COMPILERS

DEPARTMENT OF INFORMATION AND COMPUTER ENGINEERING

PART B

# BUILDING A PARSER WITH THE BISON GENERATOR

**TEAM / WORK DETAILS**

**TEAM NUMBER:** 2
**STUDENT DETAILS 1:** Athanasiou Vasileios Evangelos (UNIWA-19390005)
**STUDENT DETAILS 2:** Theocharis Georgios (UNIWA-19390283)
**STUDENT DETAILS 3:** Tatsis Pantelis (UNIWA-20390226)
**STUDENT DETAILS 4 :** Iliou Ioannis (UNIWA-19390066)
**STUDENT DETAILS 5:** Dominaris Vasileios (UNIWA-21390055)

**WORKING DEPARTMENT:** B1 Wednesday 12:00-14:00
**LAB INSTRUCTOR:** Iordanakis Michael

# COMPILERS

## CONTENTS

***Introduction*** ........ ***3***

Reference ........ 3

Team organization ........ 3

Reference content ........ 3

***Analysis of responsibilities/roles of all team members*** ........ ***4***

Athanasiou Vasileios Evangelos (UNIWA-19390005) ........ 4

Theocharis Georgios (UNIWA-19390283) ........ 4

Tatsis Pantelis (UNIWA-20390226) ........ 5

Iliou Ioannis (UNIWA-19390066) ........ 5

Dominaris Vasileios (UNIWA-21390055) ........ 6

***Documentation of grammar and design*** ........ ***7***

Grammar ........ 7

Planning ........ 8

***Documentation of FLEX and BISON code*** ........ ***9***

Verbal Analyzer ( FLEX ) ........ 9

Statements Area ........ 9

Definitions Area ........ 10

Region Rules ........ 10

User Code ........ 12

Editorial Analyst ( BISON ) ........ 13

Statements Area ........ 13

Definition of Tokens ........ 14

Priority of Performers ........ 15

Start program ........ 15

Grammar Rules ........ 15

User Code ........ 22

***Control cases and commentary*** ........ ***23***

2.2 Declarations of Variables ........ 23

Input File ( input . txt ) ........ 23

File Output (output.txt) ........ 24

2. 3 Tables ........ 26

Input File ( input . txt ) ........ 26

Output File ( output . txt ) ........ 26
2.4 Built-in simple functions ........ 29
Input File ( input . txt ) ........ 29
File Output (output.txt) ........ 30
2.5 Declaration of User Functions ........ 36
Input File ( input . txt ) ........ 36
File Output (output.txt) ........ 36
2. 6 Statements of Simple Expressions ........ 39
Input File ( input . txt ) ........ 39
File Output (output.txt) ........ 41
2. 7 Compound Statements ........ 56
Input File ( input . txt ) ........ 56
File Output (output.txt) ........ 57
2.8 Syntax Warning Errors ( Warnings ) ........ 63
Input File ( input . txt ) ........ 63
File Output (output.txt) ........ 65
2.9 Syntax Errors ........ 73
Input File ( input . txt ) ........ 73
File Output (output.txt) ........ 75
2.10 Final ........ 85
Input File ( input . txt ) ........ 85
Output File ( output . txt ) ........ 86
Analysis Statistics ........ 96
Output File ( output . txt ) ........ 96
***Explicit code execution reference ........ 98***

# Introduction

## Reference

Part "B Creating a Parser with the BISON generator" is the final stage of building the Uni - C language compiler . The report focuses on grammar documentation for verbal, syntactic and semantic analysis, as well as planning. The parser built with the FLEX generator in part “A3 Completing FLEX Sample Code ” is linked to the parser built with the BISON generator . The codes given as inputs to the generators are presented sufficiently commented and documented. Also, exhaustive tests with annotated results are presented to prove the reliability of the compiler. Finally, the explicit reference to the execution of the code is presented.

## Team organization

Through the climate of cooperation, all team members contributed to the completion of the BISON code , so that its operation could be understood. The tasks for writing the documentation document were divided as in part A into sub-groups, tasks such as code commenting, input test options, but also extensive commenting of the results. The work of writing the BISON code and adapting the FLEX code to work with BISON was done by all team members.

## Reference content

The presentation focuses on the functionality of the parser with the BISON generator , which recognizes Uni - C language expressions . The reference funds are as follows:

1. Documentation of grammar and design
2. FLEX and BISON code
3. Control cases and commentary

In addition, for each task are listed:

1. Which sub-group implemented it?
2. Annotating the control cases
3. Feedback on how the simulation works, any difficulties encountered and workarounds
4. Reporting any deficiencies and code execution correctness

## Analysis of responsibilities/roles of all team members

### Athanasiou Vasileios Evangelos (UNIWA-19390005)

In the role of general coordinator, the student participated in the tasks:

1. **FLEX code**
    - Adaptation to work with BISON code
    - Handling bad strings
2. **Code BISON**
    - Writing syntax rules
    - Handling incorrect expressions
    - Managing syntax warnings
    - Selection of test runs
3. **Documentation document**
    - Import
    - Documentation of grammar and design
    - Explicit code execution reference
    - In the implementation of the final version of the document

In implementing the final versions of FLEX and BISON

### Theocharis Georgios (UNIWA-19390283)

In the role of developer, the student participated in the tasks:

1. **FLEX code**
    - BISON code
    - Handling bad strings
2. **Code BISON**
    - Writing syntax rules
    - Handling incorrect expressions
    - Managing syntax warnings
    - Selection of test runs

3. **Documentation document**

   - FLEX documentation
   - 2.5 Declaration of User Functions
   - 2.6 Declarations of simple expressions

## Tatsis Pantelis (UNIWA-20390226)

In the role of tester , the student participated in the tasks:

1. **FLEX code**

   - BISON code
   - Handling bad strings

2. **Code BISON**

   - Writing syntax rules
   - Handling incorrect expressions
   - Managing syntax warnings
   - Selection of test runs

3. **Documentation document**

   - 2.3 Tables
   - 2.4 Built-in simple functions
   - 2.10 Final

## Iliou Ioannis (UNIWA-19390066)

In the role of commentator, the student participated in the tasks:

1. **FLEX code**

   - BISON code
   - Handling bad strings

2. **Code BISON**

   - Writing syntax rules
   - Handling incorrect expressions
   - Managing syntax warnings

- Selection of test runs

3. **Documentation document**

- 2.2 Declarations of Variables
- 2.8 Syntax Warning Errors
- 2.9 Syntax Errors

## Dominaris Vasileios (UNIWA-21390055)

In the role of analyst, the student participated in the tasks:

1. **FLEX code**

- BISON code
- Handling bad strings

2. **Code BISON**

- Writing syntax rules
- Handling incorrect expressions
- Managing syntax warnings
- Selection of test runs

3. **Documentation document**

- BISON documentation
- 2.7 Composite Statements
- Annotating Analysis Statistics

# COMPILERS

## Documentation of grammar and design

**In red letters, the grammar principles specified by Uni - C and also the design principles specified by the construction of a Uni - C language compiler , where the team failed to implement, are captured** .

## Grammar

Documenting the grammar for a programming language such as Uni - C , which is a subset of the C language , involves analyzing the elements of the language at the lexical, syntactic, and semantic levels. Here is a detailed description of each level:

1. **Verbal Analysis**

   Lexical analysis involves identifying Uni - C language units such as:

   - **Identifiers :** Used to name variables, functions, etc. (eg var, foo) **.**
   - **Keywords :** These are reserved words that have a special meaning in the Uni-C language **and** for this reason it is forbidden to use them as the name of a user variable or as the name of one's own function (e.g. if , else , while )
   - **Literal literals ( Strings ):** These are simple strings that are enclosed in double apostrophes and include any character (e.g. " Hello ", " World ").
   - **Integers :** These are fixed integer numerical values that are used in a variable to perform operations (numerical, relational, logical) or to assign to a variable (e.g. 1, 10, 50 **)**
   - **Floating point numbers ( Floats ):** These are constant numerical values separated by a decimal point or expressed in power form with the scientific form of the power exponent " E or e " (eg 3.14, 4 e 1, 3.5 e -1)
   - **Operators :** Used to implement numerical, relational and logical operations (e.g. >, +, && **)**
   - **Delimiters :** Used to separate two or more Uni - **C** commands (eg ; )
   - **Special characters ( Special ):** These are special characters used to write some expressions such as the declaration of arrays where the special character [ and ] is used.

     Verbal analysis also recognizes other verbal structures such as comments and white _ space characters, however, it does not return them as verbal units and simply ignores them.

     Finally, parsing also handles the situation where a mismatched input string is recognized with the aforementioned verbal rules. This string is recognized as TOKEN _ ERROR and it is at the discretion of the parser how to handle this string.

2. **Syntax analysis**

   The editorial analysis concerns the structure of the programs and the rules governing the Uni - C 's editorial such as:

   - **Logical Lines:** These are expressions that are completed with the semi - colon separator (?).

- **Natural Lines:** This is a sequence of characters terminated by the newline character (\ n )
- **Concatenation of Natural Lines:** This is the concatenation of 2 natural lines with the backslash character (\) at the end of the first line.
- **Blank lines:** This is a line that does not contain any word units recognized by the word analysis.
- **Variable declarations:** This is a command in which the user declares a variable using a keyword that captures the data type of the variable (eg float , int , double , short , long )
- **Arrays:** This is a data structure that stores variables, constant numeric values and strings.
- **Built-in simple functions:** These are built-in functions defined in Uni - C that have a special function with a special syntax (eg scan , len , cmp , print )
- **Declaration of user functions:** These are functions that the user can define with their own parameters, their own name and their own code as long as they start with the keyword func .
- **Simple Expression Statements:** These are numeric expressions, assignment of values to variables, comparisons and concatenation of arrays.
- **Compound Statements:** These are control statements with if and looping statements with while and for

3. **Semantic analysis**

Semantic analysis deals with the meaning and interpretation of programs in Uni - C based on semantic rules such as

- **Type Checking:** Confirming that operations are performed between compatible data types (eg integer to integer)
- **Variable Scope:** Defines the visibility and lifetime of variables
- **Function Scope:** Defines the visibility of a function and how many parameters and what data type parameters it can accept
- **Array Elements:** Specifies the array elements to be of the same data type

## Planning

Uni - C compiler involves several stages and subsystems that work together to convert source code into an executable or intermediate code. In detail, the design includes the following stages:

1. **Building the Verbal Parser with the FLEX generator**

   The parser breaks the code into tokens .

2. **Building the Editorial Parser with the BISON generator**

   The parser checks the structure of the code according to the syntax rules.

3. **Symbol Table Construction**

The symbol table contains the names chosen by the programmer for the various entities that the compiler is to process and information about them.

4. **Construction of an Abstract Syntax Tree**

   The abstract syntax tree is constructed by connecting the terminal or non-terminal symbols resulting from a production with the non-terminal symbol from which they are derived

5. **Production of Intermediate Code**

   Intermediate code is a form of code between the source code and machine language level and is used to allow room for optimization.

## *Documentation of* FLEX and BISON code

### Verbal Analyzer ( FLEX )

The following code implements a Flex scanner to recognize various tokens (keywords, operators, identifiers, etc.) from an input file and report possible errors.

#### Statements Area

```
/* Reading is limited to a single file and terminates at the first EOF */
% option noyywrap
% x error

C code to define required headers files and variables.
   Anything between %{ and %} is automatically transferred to the C file which
   will create the Flex . */

%{

# include < stdio . h >
# include < string.h >
# include < stdlib.h >


/* Header file containing list of all tokens */
# include " simple-bison-code.tab.h "

/* Set current line counter */
extern int line ;
extern int flags ;
extern int correct_words ;
extern int lex_warnings ;

void yyerror ( const char * msg );
void prn ( char * token );

%}
```

It includes the necessary header files for I/O and string manipulation functions.

It declares external variables and functions that are used in the rest of the program, as well as in the SA program.

## Definitions Area

```
/* Names and corresponding definitions (in regular expression form).
   After that , the names ( left ) can be used instead of ,
   therefore particularly long and difficult to understand , regular expressions */

IDENTIFIER [a-zA-Z_][a-zA-Z0-9_]{0,31}
STRING \"([^"\\]*(\\[\\n"][^"\\]*)*)\"
INTEGER ([1-9][0-9]*|0[x|X][0-9A-F]+|0[0-7]+|0)
FLOAT (?:[1-9][0-9]*|0)(?:\.(?:[1-9][0-9]*|0*[1-9]+))?( ?:[eE](?:-?[1-9][0-9]*|0))?
COMMENT (\/\/.*|\/\*[^*]*\*+([^/*][^*]*\*+)*\/)
DELIMITER \;
WHITESPACE [ \t]+
NEWLINE \n
TOKEN_ERROR [^ \t\r\n?]


%%
```

In the definitions area, regular expressions are defined for various lexical units such as identifiers, strings, integers, floating-point numbers, comments, separators, spaces, and newline.

## Region Rules

```
break          { correct_words ++; prn ( " KEYWORD " ); return SBREAK ; }
case           { correct_words ++; prn ( " KEYWORD " ); return CASE ; }
"constant"          { correct_words ++; prn ( " KEYWORD " ); return SCONST ; }
continue       { correct_words ++; prn ( " KEYWORD " ); return SCONTINUE ; }
"do"             { correct_words ++; prn ( " KEYWORD " ); return SDO ; }
double         { correct_words ++; prn ( " KEYWORD " ); return SDOUBLE ; }
"else"           { correct_words ++; prn ( " KEYWORD " ); return SELSE ; }
float          { correct_words ++; prn ( " KEYWORD " ); return SFLOAT ; }
"for"            { correct_words ++; prn ( " KEYWORD " ); return SFOR ; }
if             { correct_words ++; prn ( " KEYWORD " ); return SIF ; }
"int"            { correct_words ++; prn ( " KEYWORD " ); return SINT ; }
long           { correct_words ++; prn ( " KEYWORD " ); return SLONG ; }
```

```
"return"        { correct_words ++; prn ( " KEYWORD " ); return SRETURN ; }
"sizeof"        { correct_words ++; prn ( " KEYWORD " ); return SSIZEOF ; }
"structure"        { correct_words ++; prn ( " KEYWORD " ); return SSTRUCT ; }
switch        { correct_words ++; prn ( " KEYWORD " ); return SSWITCH ; }
"void"          { correct_words ++; prn ( " KEYWORD " ); return SVOID ; }
while         { correct_words ++; prn ( " KEYWORD " ); return SWHILE ; }
"func"          { correct_words ++; prn ( " KEYWORD " ); return SFUNC ; }
short         { correct_words ++; prn ( " KEYWORD " ); return SHORT ; }
"+"             { prn ( " OPERATOR " ); return PLUS ; }
"*="            { prn ( " OPERATOR " ); return MULEQ ; }
"--"            { prn ( " OPERATOR " ); return PMINEQ ; }
"-"             { prn ( " OPERATOR " ); return MINUS ; }
"/="            { prn ( " OPERATOR " ); return DIVEQ ; }
"<"             { prn ( " OPERATOR " ); return LT ; }
"*"             { prn ( " OPERATOR " ); return MUL ; }
"!"             { prn ( " OPERATOR " ); return NOT ; }
">"             { prn ( " OPERATOR " ); return GT ; }
"/"             { prn ( " OPERATOR " ); return DIV ; }
"&&"            { prn ( " OPERATOR " ); return AND ; }
"<="            { prn ( " OPERATOR " ); return LEQ ; }
"%"             { prn ( " OPERATOR " ); return MOD ; }
"||"            { prn ( " OPERATOR " ); return OR ; }
">="            { prn ( " OPERATOR " ); return GREQ ; }
"="             { prn ( " OPERATOR " ); return ASSIGN ; }
"=="            { prn ( " OPERATOR " ); return EQUAL ; }
"&"             { prn ( " OPERATOR " ); return ADDR ; }
"+="            { prn ( " OPERATOR " ); return PLUSEQ ; }
"!="            { prn ( " OPERATOR " ); return NOTEQ ; }
"-="            { prn ( " OPERATOR " ); return MINEQ ; }
"++"            { prn ( " OPERATOR " ); return PPLUSEQ ; }
"("             { prn ( " SPECIAL " ); return OPENPAR ; }
")"             { prn ( " SPECIAL " ); return CLOSEPAR ; }
"["             { prn ( " SPECIAL " ); return OPENSQBRA ; }
"]"             { prn ( " SPECIAL " ); return CLOSESQBRA ; }
"{"             { prn ( " SPECIAL " ); return OPENCURBRA ; }
"}"             { prn ( " SPECIAL " ); return CLOSECURRENT ; }
","             { prn ( " SPECIAL " ); return COMMA ; }
" \\ "            { prn ( " SPECIAL " ); return BACKSLASH ; }
scan          { correct_words ++; prn ( " FUNCTION " ); return SSCAN ; }
"len"           { correct_words ++; prn ( " FUNCTION " ); return SLEN ; }
print         { correct_words ++; prn ( " FUNCTION " ); return SPRINT ; }
"cmp"           { correct_words ++; prn ( " FUNCTION " ); return SCMP ; }
{DELIMITER}     { prn ( " DELIMITER " ); return DELIMITER ; }
{IDENTIFIER}    { correct_words ++; prn ( " IDENTIFIER " ); return IDENTIFIER ; }
{STRING}        { correct_words ++; prn ( " STRING " ); return STRING ; }
{INTEGER}       { prn ( " INTEGER " ); return INTEGER ; }
{FLOAT}         { prn ( " FLOAT " ); return FLOAT ; }
{COMMENT}       { ECHO ; }
```

```
{WHITESPACE}     { }
{NEWLINE}        { line ++; return NEWLINE ; }
<<EOF>>          { return EOF; }


{TOKEN_ERROR}            { lex_warnings ++; yyerror ( " Token error " ); BEGIN (
error ); return TOKEN_ERROR ; }
<error> {WHITESPACE} +     { BEGIN ( 0 ); fprintf ( yyout , " \t %d character(s)
ignored so far \n " , lex_warnings ); }
<error>.                 { lex_warnings ++; }
<error> \n                { line ++; BEGIN ( 0 ); fprintf ( yyout , " \t\t %d
character(s) ignored so far \n " , lex_warnings ); return NEWLINE ;}

%%
```

The rules area lists the rules that input strings can be mapped to. Each rule consists of the regular expression defined in the definitions area and action-commands in C .

