CONSTRUCTION OF A FRONT_END COMPILER FOR VECTOR LANGUAGES

P.Vroustouris

MSc Dissertation

University of Peloponnese
2012

CONSTRUCTION OF A FRONT_END COMPILER FOR VECTOR LANGUAGES

Panayotis Vroustouris

Dissertation submitted in partial
fulfillment of the
requirements for the
Degree of Master by Advanced Study
in Computer Science

Supervisor: Professor Dr. Costas Masselos

Department of
Computer Science and Technology

University of Peloponnese
2012

ABSTRACT

Construction of a front_end compiler for vector languages

Panayotis N. Vroustouris

Vector languages such as MATLAB have become popular for many years in prototyping algorithms in various domains. Many of these applications whose subtasks have diverse execution requirements, often employ distributed, heterogeneous, reconfigurable systems. These systems consist of an interconnected set of heterogeneous processing resources that provide a variety of architectural capabilities. The MATCH (MATlab Compiler for Heterogeneous computing systems) compiler project, developed at Northwestern University, is to make it easier for the users to develop efficient code for distributed, heterogeneous, reconfigurable computing systems.

In the current project, I use the context–free grammar of the MATCH compiler to the design and implement a front_end compiler using a full object-oriented schema. I present the execution results on some test MATLAB code with the use of this front_ end compiler.

Keywords: context-free grammar, scanner, parser, M-Files, Flex, Bison, non-terminals, statements, expressions, assignments, parse errors, intermediate representation abstract syntax tree, DOT files

To my beloved family

for their love, patience and moral support

all this time…

TABLE OF CONTENTS

Introduction…………………………………………………………………......... 1

Chapter 1 Description of the scanning and parsing processes……………… 6
- 1.1 Lexical Analysis ……………………………………………… 7
- 1.2 The Flex Tool………………………………………………….. 9
- 1.3 Syntax Analysis………………………………………………... 11
- 1.4 The Bison tool ………………………………………………… 13
- 1.5 Putting all together 17
- 1.6 Conclusions

Chapter 2. Design of the MATLAB Parser – General Description…………. 18
- 2.1 Conclusions…………………………………………………… 25

Chapter 3. Design of Flex and Bison files ………………………………….. 26
- 3.1 The Flex file (MATFE.l) …………………………………….. 26
- 3.2 The Bison file (MATFE.y) ……………………………………. 29
- 3.3 Conclusions…………………………………………………… 40

Chapter 4. Production of the abstract syntax tree…………………………… 41
- 4.1 Conclusions…………………………………………………… 50

Chapter 5. Design of the interface ………………………………………….. 51
- 5.1 The driver files ………………………………………………… 51
- 5.2 The main function …………………………………………….. 53
- 5.3 Conclusions ………………………………………………….. 54

Chapter 6. AST Graphical Representation………………………………….. 55
- 6.1 A simple IR visualization …………………………………….. 56
- 6.2 Production of the DOT layout ………………………………… 57
- 6.3 Conclusions ……………………………….. ………………. 61

Epilogue: Recommendations for further work…………………………….. 63

Appendix 1…………………………………………………………………. 65

Appendix 2…………………………………………………………………. 72

LIST OF FIGURES

Figure 1-Scanning and Parsing processes…………………………………… 7

Figure 2.-Graphviz layout of a simple dependency tree ……………… …… 55

Figure 3.-Graphviz layout of a simple IR…………………………………... 56

Figure 4.-AST Graphical Representation ………………………………….. 62

Figure 5.-input.m AST graphical representation…………………………… 65

Figure 6.-input3.m AST graphical representation………………………….. 66

Figure 7-input8.m AST Graphical Representation………………………….. 67

Figure 8-input9.m AST Graphical Representation………………………….. 68

Figure 9-input14.m AST Graphical Representation……………………….. 70

Figure 10.-input15.m AST Graphical Representation……………………… 71

Figure 11.-input23.m AST Graphical Representation…………………….. 72

ACKNOWLEDGEMENTS

I would like to thank my supervisors Professor Dr. Costas Masselos and Research Assistant Dr. Grigoris Dimitroulakos for giving me the opportunity to work on the area of computer architecture and compiler techniques and for their support and interventions whenever I needed.
Also I want to thank my colleague and good friend John Bourlakos for his help and useful remarks all this time.

Introduction

In the early days of Computer Science, software was written in assembly language, which means that a program works on a specific machine. The need of a machine-independent representation was not so intense, until the benefits of being able to reuse software on different kinds of CPUs started to become very important. Moreover the very limited memory capacity of early computers created many technical problems when implementing a compiler.

The first higher level (machine-independent) programming languages have been proposed in late fifties. The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957. COBOL was an early language to be compiled on multiple architectures, in 1960. Early compilers were written in assembly language. The first *self-hosting* compiler — capable of compiling its own source code in a high-level language — was created for Lisp by Tim Hart and Mike Levin at MIT in 1962. Building a self-hosting compiler is a bootstrapping problem—the first such one for a language must be checked either by a compiler written in a different language, or (as in Hart and Levin's Lisp compiler) by running in an interpreter.

The compilers connect the source programs in high–level languages with the machine hardware. The majority of them produce outputs that are intended to directly run on the same type of computer and operating system that the compiler itself runs on. These are known as *native* or *hosted* compilers. Other ones produce outputs for a virtual machine (VM) and may not be executed on the same platform as the compiler that produced it.

A typical compiler requires:

- determining the correctness of the syntax of a program, based on a specific grammar
- generating correct and efficient object code
- run-time organization
- formatting output according to assembler and/or linker conventions

A compiler consists of three main parts: the *front-end*, the *middle-end*, and the *back-end*.

- The **front_end** checks whether the program is correctly written in terms of the programming language grammar, syntax and semantics. At this stage, legal statements are produced and errors, if any, are reported in a useful and predefined way. The *front-end* then generates an *intermediate representation* or *IR* of the source code.
- The **middle_end** is where optimization of the received IR occurs. Typical transformations for optimization are removal of useless or unreachable code, discovery

and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of a loop). It is also possible a total conversion of the IR to another form, if it is more convenient to the final application.

- The **back_end** is the stage where the final *IR* received by the middle-end is translated into assembly code. The target instruction(s) are chosen for each IR instruction. Register allocation assigns processor registers for the program variables where possible. The *back-end* uses the hardware by figuring out how to keep parallel execution units busy, filling delay slots, and so on. Although most algorithms for optimization are NP problems, heuristic techniques are developed with good results.

The human brain acts somehow as a compiler. Everyone can recognize a correct English text. For example the sentence "A tiger flies in the sky" is syntactically correct (regardless if its declaration is true or not). When we read such a sentence, we can intuitively decide if it is syntactically right or wrong. From this viewpoint we act as a *language recognizer*, a machine that receives correct sentences. Deterministic Finite Automata are also types of language recognizers. Moreover we are able to produce correct English sentences. We therefore behave as a *language producer.* A language producer begins to construct *strings* when it receives a kind of "start signal". Its function is controlled by a set of rules. When this process stops, a complete set of strings is produced at the output.

Obviously it is very difficult to construct a recognizer and a producer for the English language (it is like constructing a robot having the abilities of the human brain!). But the idea of a language producer has to play a very important role as far as computers are concerned. The core of these ideas is the techniques of producing grammars for artificial and context-free languages. These theories have a practical value at the determination and the analysis of the programming languages.

Normal expressions can be regarded as language producers. For example let's consider the normal expression a(a*Èb*)b. A lexical formulation of how a string is constructed according to this expression is:

- An "a" is extracted at the output. Then one of the following occurs:
- A string beginning from "a" is extracted at the output or
- A string beginning from "b" is extracted at the output
- Finally a "b" is extracted at the output

The language related with this *language producer* – the whole set of strings produced by the above procedure – is the *normal language* defined by the normal expression $a(a^{*} \cup b^{*})b$.

Normal expressions are served as language producers in a way that strings are extracted from left to right. We can therefore assume that every symbol is produced at the output *immediately* when selected. On the other hand, more complex language producers exist (called *context-free grammars*) which are based at a more complete comprehension of the structure which describe the strings of the produced language. For example from a more detailed examination of the language produced by the normal expression $a(a^{*} \cup b^{*})b$, we can observe that every string consists from:

- an initial “a”
- an *intermediate part* produced by an “$a \cup b$”
- a final “b”

Now we can introduce two new symbols. The first is “M” and represents the intermediate part and the second is “S” which is translated as *a language string*. We can therefore express the set of strings as:

$S \rightarrow aMb$

where “$\rightarrow$” means “it may be”. An expression of this form is called *a rule*. The medium part “M“ can be a string of “a” or a string of “b”. This can be expressed as:

$M \rightarrow A$ and $M \rightarrow B$

where A and B represent strings of “a” and “b” respectively. Now a string of “a” can be the null string ($A \rightarrow \varepsilon$) or a string that consists from an initial “a” and a following string of “a”. ($A \rightarrow aA$). Similarly a string of “b” is ($B \rightarrow \varepsilon$) or ($B \rightarrow bB$).

Therefore we produce strings of the formula “aaab”, “abbbbb” and so on. For instance, in order to produce the string “aaaaab”, we begin from the language string S and following the rules we make the production of “aaaaab”, using the following listing:

- $S \rightarrow aMb$. (First Rule)
- $M \rightarrow A$ (Second Rule)

- Sà aAb (Third Rule)
- Aà aA (Forth Rule four times)
- Aà e)
- SàaaaaAb the production is finished.

Therefore the normal expression a(a*Èb*)b *produced a language*. The "words" of this language are the strings of the formula "aaab", "abbbbb" and so on. For instance, in order to produce the string "aaab", we begin from the language string S and we follow the rules.

In conclusion, a context- free grammar can be considered as a language producer which uses a set of rules. But is the opposite valid? Does a language producer come from a context – free grammar? The answer is yes. Consider the string "aaAb" which is an intermediate stage in the production of "aaab". It's obvious that we can name the strings "aa" and "b" which surround A, the *contexts* of A at this string. Now the rule AàaA says that we can replace "A" with "aA", regardless of which strings surround it, in other words *independently of the contexts of A*. We have therefore a *context- free* grammar.

In a context- free grammar, some symbols appear at the left and at the right part of production grammar rules and some others only at the right. The symbols of the first kind are called *non-terminals.* The symbols of the second kind are called *terminals,* since the production of a string that consists only of symbols of that type, implies the end of production stage.

In this project all this theory is used, in order to construct the front_end stage of a compiler for vector languages i.e. the MATLAB, the SVL (Scientific Vector Language), etc.

Chapter One is a general description of the scanning and parsing stages as well as the Flex and Bison relevant tools in a common compiler function.

Chapter Two is an introduction to the current project and a general outline of the construction as well as the definitions of the main classes.

Chapter Three describes in detail the files that Flex and Bison tools elaborate for the production of *terminal* and non-terminal elements.

Chapter Four includes the implementation of the classes and explains the code of the relevant constructors and functions, via illustrative examples.

Chapter Five is referred to the construction of *the interface* which permits the interaction with the front_end compiler and includes the *main function* of the project.

Chapter Six explains how the produced abstract syntax tree can be visualized via a `DOT` graph description language and analyses the relevant technique.

Finally some important issues are expounded at the epilogue and certain ideas and recommendations for further work are proposed.

Chapter 1

Description of the scanning and parsing processes

To translate a program from one language to another, a compiler must first pull it apart and understand its structure and meaning, then put them together in a special way. The analysis is performed by the front_end stage, while the synthesis is performed by the back end.

The analysis stage is usually broken up into six stages.

- **The lexical phase (scanning process):** It groups characters into lexical units or tokens. The input to the lexical phase is a character stream. The output is a stream of tokens. Regular expressions are used to define the tokens recognized by a scanner (or lexical analyzer). The scanner is implemented as a finite state machine.
- **Syntax analysis (parsing process):** Here the tokens are grouped into syntactical units. The output of the parser is an intermediate representation of the program. Context-free grammars are used to define the program structure recognized by a parser. The parser is implemented as push-down automata.
- **Semantic analysis:** understanding the program’s meaning. The output of the semantic analysis phase is an annotated parse tree. Attribute grammars are used to describe the static semantics of a program.
- **The optimizer** applies semantics preserving transformations to the annotated parse tree to simplify the structure of the tree and to facilitate the generation of more efficient code.
- **The code generator** transforms the simplified annotated parse tree into object code, using rules which denote the semantics of the source language. The code generator may be integrated with the parser.
- **The peep-hole optimizer** examines the object code, a few instructions at a time, and attempts to do machine dependent code improvements.

The lexical analyzer and grammar parser are two of the phases in compiler construction, but for the purposes of this project, they are all that are required. For the parsing of configuration files, it is needed to break the file down into tokens and then parse these into a parse tree. This is illustrated in the following image.

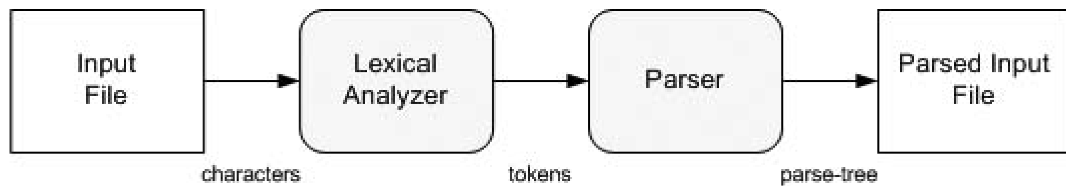

Figure1. Scanning and parsing processes

1.1 Lexical Analysis

The lexical analysis takes a stream of characters and produces a stream of names, keywords, punctuation marks and generally all the principal elements of the programming language, now called "lexical tokens". Moreover it discards white spaces and comments between these tokens. A token is sequence of characters that can be treated as a unit in the grammar of the programming language. A programming language classifies lexical tokens into a finite set of token types. For example, some of the token types of a typical programming language are:

Type	Examples
ID	fee k43 first med23
NUM	55 0 324 010
DOUBLE	444.5 .87 2.34 44e-10 2.5e32
IF	if
COMMA	,
SEMICOLON	;
NOTEQ	!=
OPEN_PAREN	(
CLOSE_PAREN	)
GTHAN	>
LTHAN	<

Reserved words as `IF`, `FOR`, `VOID`, `RETURNED` in most languages cannot be used as ID (identifiers). Expressions that cannot be used as tokens are:

Type	Examples
macros	YY_LEX
preprocessor directive	#include<stdio.h>,
define namespace	test
blanks, tabs, etc	

In languages weak enough to require a macro preprocessor, the preprocessor operates on the source character stream, producing another character stream that is then fed to the lexical analyzer. It is also possible to integrate macro processing with lexical analysis.

Given a fragment of a C source such as:

```
if (counter < 5) counter++;
```

The lexical analyzer will produce the stream of tokens: ‘if’, ‘(‘, ‘<’, ‘5’, ‘)’, ‘counter’, ‘++’, ‘;’ . That is quite a few tokens, but this process is necessary to identify how the tokens are made up. This is the specification part of lexical analysis. Furthermore the lexical analyzer returns metadata that describes what was parsed. For example, rather than the tokens, additional data is returned such as:

```
IF_TOKEN OPEN_PAREN ID LTHAN  NUM  CLOSE_PAREN  ID  UNARY_INC
SEMICOLON
```

This is useful because when we are trying to understand whether the tokens have meaning, it’s not important which variable you ’re dealing with, but just that you have a variable (identified in this example by the ID token metadata). Some of the tokens such as identifiers and literals have semantic values attaching to them, giving auxiliary information in addition to the token type.

We can describe the lexical tokens of a programming language. Here is a brief description of identifiers in C++:

An identifier is a sequence of letters and digits, the first character of which must be a letter. The underscore _ counts as a letter. Upper and lowercase letters are different. If the input stream has been scanned into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs, new lines and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords and constants.

The lexical analyzer (from now we refer to it for simplicity as “lexer”) is tasked with determining that the input stream can be divided into valid symbols in the source language, but has no smarts about which token should come where. Few errors can be detected at the lexical level alone because the scanner has a much localized view of the source program without any context. The scanner can report about characters that are not valid tokens (e.g., an illegal or unrecognized symbol) and a few other malformed entities (illegal characters within a string constant, unterminated comments, etc.) It does

not look for or detect garbled sequences, tokens out of place, undeclared identifiers, misspelled keywords, mismatched types and the like.

Let's consider the following input:

```
int b float } switch b[2] =;
```

The lexer will not generate any errors in the lexical analysis phase, because it has no idea of the appropriate arrangement of tokens for a declaration. The errors will be caught later in the parsing phase. Furthermore, the scanner has no idea how tokens are grouped. For instance, in the above sequence, it returns **b**, **[**, **2**, and **]** as four separate tokens, having no idea they collectively form an array access. The lexer can be a convenient place to carry out some other chores like stripping out comments and white space between tokens and perhaps even some features like macros and conditional compilation (although often these are handled by some sort of pre-processors which filters the input before the compiler runs).

