CENTRO *UNIVERSITÁRIO* DE BARRA MANSA
ACADEMIC PRO-RECTORY

COMPUTER ENGINEERING COURSE

A SYNTACTIC ANALYZER BUILT WITH THE CASE TOOLS LEX AND YACC

By:

Leniel Braz de Oliveira Macaferi
Wellington Magalhães Leite

Barra Mansa
April 18th, 2006

CENTRO *UNIVERSITÁRIO* DE BARRA MANSA
ACADEMIC PRO-RECTORY

COMPUTER ENGINEERING COURSE

A SYNTACTIC ANALYZER BUILT WITH THE CASE TOOLS LEX AND YACC

By:

Leniel Braz de Oliveira Macaferi
Wellington Magalhães Leite

Paper presented to the Computer Engineering course at Centro Universitário de Barra Mansa, as a partial requisite to the obtention of the first grade pertaining the Compilers Construction discipline, under prof. José Nilton Cantarino Gil supervision.

Barra Mansa
April 18th, 2006

ABSTRACT

The function of a syntactic analyzer in a compiler is to verify the syntactic structure of a program's source code. It then detects, signalize and handle the syntactic errors found in the source code and at the same time servers as the framework for the front-end (user interface) of the program. So, its construction helps with the familiarization regarding the tasks included in the project of a compiler.

The language used in this paper does not have left recursion. The language neither presents subprograms, nor indexed variables and its only loop command is while for the sake of simplicity. The syntactic analyzer implemented uses the bottom-up parsing approach.

This paper presents the steps to the construction of a syntactic analyzer, which serves as a base element for a compiler implementation.

Keywords: syntactic analyzer, syntactical analysis, compiler construction, case tools, flex, yacc

LIST OF FIGURES

LIST OF TABLES

CONTENTS

Page

1 INTRODUCTION

1.1 Objective

Our objective is to construct a syntactic analyzer for the proposed grammar (structured English).

1.2 Definition

1.2.1 Grammar

The grammar used in this paper does not have left recursion, but presents the so known ambiguity of conditional commands.

The language neither presents subprograms, nor indexed variables and its only loop command is while.

The non-terminal symbols are presented in *italic*.

The terminal symbols are atoms obtained from the lexical analyzer; they are presented in UPPER case letters, or with the characters that indentify them.

1.2.1.1 Grammar productions

#	**Terminals**		
1	<*prog*>	→	PROGRAM ID SEMIC <*decls*> <*compcmd*> DOT
2	<*decls*>	→	λ | VAR <*decl_list*>
3	<*decl_list*>	→	<*decl_type*> | <*decl_type*> <*decl_list*>
4	<*decl_type*>	→	<*id_list*> COLON <*type*> PVIRG
5	<*id_list*>	→	ID | ID COMMA <*id_list*>
6	<*type*>	→	INTEGER | REAL | BOOL
7	<*compcmd*>	→	OCBRA <*cmd_list*> CCBRA
8	<*cmd_list*>	→	<*cmd*> | <*cmd*> SEMIC <*cmd_list*>
9	<*cmd*>	→	<*If_cmd*> | <*While_cmd*> | <*Read_cmd*> | <*Write_cmd*> | <*Atrib_cmd*> | <*compcmd*>
10	<*If_cmd*>	→	IF <*exp*> THEN <*cmd*> | IF <*exp*> THEN <*cmd*> ELSE <*cmd*>
11	<*While_cmd*>	→	WHILE <*exp*> DO <*cmd*>
12	<*Read_cmd*>	→	READ OPAR <*id_list*> CPAR
13	<*Write_cmd*>	→	WRITE OPAR <*w_list*> CPAR
14	<*w_list*>	→	<*w_ elem*> | <*w_elem*> COMMA <*w_list*>
15	<*w_elem*>	→	<*exp*> | STRING
16	<*Atrib_cmd*>	→	ID ATRIB <*exp*>
17	<*exp*>	→	<*simple_e*> | <*simple_exp*> RELOP <*simple_exp*>
18	<*simple_exp*>	→	<*term*> | <*term*> ADDOP <*simple_exp*>
19	<*term*>	→	<*fac*> | <*fac*> MULTOP <*term*>
20	<*fac*>	→	ID | CONS | RCONS | OPAR <*exp*> CPAR | TRUE | FALSE | NEGOP <*fac*>

Table 1 - Grammar productions

1.2.1.2 Lexical speceficiations

Each atom has at least an attribute called type.

According to the atom type, it will be capable of having other attributes.

If the atom is a reserved word (keyword), its type is the keyword and there is no other attribute.

<table>
<tr><th>#</th><th>Type</th><th></th><th>Syntax</th></tr>
<tr><td>1</td><td>ID</td><td>→</td><td>letter (letter | digit)*</td></tr>
<tr><td>2</td><td>CONS</td><td>→</td><td>digit+</td></tr>
<tr><td>3</td><td>RCONS</td><td>→</td><td>digit* DOT digit+</td></tr>
<tr><td>4</td><td>STRING</td><td>→</td><td>Any string surrounded by “double quotation marks”</td></tr>
<tr><td>5</td><td>RELOP</td><td>→</td><td>< | > | <= | >= | != | ==</td></tr>
<tr><td>6</td><td>ADDOP</td><td>→</td><td>OR | + | -</td></tr>
<tr><td>7</td><td>MULTOP</td><td>→</td><td>AND | * | /</td></tr>
<tr><td>8</td><td>NEGOP</td><td>→</td><td>NOT | !</td></tr>
<tr><td>9</td><td>ATRIB</td><td>→</td><td>=</td></tr>
<tr><td>10</td><td>OPAR</td><td>→</td><td>(</td></tr>
<tr><td>11</td><td>CPAR</td><td>→</td><td>)</td></tr>
<tr><td>12</td><td>OCBRA</td><td>→</td><td>{</td></tr>
<tr><td>13</td><td>CCBRA</td><td>→</td><td>}</td></tr>
<tr><td>14</td><td>COMMA</td><td>→</td><td>,</td></tr>
<tr><td>15</td><td>SEMIC</td><td>→</td><td>;</td></tr>
<tr><td>16</td><td>DOT</td><td>→</td><td>.</td></tr>
</table>

Table 2 - Lexical specifications

Note:

* means 0 or more incidences of the atom

+ means at least 1 incidence of the atom

1.2.1.3 Reserved words or keywords

#	Word or keyword
1	PROGRAM
2	VAR
3	INTEGER
4	REAL
5	BOOL
6	IF
7	THEN
8	ELSE
9	WHILE
10	DO
11	READ
12	WRITE
13	TRUE
14	FALSE

Table 3 - Reserved words or keywords

If the atom is an identifier, its type is ID and the other attribute is the string of its characters; its syntax is: letter (letter | digit)*

Integer constants have a type of CONS and the other attribute is its integer value.

Character strings come surrounded by “double quotation marks”. Their type is STRING and the other attribute is the content of the string without the double quotation marks.