One action performed on some error handling rules is to update the counter variables **correct _ words** for correct words and **lex _ warnings** for incorrect words. Also, another important action that is performed in the rules of lexical units is the **return command** which returns the token to SA and is the main communication code between LA and SA.

Handles errors with TOKEN _ ERROR and transitions to error state when an unrecognized character is encountered.

## User Code

```
/* The yyerror function is used to report errors. It is specifically called
   by yyparse when there is a typo. In the following case the
   function actually prints an error message to the screen. */
void yyerror ( const char * msg )
{
   fprintf ( yyout , " !! %s !! -> at Line= %d \n " , msg , line );
   return ;
}

void prn ( char * token )
{
   fprintf ( yyout , " \t [FLEX] Line= %d , token= %s , value= \" %s \"\n " , line ,
token , yytext );
   return ;
}
```

Here the user code is absent as SA is the one that calls LA to recognize strings. There are two string manipulation functions:

**void yyerror ( const char * msg ):** Prints an error message with the current line.

**void prn ( char * token ):** Prints the recognized tokens with the current line and the content of the token .

This code is designed to recognize various patterns from the input and generate the corresponding tokens , while also monitoring and reporting errors and other information such as the line number.

## Editorial Analyst ( BISON )

Bison code defines a parser for a custom Uni - C language . The code consists of several main parts: preamble, tokens, precedence rules, grammar rules and the main function. Below is a detailed explanation of each section.

### Statements Area

```
%{
C language definitions and declarations . Anything to do with definition or
initialization
   variables & functions, header files and declarations # define enters at this
point */

        # include < stdio.h >
        # include < string.h >
        # include < stdlib.h >
        # define YYSTYPE char *
        # define YYDEBUG 1

        int lines = 1 ;
        int errflag = 0 ;

        extern char * yytext ;
        int correct_words = 0 ;
        int correct_exprs = 0 ;
        int fatal_errors = 0 ;
        int par_warnings = 0 ;
        int lex_warnings = 0 ;

        int yylex ();
        void yyerror ( const char * msg );

        /* The yyin pointer is what "points" to the input file. If not used
   of yyin , then the entry is made exclusively by the standard input (keyboard) */

        extern FILE * yyin ;
        extern FILE * yout ;
%}
```

**Definitions and Declarations in the C language : Includes the** C standard libraries , defines the YYSTYPE constant for semantic values, and sets the program debug constant with the YYDEBUG constant .

**Global Variables:** Declare global variables to track lines, errors, warnings, and correct expressions.

**External Declarations:** Declares external ( extern ) variables and functions that are also used in the parser ( FLEX ).

## Definition of Tokens

```
/* Definition of recognizable lexical units. */
%token IDENTIFIER STRING
%token INTEGER
%token FLOAT
%token SBREAK SDO SIF SSIZEOF SCASE SDOUBLE SINT SSTRUCT SFUNC SELSE SLONG SSWITCH
SCONST SFLOAT SRETURN SVOID SCONTINUE SFOR SSHORT SWHILE
%token PLUS "+"
%token MULEQ "*="
%token PMINEQ "--"
%token MINUS "-"
%token DIVEQ "/="
%token LT "<"
%token MUL "*"
%token NOT "!"
%token GT ">"
%token DIV "/"
%token AND "&&"
%token LEQ "<="
%token MOD "%"
%token OR "||"
%token GREQ ">="
%token ASSIGN "="
%token EQUAL "=="
%token ADDR "&"
%token PLUSEQ "+="
%token NOTEQ "!="
%token MINEQ "-="
%token PPLUSEQ "++"
%token OPENPAR "("
%token CLOSEPAR ")"
%token OPENSQBRA "["
%token CLOSESQBRA "]"
%token OPENCURBRA "{"
%token CLOSECURBRA "}"
%token COMMA ","
%token BACKSLASH "\\"
%token DELIMITER ";"
%token SSCAN SPRINT SLEN SCMP
% tokens NEWLINE
% tokens TOKEN _ ERROR
```

**Tokens Declarations :** Lists all tokens recognized by the parser, with some tokens associated with specific strings.

## Priority of Performers

```
/* Prioritize tokens */
%left ","
%right "*=" "/=" "+=" "-=" "="
%left "||"
%left "&&"
%left "==" "!="
%left "<" ">" "<=" ">="
%left "+" "-"
%left "*" "/" "%"
%right "&" "!"
%left "++" "--"
```

**Precedence and Connectivity:** Sets the priority and connectivity for operators to resolve ambiguities.

## Start program

```
/* Start program*/
% start program

%%
```

It states that the rule **program** it will be the first symbol to enter the Stack.

## Grammar Rules

```
* Definition of grammar rules. Whenever a grammar is matched
   rule with the input data, the C code between the
   hugs. The expected pension is:
                               name : rule { C code } */
program :
        program decl_statements NEWLINE { correct_exprs ++; if ( $ 2 != " \n " )
fprintf ( yyout , " [BISON] Line= %d , expression= %s \n\n " , line - 1 , $ 2 ); }
        | program error NEWLINE { fatal_errors ++; errflag = 1 ; yyerrok ; }
        | program merge_arr TOKEN_ERROR merge_arr NEWLINE { yyerrok ; }
        |                                                  { }
        ;
```

**Program Structure:** defines ' **program** ' as a series of ' **decl _ statements** ' separated by ' **NEWLINE** '.

```
/* ============== [2.2] Variable Declarations ============== */
decl _ var :
        type var { $$ = strdup ( yytext ); }
        ;

Type :
        SINT                  { $$ = strdup ( yytext ); }
        | SFLOAT              { $$ = strdup ( yytext ); }
        | SDOUBLE             { $$ = strdup ( yytext ); }
        | SSHORT              { $$ = strdup ( yytext ); }
        | SLONG               { $$ = strdup ( yytext ); }
       /* ## Warning ## -> Extra keyword in variable declaration */
        | SFLOAT SFLOAT { par_warnings ++; $$ = strdup ( yytext ); fprintf ( yyout ,
" ## Warning ## -> Double float detected at Line= %d \n " , line ); }
        | SDOUBLE SDOUBLE { par_warnings ++; $$ = strdup ( yytext ); fprintf ( yyout
, " ## Warning ## -> Double double detected at Line= %d \n " , line ); }
        | SINT SINT { par_warnings ++; $$ = strdup ( yytext ); fprintf ( yyout , "
## Warning ## -> Double int detected at Line= %d \n " , line ); }
        | SLONG SLONG { par_warnings ++; $$ = strdup ( yytext ); fprintf ( yyout , "
## Warning ## -> Double long detected at Line= %d \n " , line ); }
        | SSHORT SSHORT { par_warnings ++; $$ = strdup ( yytext ); fprintf ( yyout ,
" ## Warning ## -> Double short detected at Line= %d \n " , line ); }
/* ################################################## ## */
        ;

var:
IDENTIFIER { $$ = strdup ( yytext ); }
        | var " , " var { $$ = strdup ( yytext ); }
        ;
```

**Variable Declarations:** Defines how variables and their types are declared, including warnings about duplicate keywords.

```
/* ============= [2.3] Arrays ============= */
pos_elem:
IDENTIFIER " [ " INTEGER " ] "      { $$ = strdup ( yytext ); }
        | IDENTIFIER " [ " IDENTIFIER " ] " { $$ = strdup ( yytext ); }
        ;


arr_elements:
        " [ " " ] "           { $$ = strdup ( yytext ); }
        | " [ " integ " ] "   { $$ = strdup ( yytext ); }
        | " [ " fl " ] "      { $$ = strdup ( yytext ); }
        | " [ " str " ] "     { $$ = strdup ( yytext ); }
        | " [ " var " ] "     { $$ = strdup ( yytext ); }
        ;

integer:
```

```
INTEGER { $$ = strdup ( yytext ); }
        | integ " , " integ { $$ = strdup ( yytext ); }
        ;

fl:
FLOAT { $$ = strdup ( yytext ); }
        | fl " , " fl { $$ = strdup ( yytext ); }
        ;

str:
STRING { $$ = strdup ( yytext ); }
        | str " , " str { $$ = strdup ( yytext ); }
        ;
```

**Table Manipulation:** Defines rules for table elements and their positions.

```
/* ============== [2.4] Built-in simple functions ============== */
build_func :
        func { $$ = strdup ( yytext ); }
        ;

function:
SSCAN " ( " scan_params " ) "         { $$ = strdup ( yytext ); }
        | SLEN " ( " len_params " ) "          { $$ = strdup ( yytext ); }
        | SCMP " ( " cmp_params " ) "          { $$ = strdup ( yytext ); }
        | SPRINT " ( " print_params " ) "      { $$ = strdup ( yytext ); }
        | IDENTIFIER " ( " print_params " ) " { $$ = strdup ( yytext ); }
        ;

scan_params:
IDENTIFIER { $$ = strdup ( yytext ); }
        ;

len_params:
arr_elements { $$ = strdup ( yytext ); }
        | STRING { $$ = strdup ( yytext ); }
        | IDENTIFIER { $$ = strdup ( yytext ); }
        ;

cmp_params:
STRING { $$ = strdup ( yytext ); }
        | IDENTIFIER { $$ = strdup ( yytext ); }
        | cmp_params " , " cmp_params { $$ = strdup ( yytext ); }
        ;

print_params:
STRING { $$ = strdup ( yytext ); }
        | IDENTIFIER { $$ = strdup ( yytext ); }
        | INTEGER { $$ = strdup ( yytext ); }
        | FLOAT { $$ = strdup ( yytext ); }
        | func { $$ = strdup ( yytext ); }
        | pos_elem { $$ = strdup ( yytext ); }
        | print_params " , " print_params { $$ = strdup ( yytext ); }
        ;
```

**Built-in Simple Function Calls: Handles** Uni - C built-in simple function definitions such as **print , len , cmp , scan**

```
/* ============== [2.5] Declaring user functions ============== */
decl_func :
        name_func decl_statement { $$ = strdup ( yytext ); }
        ;

name_func:
SFUNC { $$ = strdup ( yytext ); }
        /* ## Warning ## -> Return type in functions */
        | SFUNC type { par_warnings ++; $$ = strdup ( yytext ); fprintf ( yyout , "
## Warning ## -> Return type unnecessary at Line= %d \n " , line ); }
/* ################################################# ## */
        | name_func IDENTIFIER params NEWLINE { $$ = strdup ( yytext ); }
        ;

parameters:
        " ( " " ) "                   { $$ = strdup ( yytext ); }
        | " ( " type_params " ) "      { $$ = strdup ( yytext ); }
        ;

type_params:
type IDENTIFIER { $$ = strdup ( yytext ); }
        | type_params " , " type_params { $$ = strdup ( yytext ); }
        ;
```

**User Function Handling: Handles user function definitions according to** Uni - C language standards .

```
/* ============== [2.6] Declarations of simple expressions ============== */
/* [2.6.1] Numeric expressions */
sign:
INTEGER { $$ = strdup ( yytext ); }
        | FLOAT { $$ = strdup ( yytext ); }
        | IDENTIFIER { $$ = strdup ( yytext ); }
        | " + " sign { $$ = strdup ( yytext ); }
        | " - " sign { $$ = strdup ( yytext ); }
        ;

arithmetic_expr:
sign { $$ = strdup ( yytext ); }
        | arithmetic_expr " + " arithmetic_expr { $$ = strdup ( yytext ); }
        | arithmetic_expr " - " arithmetic_expr { $$ = strdup ( yytext ); }
        | arithmetic_expr " * " arithmetic_expr { $$ = strdup ( yytext ); }
        | arithmetic_expr " / " arithmetic_expr { $$ = strdup ( yytext ); }
        | arithmetic_expr " % " arithmetic_expr { $$ = strdup ( yytext ); }
        ;

number:
INTEGER { $$ = strdup ( yytext ); }
        | FLOAT { $$ = strdup ( yytext ); }
        | pos _ elem  { $$ = strdup ( yytext ); }
        ;


/* [2.6.2] Variable value assignments */
assign:
var " = " val { $$ = strdup ( yytext ); }
        | var " = " cmp_expr { $$ = strdup ( yytext ); }
        | var " = " arithmetic_expr { $$ = strdup ( yytext ); }
        | var " = " merge_arr { $$ = strdup ( yytext ); }
        | oper_eq { $$ = strdup ( yytext ); }
        ;

oper_eq:
var " ++ "                { $$ = strdup ( yytext ); }
        | var " -- "              { $$ = strdup ( yytext ); }
        | " ++ " var { $$ = strdup ( yytext ); }
        | " -- " var { $$ = strdup ( yytext ); }
        | var " += " val { $$ = strdup ( yytext ); }
        | var " -= " val2 { $$ = strdup ( yytext ); }
        | var " *= " val2 { $$ = strdup ( yytext ); }
        | var " /= " val2 { $$ = strdup ( yytext ); }
        ;

val2:
number { $$ = strdup ( yytext ); }
    | IDENTIFIER { $$ = strdup ( yytext ); }
    ;

val:
number { $$ = strdup ( yytext ); }
    | IDENTIFIER { $$ = strdup ( yytext ); }
    | STRING { $$ = strdup ( yytext ); }
    | arr_elements { $$ = strdup ( yytext ); }
    | val " , " val { $$ = strdup ( yytext ); }
    ;

/* [2.6.3] Comparisons */
cmp_expr:
INTEGER { $$ = strdup ( yytext ); }
```

```
        | FLOAT { $$ = strdup ( yytext ); }
        | IDENTIFIER { $$ = strdup ( yytext ); }
        | cmp_expr " > " cmp_expr { $$ = strdup ( yytext ); }
        | cmp_expr " < " cmp_expr { $$ = strdup ( yytext ); }
        | cmp_expr " <= " cmp_expr { $$ = strdup ( yytext ); }
        | cmp_expr " >= " cmp_expr { $$ = strdup ( yytext ); }
        | cmp_expr " == " cmp_expr { $$ = strdup ( yytext ); }
        | cmp_expr " != " cmp_expr { $$ = strdup ( yytext ); }
        | cmp_expr " || " cmp_expr { $$ = strdup ( yytext ); }
        | cmp_expr " && " cmp_expr { $$ = strdup ( yytext ); }
        | " ! " cmp_expr { $$ = strdup ( yytext ); }
/* ## Warning ## -> Double comparison symbol */
        | cmp_expr " > " " > " cmp_expr { par_warnings ++; $$ = strdup ( yytext );
fprintf ( yyout , " ## Warning ## -> Double > detected at Line= %d \n " , line - 1
); }
        | cmp_expr " < " " < " cmp_expr { par_warnings ++; $$ = strdup ( yytext );
fprintf ( yyout , " ## Warning ## -> Double < detected at Line= %d \n " , line - 1
); }
/* ################################################# ## */
        ;

/* [2.6.4] Join Tables */
merge_arr:
arr_elements { $$ = strdup ( yytext ); }
        | merge_arr " + " merge_arr { $$ = strdup ( yytext ); }
        /* ## Warning ## -> Invalid characters in array concatenation */
        | merge_arr TOKEN_ERROR " + " merge_arr { par_warnings ++; $$ = strdup (
yytext ); fprintf ( yyout , " ## Warning ## -> Invalid character in array merge
detected at Line= %d \n " , line ); }
        | merge_arr " + " TOKEN_ERROR merge_arr { par_warnings ++; $$ = strdup (
yytext ); fprintf ( yyout , " ## Warning ## -> Invalid character in array merge
detected at Line= %d \n " , line ); }
        /* ################################################# ## */
        ;
```

