COMPILERS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

DEPARTMENT OF INFORMATION AND COMPUTER ENGINEERING

PART A3

FLEX CODES

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

Reference

FLEX template code ” is the 2nd part of the task “Creating an Independent Parser with the FLEX generator ”. The report focuses on the presentation of the FLEX code that recognizes Uni - C language units through the regular expressions implemented by the team in the 1st part “A2 Coding of automatic finite states via FSM ”. Through exhaustive test executions carried out in this report, the correct operation of the verbal analyzer is clarified which will form the basis for the implementation of the syntactic analysis in part B.

Organization let's team

Through the collaborative atmosphere, all team members contributed to completing the FLEX code to understand how it works and how exactly it connects to the work we did in the 1st part of the finite automaton work. Then, the tasks were divided as in the 1st part into sub-groups, tasks such as commenting the code, the options of the input tests, but also the extensive commenting of the results.

Reference content

The presentation focuses on the functionality of the parser with the FLEX generator, which recognizes Uni - C language units . The heads are one at a time are the following :

1. code FLEX
2. Control cases
3. Commentary on results

In addition, for each task are listed:

1. Which sub - group ? he wrote
2. Commenting on 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:

- Introduction
- 1. Tokens
- FLEX Code
- 3. Control cases
- 4. Commenting on results
- Writing a report

Theocharis Georgios (UNIWA-19390283)

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

- Introduction
- 1. Tokens
- FLEX Code
- 3. Control cases
- 4. Commenting on results

Tatsis Pantelis (UNIWA-20390226)

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

- Introduction
- 1. Tokens
- FLEX Code
- 3. Control cases
- 4. Commenting on results

Iliou Ioannis (UNIWA-19390066)

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

- Introduction
- 1. Tokens
- FLEX Code
- 3. Control cases
- 4. Commenting on results

Dominaris Vasileios (UNIWA-21390055)

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

- Introduction
- 1. Tokens
- FLEX Code
- 3. Control cases
- 4. Commenting on results

1. Tokens

The table of tokens is in the **token** header file **. h** .

```
#define DELIMITER 1

#define IDENTIFIER 2

#define STRING 3

#define INTEGER 4

#define FLOAT 5

#define KEYWORD 6

#define OPERATOR 7

#define NEWLINE 8

#define END_OF_FILE 9

# define UNKNOWN 10
```