Any reasonable programming language serves to implement a lexer. Nevertheless we usually specify lexical tokens using the formal language of ***regular expressions*** and ***deterministic finite automata.*** This will lead to simpler and more readable lexical analyzers.

1.2 The Flex tool

The fast lexical analyzer generator (or flex) is a tool that takes as input a collection of regular expressions provided by the user to produce a lexical analyzer. The produced application can then be used to tokenize an input file from a sequence of characters to a sequence of tokens. The generated application is really nothing more than a finite state automaton derived from the set of regular expressions.

To understand what all of this means, it's useful to take a look at a simple example.

A flex input file (usually has the extension *.l) has the format:

```
%{
<C declarations>
%}
<definitions>
%%
<rules>
%%
<user code>
```

The first section introduces a set of C declarations into the resulting lexer. Next are definitions or simple name declarations that are used in the rules section (we can

consider them as symbolic constant regular expressions). This section also contains start conditions for the lexical analysis, used to conditionally activate rules in the rules section. Definitions have the form

```
name       definition
```

The second section describes the rules which define the set of regular expressions used to produce the tokens from the input. These rules define the finite automata to parse the tokens and take the form:

```
pattern     action
```

The following paragraph suffices to track locations accurately. Each time `yylex` is invoked, the `begin` position is moved onto the `end` position. Then when a pattern is matched, the end position is advanced of its width. In case it matched ends of lines, the end cursor is adjusted, and each time blanks are matched, the begin cursor is moved onto the end cursor to effectively ignore the blanks preceding tokens. Comments would be treated equally.

```
%{
# define YY_USER_ACTION yylloc->columns (yyleng);
%}
%%
%{
yylloc->step ();
%}
{blank}+ yylloc->step ();
[\n]+ yylloc->lines (yyleng); yylloc->step ();
```

Finally, the last section is C code produced by the user that is integrated into the resulting lexical
analyzer code.

A simple but complete and valid Flex file is described in the following listing.

```
File   mylexer.l
%{
/* <C declarations> */

# include <stdio.h>
# include <cstdlib>
# include <errno.h>
# include <limits.h>
# include <string>
%}
{%
/* <definitions>*/
# undef yywrap
# define yywrap() 1
```

```
#define yyterminate() return token::END
%}
%%
/* Rules */
id          [a-zA-Z][a-zA-Z_0-9]*
int         [0-9]+
blank        [ \t]

%%

/* c code */
set printf( "(STMT set) " );
{NUM}+ printf( "(NUM %s) ", yytext );
{VAR}+ printf( "(VAR %s) ", yytext );
printf( "(OP minus) " );
printf( "(OP plus) " );
printf( "(OP equal) " );
printf( "(END stmt) " );
\n printf( "\n\n" );
<<EOF>> printf( "End of parse\n." );
yyterminate();
```

Now we can use the `flex tool`. To build the lexer, it's enough to provide the file (in this example: `mylexer.l`) and invoke the flex utility, as follows:

```
$ flex mylexer.l
```

That produces a file called `lex.mylexer.c` which contains the lexer.

1.3 Syntax analysis

The syntax analysis phase of a compiler verifies that the sequence of tokens extracted by the scanner represents a valid sentence in the context-free grammar of our programming language. There are two major parsing approaches: ***top-down*** and ***bottom-up***.

In **top-down parsing**, we start with the start symbol and apply the productions until we arrive at the desired string.

In **bottom-up parsing**, we start with the string and reduce it to the start symbol by applying the productions backwards.

As an example, we can implement these two parsing approaches on a simple grammar that recognizes strings consisting of any number of a's followed by at least one (and possibly more) b's:

```
S -> AB
A -> aA | e
B -> b | bB
```

Here is a top-down parse of "aaab". We begin with the start symbol and at each step, expand one of the remaining non-terminals symbols by replacing it with the right side of

one of its productions. We repeat until only terminals remain. The top-down parse produces a leftmost derivation of the sentence.

```
S
AB          S -> AB
aAB         A -> aA
aaAB        A -> aA
aaaAB       A -> aA
aaaeB       A -> e
aaab        B -> b
```

The bottom-up parse works conversely. The input is the sentence of terminals and each step applies a production in reverse, replacing a substring that matches the right side with the non-terminal on the left. We continue until we have substituted our way back to the start symbol. If we read from the bottom to top, the bottom-up parse prints out a rightmost derivation of the sentence.

```
aaab
aaaeb        (insert e)
aaaAb       A -> e
aaAb          A -> aA
aAb           A -> aA
Ab            A -> aA
AB            B -> b
S             S -> AB
```

We can now form another more "arithmetic" example by modelling a grammar to represent a set of `set operations` as follows:

```
set counter = 1;
set counter = counter + 1;
set counter = lastcount;
set delta = counter - lastcount;
```

In this very simple grammar, we can reduce to the following rules. First each statement starts with a **set command** followed by a variable and an "**=**" symbol. After the assignment operator, we have what is called an `expression` and terminates with a `;`. So the first rule can take the form:

```
'set' VARIABLE '=' EXPRESSION ';'
```

The `'EXPRESSION'` has one of four forms:

```
NUMBER
VARIABLE
VARIABLE OPERATOR NUMBER
VARIABLE OPERATOR VARIABLE
```

We can therefore support two operators within an expression. It can be either an addition or a subtraction:

'+'

```
'-'
```

This is reasonable so far for this very simple grammar.

It should be clear now how the lexer and grammar parser work together. At the scanning stage the lexer breaks the input down into the necessary tokens, while at the parsing stage the grammar parser takes these tokens and, using its rules, determines if the symbols can be recognized by the grammar.

1.4 The Bison tool

Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR parser. In computer science, an **LR parser** is a parser that reads an input from **L**eft to **R**ight (as it would appear if visually displayed) and produces a rightmost derivation. The term **LR(*k*) parser** is also used; where the *k* refers to the number of unconsumed "look ahead" input symbols that are used in making parsing decisions. Usually *k* is 1 and the term **LR parser** is often intended to refer to this case.

Bison is a program which, given a context-free grammar, constructs a C program that will parse the input according to the grammar rules. It was primarily written by Richard Stalleman as part of the GNU project, while flex was authored by Vern Paxson at Lawrence Berkley Laboratory. Bison is a newer form of yacc developed by S. C. Johnson an others at AT&T Bell Laboratories. .

An input file for Bison is of the form:

```
C and parser declarations
%%
Grammar rules and actions
%%
C subroutines
```

The first section of the Bison file consists of a list of tokens (other than single characters) that are expected by the parser and the specification of the start symbol of the grammar. This section of the Bison file may contain specification of the precedence and associativity of operators. This permits greater flexibility in the choice of a context-free grammar. Addition and subtraction are declared to be left associative and of lowest precedence while exponentiation is declare to be right associative and to have the highest precedence.

Since Bison files are similar to Flex ones but more complicated, it is time now to proceed to a valid example. In the following listing a simple but well-formed file suitable for Bison input is presented. It provides a grammar for the previous set example

```
File mybison.y
%{
```

```
  #include <stdio.h>
   void yyerror(const char *str )
   {
    fprintf(stderr, "error: %s\n", str );
   }
   int main()
     {
yyparse();

return 0
}
 %}
 %token SET NUMBER VARIABLE OP_MINUS OP_PLUS ASSIGN END

%%

statements:
     | statements statement
     ;
 statement:
SET VARIABLE ASSIGN expression END
{
}
;
expression:

NUMBER
|
VARIABLE
|
VARIABLE operator NUMBER
|
VARIABLE operator VARIABLE
;
operator:
OP_MINUS
|
OP_PLUS
;
```

The code in the first section (C declarations) is to be ported to the generated parser. Any header files that are referenced in this C declarations section or in code in the rules section are included here. Moreover there are *two important functions* that we provide here for the parser. It is worth to analyze them now.

The first is a special function called yyerror.

This function is called by the generated parser whenever an error occurs. Bison supports a form of error resynchronization that allows you to define where in the stream to give up on an unsuccessful parse and how far to scan ahead and try to cleanup to allow the parse to restart. The special error token can be used in the right side of a production to mark a context where an error recovery is to be attempted. The usual use is to add the error token

possibly followed by a sequence of one or more synchronizing tokens, which tell the parser to discard any tokens until it sees that "familiar" sequence that allows the parser to continue. A simple and useful strategy might be simply to skip the rest of the current input line or current statement when an error is detected.

When an error is encountered (and reported via yyerror) the parser will discard any partially parsed rules (i.e. pop stacks from the parse stack) until it finds one in which it can shift an error token. It then reads and discards input tokens until it finds one that can follow the error token in that production. For a better understanding we can proceed to the following example:

```
Var: Modifiers Type IdentifierList ';'
| error ';'
;
```

The second production allows for handling errors encountered when trying to recognize `Var` by accepting the alternate sequence error followed by a semicolon. What happens if the parser in the middle of processing the `IdentifierList` encounters an error? The error recovery rule, interpreted strictly, applies to the precise sequence of an error and a semicolon. If an error occurs in the middle of an `IdentifierList`, there are `Modifiers` and a `Type` and what not on the stack, that doesn't seem to fit the pattern. However, bison can force the situation to fit the rule, by discarding the partially processed rules (i.e. popping states from the stack) until it gets back to a state in which the error token is acceptable (i.e. all the way back to the start state before matching the `Modifiers`). At this point the error token can be shifted. Then, if the `lookhead` token couldn't possibly be shifted, the parser reads tokens, discarding them until it finds a token that is acceptable. In this example, bison reads and discards input until the next semicolon so that the error rule can apply. It then reduces that to `Var`, and continues on from there. This basically allowed for a mangled variable declaration to be ignored up to the terminating semicolon, which was a reasonable attempt at getting the parser back on track. Note that if a specific token follows the error symbol it will discard until it finds that token, otherwise it will discard until it finds any token in the `lookahead` set for the nonterminal (for example, anything that can follow `Var` in the example above).

The second function is the *main* ..

At most applications we normally form an independent function to call the parser, but for this simple example, we can embed the main within the parser itself. The main calls the function `yyparse`, the grammar parser. The grammar parser invokes the lexer internally.

In the next section, all the tokens that are used by the parser to recognize the grammar are identified. Of course the lexer (the flex file) must know about these tokens and must be built to return them. So we have to check again the lexer and alter it, in order to support connectivity with the scanner.

The main part of our bison file is the rules that form the parser's grammar. This section begins at the first `%%` symbol of the file. A rule is formed by a name followed by a colon and then a series of tokens or non-terminals with an optional action sequence terminated by a semicolon. For example the statements rule has two possibilities. The blank after the colon means that there might be no more tokens to parse (end of file) or the possibility to have more than one adjacent statements to parse. If we had placed only the rule statement without any possibility, then we could parse a single statement and nothing more. By specifying statements before statement, we allow the possibility of parsing more than one statement in the input. But how this "statement" rule is formed?

This is the content of the second rule. The statement rule defines the parse of a single kind of statement (a variable assignment). The statement must begin with a set of commands followed by a variable name and an assignment operator. The expression rule that follows covers a number of different possibilities,. The rule ends with a semicolon that is treated as an `END` token.

The next term undefined in our simple grammar, the term "expression". The expression rule defines the possible varieties of expressions (the right value, or `rvar` of the statement). Four possibilities are considered as "legal". It can be a number, another variable, a sum of a variable and a variable or number, or a difference between a variable and a variable or number.

Finally there is the operator rule which covers the two types of legal operators (a – as defined by the `OP_MINUS` token or a + as defined by `OP_PLUS` token).

Therefore the input file is ready. Now we can use the **bison** tool. It is enough to provide the file (in this example: myparser.y) and invoke the bison utility, as follows:

```
$ bison -d myparser.y
```

Using the –d flag we tell bison to generate the extra output file which contains the macro definitions we have used. This file will be hooked up to the lexer. Two files are produced: `myparser.tab.c` and `myparser.tab.h` The C file is the parser source while the header file contains the macro definitions.

1.5 Putting all together

Now it's time for testing and compilation. Firstly we inform the lexer file (mylexer.l) to include the macro definitions. We have therefore to reproduce the scanner file lex.mylexer.c in order to include the new files by invoke again the flex tool. So the whole process is as follows:

```
# ls
  myparser.y myflexer.l
# bison -d myparser y.
# ls
myparser.y.tab.c  myparser.y.tab.h  myparser.y myflexer.l
# flex myflexer.l`
# ls
#  myparser.y.tab.c    myparser.y.tab.h    myparser.y  myflexer.l
lex.mylexec.c
The  lexer  can  now  be  compiled  in  order  to  produce  the
executable.
$ gcc -o m  lex.yy.c -o myparser -lfl
```

The -lfl parameters tells the `g++` compiler to link in the flex library (libfl.a). Normally an executable `myparser` is produced. We use this executable to test the whole process.

1.6 Conclusions

In this chapter we introduced the scanning and parsing processes as the initial stages of the front_end part of a compiler. The presentation has been quite detailed via a simple example, showing how flex and bison, working in tandem, produce a "simple compiler" based on a defined context- free grammar and it will be the basis to perform a far more complicated construction: the design of the front_end compiler for the MATLAB language.

Chapter 2

Design of the MATLAB Parser – General Description

Last generation collection-oriented languages are efficient to aggregate complex data structures (e.g., sets, sequences, matrices, vectors, graphs, lists, trees, etc) as well operations for handling them "as a whole" (summing, subtracting, sorting, permuting, etc) and generally applying functions to each member of these structures. They can therefore be considered as having the ability to carry these forms of data as "parameters" in an n-dimensional vector space. Thus we refer to them as "vector languages".

MATLAB is such a high performance vector language. In order to construct a front_end of a compiler that satisfies a subset of MATLAB capacities[1], I have to clearly define:

- The terminal symbols of the language (tokens)
- The non-terminal symbols of the language (types)
- The production rules of the non-terminal symbols (grammar rules)

The software construction is object-oriented and uses the C++ language. Every element therefore, (terminal or non terminal) will be defined as an object (more accurately: *as a pointer to an object*) of a corresponding ***class***. On the other hand, considering that the desired final *IR* will be a tree data structure (an abstract syntax tree, AST), I need a corresponding AST class. The name of this class is ***CASTSyntaxElement*** and all the elements must be of that type too. Therefore all the element classes must be subclasses of *CASTSyntaxElement* class and inherit its members and functions.