**Handling Expressions:** Deals with various expressions, including numeric, comparative, and variable assignment expressions.

```
/* ============== [2.7] Compound Statements ============== */
decl_statements:
decl_statement { $$ = $ 1 ;  }
        | decl_statements decl_statement { $$ = $ 2 ;  if ( $ 2 != " \n " ) fprintf
( yyout , " [BISON] Line= %d , expression= %s \n\n " , line - 1 , $ 2 ); }
        ;

decl_statement:
if_statement { $$ = " \" Statement if \" " ; }
        | while_statement { $$ = " \" While Statement \" " ; }
        | for_statement { $$ = " \" Statement for \" " ; }
        | decl_var " ; "                { $$ = " \" Variable declaration \" " ; }
        | build_func " ; "              { $$ = " \" Function call \" " ; }
        | decl _ func                   { $$ = " \" Declaration of user functions
\" " ; }
        | assign " ? "                  { $$ = " \" Assign value to variable \" "
; }
        | arithmetic _ expr                 { $$ = " \" Numeric expression \" " ;
}
        | cmp_expr                    { $$ = " \" Compare \" " ; }
        | merge _ arr                   { $$ = " \" Join Tables \" " ; }
        | block_statement             { $$ = " \" Compound Statements \" " ; }
        | NEWLINE                     { $$ = " \ n " ; }
        ;

/* [2.7.1] The if statement */
if_statement:
SIF condition decl_statement { $$ = strdup ( yytext ); }
        ;

conditions:
cmp_expr { $$ = strdup ( yytext ); }
        | " ( " condition " ) "    { $$ = strdup ( yytext ); }
/* ## Warning ## -> Extra parentheses in if, while statements */
        | " ( " " ( " condition " ) " " ) "    { par_warnings ++; $$ = strdup (
yytext ); fprintf ( yyout , " ## Warning ## -> Double parethensis detected at Line=
%d \n " , line ); }
/* ################################################## ## */
        ;

block_statement:
        " { " decl_statements " } " { $$ = strdup ( yytext ); }
        ;

/* [2.7.2] The while statement */
while_statement:
SWHILE condition decl_statement { $$ = strdup ( yytext ); }
        ;

/* [2.7.3] The for statement */
for_statement:
SFOR " ( " assign " ; " cmp_expr " ; " oper_eq " ) " decl_statement { $$ = strdup (
yytext ); }
        ;

%%
```

**Handling Complex Statements:** Deals with the composition of if control structures and while and for loop structures .

## User Code

```
/* The main function which is also the starting point of the program.
   In this particular case it simply calls Bison 's yyparse function
   to start the syntactic analysis. */
int main ( int argc , char ** argv )
{
yydebug = 0 ;

        if ( argc == 3 )
        {
                if (!( yyin = fopen ( argv [ 1 ], " r " )))
                {
                        fprintf ( stderr , " Cannot read file: %s \n " , argv [ 1
]);
                        return 1 ;
                }
                if (!( yout = fopen ( argv [ 2 ], " w " )))
                {
                        fprintf ( stderr , " Cannot create file: %s \n " , argv [ 2
]);
                        return 1 ;
                }
        }

        int parse = yyparse ();

        fprintf ( yyout , " \n\n\t\t PARSING STATISTICS \n\n " );

        if ( errflag == 0 && parse == 0 )
                fprintf ( yyout , " BISON -> Parsing completed successfully \ n " );
        else
                fprintf ( yyout , " BISON -> Parsing failed \ n " );

        if ( par_warnings > 0 )
                fprintf ( yyout , " \t\t (with %d warnings) \n\n " , par_warnings );


        fprintf ( yyout , " \t\t CORRECT WORDS: %d \n " , correct_words );
        fprintf ( yyout , " \t\t CORRECT EXPRESSIONS: %d \n " , correct_exprs );
        fprintf ( yyout , " \t\t WRONG WORDS: %d \n " , lex_warnings );
        fprintf ( yyout , " \t\t ERROR EXPRESSIONS: %d \n " , fatal_errors );
        fprintf ( yyout , " \n " );
        fclose ( yyin );
        fclose ( yout );

        return 0 ;
}
```

**Main Function Procedure:** Checks command line arguments, opens I/O files to read input data, and prints the analysis from LA and from SA for the input strings. The SA is performed by calling the **yyparse function** . Finally, the Analysis Statistics are printed on the output which include the results from error management (correct words, correct expressions, wrong words, wrong expressions)

Bison code defines a parser for the Uni - C language with specific tokens, precedence rules, grammar rules, and error handling. The parser reads from an input file and writes the parse results to an output file, reporting the number of processed lines, critical errors, warnings, and correct expressions. The code includes detailed handling for variables, arrays, functions, numeric and logical expressions, and control statements.

## Control cases and commentary

### 2.2 Declarations of Variables

#### Input File ( input . txt )

```
int a ?

double var;

float c;

int a, b, c;

long var1,var2,var3,var4;

short sh1;
```

## File Output (output.txt)

```
       [FLEX] Line=3, token=KEYWORD, value="int"

       [FLEX] Line=3, token=IDENTIFIER, value="a"

       [FLEX] Line=3, token=DELIMITER, value=";"



[BISON] Line=3, expression="Variable Declaration"
```

This line contains a simple declaration of a variable of type int named a . The lexer ( FLEX ) correctly detects the tokens : keyword int , identifier a , and the delimiter ? . The parser ( BISON ) recognizes this declaration as a "Variable Declaration".

```
       [FLEX] Line=4, token=KEYWORD, value="double"

       [FLEX] Line=4, token=IDENTIFIER, value="var"

       [FLEX] Line=4, token=DELIMITER, value=";"



[BISON] Line=4, expression="Variable declaration"
```

This line declares a variable of type double named var . Tokens are detected correctly and the parser recognizes the declaration as "Variable Declaration"

```
       [FLEX] Line=5, token=KEYWORD, value="float"

       [FLEX] Line=5, token=IDENTIFIER, value="c"

       [FLEX] Line=5, token=DELIMITER, value=";"



[BISON] Line=5, expression="Variable declaration"
```

A variable of type float named c is declared here . The tokens are detected correctly and the parser recognizes the declaration as a "Variable Declaration".

```
       [FLEX] Line=6, token=KEYWORD, value="int"

       [FLEX] Line=6, token=IDENTIFIER, value="a"

       [FLEX] Line=6, token=SPECIAL, value=","
```

```
      [FLEX] Line=6, token=IDENTIFIER, value="b"

      [FLEX] Line=6, token=SPECIAL, value=","

      [FLEX] Line=6, token=IDENTIFIER, value="c"

      [FLEX] Line=6, token=DELIMITER, value=";"



[BISON] Line=6, expression="Variable declaration"
```

This line declares several variables of type int ( a , b , c ). Tokens are correctly detected: keyword int , identifiers a , b , c , and the delimiters , and ? . The parser recognizes the declaration as a "Variable Declaration".

```
      [ FLEX ] Line =7, token = KEYWORD , value =" long "

      [FLEX] Line=7, token=IDENTIFIER, value="var1"

      [FLEX] Line=7, token=SPECIAL, value=","

      [FLEX] Line=7, token=IDENTIFIER, value="var2"

      [FLEX] Line=7, token=SPECIAL, value=","

      [FLEX] Line=7, token=IDENTIFIER, value="var3"

      [FLEX] Line=7, token=SPECIAL, value=","

      [FLEX] Line=7, token=IDENTIFIER, value="var4"

      [FLEX] Line=7, token=DELIMITER, value=";"



[BISON] Line=7, expression="Variable declaration"
```

This line declares several variables of type long ( var 1 , var 2 , var 3 , var 4 ). Tokens are correctly detected : keyword long , identifiers var 1 , var 2 , var 3 , var 4 , and the delimiters , and ? . The parser recognizes the declaration as a "Variable Declaration".

```
      [ FLEX ] Line =8, token = KEYWORD , value = " short "

      [FLEX] Line=8, token=IDENTIFIER, value="sh1"

      [FLEX] Line=8, token=DELIMITER, value=";"



[BISON] Line=8, expression="Variable declaration"
```

A variable of type short named sh 1 is declared here . The tokens are detected correctly and the parser recognizes the declaration as a "Variable Declaration".

## 2. 3 Tables

### Input File ( input . txt )

```
pin1 = [1, 2, 3, 4, 5 ];

pin2 = ["a", "b", "c", "d"];

pin3 = [4.5, 4e1, 4E-1, 0e0, 5.67];

pin4 = [a, b, c, d];
```

### Output File ( output . txt )

We see that the above inputs are correctly recognized by the parser as a value assignment to a variable, as this is how we define arrays in the Uni - C language .

```
      [FLEX] Line=12, token=IDENTIFIER, value="pin1"

      [FLEX] Line=12, token=OPERATOR, value="="

      [FLEX] Line=12, token=SPECIAL, value="["

      [FLEX] Line=12, token=INTEGER, value="1"

      [FLEX] Line=12, token=SPECIAL, value=","

      [FLEX] Line=12, token=INTEGER, value="2"

      [FLEX] Line=12, token=SPECIAL, value=","

      [FLEX] Line=12, token=INTEGER, value="3"

      [FLEX] Line=12, token=SPECIAL, value=","

      [FLEX] Line=12, token=INTEGER, value="4"

      [FLEX] Line=12, token=SPECIAL, value=","

      [FLEX] Line=12, token=INTEGER, value="5"

      [FLEX] Line=12, token=SPECIAL, value="]"
```

```
        [FLEX] Line=12, token=DELIMITER, value=";"

[BISON] Line=11, expression="Assign value to variable"




[BISON] Line=12, expression="Assign value to variable"
```

The tokens are detected one by one by the parser. This is a simple declaration of an array of integers in variable pin 1. The end of the statement is delimited by the delimiter (?), and the elements of the array inside the brackets are separated by ','.

```
        [FLEX] Line=13, token=IDENTIFIER, value="pin2"

        [FLEX] Line=13, token=OPERATOR, value="="

        [FLEX] Line=13, token=SPECIAL, value="["

        [FLEX] Line=13, token=STRING, value=""a""

        [FLEX] Line=13, token=SPECIAL, value=","

        [FLEX] Line=13, token=STRING, value=""b""

        [FLEX] Line=13, token=SPECIAL, value=","

        [FLEX] Line=13, token=STRING, value=""c""

        [FLEX] Line=13, token=SPECIAL, value=","

        [FLEX] Line=13, token=STRING, value=""d""

        [FLEX] Line=13, token=SPECIAL, value="]"

        [FLEX] Line=13, token=DELIMITER, value=";"


[ BISON ] Line =13, expression ="Assign value to variable"
```

Tokens are detected by the lexical analyzer and the expression by the syntactic analyzer. It's a simple declaration of strings inside an array and assigning those elements to an array-variable on pin 2. The rules are the same as the previous example.

```
        [ FLEX ] Line =14, token = IDENTIFIER , value =" pin 3"

        [FLEX] Line=14, token=OPERATOR, value="="

        [FLEX] Line=14, token=SPECIAL, value="["
        [FLEX] Line=14, token=FLOAT, value="4.5"

        [FLEX] Line=14, token=SPECIAL, value=","

        [FLEX] Line=14, token=FLOAT, value="4e1"

        [FLEX] Line=14, token=SPECIAL, value=","

        [FLEX] Line=14, token=FLOAT, value="4E-1"

        [FLEX] Line=14, token=SPECIAL, value=","

        [FLEX] Line=14, token=FLOAT, value="0e0"

        [FLEX] Line=14, token=SPECIAL, value=","

        [FLEX] Line=14, token=FLOAT, value="5.67"

        [FLEX] Line=14, token=SPECIAL, value="]"

        [FLEX] Line=14, token=DELIMITER, value=";"



[ BISON ] Line =14, expression ="Assign value to variable"
```

The tokens are again correctly approved by the parser and the. This time it is the declaration of an array of float values, in a variable pin 3. The delimitation is similar to the previous examples of this rule.

```
        [FLEX] Line=15, token=IDENTIFIER, value="pin4"

        [FLEX] Line=15, token=OPERATOR, value="="

        [FLEX] Line=15, token=SPECIAL, value="["

        [FLEX] Line=15, token=IDENTIFIER, value="a"

        [FLEX] Line=15, token=SPECIAL, value=","

        [FLEX] Line=15, token=IDENTIFIER, value="b"

        [FLEX] Line=15, token=SPECIAL, value=","

        [FLEX] Line=15, token=IDENTIFIER, value="c"

        [FLEX] Line=15, token=SPECIAL, value=","

        [FLEX] Line=15, token=IDENTIFIER, value="d"

        [FLEX] Line=15, token=SPECIAL, value="]"

        [FLEX] Line=15, token=DELIMITER, value=";"
```

```
[ BISON ] Line =15, expression ="Assign value to variable"
```

The tokens are recognized by the speech analyzer. The values we now declare in an array-variable 3 are other variables. The delimitation is common to the previous examples however we do not know if all the variables are of the same type. We are not concerned with this, however, as we are concerned up to the level of the recognition of the syntactic rule.

## 2.4 Built-in simple functions

### Input File ( input . txt )

```
/* --- Test case [#2.4.1] : SCAN --- */


scan(x);

scan(MyVariable);



/* --- Test case [#2.4.2] : LEN --- */


len([10, 20, 30, 40, 50]);

len("This is a string");

len(StringVariable);



/* --- Test case [#2.4.3] : CMP --- */


cmp("test", "best");

cmp(str1, str2);



/* --- Test case [#2.4.4] : PRINT --- */
```

```
print("Hello World");

print(x, "=", 100);

print(cmp(str1, str2));

print(len("This is a string"));

print(pin[0]);
```

## File Output (output.txt)

The parser recognizes the call of built-in functions (scan,len,cmp,print,scan). The syntax rules are common: function name → parenthesis → function content → parenthesis → delimiter(?).

```
/* --- Test case [#2.4.1] : SCAN --- */


      [FLEX] Line=20, token=FUNCTION, value="scan"

      [FLEX] Line=20, token=SPECIAL, value="("

      [FLEX] Line=20, token=IDENTIFIER, value="x"

      [FLEX] Line=20, token=SPECIAL, value=")"

      [FLEX] Line=20, token=DELIMITER, value=";"


[BISON] Line=20, expression="Function call"
```

```
      [ FLEX] Line=21, token=FUNCTION, value="scan"

      [FLEX] Line=21, token=SPECIAL, value="("

      [FLEX] Line=21, token=IDENTIFIER, value="MyVariable"

      [FLEX] Line=21, token=SPECIAL, value=")"

      [FLEX] Line=21, token=DELIMITER, value=";"


[BISON] Line=21, expression="Function call"
```

Recognition of tokens by the speech analyzer in 2 scan () functions. The expression recognized is printing the value of a variable x (on line 20) and the value of a variable MyVariable (on line 21). Expressions are recognized as 'Function Call'.

```
/* --- Test case [#2.4.2] : LEN --- */


      [ FLEX ] Line =25, token = FUNCTION , value =" len "

      [FLEX] Line=25, token=SPECIAL, value="("

      [FLEX] Line=25, token=SPECIAL, value="["

      [FLEX] Line=25, token=INTEGER, value="10"

      [FLEX] Line=25, token=SPECIAL, value=","

      [FLEX] Line=25, token=INTEGER, value="20"

      [FLEX] Line=25, token=SPECIAL, value=","

      [FLEX] Line=25, token=INTEGER, value="30"

      [FLEX] Line=25, token=SPECIAL, value=","

      [FLEX] Line=25, token=INTEGER, value="40"

      [FLEX] Line=25, token=SPECIAL, value=","

      [FLEX] Line=25, token=INTEGER, value="50"

      [FLEX] Line=25, token=SPECIAL, value="]"

      [FLEX] Line=25, token=SPECIAL, value=")"

      [FLEX] Line=25, token=DELIMITER, value=";"

[BISON] Line=24, expression="Function call"




[BISON] Line=25, expression="Function call"
```

```
      [FLEX] Line=26, token=FUNCTION, value="len"

      [FLEX] Line=26, token=SPECIAL, value="("

      [FLEX] Line=26, token=STRING, value=""This is a string""
      [FLEX] Line=26, token=SPECIAL, value=")"

      [FLEX] Line=26, token=DELIMITER, value=";"



[BISON] Line=26, expression="Function call"
```

```
      [FLEX] Line=27, token=FUNCTION, value="len"

      [FLEX] Line=27, token=SPECIAL, value="("

      [FLEX] Line=27, token=IDENTIFIER, value="StringVariable"

      [FLEX] Line=27, token=SPECIAL, value=")"

      [FLEX] Line=27, token=DELIMITER, value=";"



[BISON] Line=27, expression="Function call"
```

Recognition of tokens by the parser in 3 len () functions. The expression that is recognized is the count of the number of elements of the table (line 25), the count of the characters of the string (line 26) and finally the count of the number of characters of a string variable (line 27). Expressions are also correctly recognized as “Function calls”.

```
/* --- Test case [#2.4.3] : CMP --- */



      [FLEX] Line=31, token=FUNCTION, value="cmp"

      [FLEX] Line=31, token=SPECIAL, value="("

      [FLEX] Line=31, token=STRING, value=""test""

      [FLEX] Line=31, token=SPECIAL, value=","

      [FLEX] Line=31, token=STRING, value=""best""

      [FLEX] Line=31, token=SPECIAL, value=")"

      [FLEX] Line=31, token=DELIMITER, value=";"

[BISON] Line=30, expression="Function call"
```

```
[BISON] Line=31, expression="Function call"
```

```
       [FLEX] Line=32, token=FUNCTION, value="cmp"

       [FLEX] Line=32, token=SPECIAL, value="("

       [FLEX] Line=32, token=IDENTIFIER, value="str1"

       [FLEX] Line=32, token=SPECIAL, value=","

       [FLEX] Line=32, token=IDENTIFIER, value="str2"

       [FLEX] Line=32, token=SPECIAL, value=")"

       [FLEX] Line=32, token=DELIMITER, value=";"



[BISON] Line=32, expression="Function call"
```