FLEX Code

The FLEX code is in the **simple - flex - code** source file **. l**

```
/* File name : simple-flex - code.l

   Description: Sample for developing a verbal parser using the Flex tool

Author: Compilers Laboratory, Department of Informatics and Computer Engineering,

University of West Attica

Comments: This program implements (using flex ) a simple parser

which recognizes spaces (space and tab ) and numbers (decimal

only!) for the Uni - C language while handling special characters

                         no let 's '\n' (new line) and 'EOF' (end of file) characters .
There are reports

to identify variables, with the actual code replaced

from the verbal FILL ME so that it can be completed by you. Optionally the verbal

parser accepts file arguments for input and output.
```

```
Execution instructions: Enter " make " without the quotes in the current directory. In
change:

                         flex -o simple-flex- code.c simple-flex- code.l

                         gcc -o simple-flex-code simple-flex- code.c

                         ./ simple - flex - code

*/


/* Reading is limited to a single file and terminates at the first EOF */

% option noyywrap



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 " token . h "


/* Set current line counter */

int line = 1;


%}



/* 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 */



DELIMITER [?]

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))?

KEYWORD
(break|case|const|continue|do|double|else|float|for|if|int|long|return|sizeof|struct|swi
tch|void|while|func)

OPERATOR             (\+|-|\*|\/|%|=|\+=|-=-|\*=|\/=|!|&&|\|\||==|!=|\+             \+|--
|<|>|<=|>=|&|\||\^|<<|>>|&)

COMMENT (\/\/.*\n|\/\ *[ ^*]*\*+([^/*][^*]*\*+)*\/)

WHITESPACE [ \ t]+

NEWLINE [\n]

UNKNOWN [^ \t\r\n;]+



/* For each pattern (left) that matches, the corresponding one is executed

code inside the brackets. The return command allows going back

of a numeric value via the yylex () function */


%%


{DELIMITER} { return DELIMITER; }

{IDENTIFIER} { return IDENTIFIER; }

{STRING} { return STRING; }

{INTEGER} { return INTEGER; }

{FLOAT} { return FLOAT; }

{KEYWORD} { return KEYWORD; }

{OPERATOR} { return OPERATOR; }

{COMMENT} { ECHO ; line++; }
```

```
{WHITESPACE} { }

{NEWLINE} { return NEWLINE; }

<<EOF>> { return END_OF_FILE; }

{ UNKNOWN } { return UNKNOWN ? }


%%


/* Table with all tokens corresponding to the definitions in token . h */

char * tname [] = {

"DELIMITER",

"IDENTIFIER",

"STRING",

"INTEGERS",

"FLOAT",

"KEYWORD",

"OPERATOR",

"NEWLINE",

   " END _ OF _ FILE ",

" UNKNOWN "

};



main function : The following code will be automatically placed in the

C program that will create Flex and be the initial

execution point of the parser application. */


int main( int argc , char ** argv ){

        int token ;


/* Check command line arguments. If the

arguments are 3, the program reads from the file of the 2nd

argument and writes to the 3rd file. If the arguments are
```

```
2, reads from the 2nd argument file and writes to the screen.

It is assumed that the 1st argument ( argv [0]) in C is the name

of the executable file itself. */


        if( argc == 3 ){

                if( !( yyin = fopen ( argv [1], "r"))) {

                        fprintf ( stderr , "Cannot read file: %s\n" , argv [ 1 ]);

                        return 1;

}

                if( !( yyout = fopen ( argv [2], "w"))) {

                        fprintf ( stderr , "Cannot create file: %s\n" , argv [ 2 ]);

                        return 1;

}

}

        else if( argc == 2 ){

                if( !( yyin = fopen ( argv [1], "r"))) {

                        fprintf ( stderr , "Cannot read file: %s\n" , argv [ 1 ]);

                        return 1;

}

}


yylex function reads characters from the input and tries

to recognize tokens . The tokens it recognizes are the ones they have

set in this file, between %% and %%. If the code that

matches a template contains the command ' return VALUE', yylex ()

returns this value and stores the result in the variable token . */


        while ( ( token = yylex ()) >= 0 ){

/* For each recognized token , the line it was found on is printed

and his name along with his honor. */

                if (token)

{
```

```
                  switch ( token)

{

                        case NEWLINE:

                           line ++;

                           break

                        case END_OF_FILE:

                           fprintf ( yyout , "\ tLine =%d, token=%s, value=\"%s\"\n", line,
tname [token-1], yytext );

                           printf ( "#END-OF-FILE#\n" );

                           exit( 0 );

                           break

                        case UNKNOWN:

                           fprintf ( yyout , "\ tLine =%d, token=%s TOKEN, value=\"%s\"\n",
line, tname [token-1], yytext );

                           break

                        default :

                           fprintf ( yyout , "\ tLine =%d, token=%s, value=\"%s\"\n", line,
tname [token-1], yytext );

                           break

}

}


}

        return 0;

}
```