1.2.1.4 Operator types and attributes

#	Atom	Type	Attribute
1	+	ADDOP	PLUS
2	-	ADDOP	MINUS
3	or	ADDOP	OR
4	*	MULTOP	TIMES
5	/	MULTOP	DIV
6	and	MULTOP	AND
7	!	NEGOP	NEG
8	not	NEGOP	NOT
9	<	RELOP	LESS
10	<=	RELOP	LESSEQ
11	>	RELOP	GREATER
12	>=	RELOP	GREATEQ
13	==	RELOP	EQUAL
14	!=	RELOP	DIFF

Table 4 - Operator types and attributes

Note:

ADDOP stands for additive operator.

MULTOP stands for multiplicative operator.

NEGOP stands for negative operator.

RELOP stands for relational operator.

1.2.1.5 Separator types

The separators don’t have other attributes.

#	Atom	Type
1	;	SEMIC
2	.	DOT
3	:	COLON
4	,	COMMA
5	(	OPAR
6	)	CPAR
7	=	ATRIB
8	{	OCBRA
9	}	CCBRA

Table 5 - Separator types

The language doesn’t have comments.

Blank spaces between atoms are optional, except reserved words that can’t be concatenated with others and with identifiers and integer constants.

2 DEVELOPMENT

2.1 Lexical analysis

Groups the program’s characters into atoms.

Verifies if the atom is valid.

Classify the valid atoms, passing to them their attributes.

2.1.1 Sample source code of a factorial program

```
PROGRAM factorial;

VAR n, fa, i: INTEGER;
{
  READ(n);

  fa = 1;
  i = 1;

  WHILE i <= n DO
  {
    fa = fa * i;
    i = i + 1
  };

  WRITE(“The factorial of ”, n, “ is: ”, fat)
}.
```

The following table shows the atoms and attributes gathered from the source code above.

#	Atom	Type	Attribute
1	PROGRAM	PROGRAM	
2	factorial	ID	factorial
3	;	SEMIC	
4	VAR	VAR	
5	n	ID	n
6	,	COMMA	
7	fa	ID	fa
8	,	COMMA	
9	i	ID	i
10	:	COLON	
11	INTEGER	INTEGER	
12	;	SEMIC	
13	{	OCBRA	
14	READ	READ	
15	(	OPAR	
16	n	ID	n
17	)	CPAR	
18	;	SEMIC	
19	fa	ID	fa
20	=	ATRIB	
21	1	CONS	1
22	;	SEMIC	
23	i	ID	i
24	=	ATRIB	
25	1	CONS	1
26	;	SEMIC	
27	WHILE	WHILE	
28	i	ID	i

#	Atom	Type	Attribute
29	<=	RELOP	LESSEQ
30	n	ID	n
31	DO	DO	
32	{	OCBRA	
33	fa	ID	fa
34	=	ATRIB	
35	fa	ID	fa
36	*	MULTOP	TIMES
37	i	ID	i
38	;	SEMIC	
39	i	ID	i
40	=	ATRIB	
41	i	ID	i
42	+	ADDOP	PLUS
43	1	CONS	1
44	}	CCBRA	
45	;	SEMIC	
46	WRITE	WRITE	
47	(	OPAR	
48	“The factorial of ”	STRING	The factorial of_
49	,	COMMA	
50	n	ID	n
51	,	COMMA	
52	“ is: ”	STRING	_is:_
53	,	COMMA	
54	fa	ID	fa
55	)	CPAR	
56	}	CCBRA	

#	Atom	Type	Attribute
57	.	DOT	

Table 6 - Atoms and attributes from the sample source code

2.1.2 Flex

Considerations about the Flex tool [1]:

- Generates a lexical analyzer. Gera um analisador léxico. To that end it receives as input regular expressions in a specific notation and produces a finite automata, (transitions diagram);
- Has been frequently used in the UNIX system to create lexical analyzers for a vast variety of languages;
- Is a version of Lex for DOS;
- To work with Flex, it’s necessary to prepare a specification of the desired lexical analyzer, creating a program written in the Lex language and saving the same in a file with the .lex extension, lexan.lex, for example.

2.2 Syntactical analysis

Verifies the syntactical structure of a program.

Serves as the framework for: semantic analysis, construction of the symbols table and generation of intermediate code.

2.2.1 Sample syntactic tree

The syntactic tree extracted from the factorial program (section 2.1.1).

The tree is divided in 6 parts for better fitting in this document. We used the Microsoft Office Visio to build it.

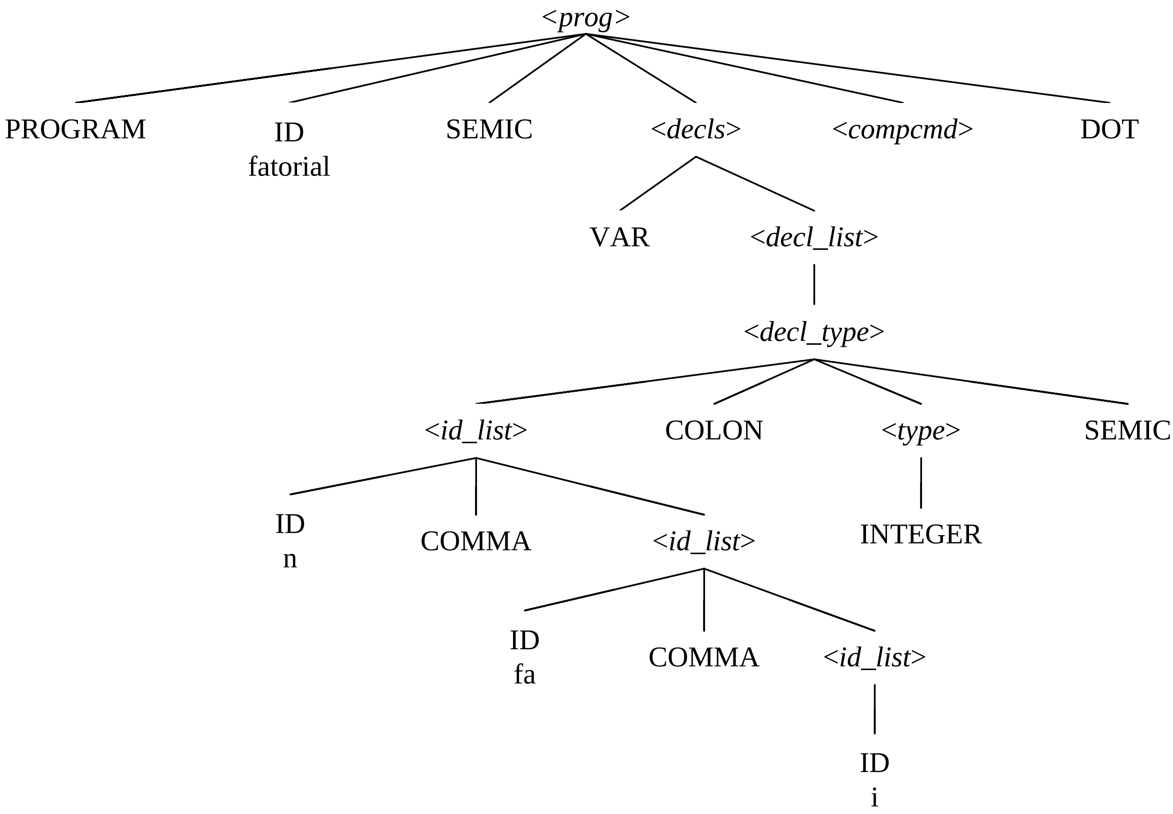

Figure 1 - Syntactic tree - part 1

The terminal <*compcmd*> is the next to be expanded.