Recognition of tokens by the parser in 2 cmp () functions. The expression recognized is the comparison of 2 strings (line 31) and the comparison of the value of 2 variables (line 32). Expressions are correctly recognized as “Function Call”.

```
/* --- Test case [#2.4.4] : PRINT --- */



       [FLEX] Line=36, token=FUNCTION, value="print"

       [FLEX] Line=36, token=SPECIAL, value="("

       [FLEX] Line=36, token=STRING, value=""Hello World""

       [FLEX] Line=36, token=SPECIAL, value=")"

       [FLEX] Line=36, token=DELIMITER, value=";"

[BISON] Line=35, expression="Function call"





[BISON] Line=36, expression="Function call"
```

```
       [FLEX] Line=37, token=FUNCTION, value="print"
```

```
   [FLEX] Line=37, token=SPECIAL, value="("

       [FLEX] Line=37, token=IDENTIFIER, value="x"

       [FLEX] Line=37, token=SPECIAL, value=","

       [FLEX] Line=37, token=STRING, value=""=""

       [FLEX] Line=37, token=SPECIAL, value=","

       [FLEX] Line=37, token=INTEGER, value="100"

       [FLEX] Line=37, token=SPECIAL, value=")"

       [FLEX] Line=37, token=DELIMITER, value=";"



[BISON] Line=37, expression="Function call"
```

```
       [FLEX] Line=38, token=FUNCTION, value="print"

       [FLEX] Line=38, token=SPECIAL, value="("

       [FLEX] Line=38, token=FUNCTION, value="cmp"

       [FLEX] Line=38, token=SPECIAL, value="("

       [FLEX] Line=38, token=IDENTIFIER, value="str1"

       [FLEX] Line=38, token=SPECIAL, value=","

       [FLEX] Line=38, token=IDENTIFIER, value="str2"

       [FLEX] Line=38, token=SPECIAL, value=")"

       [FLEX] Line=38, token=SPECIAL, value=")"

       [FLEX] Line=38, token=DELIMITER, value=";"



[BISON] Line=38, expression="Function Call"
```

```
       [FLEX] Line=39, token=FUNCTION, value="print"

       [FLEX] Line=39, token=SPECIAL, value="("

       [FLEX] Line=39, token=FUNCTION, value="len"
```

```
      [FLEX] Line=39, token=SPECIAL, value="("

      [FLEX] Line=39, token=STRING, value=""This is a string""

      [FLEX] Line=39, token=SPECIAL, value=")"

      [FLEX] Line=39, token=SPECIAL, value=")"

      [FLEX] Line=39, token=DELIMITER, value=";"


[BISON] Line=39, expression="Function call"
```

```
      [FLEX] Line=40, token=FUNCTION, value="print"

      [FLEX] Line=40, token=SPECIAL, value="("

      [FLEX] Line=40, token=IDENTIFIER, value="pin"

      [FLEX] Line=40, token=SPECIAL, value="["

      [FLEX] Line=40, token=INTEGER, value="0"

      [FLEX] Line=40, token=SPECIAL, value="]"

      [FLEX] Line=40, token=SPECIAL, value=")"

      [FLEX] Line=40, token=DELIMITER, value=";"


[BISON] Line=40, expression="Function call"
```

Recognition of tokens by the parser in 5 print () functions. The expression recognized is print a string (line 36), print a variable, character, value (line 37). A cmp function within the print expression (line 38) is also recognized , as is the len function within print (line 39). Finally, line 40 recognizes the printing of the first element of an array named pin . All the above expressions are correctly recognized as “Function call”.

## 2.5 Declaration of User Functions

### Input File ( input . txt )

```
func myfunc ()

{

}

func myfunc2(int paramA, long paramB, short paramC)

{



}

func main()

{

print("Hello World\n");

}
```

### File Output (output.txt)

```
        [FLEX] Line=44, token=KEYWORD, value="func"

        [FLEX] Line=44, token=IDENTIFIER, value="myfunc"

        [FLEX] Line=44, token=SPECIAL, value="("

        [FLEX] Line=44, token=SPECIAL, value=")"



        [FLEX] Line=45, token=SPECIAL, value="{"



        [FLEX] Line=46, token=SPECIAL, value="}"
```

```
[BISON] Line=45, expression="Declaration of user functions"




[BISON] Line=46, expression="Declaration of user functions"
```

The tokens recognized by Flex are correct: func as KEYWORD , myfunc as IDENTIFIER , the (, ) and {, } as SPECIAL .

The expressions recognized by Bison (Lines 45 and 46) are correct as "Declaration of User Functions", since the function myfunc was correctly declared without parameters.

```
      [FLEX] Line=47, token=KEYWORD, value="func"

      [FLEX] Line=47, token=IDENTIFIER, value="myfunc2"

      [FLEX] Line=47, token=SPECIAL, value="("

      [FLEX] Line=47, token=KEYWORD, value="int"

      [FLEX] Line=47, token=IDENTIFIER, value="paramA"

      [FLEX] Line=47, token=SPECIAL, value=","

      [FLEX] Line=47, token=KEYWORD, value="long"

      [FLEX] Line=47, token=IDENTIFIER, value="paramB"

      [FLEX] Line=47, token=SPECIAL, value=","

      [FLEX] Line=47, token=KEYWORD, value="short"

      [FLEX] Line=47, token=IDENTIFIER, value="paramC"

      [FLEX] Line=47, token=SPECIAL, value=")"


      [FLEX] Line=48, token=SPECIAL, value="{"




      [FLEX] Line=50, token=SPECIAL, value="}"



[BISON] Line=50, expression="Declaration of user functions"
```

The tokens that were recognized from the Flex is correct : func as KEYWORD, myfunc2 as IDENTIFIER, the (, ), {, }, , as SPECIAL, and the types data int, long, short as KEYWORD.

The expression recognized by Bison (Line 50) is correct as "Declaration of User Functions", since the function myfunc 2 was correctly declared with parameters.

```
       [FLEX] Line=51, token=KEYWORD, value="func"

       [FLEX] Line=51, token=IDENTIFIER, value="main"

       [FLEX] Line=51, token=SPECIAL, value="("

       [FLEX] Line=51, token=SPECIAL, value=")"



       [FLEX] Line=52, token=SPECIAL, value="{"



       [FLEX] Line=53, token=FUNCTION, value="print"

       [FLEX] Line=53, token=SPECIAL, value="("

       [FLEX] Line=53, token=STRING, value=""Hello World\n""

       [FLEX] Line=53, token=SPECIAL, value=")"

       [FLEX] Line=53, token=DELIMITER, value=";"

[BISON] Line=52, expression="Function call"




       [FLEX] Line=54, token=SPECIAL, value="}"



[BISON] Line=54, expression=" Declaration functions user "
```

The tokens that were recognized from the Flex is correct : func as KEYWORD, main as IDENTIFIER, ta (, ), {, }, ; as SPECIAL, n function print as FUNCTION, and or string "Hello World\n" as STRING;

The expression recognized by Bison (Line 52) as "Function Call" is correct, since the print function call is valid inside the main function .

The expression recognized by Bison (Line 54) as "Declaration of user functions" is also correct, as the main function was declared and closed correctly.

## 2. 6 Statements of Simple Expressions

**Input File ( input . txt )**

```
/* --- Test case [#2.6.1] : ARITHMETIC-EXPRESSIONS --- */


a1 * a2

a3 + a4

-5 + 10

15 + a5 - 9

+5 - a1 -16

+10 -5 + myvar - myvar2

5 % 1 / 2

+1 -4 / 7

-varA +varB * 1


/* --- Test case [#2.6.2] : VARIABLES-INITIALIZE --- */


x1=0;

x1, x2 = 0, 1;

x1, list1 = 0, ["A", "B", "C"];

x1, list1, string = 0, ["A", "B", "C"], "HELLO"?

var++;

++var;

--var;

var--;

list1 += [1, 2, 3];
```

```
x -= 4e1;

y *= 6.7;

c /= b;


/* --- Test case [#2.6.3] : ARRAY-MERGE --- */


[1, 2, 3] + [4, 5, 6]

[1, 3] + [5, 7, 9, 11]

["What", "a"] + ["Wonderful", "World"]

[4.5, 5.6, 1.1] + [0e0, 4.5E-1, 4E0]


/* --- Test case [#2.6.4] : COMPARISONS --- */


x1 > x2

a < b

myvar >= 52

var1 <= var2

isEven == isOdd

a != C

isTrue1 || isTrue2

isFalse1 && isFalse2

!not
```

## File Output (output.txt)

```
/* --- Test case [#2.6.1] : ARITHMETIC-EXPRESSIONS --- */


       [FLEX] Line=59, token=IDENTIFIER, value="a1"

       [FLEX] Line=59, token=OPERATOR, value="*"

       [FLEX] Line=59, token=IDENTIFIER, value="a2"



[BISON] Line=59, expression="Numeric expression"
```

It is recognized as a numeric expression with two variables ( a 1 and a 2 ) and the multiplication operator (*).

LA recognizes a 1 and a 2 as IDENTIFIER and * as OPERATOR .

SA identifies the line as "Numeric Expression".

```
       [FLEX] Line=60, token=IDENTIFIER, value="a3"

       [FLEX] Line=60, token=OPERATOR, value="+"

       [FLEX] Line=60, token=IDENTIFIER, value="a4"



[BISON] Line=60, expression="Numeric expression"
```

It is recognized as a numeric expression with two variables ( a 3 and a 4 ) and the addition operator (+).

LA recognizes a 3 and a 4 as IDENTIFIER and + as OPERATOR .

SA identifies the line as "Numeric Expression".

```
       [FLEX] Line=61, token=OPERATOR, value="-"

       [FLEX] Line=61, token=INTEGER, value="5"

       [FLEX] Line=61, token=OPERATOR, value="+"

       [FLEX] Line=61, token=INTEGER, value="10"
```

```
[BISON] Line=61, expression="Numeric expression"
```

Recognized as a numeric expression with two integers (5 and 10) and the subtraction (-) and addition (+) operators.

LA recognizes 5 and 10 as INTEGER and - and + as OPERATOR .

SA identifies the line as "Numeric Expression".

```
        [FLEX] Line=62, token=INTEGER, value="15"

        [FLEX] Line=62, token=OPERATOR, value="+"

        [FLEX] Line=62, token=IDENTIFIER, value="a5"

        [FLEX] Line=62, token=OPERATOR, value="-"

        [FLEX] Line=62, token=INTEGER, value="9"



[BISON] Line=62, expression="Numeric expression"
```

It is recognized as a numeric expression with an integer (15), a variable ( a 5), and another integer (9) and the addition (+) and subtraction (-) operators.

LA recognizes 15 and 9 as INTEGER , a 5 as IDENTIFIER and + and - as OPERATOR .

SA identifies the line as "Numeric Expression".

```
        [FLEX] Line=63, token=OPERATOR, value="+"

        [FLEX] Line=63, token=INTEGER, value="5"

        [FLEX] Line=63, token=OPERATOR, value="-"

        [FLEX] Line=63, token=IDENTIFIER, value="a1"

        [FLEX] Line=63, token=OPERATOR, value="-"

        [FLEX] Line=63, token=INTEGER, value="16"



[BISON] Line=63, expression="Numeric expression"
```

It is recognized as a numeric expression with an integer (5), a variable ( a 1), and another integer (16) and the addition (+) and subtraction (-) operators.

LA recognizes 5 and 16 as INTEGER , a 1 as IDENTIFIER and - as OPERATOR .

SA identifies the line as "Numeric Expression".

```
       [FLEX] Line=64, token=OPERATOR, value="+"

       [FLEX] Line=64, token=INTEGER, value="10"

       [FLEX] Line=64, token=OPERATOR, value="-"

       [FLEX] Line=64, token=INTEGER, value="5"

       [FLEX] Line=64, token=OPERATOR, value="+"

       [FLEX] Line=64, token=IDENTIFIER, value="myvar"

       [FLEX] Line=64, token=OPERATOR, value="-"

       [FLEX] Line=64, token=IDENTIFIER, value="myvar2"



[BISON] Line=64, expression="Numeric expression"
```

It recognizes as a numeric expression the integers (10 and 5) and what s variables myvar and mayvar 2 and the addition (+) and subtraction (-) operators.

LA recognizes 5 and 10 as INTEGER , myvar and myvar 2 as IDENTIFIER , and + and - as OPERATOR .

SA identifies the line as "Numeric Expression".

```
       [FLEX] Line=65, token=INTEGER, value="5"

       [FLEX] Line=65, token=OPERATOR, value="%"

       [FLEX] Line=65, token=INTEGER, value="1"

       [FLEX] Line=65, token=OPERATOR, value="/"

       [FLEX] Line=65, token=INTEGER, value="2"



[BISON] Line=65, expression="Numeric expression"
```

The integers (5, 1 and 2) and the division operators % and / are recognized as numerical expressions

LA recognizes 5 as 1 and 2 as INTEGER, % and / as OPERATOR .

SA identifies the line as "Numeric Expression".

```
       [FLEX] Line=66, token=OPERATOR, value="+"
       [FLEX] Line=66, token=INTEGER, value="1"

       [FLEX] Line=66, token=OPERATOR, value="-"

       [FLEX] Line=66, token=INTEGER, value="4"

       [FLEX] Line=66, token=OPERATOR, value="/"

       [FLEX] Line=66, token=INTEGER, value="7"



[BISON] Line=66, expression="Numeric expression"
```

It recognizes as a numerical expression the integers (1,4,7) and the addition (+), subtraction and division operators /

LA recognizes 1, 4 and 7 as INTEGER and / as OPERATOR .

SA identifies the line as "Numeric Expression".

```
       [FLEX] Line=67, token=OPERATOR, value="-"

       [FLEX] Line=67, token=IDENTIFIER, value="varA"

       [FLEX] Line=67, token=OPERATOR, value="+"

       [FLEX] Line=67, token=IDENTIFIER, value="varB"

       [FLEX] Line=67, token=OPERATOR, value="*"

       [FLEX] Line=67, token=INTEGER, value="1"



[BISON] Line=67, expression="Numeric expression"
```

Numerical expression the integers 1 and the variables varA and varB and the variables addition (+) subtraction (-) and multiplication (*) are recognized.

LA recognizes varA and varB as IDENTIFIER and 1 as INTEGER and – + and * as OPERATOR .

SA recognizes the expression as “Numeric Expression”.

```
/* --- Test case [#2.6.2] : VARIABLES-INITIALIZE --- */



       [FLEX] Line=71, token=IDENTIFIER, value="x1"
```

```
        [FLEX] Line=71, token=OPERATOR, value="="

        [FLEX] Line=71, token=INTEGER, value="0"

        [FLEX] Line=71, token=DELIMITER, value=";"

[BISON] Line=70, expression="Assign value to variable"




[BISON] Line=71, expression="Assign value to variable"
```

Recognized as an assignment expression to a variable ( x 1) with a value of 0.

LA recognizes x 1 as an IDENTIFIER , = as an OPERATOR , 0 as an INTEGER , and ? as DELIMITER .

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=72, token=IDENTIFIER, value="x1"

        [FLEX] Line=72, token=SPECIAL, value=","

        [FLEX] Line=72, token=IDENTIFIER, value="x2"

        [FLEX] Line=72, token=OPERATOR, value="="

        [FLEX] Line=72, token=INTEGER, value="0"

        [FLEX] Line=72, token=SPECIAL, value=","

        [FLEX] Line=72, token=INTEGER, value="1"

        [FLEX] Line=72, token=DELIMITER, value=";"



[ BISON ] Line =72, expression ="Assign value to variable"
```

Recognized as a multiple assignment expression to variables ( x 1 , x 2 ) with values 0 and 1 .

LA recognizes x 1 and x 2 as IDENTIFIER , = as OPERATOR , 0 and 1 as INTEGER , and , and ? and DELIMITER respectively.

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=73, token=IDENTIFIER, value="x1"

        [FLEX] Line=73, token=SPECIAL, value=","
        [FLEX] Line=73, token=IDENTIFIER, value="list1"

        [FLEX] Line=73, token=OPERATOR, value="="

        [FLEX] Line=73, token=INTEGER, value="0"

        [FLEX] Line=73, token=SPECIAL, value=","

        [FLEX] Line=73, token=SPECIAL, value="["

        [FLEX] Line=73, token=STRING, value=""A""

        [FLEX] Line=73, token=SPECIAL, value=","

        [FLEX] Line=73, token=STRING, value=""B""

        [FLEX] Line=73, token=SPECIAL, value=","

        [FLEX] Line=73, token=STRING, value=""C""

        [FLEX] Line=73, token=SPECIAL, value="]"

        [FLEX] Line=73, token=DELIMITER, value=";"


[ BISON ] Line =73, expression ="Assign value to variable"
```

Recognized as a multiple assignment expression to variables ( x 1, list 1) with values 0 and a list.

LA recognizes x 1 and list 1 as IDENTIFIER , = as OPERATOR , 0 as INTEGER , [ and ] as SPECIAL , and " A ", " B ", " C " as STRING .