3. Control cases

The test cases are in the **input** text file **. txt** .

```
// ====== Test case #1 IDENTIFIERS ======

a1

_1

Spectacular

a

A1

SPEctacular_15_

_a

_W

_ _A

a_b_c_1_2_3

AA4_F

1_id

/id

id.student

i d student

E2

_user

 user


// ====== Test case #2 STRINGS ======

"KALHMERA"

""

" hello , \"world!\"\n"

"\"\\n\"\"\\n\""

f" number of loops is {counter}"

"6/3=2"

"He said 'Why Brutus?'"

"6\3=2"

"6\\3=2"

"\n\"\\\d"
```

```
" Good morning world "

La vida location

"Hi Mark"

"Tests"

"Mark said, \"Boo!\"\n"

"Hi/n\n"


// ====== Test case #3 INTEGERS ======

0

3

214748

-50

0x4F

0X88AA

063

00

01

09

0XFGA9

01578

00xAFB1

-001

-0xAF01

0xff23

+01

+56

0x-FF


// ====== Test case #4 FLOATS ======

9e8

9E-8

3.14
```

```
10.0

10.0001

5e-15

1e100

3.1E0

0e0

-5.1e-100

0

-0.5

5.e1

1E1

00.01

0e01

5e1.5e1

1E1.2

6.20

0.0


// ====== Test case #5 COMMENTS ======

// hello world!!

/hello world!!/

# hello world!

// hello

hello world

/* hello world!! */

//hello//world//!!

\\ hello world!!

/*

* hello

* world

* !!

*/
```

```
/*

hi

*/

Hello world! *\

/*//

//hi

//* hi *//

%% hi

// hi */

/* hi //

// This is a comment

*/


// ====== Test case #6 KEYWORDS ======

int x = 5;

float y;

string str = "";

switch x?

case 1:

    break

case 2:

    break

sizeof ( str );

sizeof ;

double = 3.14;

const int z = 5;

struct Employee;

if x == 0

else x != 0

return 0;


// ====== Test case #7 FINAL ======
user ;

a =( b+c );

" theo "

45

4.8

//hi

4.7

" good moring "

4e2

good


// ====== Test case #8 FINAL ======

" paw "

/*20*/

8.5

+

+---

break

func


// ====== Test case #9 FINAL ======

@#$

8,3

1_tzigger

/f

0.0

5,001


// ====== Test case #10 OPERATORS ======

x >= 5;

5 + 3 = 8?

6/=3;
```

```
x ++;

x* y?

while 1;

void ;

TRUE && FALSE == FALSE;

TRUE || TRUE == TRUE;

y-- ;

&bit


// ====== Test case #11 CODE ======

switch 1

    case 1

        break

    case 2

        break

return 0;


// ====== Test case #12 CODE ======

int x + y = z;

var x + y = z;

for i = 1; i <= N; i ++ do

    if i % 2 == 0 then

        return 0;

    else

        return 1;

    endif ?

end for?


// ====== Test case #13 CODE ======

const double pi=3.14;

struct Math m1;

m1.count = 3.14 * 2;
```

```
m1.length = 3.14 + 2 / 5;


// ====== Test case #14 CODE/COMMENTS ======

func calculate_perimeter_of_table void

    int length = 5; // Length of table

    int width = 10; // Width of table

    int x;


x = length * 2 + width * 2; // Perimeter of table

    return x ;
```

4. Commenting on results

The results of the test cases are in the **output** text file **. txt** .

[Comment]

In line 10 the parser ignores the whitespace character (\ s) and as a result accepts the characters "_", "_ A " separately. However, it correctly recognizes them separately as identifiers . In line 16 the same happens with " i d student ", where the parser accepts " i ", " d ", " student " separately.

```
// ====== Test case #1 IDENTIFIERS ======

      Line=2, token=IDENTIFIER, value="a1"


      Line=3, token=IDENTIFIER, value="_1"


      Line=4, token=IDENTIFIER, value="Spectacular"


      Line=5, token=IDENTIFIER, value="a"


      Line=6, token=IDENTIFIER, value="A1"


      Line=7, token=IDENTIFIER, value="SPEctacular_15_"


      Line=8, token=IDENTIFIER, value="_a"
```

```
Line=9, token=IDENTIFIER, value="_W"


Line=10, token=IDENTIFIER, value="_"

Line=10, token=IDENTIFIER, value="_A"


Line=11, token=IDENTIFIER, value="a_b_c_1_2_3"


Line=12, token=IDENTIFIER, value="AA4_F"


Line=13, token=UNKNOWN TOKEN, value="1_id"


Line=14, token=UNKNOWN TOKEN, value="/id"


Line=15, token=UNKNOWN TOKEN, value=" id.student "


Line=16, token=IDENTIFIER, value=" i "

Line=16, token=IDENTIFIER, value="d"

Line=16, token=IDENTIFIER, value="student"


Line=17, token=UNKNOWN TOKEN, value="E2"


Line=18, token=IDENTIFIER, value="_user"


Line=19, token=IDENTIFIER, value="user"
```

[Comment]