Now it is the time to present two very important header files for the basic classes which support the project. The first is the `ASTdefines.h`, where three *enumeration type* declarations are defined. The first includes the terminal elements (called enum `TERMINAL_CODE`) the second includes for the non-terminal elements (called enum `NONTERMINAL_CODE`) and the third includes the grammar rules (called enum `NONTERMINAL_PRODUCTION_RULE`). The first and the second enumeration types describe the ***nodes*** of the abstrac syntax tree which will represent the final *Internediate Representation* at the end of the construction, while the third enumeration type refers to the *way* these nodes are produced, ie the corresponding grammar rules. The folowing listing presents the ASTdefines.h:

[1] The compiler that is being constructed here covers a subset of MATLAB. The implementation of a compiler which covers all the structures of MATLAB language is beyond the scope of this project.

```
#ifndef ASTDEFINES_
#define ASTDEFINES_

typedef enum nonterminals{token_ =0, translation_unit_ ,scriptMFile_,
opt_delimiter_,delimiter_, null_lines_, null_line_, empty_lines_,
statement_list_, statement_, command_form_, text_list_, expr_, reference_,
identifier_,argument_list_, matrix_, rows_, row_, row_with_commas_,
colon_expr_, assignment_, s_assignee_matrix_,
m_assignee_matrix_,reference_list_,for_command_, for_cmd_list_,
if_command_, if_cmd_list_, opt_else_, global_command_, global_decl_list_,
while_command_, while_cmd_list_, return_command_, delimited_input_,
delimited_list_, functionMFile_,f_def_line_, f_output_, f_input_,
f_argument_list_, f_body_,      parse_error_} NONTERMINAL_CODE;

typedef enum terminals {TOKEN_ = 0,LEXERROR_,LINE_,LB_,RB_,LD_,RD_,
FOR_,END_,IF_,ELSEIF_,ELSE_,WHILE_,FUNCTION_,RETURN_,TEXT_,IDENTIFIER_,INT
EGER_,DOUBLE_,IMAGINARY_,AND_,OR_,NOT_,GLOBAL_,LTHAN_,LTHANE_,GTHAN_,GTANH
E_,EQUAL_,UNEQUAL_,ASSIGN_,MINUS_,PLUS_,EMUL_,EDIV_,BIMUL_,BIDIV_,BILEFTDI
V_,ELEFTDIV_,TRANSPOSE_,CTRANSPOSE_,POWER_,EPOWER_,LPAREN_,RPAREN_,COLONOP
ERATOR_,COMMA_,SEMICOLON_} TERMINAL_CODE;

typedef enum productions {
      UNITIALIZED = 0,

      TOKEN_LEXERROR,
      TOKEN_LINE,
      TOKEN_LB,
      TOKEN_RB,
      TOKEN_LD,
      TOKEN_RD,
      TOKEN_FOR,
      TOKEN_END,
      TOKEN_IF,
      TOKEN_ELSEIF,
      TOKEN_ELSE,
      TOKEN_WHILE,
      TOKEN_FUNCTION,
      TOKEN_RETURN,
      TOKEN_TEXT,
      TOKEN_IDENTIFIER,
      TOKEN_INTEGER,
      TOKEN_DOUBLE,
      TOKEN_IMAGINARY,
      TOKEN_AND,
      TOKEN_OR,
      TOKEN_NOT,
      TOKEN_GLOBAL,
      TOKEN_GTHAN,
      TOKEN_GTHANE,
      TOKEN_LTHAN,
      TOKEN_LTANHE,
      TOKEN_EQUAL,
      TOKEN_UNEQUAL,
      TOKEN_ASSIGN,
      TOKEN_MINUS,
      TOKEN_PLUS,
      TOKEN_EMUL,
      TOKEN_EDIV,
      TOKEN_BIMUL,
      TOKEN_BIDIV,
      TOKEN_BILEFTDIV,
```

```
TOKEN_ELEFTDIV,
TOKEN_TRANSPOSE,
TOKEN_CTRANSPOSE,
TOKEN_POWER,
TOKEN_EPOWER,
TOKEN_LPAREN,
TOKEN_RPAREN,
TOKEN_COLONOPERATOR,
TOKEN_COMMA,
TOKEN_SEMICOLON,


Token__TOKEN,

TranslationUnit__ScriptMFile,
TranslationUnit__FunctionMFile,
TranslationUnit__ParseError,


ScriptMFile__OptDelimiter,
ScriptMFile__OptDelimiter_StatementList,

OptDelimiter__,
OptDelimiter__Delimiter,

Delimiter__NullLines,
Delimiter__EmptyLines,
Delimiter__NullLines_EmptyLines,

NullLines__NullLine,
NullLines__NullLines_NullLine,

NullLine__COMMA,
NullLine__SEMICOLON,
NullLine__EmptyLines_COMMA,
NullLine__EmptyLines_SEMICOLON,

EmptyLines__LINE,
EmptyLines__EmptyLines_LINE,

StatementList__Statement_OptDelimiter,
StatementList__Statement_Delimiter_StatementList,

Statement__CommandForm,
Statement__Expr,
Statement__Assignment,
Statement__ForCommand,
Statement__IfCommand,
Statement__GlobalCommand,
Statement__WhileCommand,
Statement__ReturnCommand,

 CommandForm__Identifier_TextList,

TextList__TEXT,
TextList__TextList_TEXT,

Expr__INTEGER,
Expr__DOUBLE,
Expr__IMAGINARY,
Expr__TEXT,
```

```
Expr__LPAREN_Expr_RPAREN,
Expr__Reference,
Expr__Matrix,
Expr__Expr_EPOWER_Expr,
Expr__Expr_POWER_Expr,
Expr__Expr_TRANSPOSE,
Expr__Expr_CTRANSPOSE,
Expr__NOT_Expr,
Expr__PLUS_Expr,
Expr__MINUS_Expr,
Expr__Expr_BIMUL_Expr,
Expr__Expr_BIDIV_Expr,
Expr__Expr_BILEFTDIV_Expr,
Expr__Expr_EMUL_Expr,
Expr__Expr_EDIV_Expr,
Expr__Expr_ELEFTDIV_Expr,
Expr__Expr_PLUS_Expr,
Expr__Expr_MINUS_Expr,
Expr__ColonExpr,
Expr__Expr_LTHAN_Expr,
Expr__Expr_LTHANE_Expr,
Expr__Expr_GTHAN_Expr,
Expr__Expr_GTHANE_Expr,
Expr__Expr_EQUAL_Expr,
Expr__Expr_UNEQUAL_Expr,
Expr__Expr_AND_Expr,
Expr__Expr_OR_Expr,

Reference__Identifier,
Reference__Identifier_LPAREN_ArgumentList_RPAREN,

Identifier__IDENTIFIER,

ArgumentList__COLONOPERATOR,
ArgumentList__Expr,
ArgumentList__COLONOPERATOR_COMMA_ArgumentList,
ArgumentList__Expr_COMMA_ArgumentList,

Matrix__LB_Rows_RB,

Rows__Row,
Rows__Rows_SEMICOLON,
Rows__Rows_SEMICOLON_Row,
Rows__Rows_LINE,
Rows__Rows_LINE_Row,

Row__Expr,
Row__RowWithCommas,
Row__RowWithCommas_Expr,

RowWithCommas__Expr_COMMA,
RowWithCommas__RowWithCommas_Expr_COMMA,

ColonExpr__Expr_COLONOPERATOR_Expr,
ColonExpr__ColonExpr_Expr_COLONOPERATOR_Expr,

Assignment__Reference_ASSIGN_Expr,
Assignment__SAssigneeMatrix_ASSIGN_Expr,
Assignment__MAssigneeMatrix_ASSIGN_Reference,

SAssigneeMatrix__LD_Reference_RD,
```

```
    MAssigneeMatrix__LD_Reference_COMMA_ReferenceList_RD,

    ReferenceList__Reference,
    ReferenceList__Reference_COMMA_ReferenceList,

    ForCommand__FOR_ForCmdList_END,

    ForCmdList__Identifier_ASSIGN_Expr_DelimitedInput,

    IfCommand__IF_IfCmdList_END,

    IfCmdList__Expr_DelimitedInput_OptElse,

    OptElse__ELSE_DelimitedInput,
    OptElse__ELSEIF_DelimitedInput_OptElse,

    GlobalCommand__GLOBAL_GlobalDeclList,

    GlobalDeclList__IDENTIFIER,
    GlobalDeclList__GlobalDeclList_IDENTIFIER,

    WhileCommand__WHILE_WhileCmdList_END,

    WhileCmdList__Expr_DelimitedInput,

    ReturnCommand__RETURN,

    DelimitedInput__OptDelimiter,
    DelimitedInput__OptDelimiter_DelimitedList,

    DelimitedList__Statement_Delimiter,
    DelimitedList__Statement_Delimiter_DelimitedList,

    FunctionMFile__EmptyLines_FDefLine_FBody,
    FunctionMFile__FDefLine_FBody,

    FDefLine__FUNCTION_FOutput_ASSIGN_IDENTIFIER_FInput,
    FDefLine__FUNCTION_IDENTIFIER_FInput,

    FOutput__Identifier,
    FOutput__LD_FArgumentList_RD,

    FInput__LPAREN_RPAREN,
    FInput__LPAREN_FArgumentList_RPAREN,

    FArgumentList__Identifier_COMMA_FArgumentList,
    FArgumentList__Identifier,

    FBody__Delimiter_StatementList,
    FBody__OptDelimiter,

    ParseError__LEXERROR,
    ParseError__error
} NONTERMINAL_PRODUCTION_RULE;


#endif
```

File ASTDefine. h

The second header file is the ASTSyntaxElements.h. Here are the definitions of all the classes used in this project. The main class is ***CASTSyntaxElement.*** It's important to describe it in detail.

```
class CASTSyntaxElement {
public:

CASTSyntaxElement(NONTERMINAL_PRODUCTION_RULE pr, MATFE::location * loc);
~CASTSyntaxElement();

virtual int map_syntax_element(CASTSyntaxElement *, int m=-1){return -1;}

TERMINAL_CODE m_TerminalCode;
NONTERMINAL_CODE m_NonTerminalCode;
NONTERMINAL_PRODUCTION_RULE m_NonTerminal_ProductionCode;
static unsigned int m_ObjectSerialNumberCounter;
unsigned int m_NumberOfDescendants;
unsigned int m_ObjectSerialNumber;
vector <CASTSyntaxElement *> m_Descendants;
char* graphstring;

protected:
MATFE::location *m_location;
};
```

Each of the three public variables is an object of the enumeration types defined above (`m_TerminalCode`, `m_NonTerminalCode`, `m_NonTerminal_ProductionCode`). The defined public functions are:

A function called `CASTSyntaxElement` with two arguments.

- A `NON_TERMINAL_PRODUCTION_RULE` and a pointer to a memory location. The latter points to the location in memory where the produced parsed element will be stored.
- An `int Function` called: `map_syntax_element`, which takes as arguments a pointer to a CASTSyntaxElement object and the corresponding integer from the `NONTERMINAL_CODE` enum

There are also:

- A public vector (of CASTSyntaxElements) `m_Descendants` which stores the `NONTERMINAL_CODE` values for every grammar rule used in Bison file (see next chapter).
- An integer `m_ObjectSerialNumber` used to count the nodes of the abstract syntax tree.
- A counter `m_ObjectSerialNumberCounter`.

Another important class (subclass of *CASTSyntaxElement*) defined here is class CTOKEN. All the terminal codes (tokens), produced by the scanner, are objects of this class.

```
class CTOKEN : public  CASTSyntaxElement{
public:
      CTOKEN(TERMINAL_CODE token, MATFE::location *loc,
       NONTERMINAL_PRODUCTION_RULE pr);
      ~CTOKEN();


protected:
int map_syntax_element(CASTSyntaxElement *, int c=-1);`

};
```

As far as concerns the *non-terminals,* each of them has its own class. These classes have functions that correspond to the appropriate grammar rule (lies at Bison file) for each non-terminal. For example the grammar rule for the nonterminal *text_list* is [2]:

```
text_list      : TEXT
               {

                 }
               | text_list TEXT
                {

                 }
                 ;
```

The corresponding class therefore will be defined as:

```
class CTextList : public CASTSyntaxElement{
   public:
CTextList(CTOKEN *, MATFE::location *,NONTERMINAL_PRODUCTION_RULE);
CTextList(CTextList *, CTOKEN *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

protected:
   virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);
};
```

All the classes of the non-terminal elements will be defined as above. These classes are the following.

```
class CTransUnit;
class CScriptMFile;
class CFunctionMFile;
class CParseError;
class COptDelimiter;
class CDelimiter;
class CStatementList;
class CNullLines;
class CNullLine;
class CEmptyLines;
```

[2] The code of the hole project is presented in Appendix 2

```
class CStatementList;
class CStatement;
class CCommandForm;
class CTextList;
class CExpression;
class CReference;
class CIdentifier;
class CArgumentList;
class CMatrix;
class CRows;
class CRow;
class CRowWithCommas;
class CColonExpression;
class CAssignment;
class CSassigneeMatrix;
class CMassigneeMatrix;
class CReferenceList;
class CForCommand;
class CForCmdList;
class CIfCommand;
class CIfCmdList;
class COptElse;
class CGlobalCommand;
class CGlobalDeclList;
class CWhileCommand;
class CWhileCmdList;
class CReturnCommand;
class CDelimitedInput;
class CDelimitedList;
class CFDefLine;
class CFOutput;
class CFInput;
class CFArgumentList;
class CFBody;
class CParseError;
```

The constructors and the impementations of all classes will be presented later on.

A functional remark in the construction of the code may be : *Why I use so many pointers and not simple objects?* The answer derives from C++ itself. With the help of the pointers I can pass complete structures, objects, text strings or anything I like, from one state to another, without the need for global variables. This means I can do all this without the memory (RAM) requirements I would otherwise need. It also makes my code look neater.

2.1 Conclusions

In this chapter, the basic elements of general skeleton of the compiler construction are presented. I explained briefly the way the classes are defined and I referred to some of basic files I use for the front_end construction. In the next chapter the generators of all these ideas will be analysed: The Flex and the Bison files.

Chapter 3

Design of Flex and Bison files

The analysis of previous chapter was about the definitions of the basic classes I use at the compiler construction, which describe the form of the terminals and the non-terminals elements as pointers to objects of the main class *CASTSyntaxElement.* It is time know to define *which* will be the tokens (terminals) that the scanner of my version of MATLAB compiler must return, as well as *which* will be the types (non-terminals) and *which* will be the grammar rules used for the production of these types at the parsing stage. In other words I present the form of Flex and Bison files.

3.1 The Flex file (MATFE.l)

The first bulk to solve is to deternmine the *normal expressions* that describe MATLAB code lexemes. To declare them, it is normal to accept the following admissions.

- a MATLAB identifier consists of a *letter* followed by zero or more *digits*, *letters* or *undescores.*
- a MATLAB numeric value can be free of decimal points or exponents or include decimal points, exponents or both. In the MATCH lexical specification, the name definitions INTEGER and DOUBLE correspond to the former and latter respectively. Moreover it can a complex number that is represent from the compound expression `{NUMBER}{IMAGINARYUNIT}`.
- A MATLAB horizontal space is considered as token demarketor and element separetor or in some cases influences the interpretation of a succeding character. They are usually cast away by the scanner, so the token stream passed for parsing is free of them. An horizontal space is either a blank or a horizontal tab.
- Input lines in MATLAB can be continued onto multiple lines. We achieve the breaking of long statements by using a sequence of three periods, subsequently followed by new-line, carriage return, or form feed character (`\n|\r|\f`).
- Comments in MATLAB begin at a percent character (`%`) and continue until a new-line character, or until the end of the input.

I end up therefore to the following normal expressions:

```
HSPACE          [ \t]
HSPACES         {HSPACE}+
NEWLINE         \n|\r|\f
NEWLINES        {NEWLINE}+
ELLIPSIS        \.\.\.
CONTINUATION    {ELLIPSIS}[^\n\r\f]*{NEWLINE}?
COMMENT         \%[^\n\r\f]*{NEWLINE}?
IDENTIFIER      [a-zA-Z][_a-zA-Z0-9]*
DIGIT           [0-9]
INTEGER         {DIGIT}+
EXPONENT        [DdEe][+-]?{DIGIT}+
MANTISSA        ({DIGIT}+\.)|({DIGIT}*\.{DIGIT}+)
FLOATINGPOINT   {MANTISSA}{EXPONENT}?
DOUBLE          {INTEGER}{EXPONENT}|{FLOATINGPOINT}
NUMBER          {INTEGER}|{DOUBLE}
IMAGINARYUNIT   [IiJj]
```

As stated abobe, all the terminal elements are objects of the main class *CASTSyntaxElement*. We must therefore define a new relevant type. This is the new type <*ast*> defined at Bison file. It is explained later in this chapter, but it is worth to mention its definition now:

```
CASTSyntaxElement *ast ;
```

On the other hand the tokens are elements for parsing. They *must* therefore belong to the parser class. This is defined in Flex file as follows:

```
typedef MATFE::MATParserClass::token token;
```

Now the definition of a token is a new instance of the class CTOKEN presented in previous chapter:

```
class CTOKEN : public  CASTSyntaxElement{
public:
 CTOKEN(TERMINAL_CODE token, MATFE::location *loc,
NONTERMINAL_PRODUCTION_RULE pr);
~CTOKEN();
```

The `NONTERMINAL_PRODUCTION_RULE pr` argument comes from the appropriate declaration included in file ASTDefine.h (see previous chapter).

For example, the definition of token "`if`" should be:

```
"if"                {
                     CTOKEN* newtoken;
                     newtoken = new CTOKEN(IF_,
                     (MATFE::location *)&yylloc,
                     TOKEN_IF);
                     yylval->ast = newtoken;
                     return(token::IF);
                    }
```

The function yylval (which is an argument of the macro yyin defined later inBison file) sends the produced token to the parser.
The definition of token " ;" (semicolon) should be:

```
;                       {
                         BEGIN(INITIAL);
                         CTOKEN* newtoken;
                         newtoken = new CTOKEN(SEMICOLON_,
                         (MATFE::location *)&yylloc,
                         TOKEN_SEMICOLON);
                         yylval->ast = newtoken;
                         return(token::SEMICOLON);
                        }
```

The second example includes the declaration BEGIN(INITIAL)that has to be explained.
The MATCH (MATLAB Compiler for Heterogeneous computing systems) scanner reproduces MATLAB's single quote semantics by using start conditions. The start condition mechanism basically enables the scanner to "activate" only a subset of rules in its lexical specification depending on its current state. A scanner can be transited from one start condition to another by BEGIN commands.
To imitate MATLAB's single quote behaviour, the scanner is always in one of two start conditions. These are referred as INITIAL and QuoteSC in the lexical specification. Where the scanner is in INITIAL state regards the single quote character as the demarcator of a string literal. When in QuoteSC state, the scanner considers the single quote as the CTRANSPOSE token. Thus we have the extended regular expressions `<INITIAL>'[^'\r\f\n]*'` and `<QuoteSC>'` whose action parts return the token TEXT and CTRANSPOSE respectively. Since a single quote character is scanned immediately after an INTEGER, DOUBLE, IMAGINARY, IDENTIFIER, TRANSPOSE, ']', ')', or CTRANSPOSE lexemes should be considered as the CTRANSPOSE token. Hence the production of the the corressponding tokens (exept CTRANSPOSE) must include a command which set the current state to <QuoteSC>'.
Otherwise (eg: lexemes ",", ";", "(") the current state is <INITIAL>.

```
{INTEGER}                {
                         BEGIN(QuoteSC);
                         CTOKEN* newtoken;
                         newtoken = new CTOKEN(INTEGER_,
                         (MATFE::location *)&yylloc,
                         TOKEN_INTEGER);
```

```
                        yylval->ast = newtoken;
                        return(token::INTEGER);

                         }


\*                        {
                          BEGIN(INITIAL);
                          CTOKEN* newtoken;
                          newtoken = new CTOKEN(BIMUL_,
                          (MATFE::location *)&yylloc,
                          TOKEN_BIMUL);
                          yylval->ast = newtoken;
                           return(token::BIMUL);
                           }
```

The starting state in the current Flex file has been set to `QuoteSC`.