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=74, token=IDENTIFIER, value="x1"

        [FLEX] Line=74, token=SPECIAL, value=","

        [FLEX] Line=74, token=IDENTIFIER, value="list1"

        [FLEX] Line=74, token=SPECIAL, value=","

        [FLEX] Line=74, token=IDENTIFIER, value="string"

        [FLEX] Line=74, token=OPERATOR, value="="

        [FLEX] Line=74, token=INTEGER, value="0"

        [FLEX] Line=74, token=SPECIAL, value=","

        [FLEX] Line=74, token=SPECIAL, value="["
        [FLEX] Line=74, token=STRING, value=""A""

        [FLEX] Line=74, token=SPECIAL, value=","

        [FLEX] Line=74, token=STRING, value=""B""

        [FLEX] Line=74, token=SPECIAL, value=","

        [FLEX] Line=74, token=STRING, value=""C""

        [FLEX] Line=74, token=SPECIAL, value="]"

        [FLEX] Line=74, token=SPECIAL, value=","

        [FLEX] Line=74, token=STRING, value=""HELLO""

        [FLEX] Line=74, token=DELIMITER, value=";"



[ BISON ] Line =74, expression ="Assign value to variable"
```

Recognized as a multiple assignment expression to variables ( x 1, list 1, string ) with values 0, a list, and a string.

LA recognizes x 1 , list 1 , and string as IDENTIFIER , = as OPERATOR , 0 as INTEGER , [ and ] as SPECIAL , " A ", " B ", " C " as STRING , and " HELLO " as STRING .

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=75, token=IDENTIFIER, value="var"

        [FLEX] Line=75, token=OPERATOR, value="++"

        [FLEX] Line=75, token=DELIMITER, value=";"



[ BISON ] Line =75, expression ="Assign value to variable"
```

Recognized as an expression to increment the value of the variable ( var ) by 1.

LA recognizes var as IDENTIFIER , ++ as OPERATOR and ? as DELIMITER .

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=76, token=OPERATOR, value="++"

        [FLEX] Line=76, token=IDENTIFIER, value="var"

        [FLEX] Line=76, token=DELIMITER, value=";"
```

```


[ BISON ] Line =76, expression ="Assign value to variable"
```

Recognized as an expression to increment the value of the variable ( var ) by 1.

LA recognizes ++ as OPERATOR , var as IDENTIFIER and ? as DELIMITER .

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=77, token=OPERATOR, value="--"

        [FLEX] Line=77, token=IDENTIFIER, value="var"

        [FLEX] Line=77, token=DELIMITER, value=";"



[ BISON ] Line =77, expression ="Assign value to variable"
```

Recognized as an expression to decrement the value of the variable ( var ) by 1.

LA recognizes -- as an OPERATOR , var as an IDENTIFIER , and ? as DELIMITER .

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=78, token=IDENTIFIER, value="var"

        [FLEX] Line=78, token=OPERATOR, value="--"

        [FLEX] Line=78, token=DELIMITER, value=";"



[ BISON ] Line =78, expression ="Assign value to variable"
```

Recognized as an expression to decrement the value of the variable ( var ) by 1.

LA recognizes var as an IDENTIFIER , -- as an OPERATOR , and ? as DELIMITER .

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=79, token=IDENTIFIER, value="list1"

        [FLEX] Line=79, token=OPERATOR, value="+="

        [FLEX] Line=79, token=SPECIAL, value="["

        [FLEX] Line=79, token=INTEGER, value="1"
        [FLEX] Line=79, token=SPECIAL, value=","

        [FLEX] Line=79, token=INTEGER, value="2"

        [FLEX] Line=79, token=SPECIAL, value=","

        [FLEX] Line=79, token=INTEGER, value="3"

        [FLEX] Line=79, token=SPECIAL, value="]"

        [FLEX] Line=79, token=DELIMITER, value=";"



[ BISON ] Line =79, expression ="Assign value to variable"
```

It is recognized as an expression to add elements (1, 2, 3) to the list ( list 1 ).

LA recognizes list 1 as IDENTIFIER , += as OPERATOR , [ and ] as SPECIAL , and 1, 2 and 3 as INTEGER .

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=80, token=IDENTIFIER, value="x"

        [FLEX] Line=80, token=OPERATOR, value="-="

        [FLEX] Line=80, token=FLOAT, value="4e1"

        [FLEX] Line=80, token=DELIMITER, value=";"



[ BISON ] Line =80, expression ="Assign value to variable"
```

It is recognized as an expression of reduced value assignment of the variable ( x ) by 40.

LA recognizes x as an IDENTIFIER , -= as an OPERATOR , 4 e 1 as a FLOAT , and ? as DELIMITER .

The SA identifies the line as "Assign value to variable".

```
        [FLEX] Line=81, token=IDENTIFIER, value="y"

        [FLEX] Line=81, token=OPERATOR, value="*="

        [FLEX] Line=81, token=FLOAT, value="6.7"

        [FLEX] Line=81, token=DELIMITER, value=";"
```

```
[ BISON ] Line =81, expression ="Assign value to variable"
```

It is recognized as an expression assigning the value of the variable ( y ) multiplied by 6.7

LA recognizes y as an IDENTIFIER , *= as an OPERATOR , 6.7 as a FLOAT , and ? as DELIMITER .

The SA identifies the line as "Assign value to variable".

```
       [FLEX] Line=82, token=IDENTIFIER, value="c"

       [FLEX] Line=82, token=OPERATOR, value="/="

       [FLEX] Line=82, token=IDENTIFIER, value="b"

       [FLEX] Line=82, token=DELIMITER, value=";"


[ BISON ] Line =82, expression ="Assign value to variable"
```

It is recognized as an expression assigning value to variable ( c ) divided by variable ( b ).

LA recognizes c and b as IDENTIFIER , /= as OPERATOR and ? as DELIMITER .

The SA identifies the line as "Assign value to variable".

```
/* --- Test case [#2.6.3] : ARRAY-MERGE --- */


       [FLEX] Line=86, token=SPECIAL, value="["

       [FLEX] Line=86, token=INTEGER, value="1"

       [FLEX] Line=86, token=SPECIAL, value=","

       [FLEX] Line=86, token=INTEGER, value="2"

       [FLEX] Line=86, token=SPECIAL, value=","

       [FLEX] Line=86, token=INTEGER, value="3"

       [FLEX] Line=86, token=SPECIAL, value="]"

       [FLEX] Line=86, token=OPERATOR, value="+"

       [FLEX] Line=86, token=SPECIAL, value="["

       [FLEX] Line=86, token=INTEGER, value="4"
       [FLEX] Line=86, token=SPECIAL, value=","

       [FLEX] Line=86, token=INTEGER, value="5"

       [FLEX] Line=86, token=SPECIAL, value=","

       [FLEX] Line=86, token=INTEGER, value="6"

       [FLEX] Line=86, token=SPECIAL, value="]"


[BISON] Line=86, expression="Table Concatenation"


[BISON] Line=86, expression="Table Concatenation"
```

Recognized as an array merge expression with the contents of arrays [1, 2, 3] and [4, 5, 6].

LA recognizes [ and ] as SPECIAL and 1, 2, 3, 4, 5, 6 as INTEGER .

SA identifies the line as "Merge Tables".

```
       [FLEX] Line=87, token=SPECIAL, value="["

       [FLEX] Line=87, token=INTEGER, value="1"

       [FLEX] Line=87, token=SPECIAL, value=","

       [FLEX] Line=87, token=INTEGER, value="3"

       [FLEX] Line=87, token=SPECIAL, value="]"

       [FLEX] Line=87, token=OPERATOR, value="+"

       [FLEX] Line=87, token=SPECIAL, value="["

       [FLEX] Line=87, token=INTEGER, value="5"

       [FLEX] Line=87, token=SPECIAL, value=","

       [FLEX] Line=87, token=INTEGER, value="7"

       [FLEX] Line=87, token=SPECIAL, value=","

       [FLEX] Line=87, token=INTEGER, value="9"

       [FLEX] Line=87, token=SPECIAL, value=","

       [FLEX] Line=87, token=INTEGER, value="11"

       [FLEX] Line=87, token=SPECIAL, value="]"
```

```

[BISON] Line=87, expression="Table Concatenation"
```

Recognized as an array merge expression with the contents of arrays [1, 3] and [5, 7, 9, 11].

Flex recognizes [ and ] as SPECIAL and 1, 3, 5, 7, 9, 11 as INTEGER .

Bison identifies the line as "Merge Tables".

```
        [FLEX] Line=88, token=SPECIAL, value="["

        [FLEX] Line=88, token=STRING, value=""What""

        [FLEX] Line=88, token=SPECIAL, value=","

        [FLEX] Line=88, token=STRING, value=""a""

        [FLEX] Line=88, token=SPECIAL, value="]"

        [FLEX] Line=88, token=OPERATOR, value="+"

        [FLEX] Line=88, token=SPECIAL, value="["

        [FLEX] Line=88, token=STRING, value=""Wonderful""

        [FLEX] Line=88, token=SPECIAL, value=","

        [FLEX] Line=88, token=STRING, value=""World""

        [FLEX] Line=88, token=SPECIAL, value="]"


[BISON] Line=88, expression="Table Concatenation"
```

It is recognized as an array merge expression with the contents of the arrays [" What ", " a "] and [" Wonderful ", " World "].

LA recognizes [ and ] as SPECIAL and " What ", " a ", " Wonderful ", " World " as STRING .

SA identifies the line as "Merge Tables".

```
        [FLEX] Line=89, token=SPECIAL, value="["

        [FLEX] Line=89, token=FLOAT, value="4.5"

        [FLEX] Line=89, token=SPECIAL, value=","

        [FLEX] Line=89, token=FLOAT, value="5.6"
```

```
       [FLEX] Line=89, token=SPECIAL, value=","

       [FLEX] Line=89, token=FLOAT, value="1.1"

       [FLEX] Line=89, token=SPECIAL, value="]"

       [FLEX] Line=89, token=OPERATOR, value="+"

       [FLEX] Line=89, token=SPECIAL, value="["

       [FLEX] Line=89, token=FLOAT, value="0e0"

       [FLEX] Line=89, token=SPECIAL, value=","

       [FLEX] Line=89, token=FLOAT, value="4.5E-1"

       [FLEX] Line=89, token=SPECIAL, value=","

       [FLEX] Line=89, token=FLOAT, value="4E0"

       [FLEX] Line=89, token=SPECIAL, value="]"



[BISON] Line=89, expression="Table Concatenation"
```

It is recognized as an array merge expression with the contents of arrays [4.5, 5.6, 1.1] and [0 e 0, 4.5 E -1, 4 E 0].

LA recognizes [ and ] as SPECIAL , 4.5, 5.6, 1.1, 0 e 0, 4.5 E -1 and 4 E 0 as FLOAT .

SA identifies the line as "Merge Tables".

```
/* --- Test case [#2.6.4] : COMPARISONS --- */



       [FLEX] Line=93, token=IDENTIFIER, value="x1"

       [FLEX] Line=93, token=OPERATOR, value=">"

       [FLEX] Line=93, token=IDENTIFIER, value="x2"



[BISON] Line=93, expression="Compare"



[BISON] Line=93, expression="Compare"
```

It is recognized as a comparison expression where the variable x 1 is greater than the variable x 2 .

LA recognizes x 1 and x 2 as IDENTIFIER and > as OPERATOR .

SA identifies the row as "Compare"

```
        [FLEX] Line=94, token=IDENTIFIER, value="a"

        [FLEX] Line=94, token=OPERATOR, value="<"

        [FLEX] Line=94, token=IDENTIFIER, value="b"



[ BISON ] Line =94, expression ="Compare"
```

It is recognized as a comparison expression where the variable a is less than the variable b .

LA recognizes a and b as IDENTIFIER and < as OPERATOR .

The SA identifies the row as "Compare".

```
        [FLEX] Line=95, token=IDENTIFIER, value="myvar"

        [FLEX] Line=95, token=OPERATOR, value=">="

        [FLEX] Line=95, token=INTEGER, value="52"



[ BISON ] Line =95, expression ="Compare"
```

Recognized as a comparison expression where the variable myvar is greater than or equal to the number 52.

LA recognizes myvar as an IDENTIFIER , >= as an OPERATOR , and 52 as an INTEGER .

The SA identifies the line as "Compare".

```
        [FLEX] Line=96, token=IDENTIFIER, value="var1"

        [FLEX] Line=96, token=OPERATOR, value="<="

        [FLEX] Line=96, token=IDENTIFIER, value="var2"



[ BISON ] Line =96, expression ="Compare"
```

It is recognized as a comparison expression where the variable var 1 is less than or equal to the variable var 2.

LA recognizes var 1 and var 2 as IDENTIFIER and <= as OPERATOR .

The SA identifies the row as "Compare".

```
        [FLEX] Line=97, token=IDENTIFIER, value="isEven"

        [FLEX] Line=97, token=OPERATOR, value="=="

        [FLEX] Line=97, token=IDENTIFIER, value="isOdd"



[ BISON ] Line =97, expression ="Compare"
```

Recognized as a comparison expression where the variable isEven is equal to the variable isOdd .

LA recognizes isEven and isOdd as IDENTIFIER and == as OPERATOR .

The SA identifies the line as "Compare".

```
        [FLEX] Line=98, token=IDENTIFIER, value="a"

        [FLEX] Line=98, token=OPERATOR, value="!="

        [FLEX] Line=98, token=IDENTIFIER, value="C"



[ BISON ] Line =98, expression ="Compare"
```

It is recognized as a comparison expression where the variable a is different from the variable C .

LA recognizes a and C as IDENTIFIER and != as OPERATOR .

The SA identifies the line as "Compare".

```
        [FLEX] Line=99, token=IDENTIFIER, value="isTrue1"

        [FLEX] Line=99, token=OPERATOR, value="||"

        [FLEX] Line=99, token=IDENTIFIER, value="isTrue2"



[ BISON ] Line =99, expression ="Compare"
```

Recognized as a comparison expression where either isTrue 1 or isTrue 2 is true.

LA recognizes isTrue 1 and isTrue 2 as IDENTIFIER and || as an OPERATOR .

SA identifies the row as "Compare"

```
       [FLEX] Line=100, token=IDENTIFIER, value="isFalse1"

       [FLEX] Line=100, token=OPERATOR, value="&&"

       [FLEX] Line=100, token=IDENTIFIER, value="isFalse2"


[ BISON ] Line =100, expression ="Compare"
```

Recognized as a comparison expression where both isFalse 1 and isFalse 2 are true.

LA recognizes isFalse 1 and isFalse 2 as IDENTIFIER and && as OPERATOR .

SA identifies the row as "Compare"

```
       [FLEX] Line=101, token=OPERATOR, value="!"

       [FLEX] Line=101, token=IDENTIFIER, value="not"


[ BISON ] Line =101, expression ="Compare"
```

It is recognized as a comparison expression where the variable not is true (logical negation).

LA recognizes the ! as OPERATOR and not as IDENTIFIER .

The SA identifies the line as "Compare".

## 2. 7 Compound Statements

### Input File ( input . txt )

```
/* --- Test case [#2.7.1] : IF-STATEMENTS --- */


if (var == 100) print("Value of expression is 100");

if (x < y < z) { print(x); print(y); print(z); }
if (x == 0)

{

int i;

i = 1;

if (i > 1)

print("i is greater than 1");

}



/* --- Test case [#2.7.2] : WHILE-STATEMENTS --- */

while (i < 10)

{

print("i is: ", i);

}



/* --- Test case [#2.7.3] : FOR-STATEMENTS --- */

for (i = 0; i < 10; i++)

{

print("i = ", i);

}
```

## File Output (output.txt)

```
/* --- Test case [#2.7.1] : IF-STATEMENTS --- */


      [FLEX] Line=106, token=KEYWORD, value="if"
```

```
       [FLEX] Line=106, token=SPECIAL, value="("

       [FLEX] Line=106, token=IDENTIFIER, value="var"

       [FLEX] Line=106, token=OPERATOR, value="=="

       [FLEX] Line=106, token=INTEGER, value="100"

       [FLEX] Line=106, token=SPECIAL, value=")"

       [FLEX] Line=106, token=FUNCTION, value="print"

       [FLEX] Line=106, token=SPECIAL, value="("

       [FLEX] Line=106, token=STRING, value=""Value of expression is 100""

       [FLEX] Line=106, token=SPECIAL, value=")"

       [FLEX] Line=106, token=DELIMITER, value=";"



[BISON] Line=106, expression="If Statement"
```

The line contains an if statement with a value comparison and a function call.

```
       [FLEX] Line=107, token=KEYWORD, value="if"

       [FLEX] Line=107, token=SPECIAL, value="("

       [FLEX] Line=107, token=IDENTIFIER, value="x"

       [FLEX] Line=107, token=OPERATOR, value="<"

       [FLEX] Line=107, token=IDENTIFIER, value="y"

       [FLEX] Line=107, token=OPERATOR, value="<"

       [FLEX] Line=107, token=IDENTIFIER, value="z"

       [FLEX] Line=107, token=SPECIAL, value=")"

       [FLEX] Line=107, token=SPECIAL, value="{"

       [FLEX] Line=107, token=FUNCTION, value="print"

       [FLEX] Line=107, token=SPECIAL, value="("

       [FLEX] Line=107, token=IDENTIFIER, value="x"

       [FLEX] Line=107, token=SPECIAL, value=")"

       [FLEX] Line=107, token=DELIMITER, value=";"
       [FLEX] Line=107, token=FUNCTION, value="print"

       [FLEX] Line=107, token=SPECIAL, value="("

       [FLEX] Line=107, token=IDENTIFIER, value="y"

       [FLEX] Line=107, token=SPECIAL, value=")"

       [FLEX] Line=107, token=DELIMITER, value=";"

[BISON] Line=106, expression="Function call"


       [FLEX] Line=107, token=FUNCTION, value="print"

       [FLEX] Line=107, token=SPECIAL, value="("

       [FLEX] Line=107, token=IDENTIFIER, value="z"

       [FLEX] Line=107, token=SPECIAL, value=")"

       [FLEX] Line=107, token=DELIMITER, value=";"

[BISON] Line=106, expression="Function call"


       [FLEX] Line=107, token=SPECIAL, value="}"


[BISON] Line=107, expression="If Statement"
```

The line contains an if statement with a complex logical expression and multiple function calls.