In line 26 the parser identifies the errors as unknown tokens , but splits valid expressions into identifiers due to whitespace (\ s). In line 28 it recognizes the wrong string as unknown tokens . In line 33 it also separates the words “ la ”, “ vida ”, “ loca ” due to the existence again of whitespace (\ s).

```
// ====== Test case #2 STRINGS ======

      Line=22, token=STRING, value=""KALHMERA""


      Line=23, token=STRING, value=""""


      Line=24, token=STRING, value=""hello, \"world!\"\n""


      Line=25, token=STRING, value=""\"\\n\"\"\\n\"""


      Line=26, token=UNKNOWN TOKEN, value=" f"number "

      Line=26, token=IDENTIFIER, value="of"

      Line=26, token=IDENTIFIER, value="loops"

      Line=26, token=IDENTIFIER, value="is"

      Line=26, token=UNKNOWN TOKEN, value="{counter}""


      Line=27, token=STRING, value=""6/3=2""


      Line=28, token=STRING, value=""He said ""

      Line=28, token=IDENTIFIER, value="Why"

      Line=28, token=UNKNOWN TOKEN, value="Brutus?"""


      Line=29, token=UNKNOWN TOKEN, value=""6\3=2""


      Line=30, token=STRING, value=""6\\3=2""


      Line=31, token=UNKNOWN TOKEN, value=""\n\"\\\d""


      Line=32, token=STRING, value="" Hello world " "
```

```
       Line=33, token=UNKNOWN TOKEN, value="'La"

       Line=33, token=IDENTIFIER, value=" vida "

       Line=33, token=UNKNOWN TOKEN, value=" loca '"


       Line=34, token=STRING, value=""Hi Mark""


       Line=35, token=STRING, value=""Tests""


       Line=36, token=STRING, value=""Mark said, \"Boo!\"\n""


       Line=37, token=STRING, value=""Hi/n\n""
```

[Comment]

Note that the parser works correctly in the following examples, as anything not recognized by the Uni - C language is considered unknown tokens .

```
// ====== Test case #3 INTEGERS ======

       Line=40, token=INTEGER, value="0"


       Line=41, token=INTEGER, value="3"


       Line=42, token=INTEGER, value="214748"


       Line=43, token=UNKNOWN TOKEN, value="-50"


       Line=44, token=INTEGER, value="0x4F"


       Line=45, token=INTEGER, value="0X88AA"


       Line=46, token=INTEGER, value="063"


       Line=47, token=INTEGER, value="00"
```

```
        Line=48, token=INTEGER, value="01"


        Line=49, token=UNKNOWN TOKEN, value="09"


        Line=50, token=UNKNOWN TOKEN, value="0XFGA9"


        Line=51, token=UNKNOWN TOKEN, value="01578"


        Line=52, token=UNKNOWN TOKEN, value="00xAFB1"


        Line=53, token=UNKNOWN TOKEN, value="-001"


        Line=54, token=UNKNOWN TOKEN, value="-0xAF01"


        Line=55, token=UNKNOWN TOKEN, value="0xff23"


        Line=56, token=UNKNOWN TOKEN, value="+01"


        Line=57, token=UNKNOWN TOKEN, value="+56"


        Line=58, token=UNKNOWN TOKEN, value="0x-FF"
```

[Comment]

On line 71 we notice that the parser incorrectly recognizes 0 as an integer. Based on Uni - C rules it should not be recognized. On line 74 the parser recognizes the Greek character E as a valid character, this should not be the case again based on Uni - C rules .

```
// ====== Test case #4 FLOATS ======

        Line=61, token=FLOAT, value="9e8"


        Line=62, token=FLOAT, value="9E-8"


        Line=63, token=FLOAT, value="3.14"
```

```
Line=64, token=UNKNOWN TOKEN, value="10.0"

Line=65, token=FLOAT, value="10.0001"

Line=66, token=FLOAT, value="5e-15"

Line=67, token=FLOAT, value="1e100"

Line=68, token=FLOAT, value="3.1E0"

Line=69, token=FLOAT, value="0e0"

Line=70, token=UNKNOWN TOKEN, value="-5.1e-100"

Line=71, token=INTEGER, value="0"

Line=72, token=UNKNOWN TOKEN, value="-0.5"

Line=73, token=UNKNOWN TOKEN, value="5.e1"

Line=74, token=FLOAT, value="1E1"

Line=75, token=UNKNOWN TOKEN, value="00.01"

Line=76, token=UNKNOWN TOKEN, value="0e01"

Line=77, token=UNKNOWN TOKEN, value="5e1.5e1"

Line=78, token=UNKNOWN TOKEN, value="1E1.2"

Line=79, token=UNKNOWN TOKEN, value="6 .20 "
```

```
      Line=80, token=UNKNOWN TOKEN, value="0.0"
```

[Comment]