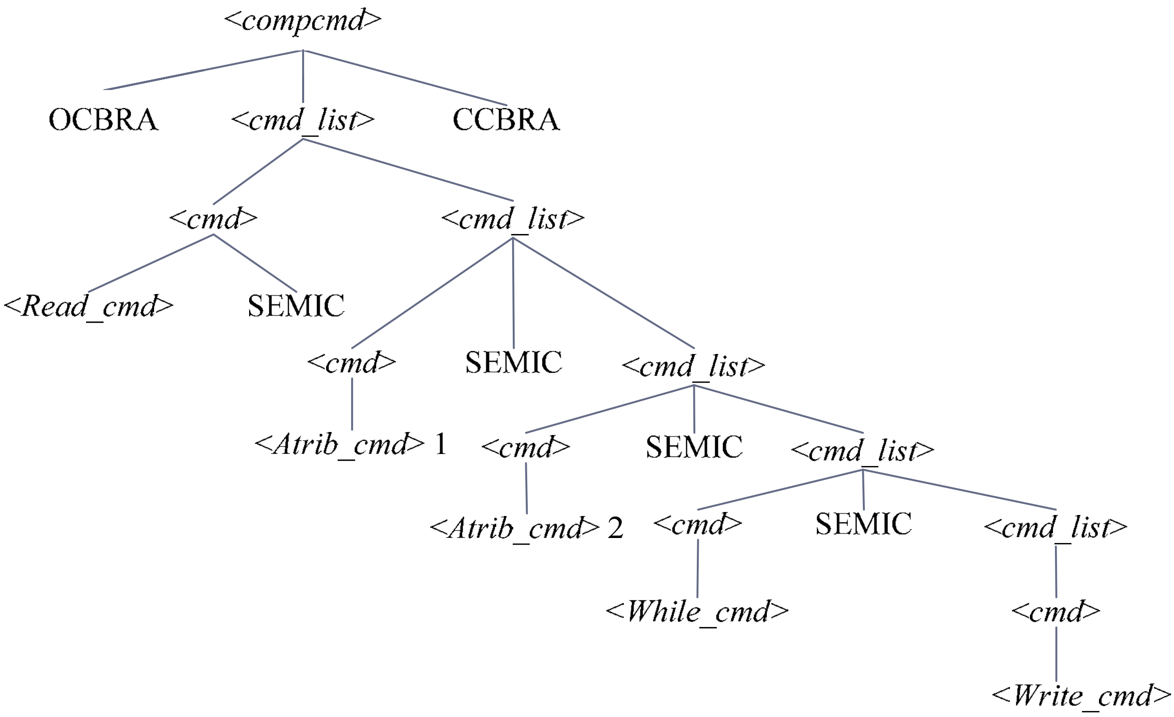

Figure 2 - Syntactic tree - part 2

The terminals <*Read_cmd*>, <*Atrib_cmd*> 1, <*Atrib_cmd*> 2, <*While_cmd*> and <*Write_cmd*> are the next to be expanded.