```
       [FLEX] Line=109, token=KEYWORD, value="if"

       [FLEX] Line=109, token=SPECIAL, value="("

       [FLEX] Line=109, token=IDENTIFIER, value="x"

       [FLEX] Line=109, token=OPERATOR, value="=="

       [FLEX] Line=109, token=INTEGER, value="0"

       [FLEX] Line=109, token=SPECIAL, value=")"


[BISON] Line=109, expression="If Statement"
```

```
      [FLEX] Line=110, token=SPECIAL, value="{"


      [FLEX] Line=111, token=KEYWORD, value="int"

      [FLEX] Line=111, token=IDENTIFIER, value="i"

      [FLEX] Line=111, token=DELIMITER, value=";"

[BISON] Line=110, expression="Variable Declaration"




      [FLEX] Line=112, token=IDENTIFIER, value="i"

      [FLEX] Line=112, token=OPERATOR, value="="

      [FLEX] Line=112, token=INTEGER, value="1"

      [FLEX] Line=112, token=DELIMITER, value=";"

[BISON] Line=111, expression="Assign value to variable"




      [FLEX] Line=113, token=KEYWORD, value="if"

      [FLEX] Line=113, token=SPECIAL, value="("

      [FLEX] Line=113, token=IDENTIFIER, value="i"

      [FLEX] Line=113, token=OPERATOR, value=">"

      [FLEX] Line=113, token=INTEGER, value="1"

      [FLEX] Line=113, token=SPECIAL, value=")"


[BISON] Line=113, expression="If Statement"


      [FLEX] Line=114, token=FUNCTION, value="print"

      [FLEX] Line=114, token=SPECIAL, value="("

      [FLEX] Line=114, token=STRING, value=""i is greater than 1""
```

```
        [FLEX] Line=114, token=SPECIAL, value=")"

        [FLEX] Line=114, token=DELIMITER, value=";"

[BISON] Line=113, expression="Function call"




        [FLEX] Line=115, token=SPECIAL, value="}"

[BISON] Line=114, expression="Compound Statements"




[BISON] Line=115, expression="Compound Statements"
```

The line contains an if statement that contains other statements and a nested if -statement.

```
/* --- Test case [#2.7.2] : WHILE-STATEMENTS --- */

        [FLEX] Line=118, token=KEYWORD, value="while"

        [FLEX] Line=118, token=SPECIAL, value="("

        [FLEX] Line=118, token=IDENTIFIER, value="i"

        [FLEX] Line=118, token=OPERATOR, value="<"

        [FLEX] Line=118, token=INTEGER, value="10"

        [FLEX] Line=118, token=SPECIAL, value=")"


        [FLEX] Line=119, token=SPECIAL, value="{"


        [FLEX] Line=120, token=FUNCTION, value="print"

        [FLEX] Line=120, token=SPECIAL, value="("

        [FLEX] Line=120, token=STRING, value=""i is: ""

        [FLEX] Line=120, token=SPECIAL, value=","

        [FLEX] Line=120, token=IDENTIFIER, value="i"

        [FLEX] Line=120, token=SPECIAL, value=")"
```

```
      [FLEX] Line=120, token=DELIMITER, value=";"

[BISON] Line=119, expression="Function call"




      [FLEX] Line=121, token=SPECIAL, value="}"

[BISON] Line=120, expression="Compound Statements"




[BISON] Line=121, expression="Compound Statements"
```

The line contains a while statement with an embedded code block that executes a function call .

```
/* --- Test case [#2.7.3] : FOR-STATEMENTS --- */

      [FLEX] Line=124, token=KEYWORD, value="for"

      [FLEX] Line=124, token=SPECIAL, value="("

      [FLEX] Line=124, token=IDENTIFIER, value="i"

      [FLEX] Line=124, token=OPERATOR, value="="

      [FLEX] Line=124, token=INTEGER, value="0"

      [FLEX] Line=124, token=DELIMITER, value=";"

      [FLEX] Line=124, token=IDENTIFIER, value="i"

      [FLEX] Line=124, token=OPERATOR, value="<"

      [FLEX] Line=124, token=INTEGER, value="10"

      [FLEX] Line=124, token=DELIMITER, value=";"

      [FLEX] Line=124, token=IDENTIFIER, value="i"

      [FLEX] Line=124, token=OPERATOR, value="++"

      [FLEX] Line=124, token=SPECIAL, value=")"


      [FLEX] Line=125, token=SPECIAL, value="{"
      [FLEX] Line=126, token=FUNCTION, value="print"

      [FLEX] Line=126, token=SPECIAL, value="("

      [FLEX] Line=126, token=STRING, value=""i = ""

      [FLEX] Line=126, token=SPECIAL, value=","

      [FLEX] Line=126, token=IDENTIFIER, value="i"

      [FLEX] Line=126, token=SPECIAL, value=")"

      [FLEX] Line=126, token=DELIMITER, value=";"

[BISON] Line=125, expression="Function call"




      [FLEX] Line=127, token=SPECIAL, value="}"

[BISON] Line=126, expression="Compound Statements"




[BISON] Line=127, expression="Compound Statements"
```

The line contains a for statement with an embedded block of code that executes a function call.

- LA and SA ( FLEX and BISON ) seem to work correctly for parsing if , while and for statements.
- Expressions are adequately recognized and categorized, and function calls are correctly identified and described.
- Complex blocks of code (such as nested - if statements and statement blocks inside loops) are parsed and rendered accurately.

The parsing process through FLEX and BISON confirms the correct operation of the parser for the given test cases, with tokens and expressions being recognized and rendered as expected.

## 2.8 Syntax Warning Errors ( Warnings )

### Input File ( input . txt )

```
/* --- Test case [#2.8.1] : DOUBLE COMPARE --- */


x >> y

x << y


/* --- Test case [#2.8.2] : DOUBLE PARENTHESIS --- */


if ((x == 0))

{

}

while ((y == 0))

{

}


/* --- Test case [#2.8.3] : RETURN TYPE IN FUNCTIONS --- */


func int main()

{

}

func float avg(int parA, int parB)

{

calc_avg(parA, parB);

}

func double pi()

{

print("3.14");

}
```

```
/* --- Test case [#2.8.4] : INVALID INPUT IN MERGE ARRAYS --- */

[3, 4, 6] # + [1, 5, 6]

[1,2,8] + $ [7,9,1]


/* --- Test case [#2.8.5] : DOUBLE USE OF KEYWORD IN DELCARATION --- */

int int x;

float float y;

double double far?

short short imp?
```

## File Output (output.txt)

```
/* --- Test case [#2.8.1] : DOUBLE COMPARE --- */

       [FLEX] Line=132, token=IDENTIFIER, value="x"

       [FLEX] Line=132, token=OPERATOR, value=">"

       [FLEX] Line=132, token=OPERATOR, value=">"

       [FLEX] Line=132, token=IDENTIFIER, value="y"

## Warning ## -> Double > detected at Line=131


[BISON] Line=132, expression="Compare"
```

```
       [FLEX] Line=133, token=IDENTIFIER, value="x"

       [FLEX] Line=133, token=OPERATOR, value="<"
       [FLEX] Line=133, token=OPERATOR, value="<"

       [FLEX] Line=133, token=IDENTIFIER, value="y"

## Warning ## -> Double < detected at Line=132



[ BISON ] Line =133, expression ="Compare"
```

Accepts the example, but notes that there is a warning about double comparison symbols ( >> and << ). Lines are recognized as comparison expressions.

```
/* --- Test case [#2.8.2] : DOUBLE PARENTHESIS --- */



       [FLEX] Line=137, token=KEYWORD, value="if"

       [FLEX] Line=137, token=SPECIAL, value="("

       [FLEX] Line=137, token=SPECIAL, value="("

       [FLEX] Line=137, token=IDENTIFIER, value="x"

       [FLEX] Line=137, token=OPERATOR, value="=="

       [FLEX] Line=137, token=INTEGER, value="0"

       [FLEX] Line=137, token=SPECIAL, value=")"

       [FLEX] Line=137, token=SPECIAL, value=")"

## Warning ## -> Double parethensis detected at Line=137



[BISON] Line=137, expression="If Statement"



       [FLEX] Line=138, token=SPECIAL, value="{"



       [FLEX] Line=139, token=SPECIAL, value="}"

[BISON] Line=138, expression="Compound Statements"
```

```
[BISON] Line=139, expression="Compound Statements"
```

```
        [FLEX] Line=140, token=KEYWORD, value="while"

        [FLEX] Line=140, token=SPECIAL, value="("

        [FLEX] Line=140, token=SPECIAL, value="("

        [FLEX] Line=140, token=IDENTIFIER, value="y"

        [FLEX] Line=140, token=OPERATOR, value="=="

        [FLEX] Line=140, token=INTEGER, value="0"

        [FLEX] Line=140, token=SPECIAL, value=")"

        [FLEX] Line=140, token=SPECIAL, value=")"

## Warning ## -> Double parethensis detected at Line=140



        [FLEX] Line=141, token=SPECIAL, value="{"



        [FLEX] Line=142, token=SPECIAL, value="}"

[BISON] Line=141, expression="Compound Statements"





[BISON] Line=142, expression="Compound Statements"
```

Accepts the example, but notes warnings about double parentheses. The lines are recognized as statements if and while.

```
/* --- Test case [#2.8.3] : RETURN TYPE IN FUNCTIONS --- */



        [FLEX] Line=146, token=KEYWORD, value="func"

        [FLEX] Line=146, token=KEYWORD, value="int"

        [FLEX] Line=146, token=IDENTIFIER, value="main"

## Warning ## -> Return type unnecessary at Line=146
```

```
       [FLEX] Line=146, token=SPECIAL, value="("

       [FLEX] Line=146, token=SPECIAL, value=")"



       [FLEX] Line=147, token=SPECIAL, value="{"



       [FLEX] Line=148, token=SPECIAL, value="}"

[BISON] Line=147, expression="Declaration of user functions"





[BISON] Line=148, expression="Declaration of user functions"
```

```
       [FLEX] Line=149, token=KEYWORD, value="func"

       [FLEX] Line=149, token=KEYWORD, value="float"

       [FLEX] Line=149, token=IDENTIFIER, value="avg"

## Warning ## -> Return type unnecessary at Line=149

       [FLEX] Line=149, token=SPECIAL, value="("

       [FLEX] Line=149, token=KEYWORD, value="int"

       [FLEX] Line=149, token=IDENTIFIER, value="parA"

       [FLEX] Line=149, token=SPECIAL, value=","

       [FLEX] Line=149, token=KEYWORD, value="int"

       [FLEX] Line=149, token=IDENTIFIER, value="parB"

       [FLEX] Line=149, token=SPECIAL, value=")"



       [FLEX] Line=150, token=SPECIAL, value="{"



       [FLEX] Line=151, token=IDENTIFIER, value="calc_avg"

       [FLEX] Line=151, token=SPECIAL, value="("
       [FLEX] Line=151, token=IDENTIFIER, value="parA"

       [FLEX] Line=151, token=SPECIAL, value=","

       [FLEX] Line=151, token=IDENTIFIER, value="parB"

       [FLEX] Line=151, token=SPECIAL, value=")"

       [FLEX] Line=151, token=DELIMITER, value=";"

[BISON] Line=150, expression="Function call"




       [FLEX] Line=152, token=SPECIAL, value="}"



[BISON] Line=152, expression="Declaration of user functions"
```

```
       [FLEX] Line=153, token=KEYWORD, value="func"

       [FLEX] Line=153, token=KEYWORD, value="double"

       [FLEX] Line=153, token=IDENTIFIER, value="pi"

## Warning ## -> Return type unnecessary at Line=153

       [FLEX] Line=153, token=SPECIAL, value="("

       [FLEX] Line=153, token=SPECIAL, value=")"



       [FLEX] Line=154, token=SPECIAL, value="{"



       [FLEX] Line=155, token=FUNCTION, value="print"

       [FLEX] Line=155, token=SPECIAL, value="("

       [FLEX] Line=155, token=STRING, value=""3.14""

       [FLEX] Line=155, token=SPECIAL, value=")"

       [FLEX] Line=155, token=DELIMITER, value=";"

[BISON] Line=154, expression="Function call"
```

```
       [FLEX] Line=156, token=SPECIAL, value="}"


[ BISON ] Line =156, expression ="Declaration of user functions"
```

Accepts function declarations, but notes warnings that return types are not necessary. They are recognized as user function declarations and function calls.

```
/* --- Test case [#2.8.4] : INVALID INPUT IN MERGE ARRAYS --- */


       [FLEX] Line=160, token=SPECIAL, value="["

       [FLEX] Line=160, token=INTEGER, value="3"

       [FLEX] Line=160, token=SPECIAL, value=","

       [FLEX] Line=160, token=INTEGER, value="4"

       [FLEX] Line=160, token=SPECIAL, value=","

       [FLEX] Line=160, token=INTEGER, value="6"

       [FLEX] Line=160, token=SPECIAL, value="]"

!! Token error!! -> at Line=160

       1 character(s) ignored so far

       [FLEX] Line=160, token=OPERATOR, value="+"

       [FLEX] Line=160, token=SPECIAL, value="["

       [FLEX] Line=160, token=INTEGER, value="1"

       [FLEX] Line=160, token=SPECIAL, value=","

       [FLEX] Line=160, token=INTEGER, value="5"

       [FLEX] Line=160, token=SPECIAL, value=","

       [FLEX] Line=160, token=INTEGER, value="6"

       [FLEX] Line=160, token=SPECIAL, value="]"


## Warning ## -> Invalid character in array merge detected at Line=161
```

```
[BISON] Line=160, expression="Table Concatenation"



[BISON] Line=160, expression="Table Concatenation"
```

```
      [FLEX] Line=161, token=SPECIAL, value="["

      [FLEX] Line=161, token=INTEGER, value="1"

      [FLEX] Line=161, token=SPECIAL, value=","

      [FLEX] Line=161, token=INTEGER, value="2"

      [FLEX] Line=161, token=SPECIAL, value=","

      [FLEX] Line=161, token=INTEGER, value="8"

      [FLEX] Line=161, token=SPECIAL, value="]"

      [FLEX] Line=161, token=OPERATOR, value="+"

!! Token error!! -> at Line=161

      2 character(s) ignored so far

      [FLEX] Line=161, token=SPECIAL, value="["

      [FLEX] Line=161, token=INTEGER, value="7"

      [FLEX] Line=161, token=SPECIAL, value=","

      [FLEX] Line=161, token=INTEGER, value="9"

      [FLEX] Line=161, token=SPECIAL, value=","

      [FLEX] Line=161, token=INTEGER, value="1"

      [FLEX] Line=161, token=SPECIAL, value="]"



## Warning ## -> Invalid character in array merge detected at Line=162

[BISON] Line=161, expression="Table Concatenation"
```

It accepts the statements, but throws ignored character errors and warnings about invalid characters in the table join. They are recognized as expressions amalgamation tables .

```
/* --- Test case [#2.8.5] : DOUBLE USE OF KEYWORD IN DELCARATION --- */
```

```


       [FLEX] Line=165, token=KEYWORD, value="int"

       [FLEX] Line=165, token=KEYWORD, value="int"

## Warning ## -> Double int detected at Line=165

       [FLEX] Line=165, token=IDENTIFIER, value="x"

       [FLEX] Line=165, token=DELIMITER, value=";"

[BISON] Line=164, expression="Variable declaration"




[BISON] Line=165, expression="Variable declaration"
```

```
       [FLEX] Line=166, token=KEYWORD, value="float"

       [FLEX] Line=166, token=KEYWORD, value="float"

## Warning ## -> Double float detected at Line=166

       [FLEX] Line=166, token=IDENTIFIER, value="y"

       [FLEX] Line=166, token=DELIMITER, value=";"


[BISON] Line=166, expression="Variable declaration"
```

```
       [FLEX] Line=167, token=KEYWORD, value="double"

       [FLEX] Line=167, token=KEYWORD, value="double"

## Warning ## -> Double double detected at Line=167

       [FLEX] Line=167, token=IDENTIFIER, value="far"

       [FLEX] Line=167, token=DELIMITER, value=";"


[BISON] Line=167, expression="Variable declaration"
```

```
       [FLEX] Line=168, token=KEYWORD, value="short"

       [FLEX] Line=168, token=KEYWORD, value="short"

## Warning ## -> Double short detected at Line=168

       [FLEX] Line=168, token=IDENTIFIER, value="imp"

       [FLEX] Line=168, token=DELIMITER, value=";"



[BISON] Line=168, expression="Variable declaration"
```

It accepts the declarations, but notes warnings about duplicate keywords . They are recognized as variable declarations.

## 2.9 Syntax Errors

### Input File ( input . txt )

```
intx a?

pin1 = [1, 3, "Hello", 4, 5];

scan(&x);

len(1variable);

cmp(x, 5);

printf("Avg is %f", avg);
```

```
func myFunc(long x, short y) {

       int c;

       c = x / y?

       return c;

}

myFunc(x, y, 10);

5++?

x -= "Hello"?

y *= [h, e, l, l, o];

50 = x?

list1, list2 = [x, b, c], 5;

x1 === x2;

array += [1,3,4];

[1,3,5] ++ [1,2];

if var == 100:

print("var", "is", var);

while(1)

{

print("Hello World\n");

}

for(??)