Any comment that is valid is displayed but not returned as a token . “ // hello ” is recognized by the parser because Greek characters are allowed by flex .

```
// ====== Test case #5 COMMENTS ======

// hello world!!

      Line=84, token=UNKNOWN TOKEN, value="/hello"

      Line=84, token=UNKNOWN TOKEN, value="world!!/"


      Line=85, token=UNKNOWN TOKEN, value="#"

      Line=85, token=IDENTIFIER, value="hello"

      Line=85, token=UNKNOWN TOKEN, value="world!"


// hello

      Line=87, token=IDENTIFIER, value="hello"

      Line=87, token=IDENTIFIER, value="world"


/* hello world!! */

//hello//world//!!

      Line=91, token=UNKNOWN TOKEN, value="\\"

      Line=91, token=IDENTIFIER, value="hello"

      Line=91, token=UNKNOWN TOKEN, value="world!!"


/*

* hello

* world

* !!

*/

/*

hi
```

```
*/
       Line=96, token=UNKNOWN TOKEN, value="\*"
       Line=96, token=IDENTIFIER, value="hello"
       Line=96, token=UNKNOWN TOKEN, value="world!"
       Line=96, token=UNKNOWN TOKEN, value="*\"


/*//
//hi
//* hi */    Line=98, token=OPERATOR, value="/"


       Line=99, token=UNKNOWN TOKEN, value="%%"
       Line=99, token=IDENTIFIER, value="hi"


// hi */
/* hi //
// This is a comment
*/
```

[Comment]

Keywords are recognized by the parser as identifiers (incorrectly) together with the separator "?". It does not recognize the special character: as we see in " case ". Operators are recognized normally.

```
// ====== Test case #6 KEYWORDS ======
       Line=105, token=IDENTIFIER, value=" int "
       Line=105, token=IDENTIFIER, value="x"
       Line=105, token=OPERATOR, value="="
       Line=105, token=INTEGER, value="5"
       Line=105, token=DELIMITER, value=";"


       Line=106, token=IDENTIFIER, value="float"
       Line=106, token=IDENTIFIER, value="y"
       Line=106, token=DELIMITER, value=";"
```

```
Line=107, token=IDENTIFIER, value="string"

Line=107, token=IDENTIFIER, value=" str "

Line=107, token=OPERATOR, value="="

Line=107, token=STRING, value=""""

Line=107, token=DELIMITER, value=";"


Line=108, token=IDENTIFIER, value="switch"

Line=108, token=IDENTIFIER, value="x"

Line=108, token=DELIMITER, value=";"


Line=109, token=IDENTIFIER, value="case"

Line=109, token=UNKNOWN TOKEN, value="1:"


Line=110, token=IDENTIFIER, value="break"

Line=110, token=DELIMITER, value=";"


Line=111, token=IDENTIFIER, value="case"

Line=111, token=UNKNOWN TOKEN, value="2:"


Line=112, token=IDENTIFIER, value="break"

Line=112, token=DELIMITER, value=";"


Line=113, token=UNKNOWN TOKEN, value=" sizeof ( str )"

Line=113, token=DELIMITER, value=";"


Line=114, token=IDENTIFIER, value=" sizeof "

Line=114, token=DELIMITER, value=";"


Line=115, token=IDENTIFIER, value="double"

Line=115, token=OPERATOR, value="="

Line=115, token=FLOAT, value="3.14"
```

```
       Line=115, token=DELIMITER, value=";"


       Line=116, token=IDENTIFIER, value=" const "

       Line=116, token=IDENTIFIER, value=" int "

       Line=116, token=IDENTIFIER, value="z"

       Line=116, token=OPERATOR, value="="

       Line=116, token=INTEGER, value="5"

       Line=116, token=DELIMITER, value=";"


       Line=117, token=IDENTIFIER, value=" struct "

       Line=117, token=IDENTIFIER, value="Employee"

       Line=117, token=DELIMITER, value=";"


       Line=118, token=IDENTIFIER, value="if"

       Line=118, token=IDENTIFIER, value="x"

       Line=118, token=OPERATOR, value="=="

       Line=118, token=INTEGER, value="0"


       Line=119, token=IDENTIFIER, value="else"

       Line=119, token=IDENTIFIER, value="x"

       Line=119, token=OPERATOR, value=" != "

       Line=119, token=INTEGER, value="0"


       Line=120, token=IDENTIFIER, value="return"

       Line=120, token=INTEGER, value="0"

       Line=120, token=DELIMITER, value=";"
```

[Comment]