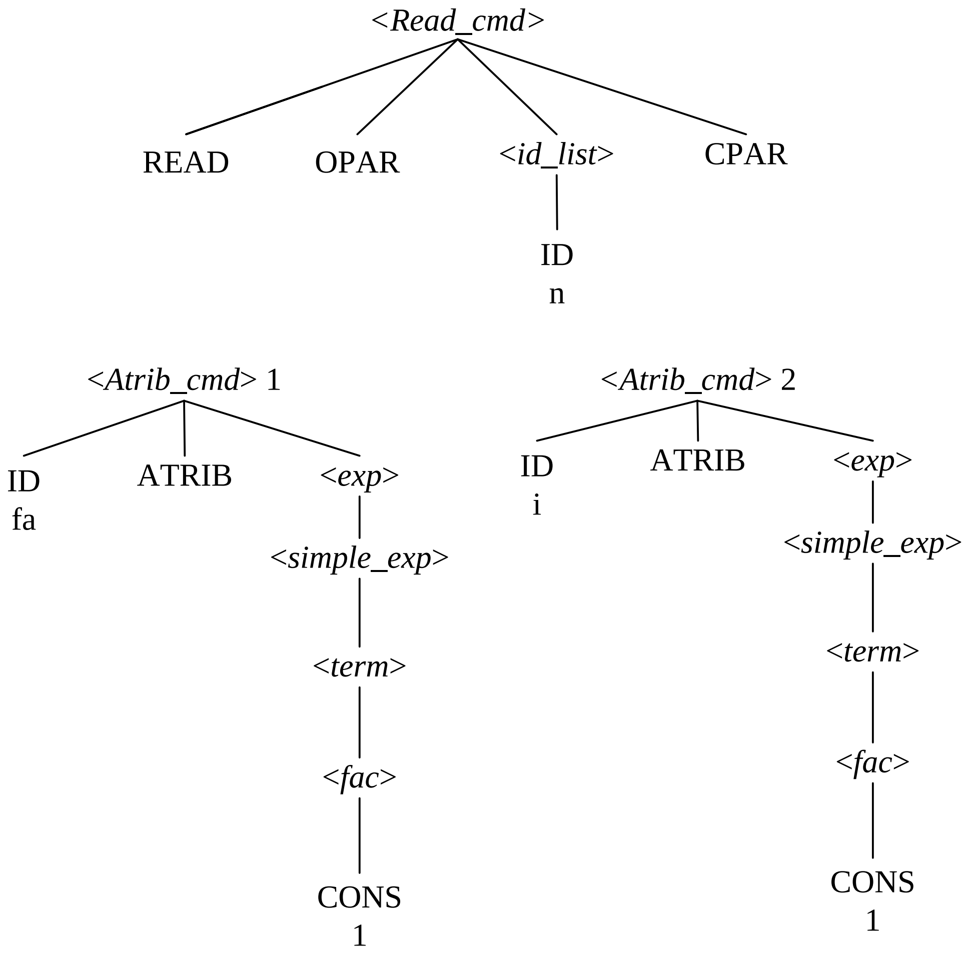

Figure 3 - Syntactic tree - part 3

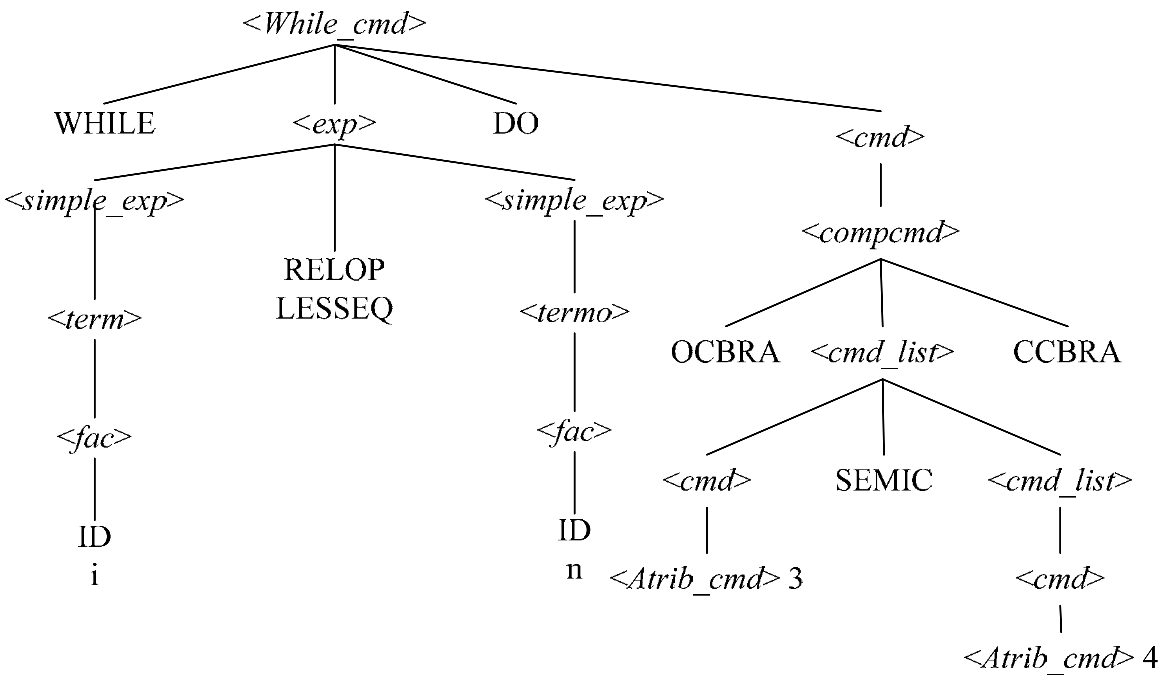

Figure 4 - Syntactic tree - part 4

The terminals <*Atrib_cmd*> 3 e <*Atrib_cmd*> 4 are the next to be expanded.