{

print("Hello World\n");

}
```

## File Output (output.txt)

```
       [FLEX] Line=172, token=IDENTIFIER, value="intx"

       [FLEX] Line=172, token=IDENTIFIER, value="a"

[BISON] Line=171, expression="Numeric expression"



       [FLEX] Line=172, token=DELIMITER, value=";"

[BISON] Line=171, expression="Numeric expression"



!! syntax error !! -> at Line = 172
```

It tries to declare an integer named a , however there is a syntax error due to x instead of int .

```
       [FLEX] Line=173, token=IDENTIFIER, value="pin1"

       [FLEX] Line=173, token=OPERATOR, value="="

       [FLEX] Line=173, token=SPECIAL, value="["

       [FLEX] Line=173, token=INTEGER, value="1"

       [FLEX] Line=173, token=SPECIAL, value=","

       [FLEX] Line=173, token=INTEGER, value="3"

       [FLEX] Line=173, token=SPECIAL, value=","

       [FLEX] Line=173, token=STRING, value=""Hello""

!! syntax error!! -> at Line=173

       [FLEX] Line=173, token=SPECIAL, value=","

       [FLEX] Line=173, token=INTEGER, value="4"

       [FLEX] Line=173, token=SPECIAL, value=","

       [FLEX] Line=173, token=INTEGER, value="5"

       [FLEX] Line=173, token=SPECIAL, value="]"

       [FLEX] Line=173, token=DELIMITER, value=";"
```

It tries to assign a list to the variable pin 1 , but " Hello " is not an integer. The contents of an array in Uni - C are strictly of the same data type.

```
       [FLEX] Line=174, token=FUNCTION, value="scan"
```

```
      [FLEX] Line=174, token=SPECIAL, value="("

      [FLEX] Line=174, token=OPERATOR, value="&"

!! syntax error!! -> at Line=174

      [FLEX] Line=174, token=IDENTIFIER, value="x"

      [FLEX] Line=174, token=SPECIAL, value=")"

      [FLEX] Line=174, token=DELIMITER, value=";"
```

It tries to scan an integer from the user and store it in the variable x . In Uni - C variables are not read with the & address operator.

```
      [FLEX] Line=175, token=FUNCTION, value="len"

      [FLEX] Line=175, token=SPECIAL, value="("

      [FLEX] Line=175, token=INTEGER, value="1"

!! syntax error!! -> at Line=175

      [FLEX] Line=175, token=IDENTIFIER, value="variable"

      [FLEX] Line=175, token=SPECIAL, value=")"

      [FLEX] Line=175, token=DELIMITER, value=";"
```

It tries to call a function named len with the parameter 1 variable , but the syntax looks correct. The reason the parser throws a syntax error is in the recognition of the identifier "1 variable ". LA does not recognize identifiers that start with a digit and for this reason it returns 2 tokens , an integer (1) and an identifier ( variable ). Thus, the SA looks for a rule in the parameters of len () that has an integer and an identifier, where it does not find and therefore considers the syntax incorrect.

```
      [FLEX] Line=176, token=FUNCTION, value="cmp"

      [FLEX] Line=176, token=SPECIAL, value="("

      [FLEX] Line=176, token=IDENTIFIER, value="x"

      [FLEX] Line=176, token=SPECIAL, value=","

      [FLEX] Line=176, token=INTEGER, value="5"

!! syntax error!! -> at Line=176

      [FLEX] Line=176, token=SPECIAL, value=")"

      [FLEX] Line=176, token=DELIMITER, value=";"
```

It tries to compare x to 5, however, cmp compares two strings if they are equal, not numbers

```
      [FLEX] Line=177, token=IDENTIFIER, value="printf"

      [FLEX] Line=177, token=SPECIAL, value="("

      [FLEX] Line=177, token=STRING, value=""Avg is %f""

      [FLEX] Line=177, token=SPECIAL, value=","

      [FLEX] Line=177, token=IDENTIFIER, value="avg"

      [FLEX] Line=177, token=SPECIAL, value=")"

      [FLEX] Line=177, token=DELIMITER, value=";"



[BISON] Line=177, expression="Function call"
```

There is a simple built-in “ print ” function, however the user could define his own function named “ printf ”. That's why SA doesn't make a typo in the name of " print "

```
      [FLEX] Line=178, token=KEYWORD, value="func"

      [FLEX] Line=178, token=IDENTIFIER, value="myFunc"

      [FLEX] Line=178, token=SPECIAL, value="("

      [FLEX] Line=178, token=KEYWORD, value="long"

      [FLEX] Line=178, token=IDENTIFIER, value="x"

      [FLEX] Line=178, token=SPECIAL, value=","

      [FLEX] Line=178, token=KEYWORD, value="short"

      [FLEX] Line=178, token=IDENTIFIER, value="y"

      [FLEX] Line=178, token=SPECIAL, value=")"

      [FLEX] Line=178, token=SPECIAL, value="{"

!! syntax error!! -> at Line=178



      [FLEX] Line=179, token=KEYWORD, value="int"

      [FLEX] Line=179, token=IDENTIFIER, value="c"

      [FLEX] Line=179, token=DELIMITER, value=";"
[BISON] Line=179, expression="Variable Declaration"



      [FLEX] Line=180, token=IDENTIFIER, value="c"

      [FLEX] Line=180, token=OPERATOR, value="="

      [FLEX] Line=180, token=IDENTIFIER, value="x"

      [FLEX] Line=180, token=OPERATOR, value="/"

      [FLEX] Line=180, token=IDENTIFIER, value="y"

      [FLEX] Line=180, token=DELIMITER, value=";"



[BISON] Line=180, expression="Assign value to variable"



      [FLEX] Line=181, token=KEYWORD, value="return"

!! syntax error!! -> at Line=181

      [FLEX] Line=181, token=IDENTIFIER, value="c"

      [FLEX] Line=181, token=DELIMITER, value=";"



      [FLEX] Line=182, token=SPECIAL, value="}"

!! syntax error !! -> at Line = 182
```

It defines a function named myFunc that accepts an integer and a short , but the syntax is incorrect because Uni - C does not support function body hooks being opened on the same line as the function parameters. Also, Uni - C does not support return values using the " return " keyword . The SA continues to recognize syntactic expressions after the syntax errors, and tries to find a rule that contains a recognizer ( c ), a delimiter (?), and a hook (}). Obviously, it doesn't find it, so we have another syntax error.

```
      [FLEX] Line=183, token=IDENTIFIER, value="myFunc"

      [FLEX] Line=183, token=SPECIAL, value="("

      [FLEX] Line=183, token=IDENTIFIER, value="x"

      [FLEX] Line=183, token=SPECIAL, value=","

      [FLEX] Line=183, token=IDENTIFIER, value="y"
```

```
       [FLEX] Line=183, token=SPECIAL, value=","

       [FLEX] Line=183, token=INTEGER, value="10"

       [FLEX] Line=183, token=SPECIAL, value=")"

       [FLEX] Line=183, token=DELIMITER, value=";"



[BISON] Line=183, expression="Function call"
```

It calls the function myFunc with three parameters. The syntax is correct, but semantically you would expect it not to be, since a function myFunc is declared with 2 parameters, while we call it with 3.

```
       [FLEX] Line=184, token=INTEGER, value="5"

       [FLEX] Line=184, token=OPERATOR, value="++"

       [FLEX] Line=184, token=DELIMITER, value=";"

!! syntax error !! -> at Line = 184
```

It tries to increment 5 , but this is a constant and cannot be modified.

```
       [FLEX] Line=185, token=IDENTIFIER, value="x"

       [FLEX] Line=185, token=OPERATOR, value="-="

       [FLEX] Line=185, token=STRING, value=""Hello""

!! syntax error!! -> at Line=185

       [FLEX] Line=185, token=DELIMITER, value=";"
```

Tries to remove the string " Hello " from the integer x . Uni - C does not support subtractions with strings but only additions e.g. “ Hello ” + “ World ”.

```
       [FLEX] Line=186, token=IDENTIFIER, value="y"

       [FLEX] Line=186, token=OPERATOR, value="*="

       [FLEX] Line=186, token=SPECIAL, value="["

!! syntax error!! -> at Line=186

       [FLEX] Line=186, token=IDENTIFIER, value="h"

       [FLEX] Line=186, token=SPECIAL, value=","

       [FLEX] Line=186, token=IDENTIFIER, value="e"
```

```
        [FLEX] Line=186, token=SPECIAL, value=","

        [FLEX] Line=186, token=IDENTIFIER, value="l"

        [FLEX] Line=186, token=SPECIAL, value=","

        [FLEX] Line=186, token=IDENTIFIER, value="l"

        [FLEX] Line=186, token=SPECIAL, value=","

        [FLEX] Line=186, token=IDENTIFIER, value="o"

        [FLEX] Line=186, token=SPECIAL, value="]"

        [FLEX] Line=186, token=DELIMITER, value=";"
```

Attempts to multiply the integer y by a list, which is not a valid operation.

```
        [FLEX] Line=187, token=INTEGER, value="50"

        [FLEX] Line=187, token=OPERATOR, value="="

!! syntax error!! -> at Line=187

        [FLEX] Line=187, token=IDENTIFIER, value="x"

        [FLEX] Line=187, token=DELIMITER, value=";"
```

It tries to set the value 50 to the integer x , but the reversal of the objects in the assignment is incorrect.

```
        [FLEX] Line=188, token=IDENTIFIER, value="list1"

        [FLEX] Line=188, token=SPECIAL, value=","

        [FLEX] Line=188, token=IDENTIFIER, value="list2"

        [FLEX] Line=188, token=OPERATOR, value="="

        [FLEX] Line=188, token=SPECIAL, value="["

        [FLEX] Line=188, token=IDENTIFIER, value="x"

        [FLEX] Line=188, token=SPECIAL, value=","

        [FLEX] Line=188, token=IDENTIFIER, value="b"

        [FLEX] Line=188, token=SPECIAL, value=","

        [FLEX] Line=188, token=IDENTIFIER, value="c"

        [FLEX] Line=188, token=SPECIAL, value="]"
        [FLEX] Line=188, token=SPECIAL, value=","

        [FLEX] Line=188, token=INTEGER, value="5"

        [FLEX] Line=188, token=DELIMITER, value=";"



[ BISON ] Line =188, expression ="Assign value to variable"
```

It tries to define two lists and assign them to two variables. But the second variable is an integer, so semantically the expression is not correct. It's syntactical and that's why SA doesn't complain.

```
        [ FLEX ] Line =189, token = IDENTIFIER , value =" x 1"

        [FLEX] Line=189, token=OPERATOR, value="=="

        [FLEX] Line=189, token=OPERATOR, value="="

!! syntax error!! -> at Line=189

        [FLEX] Line=189, token=IDENTIFIER, value="x2"

        [FLEX] Line=189, token=DELIMITER, value=";"
```

It tries to perform a triple equality check, but this is not valid in the case of Uni - C .

```
        [FLEX] Line=190, token=IDENTIFIER, value="array"

        [FLEX] Line=190, token=OPERATOR, value="+="

        [FLEX] Line=190, token=SPECIAL, value="["

        [FLEX] Line=190, token=INTEGER, value="1"

        [FLEX] Line=190, token=SPECIAL, value=","

        [FLEX] Line=190, token=INTEGER, value="3"

        [FLEX] Line=190, token=SPECIAL, value=","

        [FLEX] Line=190, token=INTEGER, value="4"

        [FLEX] Line=190, token=SPECIAL, value="]"

        [FLEX] Line=190, token=DELIMITER, value=";"



[ BISON ] Line =190, expression ="Assign value to variable"
```

Attempts to add a list to an array. This operation is syntactically correct as it is a variant of table join. Semantically, the content of the " array " could be checked .

```
      [ FLEX ] Line =191, token = SPECIAL , value ="["

      [FLEX] Line=191, token=INTEGER, value="1"

      [FLEX] Line=191, token=SPECIAL, value=","

      [FLEX] Line=191, token=INTEGER, value="3"

      [FLEX] Line=191, token=SPECIAL, value=","

      [FLEX] Line=191, token=INTEGER, value="5"

      [FLEX] Line=191, token=SPECIAL, value="]"

      [FLEX] Line=191, token=OPERATOR, value="++"

      [FLEX] Line=191, token=SPECIAL, value="["

!! syntax error!! -> at Line=191

      [FLEX] Line=191, token=INTEGER, value="1"

      [FLEX] Line=191, token=SPECIAL, value=","

      [FLEX] Line=191, token=INTEGER, value="2"

      [FLEX] Line=191, token=SPECIAL, value="]"

      [FLEX] Line=191, token=DELIMITER, value=";"
```

Attempts to concatenate two lists with the ++ operator , but this syntax is not valid in Uni - C . Arrays are concatenated using the + operator and not the increment/transcrement operator ++.

```
      [FLEX] Line=192, token=KEYWORD, value="if"

      [FLEX] Line=192, token=IDENTIFIER, value="var"

      [FLEX] Line=192, token=OPERATOR, value="=="

      [FLEX] Line=192, token=INTEGER, value="100"

!! Token error!! -> at Line=192

!! syntax error!! -> at Line=192

            4 character(s) ignored so far

      [FLEX] Line=193, token=FUNCTION, value="print"

      [FLEX] Line=193, token=SPECIAL, value="("

      [FLEX] Line=193, token=STRING, value=""var""
```

```
       [FLEX] Line=193, token=SPECIAL, value=","

       [FLEX] Line=193, token=STRING, value=""is""

       [FLEX] Line=193, token=SPECIAL, value=","

       [FLEX] Line=193, token=IDENTIFIER, value="var"

       [FLEX] Line=193, token=SPECIAL, value=")"

       [FLEX] Line=193, token=DELIMITER, value=";"



[BISON] Line=193, expression="Function call"
```

The if syntax is wrong, as the parentheses in the " var == 100" condition are missing. Also, LA recognizes an invalid colon character ":" and therefore returns a token error . The syntax of the " print " function is correct.

```
       [ FLEX ] Line =194, token = KEYWORD , value =" while "

       [FLEX] Line=194, token=SPECIAL, value="("

       [FLEX] Line=194, token=INTEGER, value="1"

       [FLEX] Line=194, token=SPECIAL, value=")"



       [FLEX] Line=195, token=SPECIAL, value="{"



       [FLEX] Line=196, token=FUNCTION, value="print"

       [FLEX] Line=196, token=SPECIAL, value="("

       [FLEX] Line=196, token=STRING, value=""Hello World\n""

       [FLEX] Line=196, token=SPECIAL, value=")"

       [FLEX] Line=196, token=DELIMITER, value=";"

[BISON] Line=195, expression="Function call"





       [FLEX] Line=197, token=SPECIAL, value="}"

[BISON] Line=196, expression="Compound Statements"
```

```



[BISON] Line=197, expression="Compound Statements"
```

while loop . SA would be expected to throw a syntax error, since while has no condition. However, the syntax is correct, as constants in the conditions of compound statements ( if , while ), are allowed in Uni - C . Any condition that returns a result of 1 or 0 is acceptable even if it is a constant.

```
	[FLEX] Line=198, token=KEYWORD, value="for"

	[FLEX] Line=198, token=SPECIAL, value="("

	[FLEX] Line=198, token=DELIMITER, value=";"

!! syntax error!! -> at Line=198

	[FLEX] Line=198, token=DELIMITER, value=";"

	[FLEX] Line=198, token=SPECIAL, value=")"


	[FLEX] Line=199, token=SPECIAL, value="{"


	[FLEX] Line=200, token=FUNCTION, value="print"

	[FLEX] Line=200, token=SPECIAL, value="("

	[FLEX] Line=200, token=STRING, value=""Hello World\n""

	[FLEX] Line=200, token=SPECIAL, value=")"

	[FLEX] Line=200, token=DELIMITER, value=";"

[BISON] Line=199, expression="Function call"




	[FLEX] Line=201, token=SPECIAL, value="}"



[BISON] Line=201, expression="Compound Statements"
```

It creates an endless execution for loop , but SA throws a syntax error. Uni - C does not support infinite loops in for statements , only in while statements .

## 2.10 Final

### Input File ( input . txt )

```
func myAdd(int num1, int num2)

{

        int sum;

        sum = a + b;

print("Sum is: ", sum);

}


func main()

{

        int a;

        int b;

scan(a);

scan(b);

myAdd(a,b);

}


func errorHandling()

{

        long a, b, c, d, e;

        int i;

        int x;
scan(a, b, c, d, e);

	pin1 = [a, b, c];

	pin2 = [d, e];

pinFinal = [a, b, c] + $[d, e];



	for(i = 0; i < pinFinal.N; i++)

	{

	x = pinFinal[i];

	x /= 2;

	if ((x == 0))

print("%d is Even\n", pinFinal[i]);

	}

}
```

## Output File ( output . txt )

We tested the capabilities of the syntax and word analyzer with an example of some functions that a user could create.