In the following examples the parser works correctly based on the naming conventions in the Uni language .

```
// ====== Test case #7 FINAL ======

        Line=123, token=IDENTIFIER, value="user"

        Line=123, token=DELIMITER, value=";"


        Line=124, token=UNKNOWN TOKEN, value="a =( b+c )"

        Line=124, token=DELIMITER, value=";"


        Line=125, token=STRING, value="" theo ""


        Line=126, token=INTEGER, value="45"


        Line=127, token=FLOAT, value="4.8"


//hi

        Line=129, token=FLOAT, value="4.7"


        Line=130, token=STRING, value=""good moring ""


        Line=131, token=FLOAT, value="4e2"


        Line=132, token=IDENTIFIER, value="good"
```

[Comment]

The parser does not recognize the break on line 141 as a keyword .

```
// ====== Test case #8 FINAL ======

        Line=135, token=STRING, value="" pao ""


/*20*/

        Line=138, token=FLOAT, value="8.5"
```

```
      Line=139, token=OPERATOR, value="+"


      Line=140, token=UNKNOWN TOKEN, value="+---"


      Line=141, token=IDENTIFIER, value="break"


      Line=142, token=IDENTIFIER, value=" func "
```

[Comment]

The parser results in the examples below are correct according to Uni - C recognition standards .

```
// ====== Test case #9 FINAL ======
      Line=145, token=UNKNOWN TOKEN, value="@#$"


      Line=146, token=UNKNOWN TOKEN, value="8 .3 "


      Line=147, token=UNKNOWN TOKEN, value="1_tzigger"


      Line=148, token=UNKNOWN TOKEN, value="/f"


      Line=149, token=UNKNOWN TOKEN, value="0.0"


      Line=150, token=FLOAT, value="5.001"
```

[Comment]

In line 158 the parser does not recognize the keyword “ while ”.

```
// ====== Test case #10 OPERATORS ======
      Line=153, token=IDENTIFIER, value="x"
      Line=153, token=OPERATOR, value=">="
      Line=153, token=INTEGER, value="5"
      Line=153, token=DELIMITER, value=";"
```

```
Line=154, token=INTEGER, value="5"

Line=154, token=OPERATOR, value="+"

Line=154, token=INTEGER, value="3"

Line=154, token=OPERATOR, value="="

Line=154, token=INTEGER, value="8"

Line=154, token=DELIMITER, value=";"


Line=155, token=UNKNOWN TOKEN, value="6/=3"

Line=155, token=DELIMITER, value=";"


Line=156, token=UNKNOWN TOKEN, value="x++"

Line=156, token=DELIMITER, value=";"


Line=157, token=UNKNOWN TOKEN, value="x*y"

Line=157, token=DELIMITER, value=";"


Line=158, token=IDENTIFIER, value="while"

Line=158, token=INTEGER, value="1"

Line=158, token=DELIMITER, value=";"


Line=159, token=IDENTIFIER, value="void"

Line=159, token=DELIMITER, value=";"


Line=160, token=IDENTIFIER, value="TRUE"

Line=160, token=OPERATOR, value="&&"

Line=160, token=IDENTIFIER, value="FALSE"

Line=160, token=OPERATOR, value="=="

Line=160, token=IDENTIFIER, value="FALSE"

Line=160, token=DELIMITER, value=";"


Line=161, token=IDENTIFIER, value="TRUE"
```

```
        Line=161, token=OPERATOR, value="||"

        Line=161, token=IDENTIFIER, value="TRUE"

        Line=161, token=OPERATOR, value="=="

        Line=161, token=IDENTIFIER, value="TRUE"

        Line=161, token=DELIMITER, value=";"


        Line=162, token=UNKNOWN TOKEN, value="y--"

        Line=162, token=DELIMITER, value=";"


        Line=163, token=UNKNOWN TOKEN, value="&bit"
```

[Comment]

The parser does not recognize the keywords “ switch ”, “ case ”, “ break ”, “ return ”.

```
// ====== Test case #11 CODE ======

        Line=166, token=IDENTIFIER, value="switch"

        Line=166, token=INTEGER, value="1"


        Line=167, token=IDENTIFIER, value="case"

        Line=167, token=INTEGER, value="1"


        Line=168, token=IDENTIFIER, value="break"

        Line=168, token=DELIMITER, value=";"


        Line=169, token=IDENTIFIER, value="case"

        Line=169, token=INTEGER, value="2"


        Line=170, token=IDENTIFIER, value="break"

        Line=170, token=DELIMITER, value=";"


        Line=171, token=IDENTIFIER, value="return"

        Line=171, token=INTEGER, value="0"
        Line=171, token=DELIMITER, value=";"
```