The detailed production of all the tokens is listed at Appendix 2.

Finallly a lexeme must be responsible to designate the end of scanning process, ie the end of file. Of course this will be the token `END`. This is achieved by the following definition of the appropriate function (`yyterminate`).

```
#define yyterminate() return token::END
```

Once all the necessary terminals are produced I proceed to the parsing stage.

3.2 The Bison file (MATFE.y)

This file includes the core of the project. To begin there is a declaration for the skeleton directive that the design of Bison file follows (“`skeleton "lalr1.c"`)[1]. The working namespace is *MATFE_0* (from the initials **MAT**lab **F**ront**E**nd), the parser class name is *MATParserClass* and the Bison file name is MATFE.y. In this file I define the tokens and the types which the grammar will use as well as the grammar rules[2]. Morover I include the macro ***YY_DECL.*** This macro defines the arguments of the parsing function ***yylex*** (the semantic_type: *yylval*, the location_type: *yyloc* and the *driver* of the parser and the lexer which is a reference to MATFE_driver:

```
#define YY_DECL \
MATFE::MATParserClass::token_type  \
yylex(MATFE::MATParserClass::semantic_type* yylval,
MATFE::MATParserClass::location_type* yylloc, \
```

---

[1] See Bison documentation.

[2] The grammar rules that I use in this Bison file derives from the paper “ The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler” *Pramod Joisha , Abhay Kanhere, Prithviraj Banerjee, U.Nagaraj Shenoy, Alok Choudhary Northwestern University September 1999.*

```
MATFE_driver& driver)
```

The union stage of ***MATFE.y*** is quite simple:

```
%union{
      char* text;
      CASTSyntaxElement *ast;
       double imaginary;
       };
```

The definition of type *ast* is very useful because it declares that all elements should be objects of *CASTSyntaxElement* class. All the terminals therefore will have that type. The token produced by the scanner are:

```
%token <ast> LEXERROR
%token <ast> LINE
%token <ast> LB
%token <ast> RB
%token <ast> LD
%token <ast> RD
%token <ast> FOR
%token <ast> END  0  "end of file"
%token <ast> IF
%token <ast> ELSEIF
%token <ast> ELSE
%token <ast> WHILE
%token <ast> RETURN
%token <ast> TEXT
%token <ast> IDENTIFIER
%token <ast> INTEGER
%token <ast> DOUBLE
%token <ast> IMAGINARY
%token <ast> GLOBAL
%token <ast> ASSIGN
%token <ast> LPAREN
%token <ast> RPAREN
%token <ast> COMMA
%token <ast> SEMICOLON
%token <ast> FUNCTION
%nonassoc MINOR_PREC
%left <ast> AND OR
%left <ast> LTHAN LTHANE GTHAN GTHANE EQUAL UNEQUAL
%left <ast> COLONOPERATOR
%left <ast> PLUS MINUS
%left <ast> BIMUL EMUL BIDIV EDIV ELEFTDIV BILEFTDIV
%nonassoc <ast> NOT
%nonassoc <ast> TRANSPOSE CTRANSPOSE
%left <ast> EPOWER POWER
%nonassoc UPLUS UMINUS
%nonassoc LPAREN
```

As far as concerns the operators precedence, this is defined by the declarations `%left`, `%right` and `%nonassoc`. In more detail, assuming the case “a op b op c”, `%left` specifies left-associativity (grouping a with b first), `%right` specifies right

associativity (grouping b with c first), while `%nonassoc` specifies no associativity, which means that 'a op b op c' is considered a syntax error.

The above precedence declarations, can only be used once for a given token; so a token has only one precedence declared in this way. For context-dependent precedence, there is need of an additional mechanism: the `%prec` modifier for rules. This modifier declares the precedence of a particular rule by specifying a terminal symbol whose precedence should be used only for that case and it appears only once in the rule.. The modifier's syntax is:

```
%prec terminal-symbol
```

and it is written after the components of the rule. Its effect is to assign the precedence of terminal symbol to the rule, overriding the precedence that would be deduced for it in the ordinary way. The altered rule precedence then affects how conflicts involving that rule, are resolved.

For example, an expression in MATLAB program can be produced at the following case:

```
expr              | PLUS expr %prec UPLUS
```

where `%prec UPLUS` means that token PLUS has the precedence of UPLUS in this particular case, it is therefore the positive integer's sign and not the addition symbol.

The grammar used here uses only `left` and `%nonassoc` in a way that does not produce shift/reduce or reduce/reduce conficts at the production stage.

As stated above, token `END` designates the end of file. This is achieved by the declaration `0 "end of file"` next to its definition.

The nonterminal elements (which make up the context-free grammar) are respectively

```
%type <ast> translation_unit
%type <ast> scriptMFile
%type <ast> opt_delimiter
%type <ast> delimiter
%type <ast> null_lines
%type <ast> null_line
%type <ast> empty_lines
%type <ast> statement_list
%type <ast> statement
%type <ast> command_form
%type <ast> text_list
%type <ast> expr
%type <ast> reference
%type <ast> identifier
%type <ast> argument_list
%type <ast> matrix
%type <ast> rows
%type <ast> row
%type <ast> row_with_commas
```

```
%type <ast> colon_expr
%type <ast> assignment
%type <ast> s_assignee_matrix
%type <ast> m_assignee_matrix
%type <ast> reference_list
%type <ast> for_command
%type <ast> for_cmd_list
%type <ast> if_command
%type <ast> if_cmd_list
%type <ast> opt_else
%type <ast> global_command
%type <ast> global_decl_list
%type <ast> while_command
%type <ast> while_cmd_list
%type <ast> return_command
%type <ast> delimited_input
%type <ast> delimited_list
%type <ast> functionMFile
%type <ast> f_def_line
%type <ast> f_output
%type <ast> f_input
%type <ast> f_argument_list
%type <ast> f_body
%type <ast> parse_error
```

Bison assumes by default that the start symbol for the grammar is the first nonterminal specified. Since the parser uses bottom-up technique (see chapter 1), this is the non-terminal where the parsing stage ends up. This non-terminal here is *translation_unit*. A `%start` declaration is included to emphasize this, although it is not necessary.

```
%start translation unit
```

The MATLAB programme code input files are called M-files. M-files can be:

- Functions which may have arguments and may return ouptout arguments or nothing.
- Scripts which don’t have input or output arguments at all.
- Bad – written code which produce a parse error when parsed.

The grammar illustrates all this as:

```
translation_unit    :scriptMFile
                     | functionMFile
                     | parse_error
                     ;
```

This form of M-files consists of the corresponding non-terminal components, as follows:

```
scriptMFile       : opt_delimiter
                  | opt_delimiter statement_list
                  ;

functionMFile     : empty_lines f_def_line f_body
                  | f_def_line f_body
                  ;

parse_error       : LEXERROR
                  | error
                  ;
```

and the production goes on until we describe all the MATLAB cases (delimiters, null lines, empty lines, statements, expressions, command forms, identifiers, matrices etc). The important task here is to implement the grammar productions rules as new instances of the relevant classes so that the nodes of the abstract syntax tree wil be formed in memory. Remember that it is assumed that every non-terminal has its own class, all the tokens are objects of the class CTOKEN and all these classes are subclasses of the parent class *CASTSyntaxElement.* At the production of the non-terminals, every element is replaced by a new object of its corresponding class. For example:

```
translation_unit   :scriptMFile
                          {
                        $$ = CTransUnit  = new
                        CTransUnit((CScriptMFile*)$1
                       (MATFE::location *)&yyloc,
                        TranslationUnit__ScriptMFile);
                          }
                   | functionMFile
                          {
                           $$ = CTransUnit = new
                         CTransUnit((CFunctionMFile*)$1,
                        (MATFE::location *)&yyloc,
                         TranslationUnit__FunctionMFile);
                           }
                    | parse_error
                           {
                          $$ = CTransUnit::instance = new
                           CTransUnit((CParseError*)$1,
                          (MATFE::location *)&yyloc,
                           TranslationUnit__ParseError);
                          }
                            ;
```

The code declares that the *valuation* `$$` of the first production of `translation_unit` for a `scriptFile` M-File is achieved by the storage of the pointer `CScriptMFile*`(object of the class CScriptFile) at the parameter `$1.` It is stored at a memory location via `yyloc(see MACRO YY_DECL stated above). The relevant`

non-terminal production rule from the corresponding enumeration type defined in file ASTDefines.h which has been presented in previous chapter is:
TranslationUnit__ScriptMFile.
Two more complex examples follow. The first is

```
functionMFile        : empty_lines f_def_line f_body
                      {
                       $$ = new CFunctionMFile((CEmptyLines*)$1,
                       (CFDefLine*)$2, (CFBody*)$3,
                       (MATFE::location *)&yyloc,
                       FunctionMFile__EmptyLines_FDefLine_FBody);
                       cout << "FunctionMFile stage(1st rule)"<< endl;
                       cout << "-----------------------------" << endl;
                      }
                      | f_def_line f_body
                      {
                       $$ = new CFunctionMFile((CFDefLine*)$1,
                       (CFBody*)$2,
                       (MATFE::location *)&yyloc,
                       FunctionMFile__FDefLine_FBody);
                       cout << "FunctionMFile stage(2nd rule)"<< endl;
                       cout << "-----------------------------" << endl;
                      }
                        ;
```

The first case of production of a Function M-File uses the non-terminals empty_lines, f_def_line and f_body. Consequently the *valuation* $$ of the new instance of class CFunctionMFile is succedeed by the storage of *three parameters:* $1 (corresponds to an object of class CEmptyLines) ,$2 (corresponds to an object of class CFDefLine)and $3 (corresponds to an object of class CFDefLine). In other words when the parser produce in sequence the non-terminals empty_lines, f_def_line, f_body, the compiler understands that a functionMFile is at the input and represent it by the relevant node at the IR tree .
The second case of production of a functionMFile uses only the non-terminals f_def_line and f_body. Consequently the *valuation* $$ of the new istance of class CFunctionMFile is succedeed by the storage of *two parameters:* $1 (corresponds to an object of class CFDefLine)and $2 (corresponds to an object of class CFDefLine).Therefore when the parser produce in sequence the non-terminals f_def_line, f_body, the compiler understands again that a functionMFile is at the input .
The messages added at the end of the code (eg: FunctionMFile stage(2nd rule)") are helpful at the compilation of MATFE.y (more accurately at the

compilation of the file that Bison produces, MATFE_tab.c). They help to designate which is the appropriate case of the grammare rule that reduction uses in order to produce the relevant non-terminal.

The second example referres to the production of non-terminal element `f_def_line` (which is used at the `FunctionMFile` reduction).

```
f_def_line      : FUNCTION f_output ASSIGN IDENTIFIER f_input
                  {
                   $$ = new CFDefLine((CTOKEN*)$1,
                   (CFOutput*)$2, (CTOKEN*)$3,
                   (CTOKEN*)$4, (CFInput*)$5,
                   (MATFE::location *)&yyloc,
                  FDefLine__FUNCTION_FOutput_ASSIGN_IDENTIFIER_FInput);
                  cout << "f_def_line stage(1st rule)"<< endl;
                  cout << "-------------------------" << endl;
                }
                    | FUNCTION IDENTIFIER f_input
                {
                  $$ = new CFDefLine((CTOKEN*)$1,
                  (CTOKEN*)$2, (CFInput*)$3,
                  (MATFE::location *)&yyloc,
                   FDefLine__FUNCTION_IDENTIFIER_FInput);
                   cout << "f_def_line stage(2nd rule)"<< endl;
                   cout << "-------------------------" << endl;
                  }
                ;
```

The first case of production of a `f_def_line` uses the terminals `FUNCTION`, `ASSIGN` and `IDENTIFIER`, each being an object of class `CTOKEN`, and the non-terminals `f_output` and `f_input`. Consequently the valuation `$$` of the new instance of class `CFDefLine` is succedeed by the storage of five parameters: Three of them (`$1`, `$3`, `$4`) are objects of class `CTOKEN`, while `$2` corresponds to an object of class `CFOutput` and `$3` corresponds to an object of class `CFInput`. In other words when the parser produce in sequence a token (`FUNCTION`), the non-terminal `f_output`, another token (`ASSIGN`), another more token (`IDENTIFIER`) and the non-terminal `f_input`, the grammar procceds to a reduction, producing the non-terminal `f_def_line`.

The second case uses only the two tokens and the non-termnal `f_input`. The valuation `$$` of the new istance of class `CFDefLine` is succedeed by the storage of three parameters*:* A token (`FUNCTION`) for `$1`, another token (`IDENTIFIER`) for `$2` and an object of class `CFInput`. A sequence of these elements sequence produced by the parser results again to the production of the non-terminal `f_def_line`.

All the non-terminal elements therefore can be produced using these techniques. If the input MATLAB code is correct, the successive applications of Bison grammar rules

expressed by this way leads to the final production of `translation_unit` which will be the `root` of the abstract syntax tree that visualizes the construction in memory of desired Intermediate Representation which designates the end of the front_end compilation stage (of course the leaves of this tree are expected to be tokens).

The Flex and Bison files *MATFE.l* and *MATFE.y* are presented in Appendix 2. (note that by convention I use lowercase characters for nonterminal and capital letters for tokens).

Before closing this chapter, it is important to analyse a special parser phenomenon, which springs at the way MATLAB handles *assigments* and *matrices*.

Assigmnents in MATLAB come in three variations. The first allows the assigment of an arbritary expression to a variable of a matrix secrion. For example:

```
a(1:2:3, 1:3) =2;
```

creates a 3 by 3 matrix against the variable `a`, having the element 2 along the first and third rows and the element 0 in the remaings rows and columns. The second variation is identical except that the array section is enclosed in box brackets (tokens `LB` and `RB` respectively).

```
[a(1:2:3, 1:3)] =2;
```

The third variation allows a function to return more than one value. The right-hand side of the assignment is restricted to be a function call. For instance, the following example invokes the function `size()`.

```
[x,y] = size(1);
```

The above three assignment varieties are expressed using three productions. The non-terminals *s_assignee matrix* and *m_assignee_matrix* resemble to *matrix,* but they are not the same. For example, `SEMICOLON` and `LINE` delimiters are not allowed in an *m_asginnee_matrix*, or the elements of those special matrices can only be variables or array sections. These differences results to the need of the introdution of new nonterminals in order to decribe these kinds of matrices (*s_assignee_matrix* and *m_assignee_matrix)* cases. The production grammar rules for each case are:

```
matrix                  : LB rows RB
                        ;

assignment              : reference ASSIGN expr

                        | s_assignee_matrix ASSIGN expr
                        | m_assignee_matrix ASSIGN reference

                        ;
```

```
s_assignee_matrix        : LD reference RD
                         ;
m_assignee_matrix        : LD reference COMMA reference_list RD
                         ;
reference_list           : reference
                         | reference COMMA reference_list
                         ;
```

The tokens LD and RD correspond to the lexemes [and], as well as the tokens LB and RB, but *they are not the same,* because if they were so, a reduce-reduce coflict would occur, due to the productions reference_list à reference and exprà reference. This is because, until an assignment operator is encountered, the parser has no way to decide whether an assignment or an expression is being proccessed. The situation, before an assignment operator be returned by the scanner, is ambiguous because the parser could be in the midst of a sentence which reduces to the s_assignee_matrix or m_assignee_matrix nonterminals, or a sentence that reduces to the matrix nonterminal. Moreover a default situation cannot be adjusted, since a reduction to the the s_assignee_matrix or m_assignee_matrix nonterminals predominates in an **assignment** contect, while a reduction to the matrix nonterminal predominates in an **expression** context.