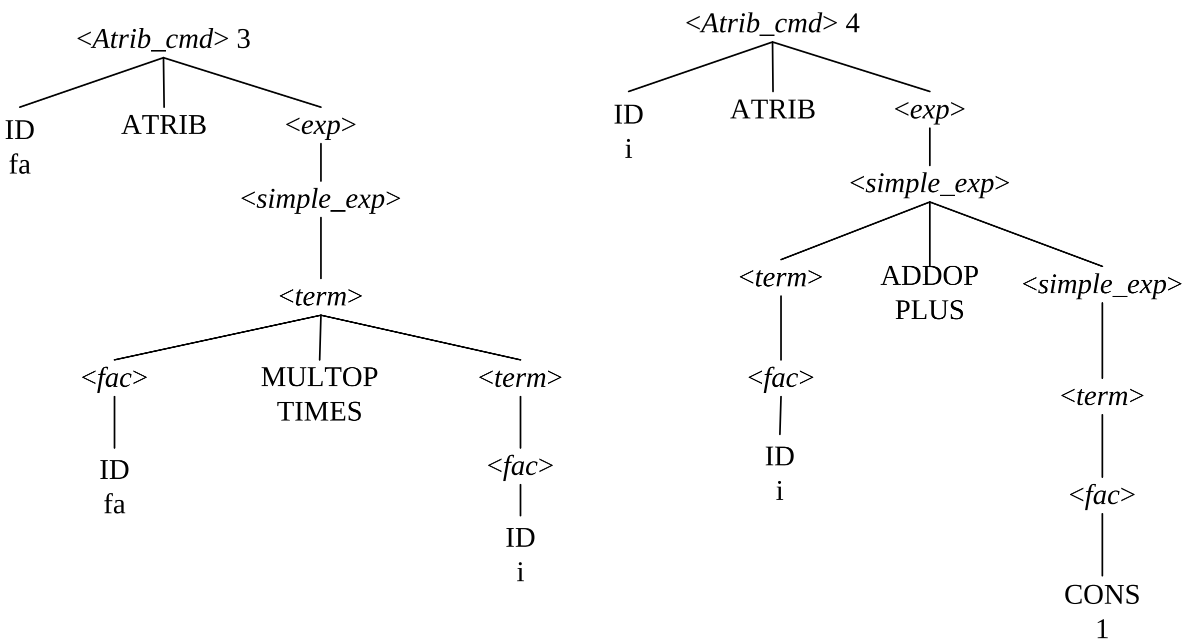

Figure 5 - Syntactic tree - part 5

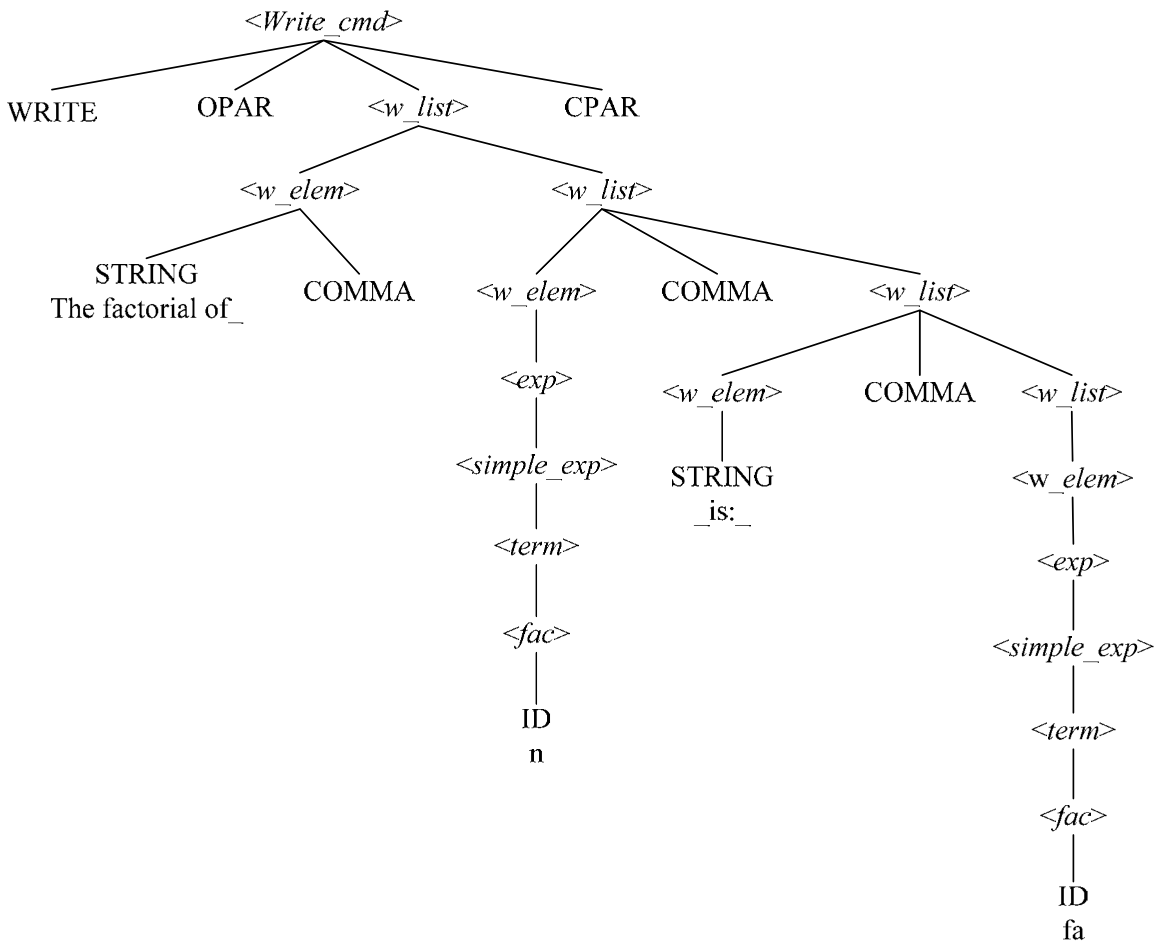

Figure 6 - Syntactic tree - part 6

2.2.2 YACC

Considerations about the YACC tool [2]:

- Generates a syntactic analyzer. To that end it receives as input the specification of a context-free grammar (CFG) [3] and produces a syntactic analyzer for such a grammar.
- YACC (Yet Another Compiler Compiler) has been used to implement hundreds of compiles for the UNIX system.
- The syntactic analyzer created by YACC performs a bottom-up parsing technique [4] and uses the Look Ahead Left Recursive (LALR) parser method [5].

To work with YACC, it’s necessary to prepare the specification of the desired syntactic analyzer, creating a program written in the YACC language and saving the same in a file with a .yacc extension, sinan.yacc, for example.

3 APPLICATION

3.1 Constructing the file for the lexical analysis (lexan.lex)

```
%{
%}
delim                                   [ \t\n\r]
ws                                      {delim}+
digit                                   [0-9]
cons                                    {digit}+
letter                                  [A-Za-z_]
id                                      {letter}({letter}|{digit})*
dot                                     [.]
%%
{ws}                                    {;}
";"                                     {return SEMIC;}
[\.]                                    {return DOT;}
var                                     {return VAR;}
":"                                     {return COLON;}
integer                                 {return INTEGER;}
real                                    {return REAL;}
{digit}*{dot}{digit}+                   {return RCONS;}
bool                                    {return BOOL;}
"{"                                     {return OCBRA;}
"}"                                     {return CCBRA;}
if                                      {return IF;}
then                                    {return THEN;}
else                                    {return ELSE;}
while                                   {return WHILE;}
do                                      {return DO;}
read                                    {return READ;}
"("                                     {return OPAR;}
")"                                     {return CPAR;}
write                                   {return WRITE;}
","                                     {return COMMA;}
[\"][^\"]*[\"]                          {return STRING;}
"="                                     {return ATRIB;}
"<"|">"|"<="|">="|"!="|"=="             {return RELOP;}
or|"+"|"-"                              {return ADDOP;}
and|"*"|"/"                             {return MULTOP;}
not|"!"                                 {return NEGOP;}
{cons}                                  {return CONS;}
true                                    {return TRUE;}
false                                   {return FALSE;}
program                                 {return PROGRAM;}
{id}                                    {return ID;}
%%
```

3.2 Constructing the file for the syntactic analysis (sinan.yacc)

```
%{
   #include <stdio.h>
   #include <stdlib.h>
%}

%token PROGRAM
%token ID
%token SEMIC
```

```
%token DOT
%token VAR
%token COLON
%token INTEGER
%token REAL
%token RCONS
%token BOOL
%token OCBRA
%token CCBRA
%token IF
%token THEN
%token ELSE
%token WHILE
%token DO
%token READ
%token OPAR
%token CPAR
%token WRITE
%token COMMA
%token STRING
%token ATRIB
%token RELOP
%token ADDOP
%token MULTOP
%token NEGOP
%token CONS
%token TRUE
%token FALSE

%%

prog : PROGRAM ID SEMIC decls compcmd DOT
       {
         printf("\n Syntactical Analisys done without erros!\n");

         return 0;
       }
     ;

decls : VAR decl_list
      |
      ;

decl_list : decl_list decl_type
          | decl_type
          ;

decl_type : id_list COLON type SEMIC
          ;

id_list : id_list COMMA ID
        | ID
        ;

type : INTEGER
     | REAL
     | BOOL
     ;

compcmd : OCBRA cmd_list CCBRA
        ;
```

```
cmd_list : cmd_list SEMIC cmd
         | cmd
         ;

cmd : If_cmd
    | While_cmd
    | Read_cmd
    | Write_cmd
    | Atrib_cmd
    | compcmd
    ;

If_cmd : IF exp THEN cmd
       | IF exp THEN cmd ELSE cmd
       ;

While_cmd : WHILE exp DO cmd
          ;

Read_cmd : READ OPAR id_list CPAR
         ;

Write_cmd : WRITE OPAR w_list CPAR
          ;

w_list : w_list COMMA w_elem
       | w_elem
       ;
w_elem : exp
       | STRING
       ;

Atrib_cmd : ID ATRIB exp
          ;

exp : simple_exp
    | simple_exp RELOP simple_exp
    ;

simple_exp : simple_exp ADDOP term
           | term
           ;

term : term MULTOP fac
     | fac
     ;

fac : fac NEGOP
    | CONS
    | RCONS
    | OPAR exp CPAR
    | TRUE
    | FALSE
    | ID
    ;

%%

#include "lex.yy.c"
```

3.3 Guide to implementation

To execute the following steps you’ll need the CASE tools [6] Flex and Yacc and a C compiler. In this paper we used the Minimalist GNU for Windows or just MinGW [7] that is a native software port of the GNU Compiler Collection (GCC) [8] to Microsoft Windows. Flex, Yacc and the C compiler are included in MinGW. If you want you can also use the DJGPP [9] development suite.

These are the steps we used to create the syntactic analyzer:

1st: Create a folder called CompCons on the root directory C:\, which will contain all the files created henceforward.

2nd: Open a command prompt and type: path=C:\MinGW\bin;%PATH%. We consider that the MinGW installation was done in the folder C:\MinGW. Change it accordingly. After completing this step the tools will be available in the command prompt.

3rd: Construct the lexical analysis file (lexan.lex), as done in section 3.1.