[Comment]

The verbal analyzer does not recognize keywords .

```
// ====== Test case #12 CODE ======

        Line=174, token=IDENTIFIER, value=" int "

        Line=174, token=IDENTIFIER, value="x"

        Line=174, token=OPERATOR, value="+"

        Line=174, token=IDENTIFIER, value="y"

        Line=174, token=OPERATOR, value="="

        Line=174, token=IDENTIFIER, value="z"

        Line=174, token=DELIMITER, value=";"


        Line=175, token=IDENTIFIER, value=" var "

        Line=175, token=IDENTIFIER, value="x"

        Line=175, token=OPERATOR, value="+"

        Line=175, token=IDENTIFIER, value="y"

        Line=175, token=OPERATOR, value="="

        Line=175, token=IDENTIFIER, value="z"

        Line=175, token=DELIMITER, value=";"


        Line=176, token=IDENTIFIER, value="for"

        Line=176, token=IDENTIFIER, value=" i "

        Line=176, token=OPERATOR, value="="

        Line=176, token=INTEGER, value="1"

        Line=176, token=DELIMITER, value=";"

        Line=176, token=IDENTIFIER, value=" i "

        Line=176, token=OPERATOR, value="<="

        Line=176, token=IDENTIFIER, value="N"

        Line=176, token=DELIMITER, value=";"

        Line=176, token=UNKNOWN TOKEN, value=" i ++"

        Line=176, token=IDENTIFIER, value="do"
```

```
Line=177, token=IDENTIFIER, value="if"

Line=177, token=IDENTIFIER, value=" i "

Line=177, token=OPERATOR, value="%"

Line=177, token=INTEGER, value="2"

Line=177, token=OPERATOR, value="=="

Line=177, token=INTEGER, value="0"

Line=177, token=IDENTIFIER, value="then"


Line=178, token=IDENTIFIER, value="return"

Line=178, token=INTEGER, value="0"

Line=178, token=DELIMITER, value=";"


Line=179, token=IDENTIFIER, value="else"


Line=180, token=IDENTIFIER, value="return"

Line=180, token=INTEGER, value="1"

Line=180, token=DELIMITER, value=";"


Line=181, token=IDENTIFIER, value="end"

Line=181, token=IDENTIFIER, value="if"

Line=181, token=DELIMITER, value=";"


Line=182, token=IDENTIFIER, value="end"

Line=182, token=IDENTIFIER, value="for"

Line=182, token=DELIMITER, value=";"
```

[Comment]

In line 185 the words “ pi ” (identifier), “=” (operator), “3.14” (float), due to the absence of a separator (e.g. space), are not recognized as valid Uni - C verbal units separately .

```
// ====== Test case #13 CODE ======

      Line=185, token=IDENTIFIER, value=" const "

      Line=185, token=IDENTIFIER, value="double"

      Line=185, token=UNKNOWN TOKEN, value="pi=3.14"

      Line=185, token=DELIMITER, value=";"


      Line=186, token=IDENTIFIER, value=" struct "

      Line=186, token=IDENTIFIER, value="Math"

      Line=186, token=IDENTIFIER, value="m1"

      Line=186, token=DELIMITER, value=";"


      Line=187, token=UNKNOWN TOKEN, value="m1.count"

      Line=187, token=OPERATOR, value="="

      Line=187, token=FLOAT, value="3.14"

      Line=187, token=OPERATOR, value="*"

      Line=187, token=INTEGER, value="2"

      Line=187, token=DELIMITER, value=";"


      Line=188, token=UNKNOWN TOKEN, value="m1.length"

      Line=188, token=OPERATOR, value="="

      Line=188, token=FLOAT, value="3.14"

      Line=188, token=OPERATOR, value="+"

      Line=188, token=INTEGER, value="2"

      Line=188, token=OPERATOR, value="/"

      Line=188, token=INTEGER, value="5"

      Line=188, token=DELIMITER, value=";"
```

[Comment]