The solution of the above probolem is to program the scanner to return a different pair of tokens when an assignment to a box-bracket structute is detected. Thus when a regular expression \[functions, the input is read until a matching close bracket is returned. Consequently the following situations may occur:

- If a matching close bracket is not found, a LEXERROR token is returned since we are in an error situation.
- If a matching close bracket is not found, the scanner checks the next horizontal space character until the end of the input. If this is found and it matters to be a "=" lexeme, an eventual assignment to a box-bracket arises. This is because the context [x,y] = size(1); corresponds to an *assignment* to a box bracket structure, while the context [x,y] == size(1); is an *expression.* When therefore the scanner meets the "=" character, it attempts to read the next lexeme in the input stream. If one exists and if it is not '=", an asignment to a box bracketed structure is detected and the scanner returns the token LD. Otherwise the scanner returns the token LB.

All this is illustrated at the following code[3] in the translation of `[` lexeme, in the Flex file :

```
\[                              {
                                 if(Depth)
                                  {
                                    Depth++;
                                    CTOKEN* newtoken;
                                    newtoken = new CTOKEN(LB_,
                                   (MATFE::location *)&yylloc,
                                    TOKEN_LB);
                                    yylval->ast = newtoken;
                                    return(token::LB);
                                   }
                                  int current=0, next=0;
                                  char* buffer=0;
                                  int level=1, length=0;
                                  while(level &&
                                     (current=yyinput())!=EOF)
                                    {

                                  buffer=(char*)realloc(buffer,++length);
                                  *(buffer+length-1)=current;
                                   if(current=='[') level++;
                                    if(current==']') level--;
                                    }
                                  if(level)
                                    {
                                      CTOKEN* newtoken;
                                      newtoken = new CTOKEN(LEXERROR_,
                                      (MATFE::location *)&yylloc,
                                       TOKEN_LEXERROR);
                                       yylval->ast = newtoken;
                                       return(token::LEXERROR);
                                     }
                                   while((current=yyinput())!=EOF)
                                  {

                                       buffer=(char*)realloc(buffer,++lengt
                                       h);
                                   *(buffer+length-1)=current;
                                       if (current!=' '&& current!='\t')
                                       break;
                                   }
                                   if((next=yyinput())!=EOF)
                                    {

                                         buffer=(char*)realloc(buffer,++len
                                         gth);
                                    *(buffer+length-1)=next;
                                      }
                                      for( ;length > 0; length-- )
                                      unput(*(buffer+length-1));
                                      free(buffer);
                                      Depth=1;
                                      if(current =='=' && next!='=')
                                    {
                                      CTOKEN* newtoken;
                                      newtoken = new CTOKEN(LD_,
```

[3] I used (slightly modified) the code included in the paper mentioned at previous footnote

```
          (MATFE::location *)&yylloc,
          TOKEN_LD);
          yylval->ast = newtoken;
          return(token::LD);
         }
        else {
        return(token::LB);
           }
        }
```

The scanner keeps the current nest level of a box bracket in the integer variable `Depth`.This information enables an optimazation that avoids reading every opening box bracket. When such a lexeme is encountered, `Depth` is incremented by one. Similarly when a closing box bracket is red, `Depth` is decremented by one. Moreover the scanner matches every opening box bracket with a closing one. Hence the value of variable `Depth` corresponds to how deep the box bracket is in the boxbracket nesting.Therefore, on meeting a box-bracket, the scanner needs to read ahead only if the current value of `Depth` is zero.

In the same way, the `RD` token is returned only when the regular expression `\]/{HSPACE}*=[^=]` is matched. Fortunately once an `LD` token is returned by the scanner, the action of the above regular expression is guaranteed. All other closing boxbrackets will match the regular expresion `\]` that returns the `RB` token.

The code therefore in the Flex file will be:

```
\]/{HSPACE}*=[^=]       {
                        BEGIN(INITIAL);
                        Depth--;
                        CTOKEN* newtoken;
                        newtoken = new CTOKEN(RD_,
                        (MATFE::location*)&yylloc,
                        TOKEN_RD);
                        yylval->ast = newtoken;
                        return(token::RD);
                               }
\]                       {
                         BEGIN(QuoteSC);
                         Depth--;
                         CTOKEN* newtoken;
                         newtoken = new CTOKEN(RB_,
                         (MATFE::location *)&yylloc,
                         TOKEN_RB);
                         yylval->ast = newtoken;
                         return(token::RB); }
```

3.3 Conclusions

This chapter is the most important part of the project. It explains in details how the scanning and parsing stages of the compiler work in the project's object oriented environment and what will be the form of terminal and non-terminal elements. In the next chapter we focus at the implementation of the classes defined in chapter 2, which leads to the vectorization of the parsing productions and the construction of the abstract syntax tree in memory.

Chapter 4

Production of the abstract syntax tree

The first part of the core of the project is now completed. Assuming that everything operates correctly so far, I am ready to construct the nodes of the *IR* abstract syntax tree (AST), via the implementation of all the classes defined in file `ASTSyntaxElements.h` in chapter 2. Thus I have to form the constructor of the terminals (in class CTOKEN), as well as the relevant constructors included in the class *of every non-terminal element,* according to the productions grammar rules described in previous chapter. Although this stage, from the code point of view, is not very complicated, it requires enhanced attention since it leads to a very long program in order to construct everything correctly.

I begin of the implementation of the main class `CASTSyntaxElement` since the others are sub classes of it. Here is its definition:

```
class CASTSyntaxElement {
public:

CASTSyntaxElement(NONTERMINAL_PRODUCTION_RULE pr, MATFE::location *
loc);
~CASTSyntaxElement();
virtual int map_syntax_element(CASTSyntaxElement *,int m=-1){return -
1;}

TERMINAL_CODE m_TerminalCode;
NONTERMINAL_CODE m_NonTerminalCode;
NONTERMINAL_PRODUCTION_RULE m_NonTerminal_ProductionCode;
static unsigned int m_ObjectSerialNumberCounter;
unsigned int m_NumberOfDescendants;
unsigned int m_ObjectSerialNumber;
vector <CASTSyntaxElement *> m_Descendants;

protected:
MATFE::location *m_location;
};
```

The relevant implementation should be:

```
CASTSyntaxElement::CASTSyntaxElement(NONTERMINAL_PRODUCTION_RULE pr,
MATFE::location * loc) {

     m_NonTerminal_ProductionCode=pr;
     m_location=loc;
     m_ObjectSerialNumberCounter++;
     m_ObjectSerialNumber=m_ObjectSerialNumberCounter;

}
```

The initialization of counter `m_ObjectSerialNumberCounter` is compulsory|:

```
unsigned int CASTSyntaxElement::m_ObjectSerialNumberCounter=0;
```

We have also to define the destructor:

```
CASTSyntaxElement::~CASTSyntaxElement() {
  m_ObjectSerialNumberCounter--;
}
```

As far as concerns the cunstructor of tokens:

```
    CTOKEN::CTOKEN(TERMINAL_CODE ct,MATFE::location *loc,
       NONTERMINAL_PRODUCTION_RULE pr)
    :CASTSyntaxElement(pr,loc){
    m_NonTerminalCode = token_;
    m_NumberOfDescendants=1;
}
```

According to the `NONTERMINAL_CODE` enum (reported in chapter 2), all the terminal elements are represented from the non-terminal **token_.** The `m_NumberOfDescendants` value is 1, since the token_ nonterminal is formed from only one element.

Now I am ready to model the constructors of every non-terminal. Since the entire program (included in Appendix 2) is very long, the analysis is based to some indicative examples.

The first example forms the constuctors of the final stage of grammar rules reduction during the parsing of an M-file, thus the terminal *translation_unit_* . Its relevant class is `class CTransUnit.` The corresponding grammar rule (stated also in chapter 2) from Bison File is:

```
translation_unit   :scriptMFile
                         {
                          $$ = CTransUnit  = new
                          CTransUnit((CScriptMFile*)$1
                         (MATFE::location *)&yyloc,
                         TranslationUnit__ScriptMFile);
                          }
                          | functionMFile
                          {
                         $$ = CTransUnit = new
                         CTransUnit((CFunctionMFile*)$1,
                        (MATFE::location *)&yyloc,
                         TranslationUnit__FunctionMFile);
                          }
                          | parse_error
                            {
                          $$ = CTransUnit::instance = new
                          CTransUnit((CParseError*)$1,
                         (MATFE::location *)&yyloc,
                          TranslationUnit__ParseError);
                          }
                          ;
```

There are three cases which reduce to `translation_unit`, therefore *three* instances of that class. Their definitions (included in header file ASTSyntaxElements.h) are below:

```
class CTransUnit:public CASTSyntaxElement{
public:
      CTransUnit(CScriptMFile *, MATFE::location *,
            NONTERMINAL_PRODUCTION_RULE);

      CTransUnit(CFunctionMFile*, MATFE::location *,
            NONTERMINAL_PRODUCTION_RULE);

      CTransUnit(CParseError*, MATFE::location *,
            NONTERMINAL_PRODUCTION_RULE);

protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c);

 }
};
```

Hence the implementation of the class must include *three* constructors which derive from the above definitions.

```
CTransUnit::CTransUnit(CScriptMFile* ScriptMFile,
                        MATFE::location *loc,
                        NONTERMINAL_PRODUCTION_RULE pr)
       :CASTSyntaxElement(pr,loc){
      m_NonTerminalCode= translation_unit_;
      int position = map_syntax_element(ScriptMFile);
      cout << "CCScriptMFile pos:" << position <<endl;
      m_Descendants.push_back(ScriptMFile);
      cout <<"Translation unit reached!" << endl;
}

CTransUnit::CTransUnit(CFunctionMFile* FunctionMFile,
                        MATFE::location *loc,
                        NONTERMINAL_PRODUCTION_RULE pr)
       :CASTSyntaxElement(pr,loc){
      m_NonTerminalCode= translation_unit_;
      int position = map_syntax_element(FunctionMFile);
      cout << "CFunctionMFile pos:" << position <<endl;
      m_Descendants.push_back(FunctionMFile);
      m_NumberOfDescendants=1;
      cout <<" Translation unit reached!" << endl;
}


CTransUnit::CTransUnit(CParseError* ParseError,
                       MATFE::location *loc,
                       NONTERMINAL_PRODUCTION_RULE pr)
       :CASTSyntaxElement(pr,loc){
      m_NonTerminalCode= translation_unit_;
      m_Descendants.push_back(ParseError);
      m_NumberOfDescendants=1;
      cout <<"Parsing errors" << endl;
}
```

The three costructors are similar. According to the case, the invoked constructor feeds the vector `m_Descendants` with objects `<CASTSyntaxElement*>` which are the `ScriptMFile`, the `FunctionMFile` and the `ParseError`. Notice that each of these elements is also pointer to an object of *its relevant class.* But what are the values of those elements? This is the work of function `map_syntax_element` which has been defined `virtual` in the base class `CASTSyntaxElement` in order to be redefined, if it is needed, in every subclass.

```
int CTransUnit::map_syntax_element(CASTSyntaxElement *el, int m){
      cout <<"CTransUnit Stage" << endl;
      cout <<"-------------" << endl;
    cout << "In TransUnit::map_syntax_element() -- " << endl;
    cout << "NTPC : " << m_NonTerminal_ProductionCode << endl;
    cout <<  "el->m_NonTerminalCode : " << el->m_NonTerminalCode <<
endl;
      switch(m_NonTerminal_ProductionCode){
            case TranslationUnit__ScriptMFile:
                  if ( el->m_NonTerminalCode == scriptMFile_ ){
                        return 0;
                      }
                  break;
        case TranslationUnit__FunctionMFile:
                  //TranslationUnit:FunctionMFile
                  if ( el->m_NonTerminalCode == functionMFile_){
                        return 0;
                      }
                  break;
        case TranslationUnit__ParseError:
                  //TranslationUnit:ParseError
                  if ( el->m_NonTerminalCode == parse_error_ ){
                        return 0;
                      }
            break;
      default:
                  cout << endl << "Error in
TranslationUnit::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);


      }
return -1;
}
```

The main part of this function is a *switch* statement that includes cases which correspond to the suitable, for each parsed M-file, grammar rule. Every case verifies that the right non-terminal is parsed and returns an integer. In *translation_unit* circumstance all integers returned are zero, since there is only one structure element in each case that forms *translation_unit* (there is a more complex example later on). If

none of the cases is matched, an error message is sent (since that means that something is wrong in parsing) and the function terminates.

Moreover `map_syntax_element` provides some messages which help to watch the non-terminal elements production when the program is executed. The usefullness of these messages will be clearer at the next example .

So the construcors are formed. Since reaching *translation_unit* means that the parsing stage has ended successfully, every constructor includes a message which .indicates the success or failure of parsing stage.

We proceed now to a more complicated example which concerns the production of the *assignment* non-terminal. The relevant class is `CAssignment.` According to the grammar, the parser yields an assignment in three cases. The following fragment of Bison file illustrates them.

```
assignment              : reference ASSIGN expr
                              {
                              $$ = new CAssignment((CReference*)$1,
                               (CTOKEN*)$2,(CExpression*)$3,
                               (MATFE::location *)&yyloc,
                                Assignment__Reference_ASSIGN_Expr);
                                cout <<"Assignment stage(1st rule)" <<
                              endl;
                              }
                        | s_assignee_matrix ASSIGN expr
                         {
                          $$ = new  CAssignment((CSassigneeMatrix*)$1,
                          (CTOKEN*)$2,(CExpression*)$3,
                          (MATFE::location *)&yyloc,
                          Assignment__SAssigneeMatrix_ASSIGN_Expr);
                          cout <<"Assignment stage(2nd rule)" << endl;
                              }
                        | m_assignee_matrix ASSIGN reference
                            {
                            $$ = nCAssignment((CMassigneeMatrix*)$1,
                            (CTOKEN*)$2,(CReference*)$3,
                            (MATFE::location *)&yyloc,
                        Assignment__MAssigneeMatrix_ASSIGN_Reference);
                        cout <<"Assignment stage(3rd rule)" << endl;
                             }
                              ;
```

According to the first rule, a reduction to an assignment occurs whenever the token *ASSIGN* succeeds a *reference* and it is followed by an *expression* (expr). The relative therefore constructor should be:

```
CAssignment::CAssignment(CReference *ref, CTOKEN* token,
                         CExpression* exp,MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = assignment_;
  int position = map_syntax_element(ref);
```

```
  cout << "Reference pos:" <<position << endl;
  m_Descendants.push_back(ref);
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp);
  m_NumberOfDescendants =3;
}
```

The structural elements of the CAssignment constructor "push_back" the vector `m_Descendants` with values returned by the function `map_syntax_element`.

The second and the third production rules of assignmnent use the non-terminals *s_assignee_matrix* and *m_assignee_matrix* which are special cases of matrices in MATLAB. Whenever a `s_assignee_matrix` is followed by the token *ASSIGN* and the non-terminal *expr* is found afterwards, a reduce to an assignment occurs. The relevant constructor therefore should be:

```
CAssignment::CAssignment(CSassigneeMatrix *csa, CTOKEN* token,
                            CExpression* exp,MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = assignment_;
  int position = map_syntax_element(csa);
  cout << "CSassigneeMatrix pos:" <<position << endl;
  m_Descendants.push_back(csa);
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp);
  m_NumberOfDescendants =3;
}
```

On the other hand, according to the third grammar rule of assignment production, whenever a *m_assignee_matrix* precedes the token *ASSIGN* and the non-terminal *expr* follows, a reduce to an assignment occurs. Hence the constructor should be:

```
CAssignment::CAssignment(CMassigneeMatrix *msa, CTOKEN* token,
                    CReference* ref,MATFE::location *loc,
                    NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = assignment_;
  int position = map_syntax_element(msa);
  cout << "CMassigneeMatrix pos" <<position << endl;
  m_Descendants.push_back(msa);
  m_Descendants.push_back(token);
  m_Descendants.push_back(ref);
  m_NumberOfDescendants=3;
};
```

Now the relevant function `map_syntax_element` differs from the corresponding one at `tranlation _unit`. Here every case of the *switch* statement includes `if` and `else if` statements which verifies the structural elements that make the production. For instance, in case `m_assignee_matrix ASSIGN reference`, `map_syntax_element` returns integer 0 for `m_assignee_matrix`, integer 1 for

token (ASSIGN) and integer 2 for reference. These are the positions of CASTSyntax elements in vector m_Descendants.