4th: Generate the file lex.yy.c in the command prompt with the following command: flex -i lexan.lex. Figure 7 shows this step:

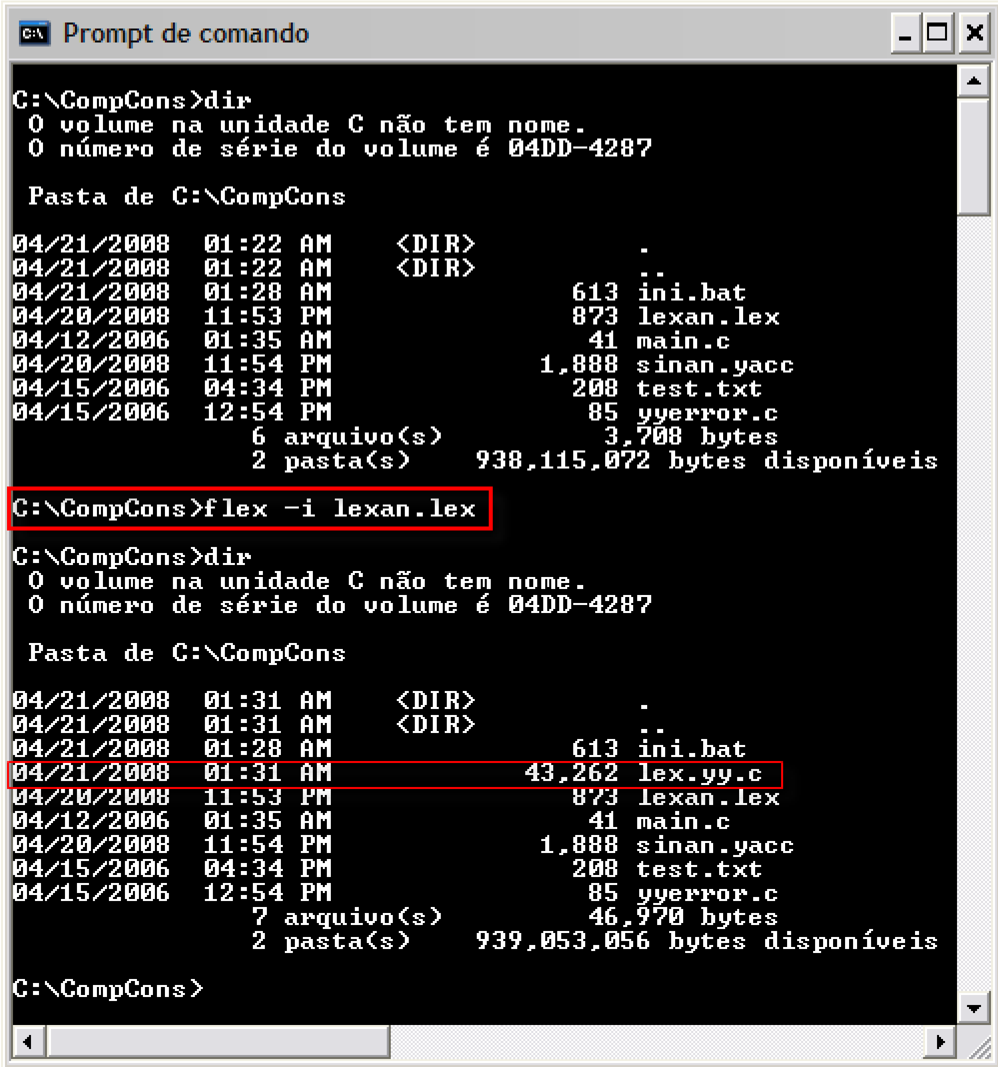

Figure 7 - Generating the file lex.yy.c in the command prompt

5th: Construct the syntactic analysis file (sinan.yacc), as done in section 3.2. Include the file lex.yy.c at the end of the sinan.yacc file with an include directive;

6th: Generate the file y.tab.c in the command prompt with following command: yacc sinan.yacc. Figure 8 shows this step:

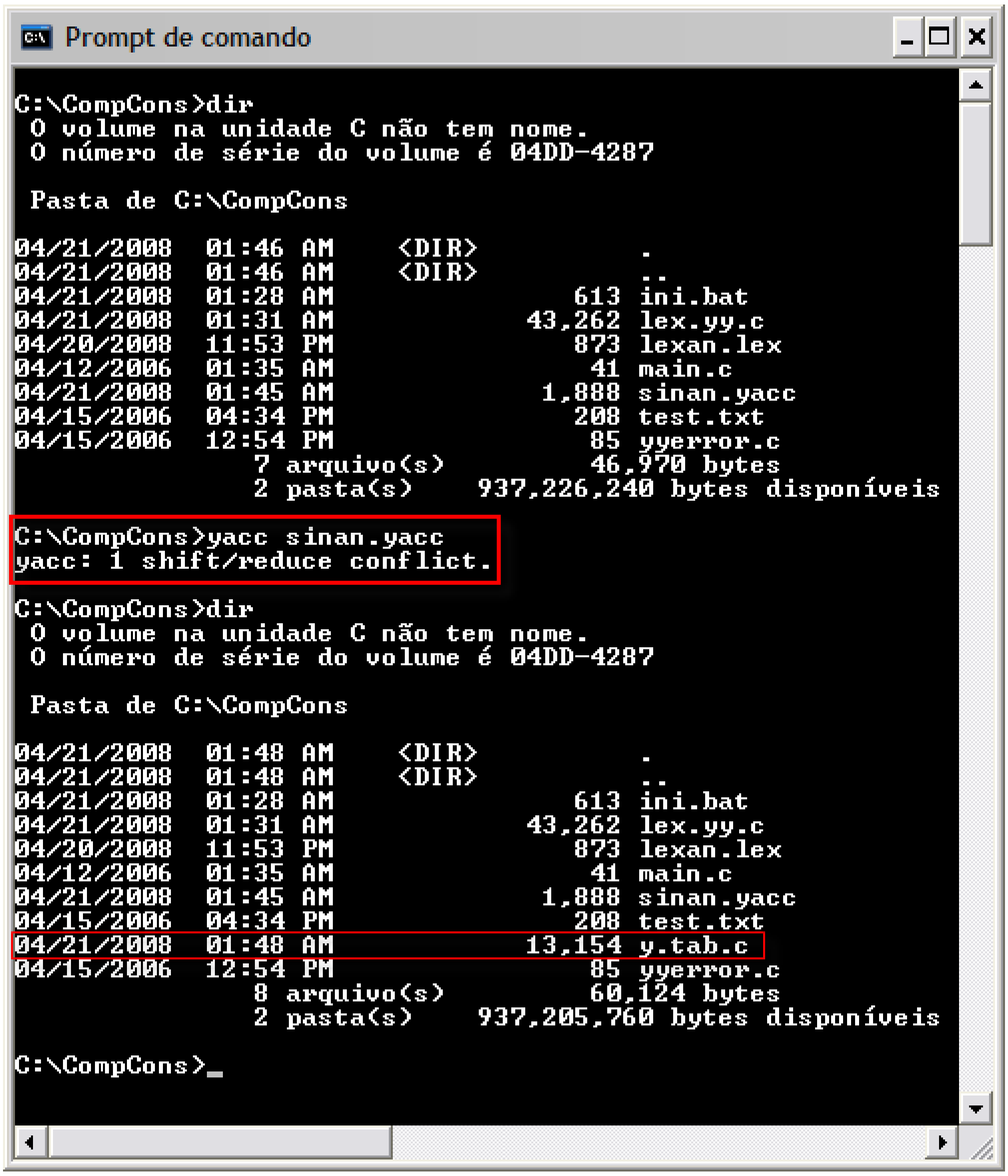

Figure 8 - Generating the file y.tab.c in the command prompt

7th: Compile the files with GCC [8] in the command prompt with the following command: gcc y.tab.c yyerror.c main.c -osinan -lfl. Figure 9 shows this step:

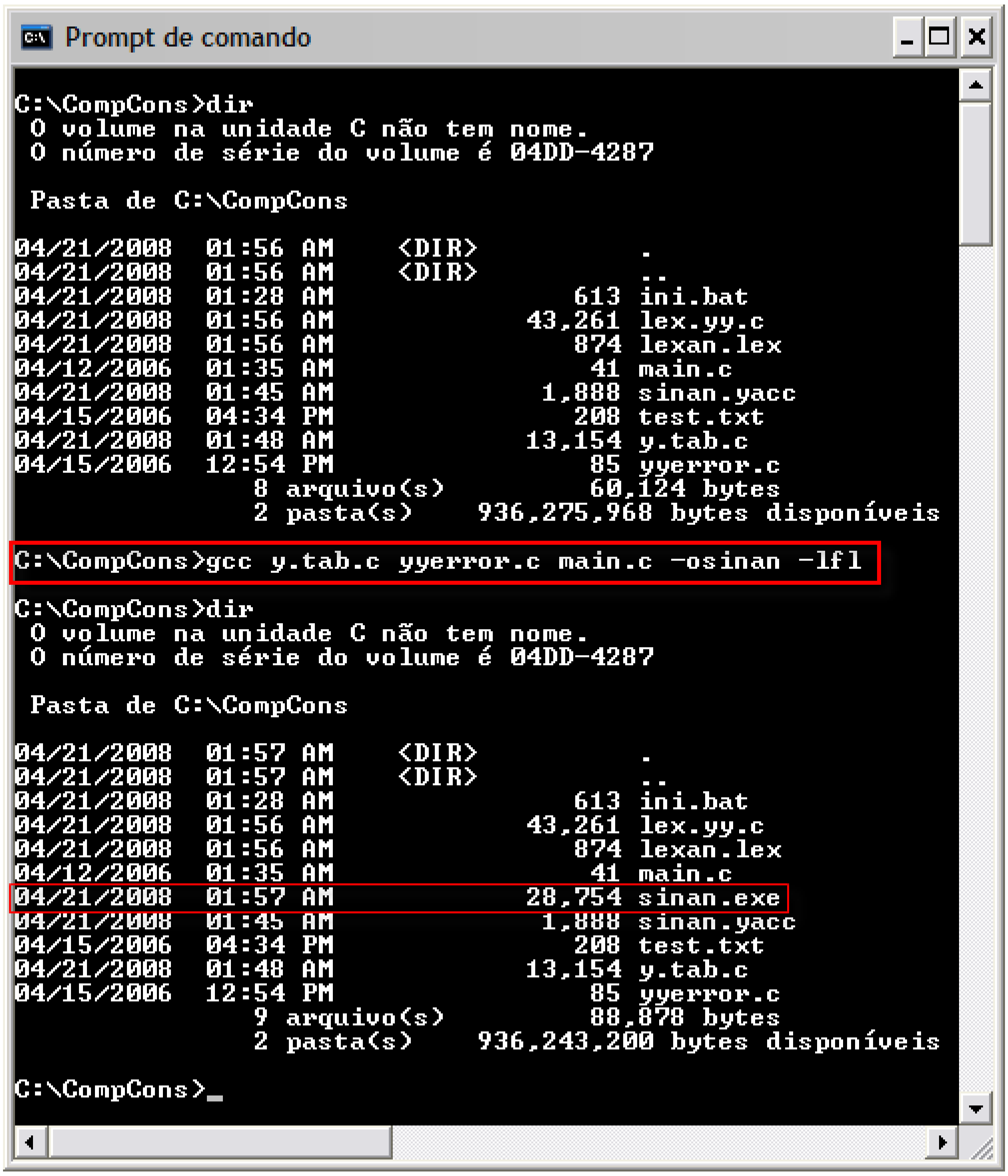

Figure 9 - Compiling the files with GCC

8th: The result file for the syntactic analyzer is sinan.exe. To use it just type sinan < test.txt. The file test.txt contains the source code to be analyzed by the syntactic analyzer. Figure 10 shows the execution of the syntactic analyzer

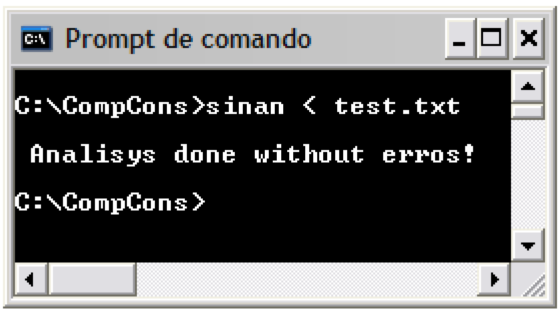

Figure 10 - Executing the syntactic analyzer

As can be seen this time there was no error with the test source code.

In the Addendum section we make available two test cases in which we proves the efficacy of the syntactic analyzer developed for this paper.

Notes:

- The parameter -i used in the command prompt for the command “flex -i lexan.lex” signalize the non differentiation of lower and upper letters. This will take effect when a file is being analyzed by the lexical analyzer;
- The files yyerror.c and main.c are auxiliary ones used and their content is included in the Addendum section;
- The parameter -lfl used in the command prompt for the command “gcc y.tab.c yyerror.c main.c -osinan -lfl” signalize the inclusion of a library that has to do with Flex.

3.4 Using a batch file to avoid boilerplate work

We also created a batch file [10] called ini.bat to add the MinGW bin folder to the OS environment variables path.

This batch file can be used to execute the steps shown in the previous section with only one command, that is, it’s not necessary to execute the commands one by one. It helps a lot while testing the source codes. Although this feature is offered, we decided to go with the step by step approach in the paper to show in a more detailed way what happens during the creation process of a syntactic analyzer.

To execute the batch file just open a command prompt and type: ini. After that we have three options as show in the following Figure 11:

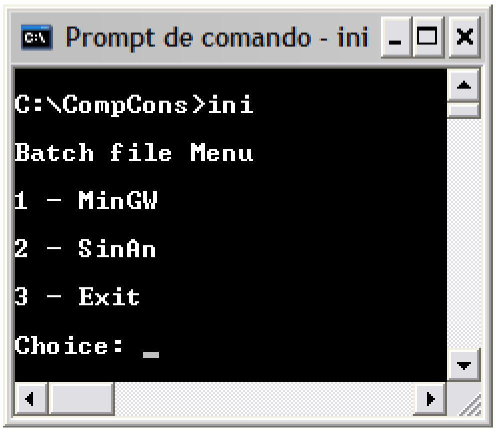

Figure 11 - Executing the ini.bat batch file

Option 1 only adds the MinGW bin folder to the operating system variables path. If this option is selected, it’ll be necessary to execute the steps 4, 6 and 7 of section 3.3.