The same error with the recognition of keywords as identifiers .

```
// ====== Test case #14 CODE/COMMENTS ======
       Line=191, token=IDENTIFIER, value=" func "
       Line=191, token=IDENTIFIER, value=" calculate_perimeter_of_table "
       Line=191, token=IDENTIFIER, value="void"


       Line=192, token=IDENTIFIER, value=" int "
       Line=192, token=IDENTIFIER, value="length"
       Line=192, token=OPERATOR, value="="
       Line=192, token=INTEGER, value="5"
       Line=192, token=DELIMITER, value=";"
// Length of table
       Line=193, token=IDENTIFIER, value=" int "
       Line=193, token=IDENTIFIER, value="width"
       Line=193, token=OPERATOR, value="="
       Line=193, token=INTEGER, value="10"
       Line=193, token=DELIMITER, value=";"
// Width of table
       Line=194, token=IDENTIFIER, value=" int "
       Line=194, token=IDENTIFIER, value="x"
       Line=194, token=DELIMITER, value=";"



       Line=196, token=IDENTIFIER, value="x"
       Line=196, token=OPERATOR, value="="
       Line=196, token=IDENTIFIER, value="length"
       Line=196, token=OPERATOR, value="*"
       Line=196, token=INTEGER, value="2"
       Line=196, token=OPERATOR, value="+"
       Line=196, token=IDENTIFIER, value="width"
       Line=196, token=OPERATOR, value="*"
       Line=196, token=INTEGER, value="2"
       Line=196, token=DELIMITER, value=";"

// Perimeter of table

       Line=197, token=IDENTIFIER, value="return"

       Line=197, token=IDENTIFIER, value="x"

       Line=197, token=DELIMITER, value=";"


       Line=198, token=END_OF_FILE, value=""
```

5. Problems and solutions

We encountered a problem mainly in the priority order of the rules in the FLEX code . Specifically, the newline character (\ n), the parser recognized and returned as NEWLINE token and as UNKNOWN TOKENS . The rules had the following order and structure.

```
NEWLINE [\n]

UNKNOWN.
```

Knowing that the UNKNOWN expression recognizes any character other than a newline, we were very concerned that it also returned the character as UNKNOWN TOKENS .

The problem was fixed by replacing the regular expression of the UNKNOWN rule with the following

```
[^ \t\r\n;]+
```

The expression recognizes any character other than the escape characters tab, space, newline, newline, and semicolon. Even though we're running in a Linux terminal , we've also used \ r for line breaks.

Another problem we faced was in the recognition of keywords as seen in the comments of the control cases. A workaround was not found due to limited time to implement this part. Keywords are recognized as identifiers.

6. Express reporting of deficiencies

The report includes, with deficiencies, what is required:

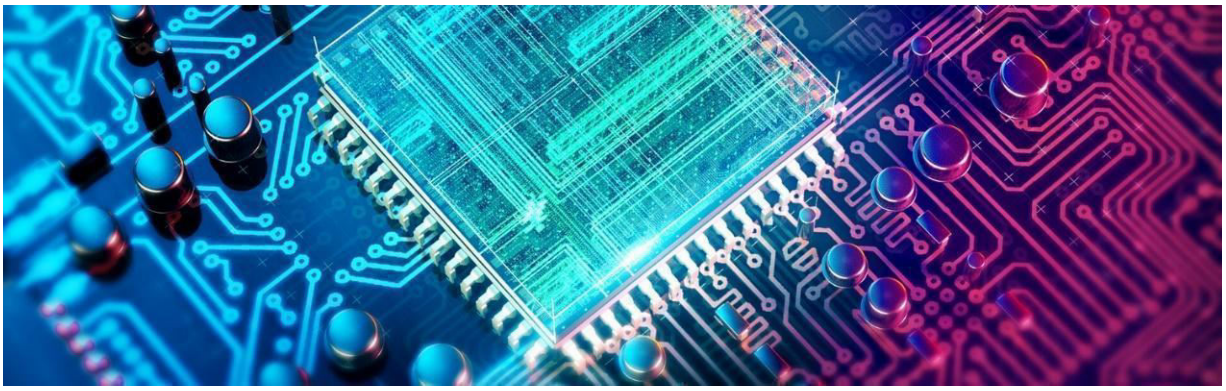
- Table with tokens
- Code FLEX
- Control cases
- Commenting on results
- Problems we faced and ways to deal with them

The code was run in a Windows environment with a Windows installation Subsystem Linux (WSL). The terminal the team ran the code on was on Ubuntu 22.04 LTS , so the commands we used to run the code are as follows:

```
make
```

```
modifier_ob.select=1
bpy.context.scene.objects.active = modifier_ob
print("Selected" + str(modifier_ob)) # modifier
```

Thank you for your attention.



COMPILERS