Hence the function map_syntax_element is formed as below:

```
int CAssignment::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "In CAssignment::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "s_assignee_matrix_: " << s_assignee_matrix_  << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     Assignment__Reference_ASSIGN_Expr:
                  //Reference_ASSIGN_Expr
                  if ( el->m_NonTerminalCode = = reference_ ){
                        return 0;}
                  else if ( el->m_NonTerminalCode = = token_){
                        return 1;}
                  else if ( el->m_NonTerminalCode == expr_ ){
                        return 2;}
        break;
            case Assignment__SAssigneeMatrix_ASSIGN_Expr:
            //SAssigneeMatrix_ASSIGN_Exp
                  if ( el->m_NonTerminalCode = = s_assignee_matrix{
                        return 0;}
                  else if ( el->m_NonTerminalCode = = token_){
                        return 1;}
                  else if ( el->m_NonTerminalCode == expr_ ){
                        return 2;}
        break;
            case Assignment__MAssigneeMatrix_ASSIGN_Reference:
            //MAssigneeMatrix_ASSIGN_Reference
                  if ( el->m_NonTerminalCode == m_assignee_matrix_ ){
                        return 0;}
                  else if ( el->m_NonTerminalCode == token_){
                        return 1;}
                  else if ( el->m_NonTerminalCode == reference_ ){
                        return 2;}
                  break;
            default:
      cout << endl << "Error in Assignment::map_syntax_element"
                  << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}
```

The initial messages are very useful when the compilation of an input file leads to a *tranlation_unit_error* or to a *segmentation fault* (run-time error). This may involve potential weaknesses of context-free grammar used or omissions in program construction. The message “NTPC:” indicates the grammar rule which may cause the error and the message “el->m_NonTerminalCode” indicates the non-terminal responsible for the irregularity.

The third example describes the production of the *reference* non-terminal. The appropriate part of Bison code is:

```
reference                : identifier
                          {
                               $$ = new CReference((CIdentifier *)$1,
                               (MATFE::location *)&yyloc,
                               Reference__Identifier);
                               cout << "Reference stage(1srule)"<<endl;
                               cout << "-------------------------"
                               endl;
                         }
                         | identifier LPAREN  argument_list RPAREN
                           {
                            $$ = new CReference((CIdentifier *)$1,
                            (CTOKEN*)$2,(CArgumentList*)$3,(CTOKEN*)$4,
                             (MATFE::location *)&yyloc,

Reference__Identifier_LPAREN_ArgumentList_RPAREN);
                            cout << "Reference stage(2nd rule)"<<endl;
                            cout << "------------------------" << endl;

                              }
                              ;
```

A grammar fuzziness phenomenon is found here. According to the first rule, an identifier *is* a reference. On the other hand the second reference production rule *begins* from an identifier. This leads to a *shift-reduce* conflict because the parser doesn't know if it must *reduce* to reference state (so the second rule becomes useless) or *shift* to the next token (potentially a `LPAREN`)to apply the second grammar rule. To remedy this problem, the non-terminal *identifier* is accompanied by the precedent operator `%nonassoc MINOR_PREC`, which declares **lower precedence** for this element.

```
reference                : identifier %prec MINOR_PREC
```

The parser therefore is forced to shift to the next token `LPAREN` which has a declared `%nonassoc` precedence (see previous chapter) and the shift/reduce conflict is resolved. The constructors and the `map_syntax_element` function for the `reference` object are formed as below:

```
CReference::CReference(CIdentifier* id,
                      MATFE::location *loc,
                      NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

   m_NonTerminalCode = reference_;
   int position = map_syntax_element(id);
   cout << "CIdentifier pos:" << position << endl;
   m_Descendants.push_back(id);
   m_NumberOfDescendants=1;

}

CReference::CReference(CIdentifier *id, CTOKEN *token,
                                 CArgumentList *arg, CTOKEN *token1,
                                 MATFE::location *loc,
                                 NONTERMINAL_PRODUCTION_RULE pr)
```

```
  :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = reference_;
  int position = map_syntax_element(id);
  cout << "CIdentifier pos:" << position << endl;
  m_Descendants.push_back(id);
  m_Descendants.push_back(token);
  m_Descendants.push_back(arg);
  m_Descendants.push_back(token1);
  m_NumberOfDescendants=4;
}

int CReference::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "In CReference::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
            case Reference__Identifier:
                  if ( el->m_NonTerminalCode == identifier_ ){
                        return 0;
                      }
        break;
            case Reference__Identifier_LPAREN_ArgumentList_RPAREN:
                  if ( el->m_NonTerminalCode == identifier_){
                        return 0;
                  }
                  else if (el->m_NonTerminalCode == token_){
                              return 1; }
                  else if (el->m_NonTerminalCode == argument_list_){
                        return 2;}
                  else if (el->m_NonTerminalCode == token_){
                        return 3;
                  }
            break;
      default:
                  cout << endl << "Error in
Reference::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
                  ;
      }
return -1;
}
```

The constructors of all the non-terminal elements of the grammar must be created as the three examples above. The entire code is presented in Appendix 2. The execution of this program (in tandem with Bison and Flex produced code) provides the construction of the Abstract Syntax Tree in memory which is the desired Intermediate Representation (IR).

4.1 Conclusions

So that's all? Is the front_end compiler ready? Of course not. A set of parsing classes is constructed but the *parse* object is still missing. There is no way to interact with the

parser, to control the start and the termination of scanning and parsing procedures, or even to declare the M-file input. An *interface* therefore must be constructed. This is the subject of the following chapter.

Chapter 5

Design of the interface

The context of previous chapters explained the structure of scanning and parsing functions of the front_end compiler which is almost ready. The target of this chapter is to construct an interface program to control the compiler. In other words we have to design the *driver*.

5.1 The driver files

This driver must include suitable functions to handle the beginning and the end of the scanning stage, the constructor of parser driver (defined as class MATFE driver in Bison file), functions for the error and debug mode and other useful functions. File *driver.h* stated below summarizes the necessary definitions.

File driver.h

```
#ifndef DRIVER_H
#define DRIVER_H

#include <string>
#include "MATFE.tab.h"
#include "ASTDefines.h"
#include "ASTSyntaxElements.h"
#include <iostream>
#include <fstream>


extern FILE* yyin;

YY_DECL;

class MATFE_driver
{
  public:
   MATFE_driver();
   virtual ~MATFE_driver();
   int result;
   std::string file;
   bool debug_mode;
   bool trace_scanning;
   bool trace_parsing;

   // Handling the scanner.
   void scan_begin();
   void scan_end();

   // Run the parser. Return 0 on success.
   int parse (const std::string& f);
```

```
    // Error handling.
    void error (const MATFE::location& l, const std::string& m);
    void error (const std::string& m);
// debug
    void printInfo(const char* msg);
    void set_debug_mode(bool flag);
    string to_string();
   };
#endif // DRIVER_H
```

Firstly it is useful to define some functions to encapsulate the coordination with the Flex scanner. These (void) functions are `scan_begin` and `scan_end`. The implementation of `scan_begin` is included in the relevant file `driver.cpp` and it uses the `f_open` function which is reponisble to start the scanning stage of the (candidate for compile) input MATLAB source code (or code of another vector language). This function is stored in variable `yyin`.

```
void MATFE_driver::scan_begin (){
  yyin = fopen(file.c_str(),"r");
}
```

Hence the variable `yyin` is declared as `extern`. The function `scan_end` simply close the file at the end of the scanning stage.

```
void MATFE_driver::scan_end (){
    fclose(yyin);
 }
```

Now it's time to start the *parsing* stage. The parsing function `yylex` must be invoked via the *macro* `YY_DECL` which defines its arguments (see Chapter 3, Bison file). Then the parser class `MATFE_driver` which is the heart of the parsing operation has to be defined.

This class includes the constructor and the relevant destructor of `MATFE_driver` and some boolean functions which contols the scanning and parsing actions at the implementation of the constructor.

```
MATFE_driver::MATFE_driver()
: trace_scanning(false), trace_parsing(false){
}
```

Morover `class MATFE_driver` includes the scanner start and end functions stated above, a group of functions to control the parsing errors, and of course the function `parse(const std::string& f)` which makes the system work. This function takes as argument a string which correspods to the input file name and returns an integer (0 if compiling is successful). It's worth to implement it here:

```
int MATFE_driver::parse (const std::string &f)
{
    file=f;
    scan_begin();
```

```
    printInfo("Creating parser object...");
    MATFE::MATParserClass parser(*this);
    parser.set_debug_level(trace_parsing);
    printInfo("Created.");
    printInfo("Initiating parsing...");
    int res = parser.parse();
    scan_end();
    return res;
}
```

The function `scan_begin()`is invoked to start scanning the input `f`.Similtaneously an object of parser class is created. Function `set_debug_`level is a parser method which sets the trace level. Its (bolean) value depends from the flag returned by function `set_debug_mode`. According to the flag sent therefore, the parser enters the running mode. When it ends, function `scan_end()` closes the file and an integer is returned which denotes the success or failure of parsing execution.

Finally the driver includes the function `error` for handling the error state. It has two defined forms, with one or two arguments. The latter (implemented in Bison file MATFE.y) registers the errors to the driver. The former sends the error message to the standard output.

```
void MATFE::MATParserClass::error(const
MATFE::MATParserClass::location_type& l, const std::string& m){
      driver.error (l, m);
      }


void MATFE_driver::error (const std::string& m){
                std::cerr << m << std::endl;
       }
```

5.2 The main function

Since the interface is ready, the remaining procedure is to write the *main* function of the project. There is not much work to do here. Firstly `driver` object of the class driver is invoked and `trace_parsing` is set to *true*, via the driver function `set_debug_mode`. After the welcoming message, a `for loop` detects the characters of `char* argv[]` matrix argument. Two switches (“-s”, “-p”) are created via `if` control statements, which are used for testing the interface function. These switches cancel the scanner and the parser, setting their *trace* parameter to false state.

File `main.cpp`

```
#include <iostream>
```

```
#include "driver.h"
#include "MATFE.tab.h"

using namespace std;

int main (int argc, char *argv[]){
    MATFE_driver driver;
    driver.set_debug_mode(true);
    char** argv_init = argv;
    cout << "Start scanning..." << endl;
    for (int i = 0; i < argc; i++)
        argv = argv_init+i;
        if (*argv == std::string ("-p")) {
            driver.trace_parsing = false;
        }
        else if (*argv == std::string ("-s")) {
            driver.trace_scanning = false;
        }
        else {
            string filename = string(*argv);
            cout << "Scanning file: "<< filename << endl;
               if (!driver.parse(filename)) {
             std::cout << "OK. End of file" << std::endl;
            }
        }
    cout << "It was a nice parsing!" << endl;
    return 0;
}
```

If none of these switches are detected, the `for loop` reads the entire argument `*argv` and the result is assigned to the string variable `filename` which is the input MATLAB M-File. Finally anotgher `if` statement indicates the parsing termination. If the parser does not return any error, some relevant messages denote that the project execution has been successful.

5.3 Conclusions

Since the interface has been constructed, the project is almost finished. We can start testing it by using MATLAB input files. The result is the formation of the desired Intermediate Representation in memory, in the form of an anstract syntax tree (AST), or the indication of errors in input code if any. Nevertheless, a visual representation of this *IR* would be very useful, in order to have an idea (in a graphic environment) of *which nonterminals and which grammar rules* Bison and Flex tools used to produce this AST and *what* is the magnitude and the shape of it. The last chapter explains how this can be achieved.

Chapter 6
AST Graphical Representation

DOT is a plain text graph description language. It is a simple way of presenting graphs that both humans and computer programs can use. DOT graphs are typically files with extension .gv or .dot. DOT files which can be processed by various programs. Some, like *dot*, *neato*, *twopi*, *circo*, *fdp*, and *sfdp*, read a DOT file and render it in graphical form. Others, like *gvpr*, *gc*, *accyclic*, *ccomps*, *sccmap*, and *tred*, read a DOT file and perform calculations on the represented graph. Finally, others, like *GVedit*, *KGraphEditor*, *lefty*, *dotty*, and *grappa*, provide an interactive interface. Most of these programs are included in **Graphviz** (abbreviation of Graph Visualization Software). This is a package of open-source tools developed by AT&T Labs Research, for drawing graphs specified in DOT language scripts.

At its simplest form, DOT can be used to describe an undirected graph. An undirected graph shows simple relations between objects such as airline connections between cities or social relations between people. A new graph is declared by the keyword *graph* and nodes are described within curly braces. A double-hyphen (--) is used to show connections between the nodes. Similarly, DOT can represent direct graphs as *dependency trees*, which is the case here. The syntax is the same as for undirected graphs, except that a new graph begins with the keyword *digraph* and an arrow (->) is used to show relationship between nodes. In detail, a DOT input for a simple dependency tree which consists of five digits as nodes, has the following form:

```
digraph G{1 ->2; 2->3; 2 ->4; 4 ->5;}
```

The above code produces the following layout:

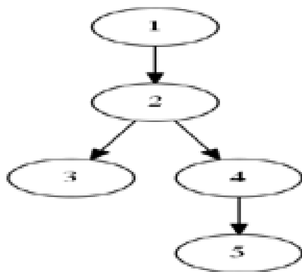

Image 2. Graphviz layout of a simple dependency tree

6.1 A simple IR visualization

Now I consider that my front_end compiler treats a *very simple input* for compilation. What should be the desired *graphviz layout* of the produced abstract syntax tree (AST)? Let's assume that the input M_File consists of only a *single lexeme* e.g. a `semicolon` and nothing else. The scanner verifies it and sends the corresponding token to the parser. According to the context- free grammar used, the productions in the parsing stage will be the following:

```
null_line         | token
null_lines        | null_line
delimiter         | null_lines
empty_lines       | token
delimiter         | empty_lines
opt_delimiter     | delimiter
scriptMFile       | opt_delimiter
translation_unit  | scriptMFile
```

The relevant `DOT` input file for the construction of the AST (with an addition of a counter to calculate the number of produced nodes) and the corresponding layout should be:

```
digraph G {
null_line_2 -> token_1;
null_lines_3 -> null_line_2;
delimiter_6 -> null_lines_3;
empty_lines_5 -> token_4;
delimiter_6 -> empty_lines_5;
opt_delimiter_7 -> delimiter_6;
scriptMFile_8 -> opt_delimiter_7;
translation_unit_9 -> scriptMFile_8;
}
```

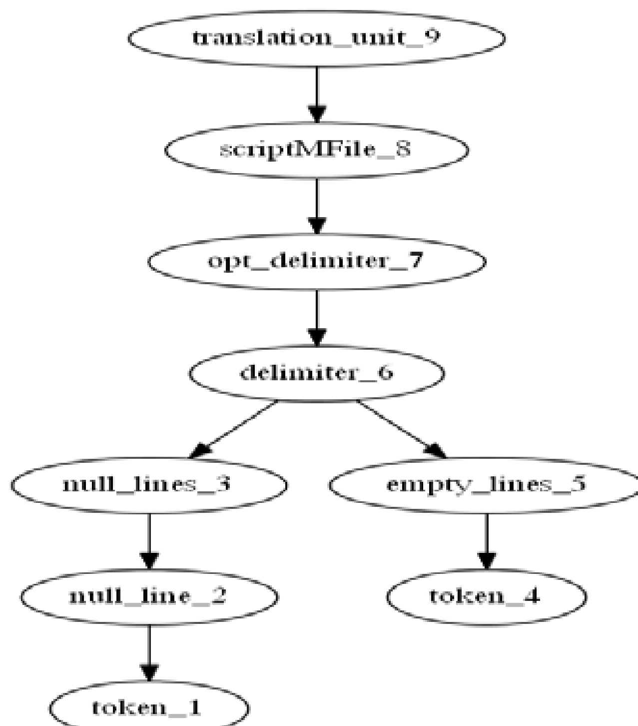

Image3. Graphviz layout of a simple IR

6.2 Production of the DOT layout

The DOT layout form illustrated at image 3 is the desired graphical representation of the abstract syntax tree. However, the project is not yet ready to produce it. There are two problems to solve. The first concerns the *names* of the non-terminal elements.

The nodes of the AST are members of the class CASTSyntaxElement. Let's recall its definition:

```
class CASTSyntaxElement {
public:

  CASTSyntaxElement(NONTERMINAL_PRODUCTION_RULE pr, MATFE::location *
loc);
  ~CASTSyntaxElement();
  virtual int map_syntax_element(CASTSyntaxElement *, int m=-1){return -
1;}

  TERMINAL_CODE m_TerminalCode;
  NONTERMINAL_CODE m_NonTerminalCode;
  NONTERMINAL_PRODUCTION_RULE m_NonTerminal_ProductionCode;
  static unsigned int m_ObjectSerialNumberCounter;
  unsigned int m_NumberOfDescendants;
  unsigned int m_ObjectSerialNumber;
  vector <CASTSyntaxElement *> m_Descendants;
```

The m_NonTerminalCode and m_TerminalCode variables are members of types NONTERMINAL_CODE and TERMINAL_CODE which have been defined as *enum data types* (see Chapter 2). Therefore they are *integers*. I do not have any structure which contains the *real names*. Consequently, I have to construct one.

Although the project includes a definition file ASTDefine.h, the corresponding source file ASTDefine.cpp. is still undone. I can embed an array which contains the names of the non-terminals in this file.

