loop statements
%token T_VAL //boolean
%token ASSIGN_OP EQUAL_OP //assignment and equality
%token LP RP LBRACE RBRACE LSQUARE RSQUARE STMT_TERMINATOR
COMMA NUMERIC//structure related
%token STRINGNL STRING NEWLINE

%start program


%%

//-------------------------------------------------program
structure-------------------------------------------------
program: MAIN LBRACE statements RBRACE;
statements:
                statement STMT_TERMINATOR
             | statement STMT_TERMINATOR statements;
statement:
              declaration
            | init
            | expression
            | ifstmt
            | loop
            | predicateInstantiation
```

```
            | listElementAssignment
            | inhale
            | exhale; // TO-DO

//-------------------------------------------------list
initialization-----------------------------------------------

listInit:
             LIST_KW VAR_ID ASSIGN_OP LBRACE listParameterList
RBRACE
           | LIST_KW VAR_ID ASSIGN_OP LBRACE  RBRACE;

listParameter:
                   T_VAL
                 | VAR_ID
                 | CONST_ID;

listParameterList:
                        listParameter
                    |  listParameter COMMA listParameterList;

listElement: VAR_ID LSQUARE NUMERIC RSQUARE;

listElementAssignment: listElement ASSIGN_OP T_VAL;


//-------------------------------------------------
declaration-------------------------------------------------------
--

//predicate declaration

declaration:
                VAR_TYPE VAR_ID  //variable declaration
             | predicate;  //predicate declaration

parameter:  VAR_ID;

parameterList:
                  parameter
              | parameter COMMA parameterList;

predicateParameterList: LP parameterList RP;

predicateName: VAR_ID ;

predicatePrototype: predicateName predicateParameterList;
```

```
predicateBody:
                    LBRACE RETURN T_VAL STMT_TERMINATOR RBRACE
                | LBRACE RETURN expression STMT_TERMINATOR
RBRACE
                  | LBRACE RETURN predicateInstantiation
STMT_TERMINATOR RBRACE
                  | LBRACE statements RETURN T_VAL
STMT_TERMINATOR RBRACE
                  | LBRACE statements RETURN expression
STMT_TERMINATOR RBRACE
                  | LBRACE statements RETURN
predicateInstantiation STMT_TERMINATOR RBRACE;


predicate: predicatePrototype predicateBody;


//-----------------------------------------------
initialization-----------------------------------------------
--

init: constInit | varInit | listInit;

constInit:
              CONST_TYPE  CONST_ID CONST_CONTENT ASSIGN_OP
T_VAL
           |  CONST_TYPE  CONST_ID ASSIGN_OP T_VAL
           |  CONST_TYPE  CONST_ID CONST_CONTENT ASSIGN_OP
expression
           |  CONST_TYPE CONST_ID ASSIGN_OP expression
           |  CONST_TYPE  CONST_ID CONST_CONTENT ASSIGN_OP
listElement
           |  CONST_TYPE  CONST_ID  ASSIGN_OP listElement
           |  CONST_TYPE  CONST_ID CONST_CONTENT ASSIGN_OP
predicateInstantiation
           |  CONST_TYPE CONST_ID ASSIGN_OP
predicateInstantiation;

varInit:   varFirstInit | varAssign

varFirstInit:
              VAR_TYPE VAR_ID ASSIGN_OP T_VAL
           |  VAR_TYPE VAR_ID ASSIGN_OP expression
           |  VAR_TYPE VAR_ID ASSIGN_OP listElement
           |  VAR_TYPE VAR_ID ASSIGN_OP predicateInstantiation

varAssign:
                   VAR_ID ASSIGN_OP T_VAL
```

```
          |  VAR_ID ASSIGN_OP expression
          |  VAR_ID ASSIGN_OP listElement
          |  VAR_ID ASSIGN_OP predicateInstantiation;


//-------------------------------------------------expression
structure-----------------------------------------

//operator precedence is included

expression: iffExpression;

iffExpression : iffExpression IFF impliesExpression
                  | impliesExpression;

impliesExpression: orExpression IMPLIES impliesExpression
                  | orExpression;

orExpression: orExpression OR xorExpression
                  | xorExpression;

xorExpression: xorExpression XOR andExpression
                  | andExpression;

andExpression : andExpression AND notExpression
                  | notExpression;

notExpression : NOT paranthesisExpression
                  | paranthesisExpression;

paranthesisExpression: LP iffExpression RP
                           | LP id RP
                           | LP predicateInstantiation RP
                           | LP T_VAL RP;

id: VAR_ID
  | CONST_ID;