```
	[FLEX] Line=204, token=KEYWORD, value="func"

	[FLEX] Line=204, token=IDENTIFIER, value="myAdd"

	[FLEX] Line=204, token=SPECIAL, value="("

	[FLEX] Line=204, token=KEYWORD, value="int"

	[FLEX] Line=204, token=IDENTIFIER, value="num1"

	[FLEX] Line=204, token=SPECIAL, value=","

	[FLEX] Line=204, token=KEYWORD, value="int"

	[FLEX] Line=204, token=IDENTIFIER, value="num2"

	[FLEX] Line=204, token=SPECIAL, value=")"
```

```
       [FLEX] Line=205, token=SPECIAL, value="{"


       [FLEX] Line=206, token=KEYWORD, value="int"

       [FLEX] Line=206, token=IDENTIFIER, value="sum"

       [FLEX] Line=206, token=DELIMITER, value=";"

[BISON] Line=205, expression="Variable declaration"



       [FLEX] Line=207, token=IDENTIFIER, value="sum"

       [FLEX] Line=207, token=OPERATOR, value="="

       [FLEX] Line=207, token=IDENTIFIER, value="a"

       [FLEX] Line=207, token=OPERATOR, value="+"

       [FLEX] Line=207, token=IDENTIFIER, value="b"

       [FLEX] Line=207, token=DELIMITER, value=";"

[BISON] Line=206, expression="Assign value to variable"



       [FLEX] Line=208, token=FUNCTION, value="print"

       [FLEX] Line=208, token=SPECIAL, value="("

       [FLEX] Line=208, token=STRING, value=""Sum is: ""

       [FLEX] Line=208, token=SPECIAL, value=","

       [FLEX] Line=208, token=IDENTIFIER, value="sum"

       [FLEX] Line=208, token=SPECIAL, value=")"

       [FLEX] Line=208, token=DELIMITER, value=";"

[BISON] Line=207, expression="Function call"
```

```
       [FLEX] Line=209, token=SPECIAL, value="}"



[ BISON ] Line =209, expression ="Declaration of user functions"
```

In the first function named Myadd , we recognize the declaration of a variable in line 206, the assignment of a value to a variable in line 207, and finally the call to the print function in line 208. Finally, in line 209 the user-declared function is defined with the character “}” which “closes” the Myadd function .

```
       [FLEX] Line=211, token=KEYWORD, value="func"

       [FLEX] Line=211, token=IDENTIFIER, value="main"

       [FLEX] Line=211, token=SPECIAL, value="("

       [FLEX] Line=211, token=SPECIAL, value=")"


       [FLEX] Line=212, token=SPECIAL, value="{"


       [FLEX] Line=213, token=KEYWORD, value="int"

       [FLEX] Line=213, token=IDENTIFIER, value="a"

       [FLEX] Line=213, token=DELIMITER, value=";"

[BISON] Line=212, expression="Variable declaration"



       [FLEX] Line=214, token=KEYWORD, value="int"

       [FLEX] Line=214, token=IDENTIFIER, value="b"

       [FLEX] Line=214, token=DELIMITER, value=";"

[BISON] Line=213, expression="Variable declaration"



       [FLEX] Line=215, token=FUNCTION, value="scan"

       [FLEX] Line=215, token=SPECIAL, value="("
       [FLEX] Line=215, token=IDENTIFIER, value="a"

       [FLEX] Line=215, token=SPECIAL, value=")"

       [FLEX] Line=215, token=DELIMITER, value=";"

[BISON] Line=214, expression="Function call"




       [FLEX] Line=216, token=FUNCTION, value="scan"

       [FLEX] Line=216, token=SPECIAL, value="("

       [FLEX] Line=216, token=IDENTIFIER, value="b"

       [FLEX] Line=216, token=SPECIAL, value=")"

       [FLEX] Line=216, token=DELIMITER, value=";"

[BISON] Line=215, expression="Function call"




       [FLEX] Line=217, token=IDENTIFIER, value="myAdd"

       [FLEX] Line=217, token=SPECIAL, value="("

       [FLEX] Line=217, token=IDENTIFIER, value="a"

       [FLEX] Line=217, token=SPECIAL, value=","

       [FLEX] Line=217, token=IDENTIFIER, value="b"

       [FLEX] Line=217, token=SPECIAL, value=")"

       [FLEX] Line=217, token=DELIMITER, value=";"

[BISON] Line=216, expression="Function call"




       [FLEX] Line=218, token=SPECIAL, value="}"

[BISON] Line=217, expression="Declaration of user functions"
```

```


[BISON] Line=218, expression="Declaration of user functions"
```

In the second function named main , commands such as the declaration of variables are recognized on lines 213 and 214. Also on line 216 we have the call to the Uni - C built-in function print . Finally, in line 217 we have the recognition of the function myAdd while the definition of the function main is done in line 218 with the character “}”.

```
        [FLEX] Line=220, token=KEYWORD, value="func"

        [FLEX] Line=220, token=IDENTIFIER, value="errorHandling"

        [FLEX] Line=220, token=SPECIAL, value="("

        [FLEX] Line=220, token=SPECIAL, value=")"



        [FLEX] Line=221, token=SPECIAL, value="{"



        [FLEX] Line=222, token=KEYWORD, value="long"

        [FLEX] Line=222, token=IDENTIFIER, value="a"

        [FLEX] Line=222, token=SPECIAL, value=","

        [FLEX] Line=222, token=IDENTIFIER, value="b"

        [FLEX] Line=222, token=SPECIAL, value=","

        [FLEX] Line=222, token=IDENTIFIER, value="c"

        [FLEX] Line=222, token=SPECIAL, value=","

        [FLEX] Line=222, token=IDENTIFIER, value="d"

        [FLEX] Line=222, token=SPECIAL, value=","

        [FLEX] Line=222, token=IDENTIFIER, value="e"

        [FLEX] Line=222, token=DELIMITER, value=";"

[BISON] Line=221, expression="Variable declaration"




        [FLEX] Line=223, token=KEYWORD, value="int"
```

```
       [FLEX] Line=223, token=IDENTIFIER, value="i"

       [FLEX] Line=223, token=DELIMITER, value=";"

[BISON] Line=222, expression="Variable declaration"




       [FLEX] Line=224, token=KEYWORD, value="int"

       [FLEX] Line=224, token=IDENTIFIER, value="x"

       [FLEX] Line=224, token=DELIMITER, value=";"

[BISON] Line=223, expression="Variable declaration"





       [FLEX] Line=226, token=FUNCTION, value="scan"

       [FLEX] Line=226, token=SPECIAL, value="("

       [FLEX] Line=226, token=IDENTIFIER, value="a"

       [FLEX] Line=226, token=SPECIAL, value=","

!! syntax error!! -> at Line=226

       [FLEX] Line=226, token=IDENTIFIER, value="b"

       [FLEX] Line=226, token=SPECIAL, value=","

       [FLEX] Line=226, token=IDENTIFIER, value="c"

       [FLEX] Line=226, token=SPECIAL, value=","

       [FLEX] Line=226, token=IDENTIFIER, value="d"

       [FLEX] Line=226, token=SPECIAL, value=","

       [FLEX] Line=226, token=IDENTIFIER, value="e"

       [FLEX] Line=226, token=SPECIAL, value=")"

       [FLEX] Line=226, token=DELIMITER, value=";"
       [FLEX] Line=227, token=IDENTIFIER, value="pin1"

       [FLEX] Line=227, token=OPERATOR, value="="

       [FLEX] Line=227, token=SPECIAL, value="["

       [FLEX] Line=227, token=IDENTIFIER, value="a"

       [FLEX] Line=227, token=SPECIAL, value=","

       [FLEX] Line=227, token=IDENTIFIER, value="b"

       [FLEX] Line=227, token=SPECIAL, value=","

       [FLEX] Line=227, token=IDENTIFIER, value="c"

       [FLEX] Line=227, token=SPECIAL, value="]"

       [FLEX] Line=227, token=DELIMITER, value=";"


[BISON] Line=227, expression="Assign value to variable"


       [FLEX] Line=228, token=IDENTIFIER, value="pin2"

       [FLEX] Line=228, token=OPERATOR, value="="

       [FLEX] Line=228, token=SPECIAL, value="["

       [FLEX] Line=228, token=IDENTIFIER, value="d"

       [FLEX] Line=228, token=SPECIAL, value=","

       [FLEX] Line=228, token=IDENTIFIER, value="e"

       [FLEX] Line=228, token=SPECIAL, value="]"

       [FLEX] Line=228, token=DELIMITER, value=";"


[BISON] Line=228, expression="Assign value to variable"


       [FLEX] Line=229, token=IDENTIFIER, value="pinFinal"

       [FLEX] Line=229, token=OPERATOR, value="="

       [FLEX] Line=229, token=SPECIAL, value="["
```

```
	[FLEX] Line=229, token=IDENTIFIER, value="a"

	[FLEX] Line=229, token=SPECIAL, value=","

	[FLEX] Line=229, token=IDENTIFIER, value="b"

	[FLEX] Line=229, token=SPECIAL, value=","

	[FLEX] Line=229, token=IDENTIFIER, value="c"

	[FLEX] Line=229, token=SPECIAL, value="]"

	[FLEX] Line=229, token=OPERATOR, value="+"

!! Token error!! -> at Line=229

	8 character(s) ignored so far

	[FLEX] Line=229, token=IDENTIFIER, value="e"

!! syntax error!! -> at Line=229

	[FLEX] Line=229, token=SPECIAL, value="]"

	[FLEX] Line=229, token=DELIMITER, value=";"



	[FLEX] Line=231, token=KEYWORD, value="for"

	[FLEX] Line=231, token=SPECIAL, value="("

	[FLEX] Line=231, token=IDENTIFIER, value="i"

	[FLEX] Line=231, token=OPERATOR, value="="

	[FLEX] Line=231, token=INTEGER, value="0"

	[FLEX] Line=231, token=DELIMITER, value=";"

	[FLEX] Line=231, token=IDENTIFIER, value="i"

	[FLEX] Line=231, token=OPERATOR, value="<"

	[FLEX] Line=231, token=IDENTIFIER, value="pinFinal"

!! Token error!! -> at Line=231

!! syntax error!! -> at Line=231

	11 character(s) ignored so far
```

```
      [FLEX] Line=231, token=IDENTIFIER, value="i"

      [FLEX] Line=231, token=OPERATOR, value="++"

      [FLEX] Line=231, token=SPECIAL, value=")"


      [FLEX] Line=232, token=SPECIAL, value="{"


      [FLEX] Line=233, token=IDENTIFIER, value="x"

      [FLEX] Line=233, token=OPERATOR, value="="

      [FLEX] Line=233, token=IDENTIFIER, value="pinFinal"

      [FLEX] Line=233, token=SPECIAL, value="["

      [FLEX] Line=233, token=IDENTIFIER, value="i"

      [FLEX] Line=233, token=SPECIAL, value="]"

      [FLEX] Line=233, token=DELIMITER, value=";"

[BISON] Line=232, expression="Assign value to variable"



      [FLEX] Line=234, token=IDENTIFIER, value="x"

      [FLEX] Line=234, token=OPERATOR, value="/="

      [FLEX] Line=234, token=INTEGER, value="2"

      [FLEX] Line=234, token=DELIMITER, value=";"

[BISON] Line=233, expression="Assign value to variable"



      [FLEX] Line=235, token=KEYWORD, value="if"

      [FLEX] Line=235, token=SPECIAL, value="("

      [FLEX] Line=235, token=SPECIAL, value="("

      [FLEX] Line=235, token=IDENTIFIER, value="x"
```

```
       [FLEX] Line=235, token=OPERATOR, value="=="

       [FLEX] Line=235, token=INTEGER, value="0"

       [FLEX] Line=235, token=SPECIAL, value=")"

       [FLEX] Line=235, token=SPECIAL, value=")"

## Warning ## -> Double parethensis detected at Line=235


[BISON] Line=235, expression="If Statement"


       [FLEX] Line=236, token=FUNCTION, value="print"

       [FLEX] Line=236, token=SPECIAL, value="("

       [FLEX] Line=236, token=STRING, value=""%d is Even\n""

       [FLEX] Line=236, token=SPECIAL, value=","

       [FLEX] Line=236, token=IDENTIFIER, value="pinFinal"

       [FLEX] Line=236, token=SPECIAL, value="["

       [FLEX] Line=236, token=IDENTIFIER, value="i"

       [FLEX] Line=236, token=SPECIAL, value="]"

       [FLEX] Line=236, token=SPECIAL, value=")"

       [FLEX] Line=236, token=DELIMITER, value=";"

[BISON] Line=235, expression="Function call"




       [FLEX] Line=237, token=SPECIAL, value="}"


[BISON] Line=237, expression="Compound Statements"


       [FLEX] Line=238, token=SPECIAL, value="}"

!! syntax error !! -> at Line = 238
```

The third function defined is errorHandling , the variable declarations on lines 221,222,223 are defined. Lines 227,228,233 define the pin 1, pin 2, x arrays respectively. On line 226 we have a syntax error as the builtin function scan is not defined correctly because it cannot accept more than 1 argument. On line 229 we also have a syntax error in adding arrays because the “$” character is not recognized by some grammar rule. We also have a syntax error in line 231, because the character '.' not provided for in variable naming. On line 234 we have a correct assignment of a value to a variable. The syntax errors on line 238 is a compiler bug .

## Analysis Statistics

### Output File ( output . txt )

```
              SYNTHETIC ANALYSIS STATISTICS



BISON -> Syntax parsing failed

              (with 14 warnings )



              CORRECT WORDS: 329

              CORRECT EXPRESSIONS: 116

              WRONG WORDS: 11

              WRONG EXPRESSIONS: 20
```

The syntax analysis statistics derived from the output file . txt describe the success and problems the parser encountered in parsing the input file . txt using the FLEX and BISON tools .

**Editorial Analysis Statistics:**

1. **BISON -> Syntax parsing failed (with 14 warnings ):**

- This indicates that the BISON parser failed to complete the syntax analysis of the code.
- The 14 warning messages indicate that there were some syntax warning errors during parsing, but the code continued to run with correct and incorrect words and expressions displayed.

2. **CORRECT WORDS: 329**

- This shows the number of word units correctly recognized by the LA.

3. **CORRECT EXPRESSIONS: 116**

- This refers to the number of syntactic expressions correctly recognized by the SA.

4. **WRONG WORDS: 10**

- It presents the number of incorrect word units detected by LA.

5. **WRONG EXPRESSIONS: 20**

- It presents the number of incorrect syntactic expressions observed by the SA.

These statistics provide a good picture of the results of the editorial analysis. The use of warnings can help troubleshoot problems and improve the accuracy of the parser, while metrics about correct and incorrect words and expressions provide important indicative information about the quality of code processing.

## Explicit code execution reference

FLEX and BISON code are executed with the Makefile which allows the code to be compiled and elected automatically.

```
compile :
bison -d simple-bison-code.y
flex simple-flex-code.l
gcc -o compiler-Uni-C simple-bison-code.tab.c lex.yy.c -lfl
./compiler-Uni-C input.txt output.txt
clean :
rm simple-bison-code.tab.c simple-bison-code.tab.h lex.yy.c compiler-Uni-C
```

However, during the code generation stage . c of BISON some warnings are presented .

```
bison -d simple-bison-code.y

simple-bison-code.y: warning: 19 shift/reduce conflicts [-Wconflicts-sr]

simple-bison-code.y: warning: 135 reduce/reduce conflicts [-Wconflicts-rr]

simple-bison-code.y: note: rerun with option '-Wcounterexamples' to generate conflict
counterexamples
```

The warning messages indicate that there are conflicts in the grammar rules, namely:

- **Shift/Reduce Conflicts**

  shift / reduce conflicts where the CA cannot decide whether to shift (read the next input symbol) or reduce (apply a production rule of the grammar)

- **Reduce/Reduce Conflicts**

  reduce / reduce conflicts where the CA cannot decide which production rule to apply when there are multiple candidate production rules that match.

The team tried to resolve these conflicts by debugging the BISON code with the command

```
bison -d simple-bison-code.y --verbose
```

in order to identify in which situations the SA has difficulty making decisions.

An illustrative example of shift / reduce conflict presented in file simple - bison - code . debug output is as follows :

```
State 8
```

```
8 type: SDOUBLE •

12 | SDOUBLE • SDOUBLE



SDOUBLE shift, and go to state 56



SDOUBLE [reduce using rule 8 (type)]

$default reduce using rule 8 (type)
```

In the variable declaration rule where the user chooses the keyword to use to declare the variable's data type, there is a conflict with the rule that handles a possible syntax warning error from the user putting the keyword twice . The SA cannot decide whether after the keyword double , will read the next input symbol ( shift ) or use the type production rule ( reduce ).

The team tried to resolve the individual conflicts , but failed, as the existing syntax expressions recognized by Uni - C expressions required modifications and time was limited to implement.

However, taking stock of the SA we found the following observations that lead to these conflicts:

1. **Very General Rules**

   Some rules are very general such as program , decl _ statements , decl _ statement and cover many different cases with common terminal and non-terminal symbols.

2. **Indistinct Rules**

   Some rules are not easily distinguished from each other due to their structure. For example, the rules assign and cmp _ expr can end up in conflicts due to using the same tokens ( var , arithmetic _ expr )

3. **Missing Function Code (in C ) in Rules**

   No special emphasis was placed on writing appropriate C code to handle symbols when the SA recognizes some syntax rule. Hence in most rules the team has chosen to copy the constant yytext to the non-terminal symbol of the rule with the strdup ( yytext ) command. Consequently, the result is not calculated in arithmetic operations for the reason given above.

The result of the aforementioned is that the SA recognizes the expression

```
func main()

{

print("Hello World\n");
}
```

and not her

```
}

func main () {

    print (" Hello World \ n ");

}
```

The team chose, in terms of time and difficulty, to give more weight to the writing of syntactic expressions and the management of syntactic warning errors which is also the mandatory part of the assignment.

bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier

Thank you for your attention.