Option 2 goes further and processes the above described steps even performing a test case with a file called test.txt that has as its content the source code to be analyzed by the syntactic analyzer.

Option 3 only exits the batch execution.

4 CONCLUSION

Developing this paper we took more one important step in our study about compilers construction, even taking into consideration the fact that the language neither has subprograms, nor indexed variables and that it only has the while loop command.

We could verify some significant points, as for example the fact that the syntactic analyzer has a better performance with grammars that don't have left recursion, the ambiguity case regarding the if else command, etc.

With respect to the acquired results, we cite the test cases executed with the "test.txt" file, which contains the factorial program from section 2.1.1. The first test case was done with errors in the source code regarding the PROGRAM keyword and the second one had no errors in the source code. The results can be checked in the Addendum section of this paper.

This paper gives us an overview of how we must proceed in more complex cases, for it helps us regarding the activities employed in the development of a syntactic analyzer project. This way we can master one of the steps involved with the compiler construction process, which showed to be really valuable for a computer engineer.

This paper and the syntactic analyzer files can be downloaded at:
http://lenielmacaferi.blogspot.com/2008/04/syntactic-analyzer-built-with-lex-yacc.html

5 REFERENCES

[1] **flex lexical analyser**. Wikipedia. Available at <http://en.wikipedia.org/wiki/Flex_lexical_analyser>. Accessed April 20, 2008.

[2] **Yacc**. Wikipedia. Available at <http://en.wikipedia.org/wiki/Yacc>. Accessed April 20, 2008.

[3] **Context-free grammar**. Wikipedia. Available at <http://en.wikipedia.org/wiki/Context-free_grammar>. Accessed April 20, 2008.

[4] **Bottom-up parsing**. Wikipedia. Available at <http://en.wikipedia.org/wiki/Bottom-up_parsing>. Accessed April 20, 2008.

[5] **LALR parser**. Wikipedia. Available at <http://en.wikipedia.org/wiki/LALR>. Accessed April 20, 2008.

[6] **Case tool**. Wikipedia. Available at <http://en.wikipedia.org/wiki/CASE_tool>. Accessed April 21, 2008.

[7] **MinGW**. Wikipedia. Available at <http://en.wikipedia.org/wiki/MinGW>. Accessed April 21, 2008.

[8] **GNU Compiler Collection**. Wikipedia. Available at <http://en.wikipedia.org/wiki/GNU_Compiler_Collection>. Accessed April 21, 2008.

[9] **DJGPP**. Wikipedia. Available at <http://en.wikipedia.org/wiki/DJGPP>. Accessed April 21, 2008.

[10] **Batch file** . Wikipedia. Available at <http://en.wikipedia.org/wiki/.bat>. Accessed April 21, 2008.

[11] Makarzel, Fabio Carneiro. **Class notes about compilers**. Available at <http://www.comp.ita.br/professores/fabio.htm>. Accessed April 19, 2008.

[12] Aho, Alfred V., Sethi, Ravi and Ullman, Jeffrey D. **Compilers: Principles, Techniques and Tools**. Reading : Addison-Wesley, 1986.

[13] Holub, A.I. **Compilers Design in C**. Englewood Cliffs : Prentice Hall, 1990.

[14] Tremblay, Jean-Paul and Sorenson, Paul G. **The Theory and Practice of Compiler Writing**. Singapore : McGraw Hill, 1989.

[15] Neto, João José. **Introdução à Compilação**. Rio de Janeiro : Livros Técnicos e Científicos, 1987.

[16] C.N., Fischer and LeBlanc Jr, R. J. **Crafting a Compiler**. Menlo Park : Benjamin/Cummings, 1988.

[17] Rechenberg, Peter and Mössenböck, Hanspeter. **A Compiler Generator for Microcomputers**. Englewood Cliffs : Prentice-Hall, 1989.

6 ADDENDUM

```
int main()
{
  yyparse();

  return 0;
}
```

Content of the file main.c

```
#include <stdio.h>

int yyerror(char *msg)
{
  fprintf(stderr, "%s\n", msg);
}
```

Content of the file yyerror.c

```
Prompt de comando

C:\CompCons>dir
 O volume na unidade C não tem nome.
 O número de série do volume é 04DD-4287

 Pasta de C:\CompCons

04/21/2008  03:54 AM    <DIR>          .
04/21/2008  03:54 AM    <DIR>          ..
04/21/2008  03:41 AM               211 factorial.txt
04/21/2008  03:50 AM               620 ini.bat
04/21/2008  02:57 AM            43,261 lex.yy.c
04/21/2008  01:56 AM               874 lexan.lex
04/12/2006  01:35 AM                41 main.c
04/21/2008  03:54 AM            28,754 sinan.exe
04/21/2008  02:58 AM             1,900 sinan.yacc
04/21/2008  03:53 AM            13,156 y.tab.c
04/15/2006  12:54 PM                85 yyerror.c
               9 arquivo(s)         88,902 bytes
               2 pasta(s)    931,438,592 bytes disponíveis

C:\CompCons>type factorial.txt
PROGRAMA factorial;

VAR n, fa, i: INTEGER;
{
  READ(n);

  fa = 1;

  i = 1;

  WHILE i <= n DO
  {
    fa = fa * i;

    i = i + 1
  };

  WRITE("The factorial of ", n, " is: ", fa)
}.
C:\CompCons>sinan < factorial.txt
 Syntax error!

C:\CompCons>_
```

Result we got when the source code of the factorial program had an error regarding the PROGRAM keyword

Prompt de comando

```
C:\CompCons>dir
 O volume na unidade C não tem nome.
 O número de série do volume é 04DD-4287

 Pasta de C:\CompCons

04/21/2008  04:02 AM    <DIR>          .
04/21/2008  04:02 AM    <DIR>          ..
04/21/2008  03:58 AM               210 factorial.txt
04/21/2008  03:50 AM               620 ini.bat
04/21/2008  02:57 AM            43,261 lex.yy.c
04/21/2008  01:56 AM               874 lexan.lex
04/12/2006  01:35 AM                41 main.c
04/21/2008  04:02 AM            28,754 sinan.exe
04/21/2008  04:00 AM             1,900 sinan.yacc
04/21/2008  04:01 AM            13,166 y.tab.c
04/15/2006  12:54 PM                85 yyerror.c
               9 arquivo(s)         88,911 bytes
               2 pasta(s)    931,692,544 bytes disponíveis

C:\CompCons>type factorial.txt
PROGRAM factorial;

VAR n, fa, i: INTEGER;
{
  READ(n);

  fa = 1;

  i = 1;

  WHILE i <= n DO
  {
    fa = fa * i;

    i = i + 1
  };

  WRITE("The factorial of ", n, " is: ", fa)
}
C:\CompCons>sinan < factorial.txt

 Syntactical Analisys done without erros!

C:\CompCons>_
```

Result we got when the source code of the factorial program had no errors