//-------------------------------------------------
decision(selection)-----------------------------------------

ifstmt: noElse
       | yesElse;

noElse: IF LP condition RP noElse ELSE noElse
      | LBRACE statements RBRACE ;
```

```
yesElse: IF LP condition RP LBRACE statements RBRACE
       | IF LP condition RP noElse ELSE yesElse;

//-------------------------------------------------condition-
-------------------------------------------------

condition: VAR_ID EQUAL_OP expression
          | CONST_ID EQUAL_OP expression;

//-------------------------------------------------loop
structure (for and while)---------------------------------

loop: for
     | while;

for: FOR LP varInit STMT_TERMINATOR condition STMT_TERMINATOR
RP LBRACE statements RBRACE;

while: WHILE LP condition RP LBRACE statements RBRACE;

//-------------------------------------------------predicate
instantiation----------------------------------------

predicateInstantiation: CALL_KW predicateName
predicateParameterList;

//-------------------------------------------------I/O-------
-------------------------------------------------

inhale:
       SCAN_KW LP VAR_ID COMMA T_VAL RP
      |SCAN_KW LP CONST_TYPE COMMA CONST_ID COMMA T_VAL RP
      |SCAN_KW LP CONST_TYPE COMMA CONST_ID COMMA CONST_CONTENT
COMMA  T_VAL RP
      |SCAN_KW LP listElement COMMA T_VAL RP;


exhale:
       PRINT_KW LP expression RP
    |PRINT_KW LP STRINGNL RP;
%%

// report errors
void yyerror(char *s)
{
  fprintf(stderr, "syntax error at line: %d %s\n", yylineno,
s);
}
```

```
int main(void){
      //#if YYDEBUG
      //    yydebug = 1;
      //#endif
      yyparse();
      if(yynerrs < 1) printf("there are no syntax errors!!\n");
}
```

Part 4: Example Program

Example Program

```
# all programs written TD/FD are parsed starting from the
start function (like a predicate with special instantiation
and
# definition). Every operation that can be done with TD/FD
MUST be done in the scope of start. Comments can be written
before the
# start tag and inside but NOT after the end of the start tag.
start { #start of the program
           # variable declaration is shown below
           var oguz1 = true$
           # const declaration is shown below
           const _doruk1_ ?first constant? = false$

           # variable assignment is shown below
           oguz1 = false$

           var oguz2 = true$
           var oguz3 = (_doruk1_) || (_doruk2_)$

           # logical compound is shown below
           var oguz4 = ((((_doruk1_) && (oguz1)) ^ (_doruk2_))
<=> (((oguz2) || (_doruk1_)) => (oguz1)))$

           # if statement without else which includes inhale
statement in it's body.
           if(oguz1 == (_doruk1_)) {
                          inhale (val1, true)$
           }$

           # if statement with else which includes inhale
statement in it's body.
           # else includes exhale statement in it's body
           if(oguz1 == (oguz2) ) {
                          inhale ( val2, false)$
           }
           else{
                     exhale((oguz2))$
           }$

           #predicate declarations with different parameter
count and different body structures
           predicate1( input1, input2, input3, input4){
                return true$
           }$
```

```
            predicate2( input1, input2, input3){
                  return (oguz1)<=>(doruk1)$
            }$
            predicate3( input1, input2){
                  return call predicate1(i1,i2,i3,i4)$
            }$
            predicate4( input1){
                  insidePredicate1 = !(doruk1)$
                  return call predicate1(i1,i2,i3,i4)$
            }$

            # for loop structure which includes initialization
of predicate1 which is declared above
            for(oguz1 = true$ oguz1 == (_doruk3_)$){
                        call predicate1(i1,i2,i3,i4)$
            }$

            # list initialization
            list l1 = {b, c, true, false}$

            # while loop structure which includes list element
assignment statement in it's body.
            while(whileVar1 == (true)){
                        l1[1] = true$
            }$

            # end of the program

}
```