File ASTDefines.cpp

```
#include "ASTDefines.h"
#include "ASTSyntaxElements.h"

 char *nonterminals[] = {"token_", "translation_unit_"
,"scriptMFile_","opt_delimiter_", "delimiter_", "null_lines_",
"null_line_",  "empty_lines_","statement_list_", "statement_",
"command_form_", "text_list_",
"expr_", "reference_", "identifier_", "argument_list_", "matrix_",
"rows_", "row_", "row_with_commas_", "colon_expr_", "assignment_",
"s_assignee_matrix_", "m_assignee_matrix_",
"reference_list_","for_command_", "for_cmd_list_", "if_command_",
"if_cmd_list_", "opt_else_",
"global_command_", "global_decl_list_", "while_command_",
"while_cmd_list_", "return_command_", "delimited_input_",
"delimited_list_", "functionMFile_",
"f_def_line_", "f_output_", "f_input_", "f_argument_list_", "f_body_",
"parse_error_"};
```

Now I have to match the members of the above array to the corresponding members of the *enum type* NONTERMINAL_CODE. This is achieved by adding a new variable at the private area of CASTSyntaxElement.

```
char* graphstring;
```

In order to perform this matching, I must include the following code lign in the implementation of *every non-terminal constrructor*.(File ASTSyntaxElements.cpp).

```
this->graphstring = nonterminals[this->m_NonTerminalCode];
```

As an example, the constructor of non-terminal *delimiter* becomes:

```
CDelimiter::CDelimiter(CEmptyLines* els,
                           MATFE::location *loc,
                           NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

  m_NonTerminalCode = delimiter_;
  int position = map_syntax_element(els);
  cout <<"Del.Position:" <<position <<endl;
  m_Descendants.push_back(els);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;

};
```

The index [this->m_NonTerminalCode]corresponds to the delimiter_ member of the enum NONTERMINAL_CODE which has the value "5". Consequently the value assigned to the variable graphstring (inherited from parent class CASTSyntaxElement) is nonterminals[5], which is the desired name "delimiter_". The matching therefore has been achieved. I can extract the names of all the non-terminals by this technique, so the first problem is solved.

The second problem is that the AST produced by the compilation of a common MATLAB source file contains hundrends, maybe thousands of nodes. It is impossible therefore to write the DOT input by hand as we did at the simple example of the preceding function. Therefore I must write code that generates the DOT file automatically during file compilation.

Algorithms that have to manage a large and rather uknown number of actions which are repetead in a self-similar way, usually use *recursion*. Besides, if we observe the parser from the Bison tool point of view, it is evident that non-terminal elements production via the context–free grammar is a *recursive* procedure. The abstract syntax tree is formed in a *recursive* way. The DOT file production function thereafter should be recursive.

Since the visualization of AST must be included in the parsing stage but it is not a procedure of a common parser, I prefer to declare it in the interface files, rather than

within the classes of Flex and Bison. In driver file `driver.h` I declare (as a member of `class MATFE_driver)`, two new functions which have the same name `ASTGraphEmmiter` but different number of arguments. One *private* where we pass two arguments and one *public* with one argument.

```
private:
  void ASTGraphEmmiter(CASTSyntaxElement* parent, ofstream* out);
public:
  void ASTGraphEmmiter(CASTSyntaxElement *parent);
```

The argument `CASTSyntaxElement* parent` is the non-terminal element of which the AST will be drawn after each recursive call of `ASTGaphEmitter` and it will be exported to a graphical file `out` (so an `ofstream` interface must be defined). The implementation therefore of private `ASTGraphEmmiter` should be:

```
void MATFE_driver::ASTGraphEmmiter(CASTSyntaxElement *parent, ofstream*
out){
    for (unsigned int i=0; i < (parent->m_Descendants.size()); i++)
      {
        (*out)  << parent->graphstring << parent->m_ObjectSerialNumber
<< ' ';
        (*out) << "-> ";
       (*out)  << parent->m_Descendants[i]->graphstring  << parent-
>m_Descendants[i]->m_ObjectSerialNumber<<";" << endl;
       ASTGraphEmmiter(parent->m_Descendants[i], out);
    }
}
```

I use the `ofstream` interface to send the form of the `DOT` syntax of the graphical file. Moreover I use a `for loop` which form the syntax:

```
parent->m_Descendants[i]->graphstring<<parent->m_Descendants[i]-
>m_ObjectSerialNumber<<";"
```

for *every* component of the relevant `m_Descendants` vector of the `parent non-terminal` element constructor. Then the function `ASTGraphEmmiter` invokes itself and all the parts of the abstract syntax tree are exported recursively until the end of the compilation.

The public function `ASTGraphEmmiter(CASTSyntaxElement *parent)` manages the remaining parts. Firstly an ofstream interface `graphout` is defined and the output graphical file `mygraph.grv` is openned. Then the private function `ASTGraphEmmiter` is invoked with the appropriate arguments and forms the AST. Finally, assuming that everything worked correctly, `graphout` interface closes the output file.

```
void MATFE_driver::ASTGraphEmmiter(CASTSyntaxElement *parent){
  ofstream graphout;
  graphout.open("mygraph.grv");
  graphout << "digraph G {" << endl;
  ASTGraphEmmiter(parent, &graphout);
```

```
  graphout << "}" << endl;
               graphout.close();
  }
```

So the second problem is solved as well. The code which can produce automatically a `DOT` graphical file is ready. But, in order to make it work, I must incorporate its function to the parsing stage.

Thanks to recursion, I do not have to invoke the function `ASTGraphmmiter` at the constructor of *every* non-terminal element. One invocation at the constructor of `CTransUnit`, which is the relevant class of non-terminal *translation_unit* (where the recursion terminates) is enough.

Firstly we define a new `CTransUnit` variable `instance`, initialized in the body of `CASTSyntaxElement`.

```
CTransUnit* CTransUnit::instance = 0;
```

The `graphstring` component of the `CtransUnit` constructors is assigned to `instance`, as follows:

```
this->graphstring = nonterminals[this->m_NonTerminalCode];
CTransUnit::instance = this;
```

Bison must be informed for the above changes in `CtransUnit` by an assigmnent of `instance` to the production of *translation_unit* element in file MATFE.y.

```
translation_unit     :scriptMFile
                    {
                       $$ = CTransUnit::instance = new
                       CTransUnit((CScriptMFile*)$1,
                        (MATFE::location *)&yyloc,
                        TranslationUnit__ScriptMFile);
                    }
```

The same for the `functionMFile` and `parse_error` cases.

Finally, the incorporation of all the above functions to the parsing stage is achieved by passing the argument `CTransUnit::instance` to a simple invocation of public function `ASTGraphEmmiter` in the body of the object *`parse`* of `MATFE_driver`:

```
ASTGraphEmmiter(CTransUnit::instance)
```

Now the project is complete. The front_end compiler is ready. I must now of course test it.

The following fragment of MATLAB code is chosen for my first compilation.

```
File input.m
for j=2:3,
 for k=3:4,
  k=k+1;
```

```
  j=j+k;
 end
end
```

The compilation has been succesful. The next page presents the graphical representation of the produced abstract syntax tree.

6.3 Conclusions

Thanks goodness the project has been completed successfully. It has been tested in an Ubuntu Linux environment as well as in a Microsoft Windows XP environment (using VisualStudio ver.8 tools). Some of the MATLAB files tested with their relevant graphical ASTs are presented in Appendix 1.

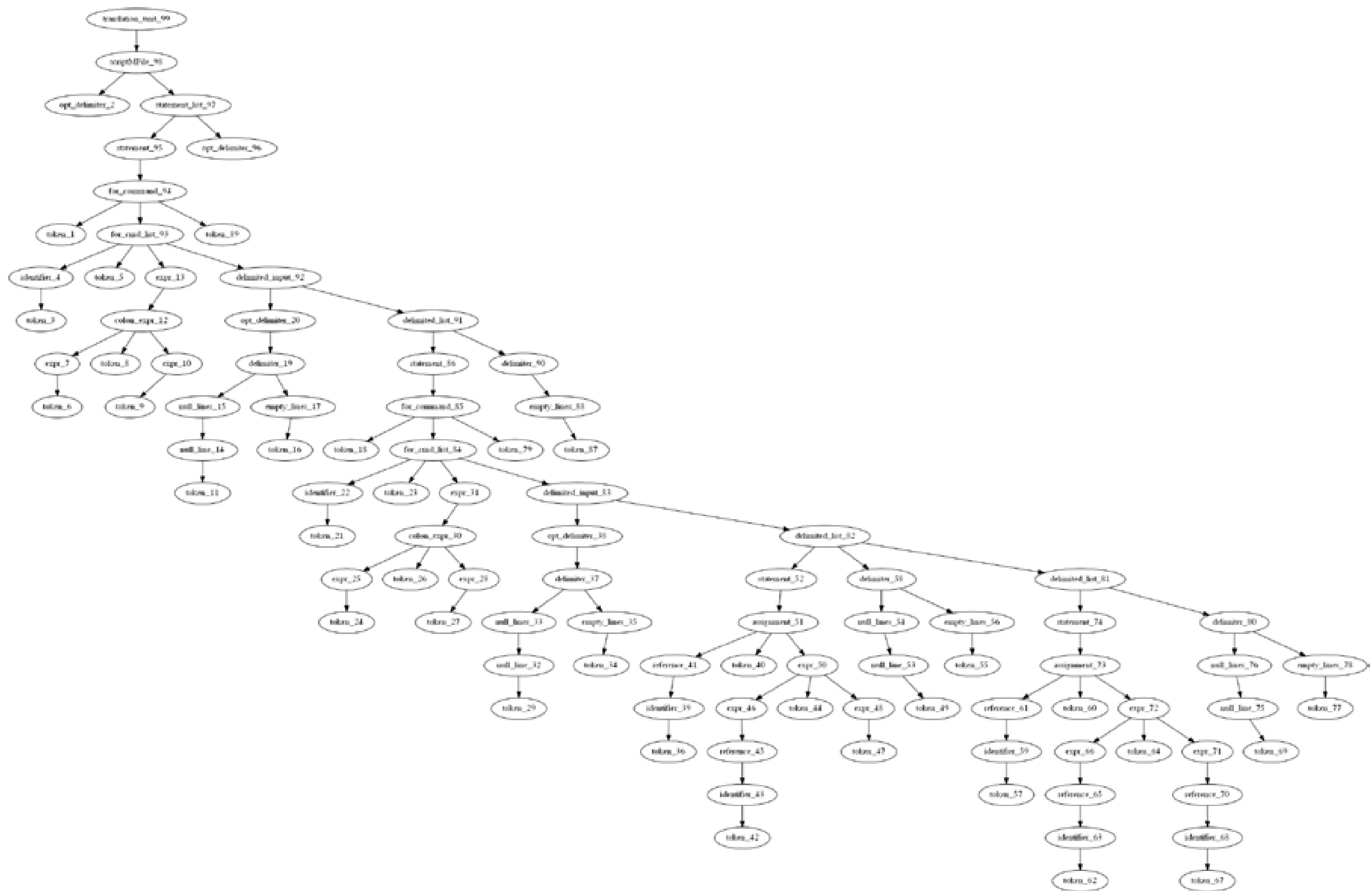

Figure 4 AST Graphical Representation

Epilogue

Recommendations for further work

Before finishing my report I would like to emphasize some important issues. Firstly, as far as concerns the optimization of the produced intermediate representation (IR), a careful observation of produced `DOT` outputs shows that, in some compilations, certain nodes are repeated. I tried to reduce these nodes by re-checking and altering my code without adequate success. Hence I conclude that the repeated nodes result due to the MATLAB input files and the grammar rules function rather than to construction bugs. I did not insist a lot, because a forced alteration at this intermediate stage of compilation in order to optimize, involves the risk to loose data.

Of course `DOT` files are not the only way to extract an IR. I selected this form because, in my view, it is the appropriate media to visualize an abstract syntax tree (AST). Nevertheless, if the target is serialization and processing rather than visualization, `XML` files are more convenient for describing the hierarchies of an AST.

Another alternative is **External Data Representation** (`XDR`) form. XDR description is a standard for the description and encoding of data, introduced by Sun Microsystems in the late 1980s. The `XDR` standard includes two key components: a language used for defining data structures and a binary format for representing those structures. Therefore an `XDR` implementation takes a definition of a data structure written as an `XDR` data description and translates it to an `XDR` binary format.

Other alternatives are high level programming languages formats (such as C or Java), which are the best selection for translation of IR to multiple final languages, at the back_end compilation stage.

Proceeding to the back_end, some interesting proposals for further work based on the front_end developed in the current project, are certain retargetable translations such as:

- An IR translation to system level C for the development of a kernel module or even a driver.
- A development of a hardware module which implements a part of the MATLAB logic, to incorporate it in an embedded system. An IR translation to VHDL or to Verilog is required in this case.

- An IR translation to Native Assembly if the development of a standalone application on a specific driver is desired.

It should be a great pleasure for me, if the front_end of the current work would be used in one of the above, or a totally different application.

Appendix 1

In this section I present the resulting graphical *IR* of certain input MATLAB files compiled by MATFE front_end compiler.

File `input.m`: `a=3+2;`

Resulting AST:

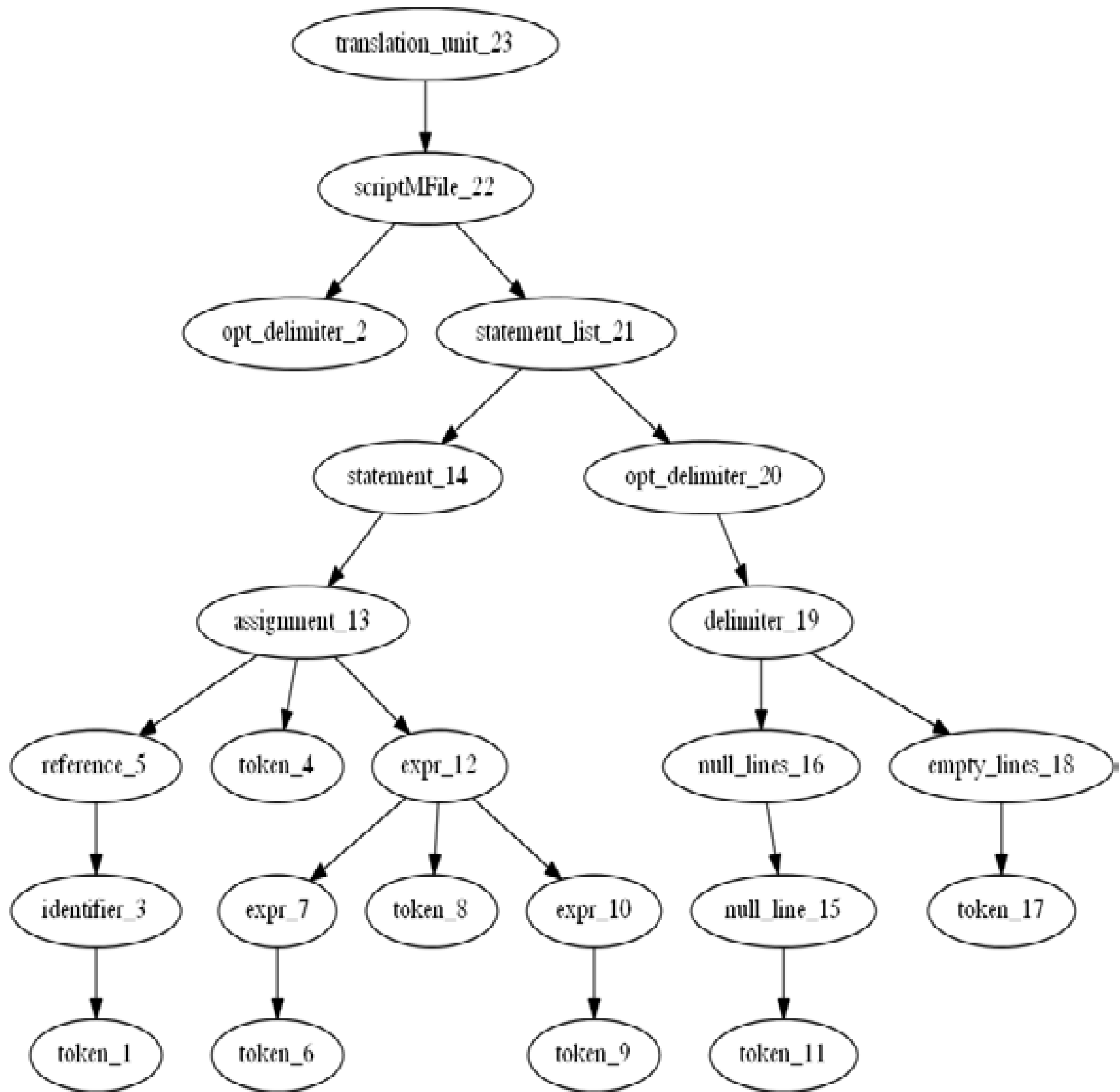

Figure 5: **input.m AST graphical representation**

File `input3.m`:

```
a(x,y)=40;
```

Resulting AST:

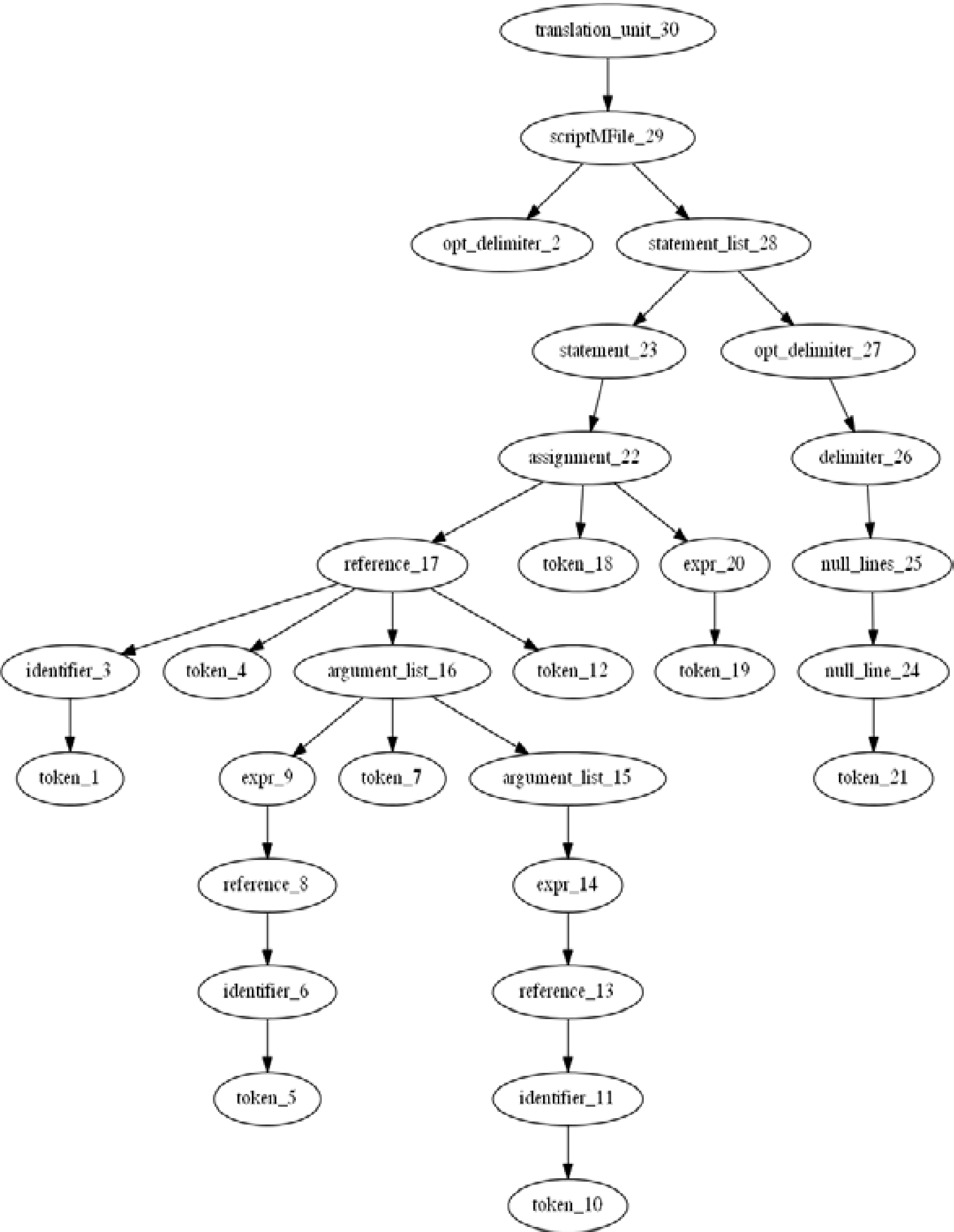

Figure 6: **input3.m AST graphical representation**

File `input8.m`:

```
if r == c
x=3;
else
x=5;
end
```

Resulting AST:

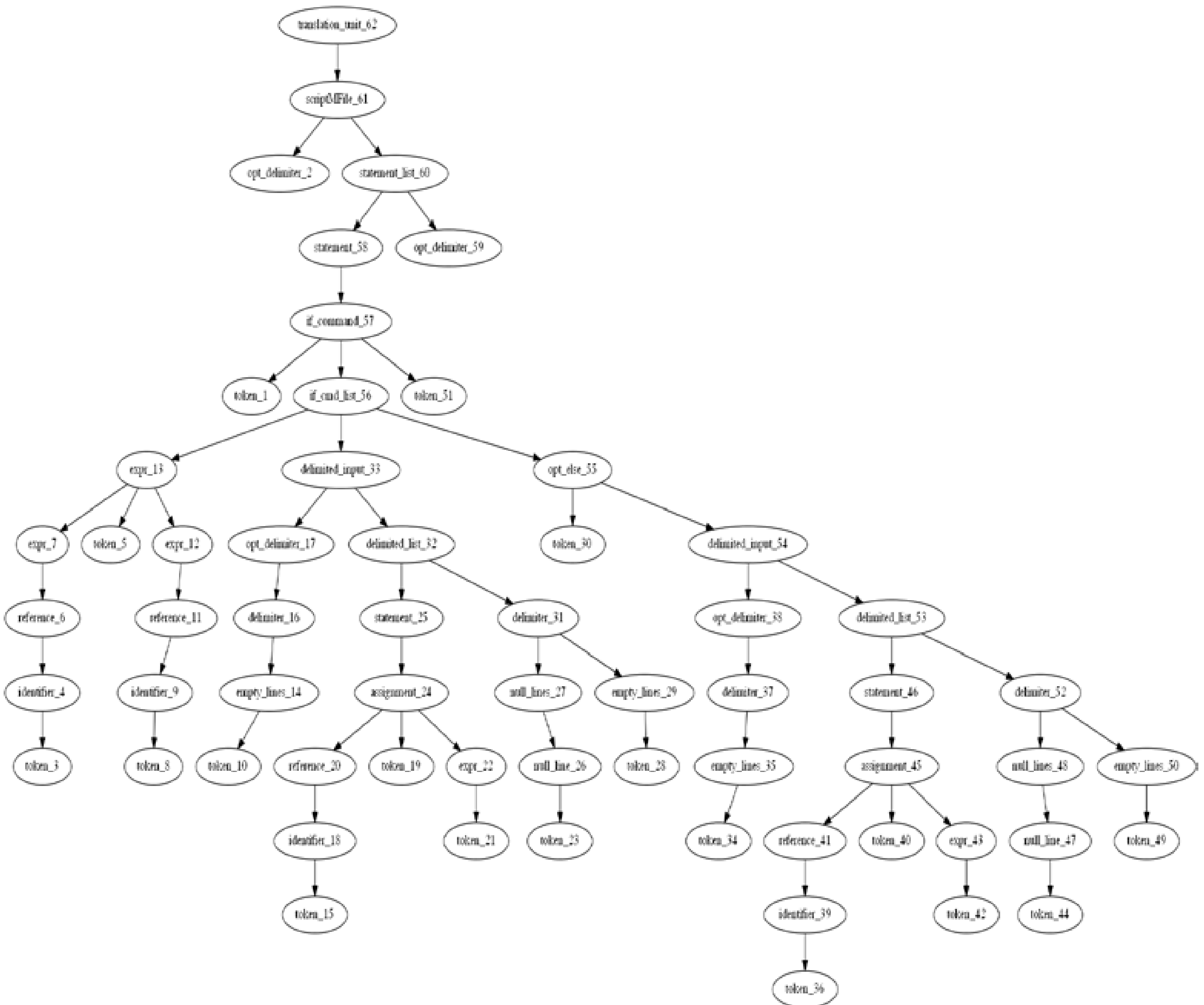

Figure 7: **input8.m AST graphical representation**

File `input9.m`:

```
for j=2:3,
     for i=j:3,
        B(i,:) = B(i,:) - B(j-1,:)*B(i,j-1)/B(j-1,j-1)
     end
   end
```

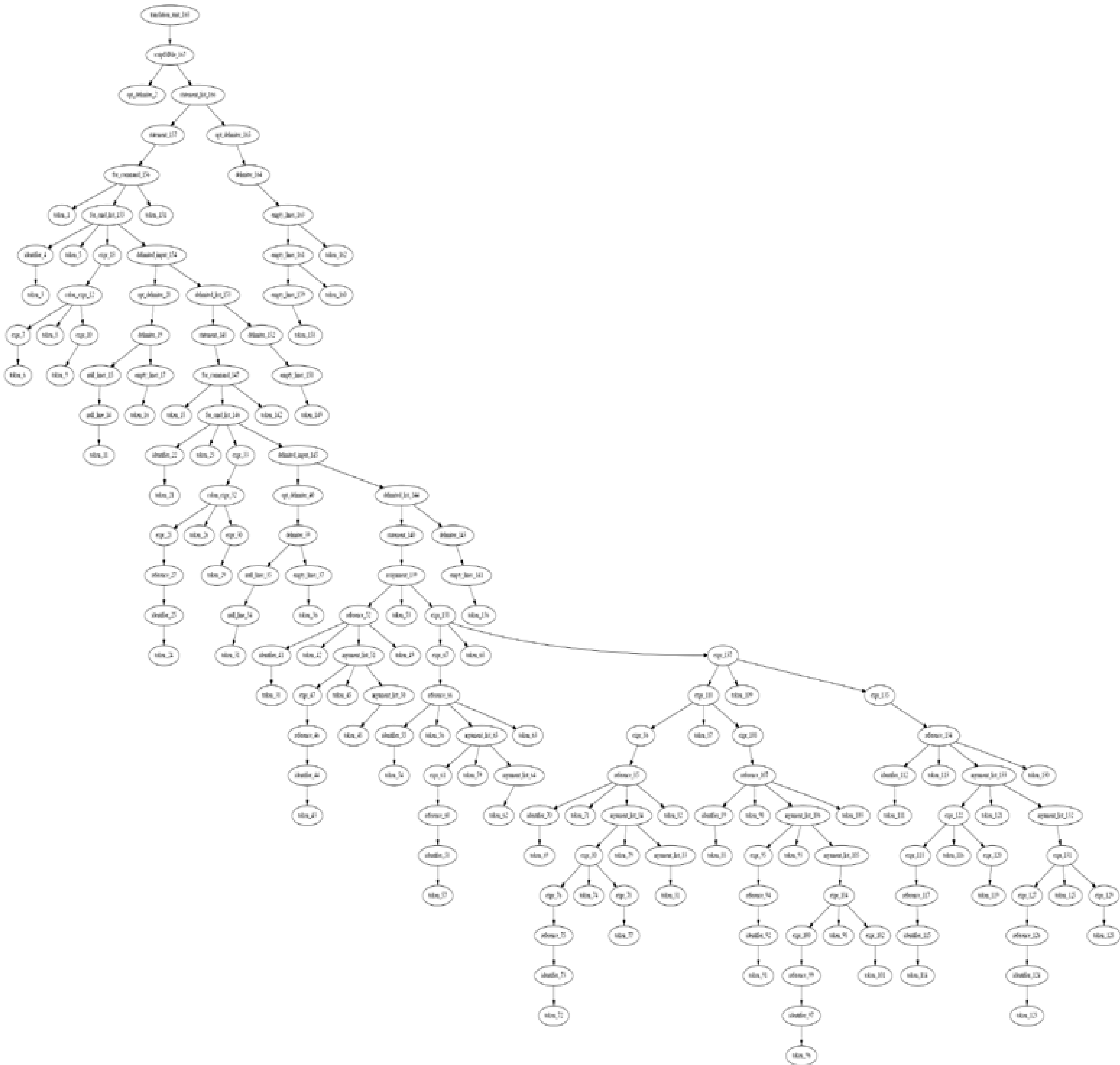

Figure 8: input9.m AST graphical representation

File input14.m:

```
n = 1;
nFactorial = 1;
while nFactorial < 1e100
    n = n + 1;
    nFactorial = nFactorial * n;
end
```

File input15.m:

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) | strncmp(line,'%',1) |~ischar(line)
     continue;
end;
    count = count + 1;
end;
fprintf('%d lines\n',count);
fclose(fid);
```

File input23.m:

```
function showImageOutsidePath(imageFile)
fig1 = figure;
% Define the cleanup routine.
cleanupObj = onCleanup(restore_env(fig1, imgpath));
% Modify the path to gain access to the image file,
% and display the image.
addpath(imgpath);
rgb = imread(imageFile);
fprintf('\n   Opening the figure %s\n', imageFile);
image(rgb);
pause(2);
   % This is the cleanup routine.
```

Figure 9:input14.m AST graphical representation

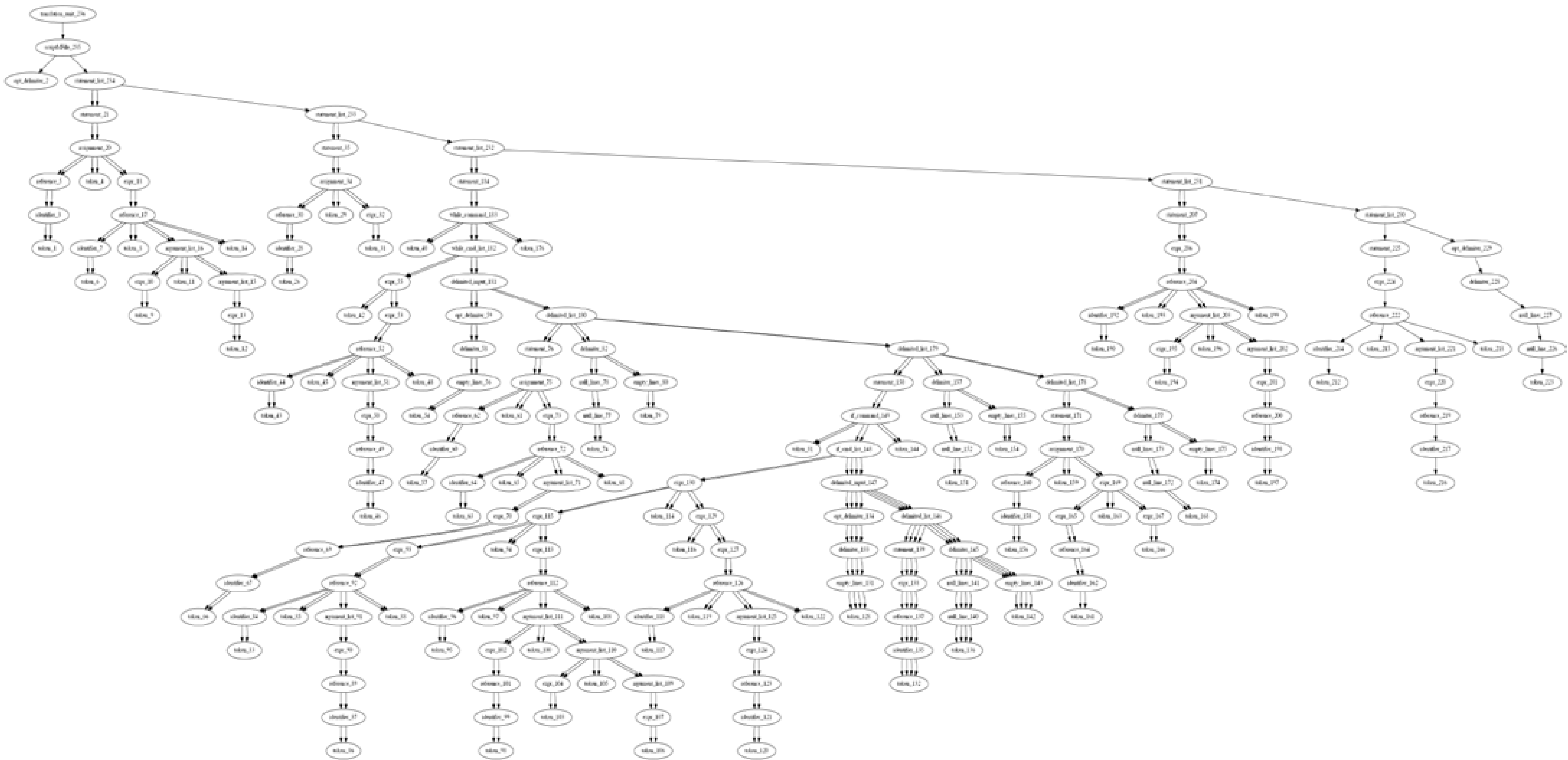

Figure 10 input15.m AST graphical representation

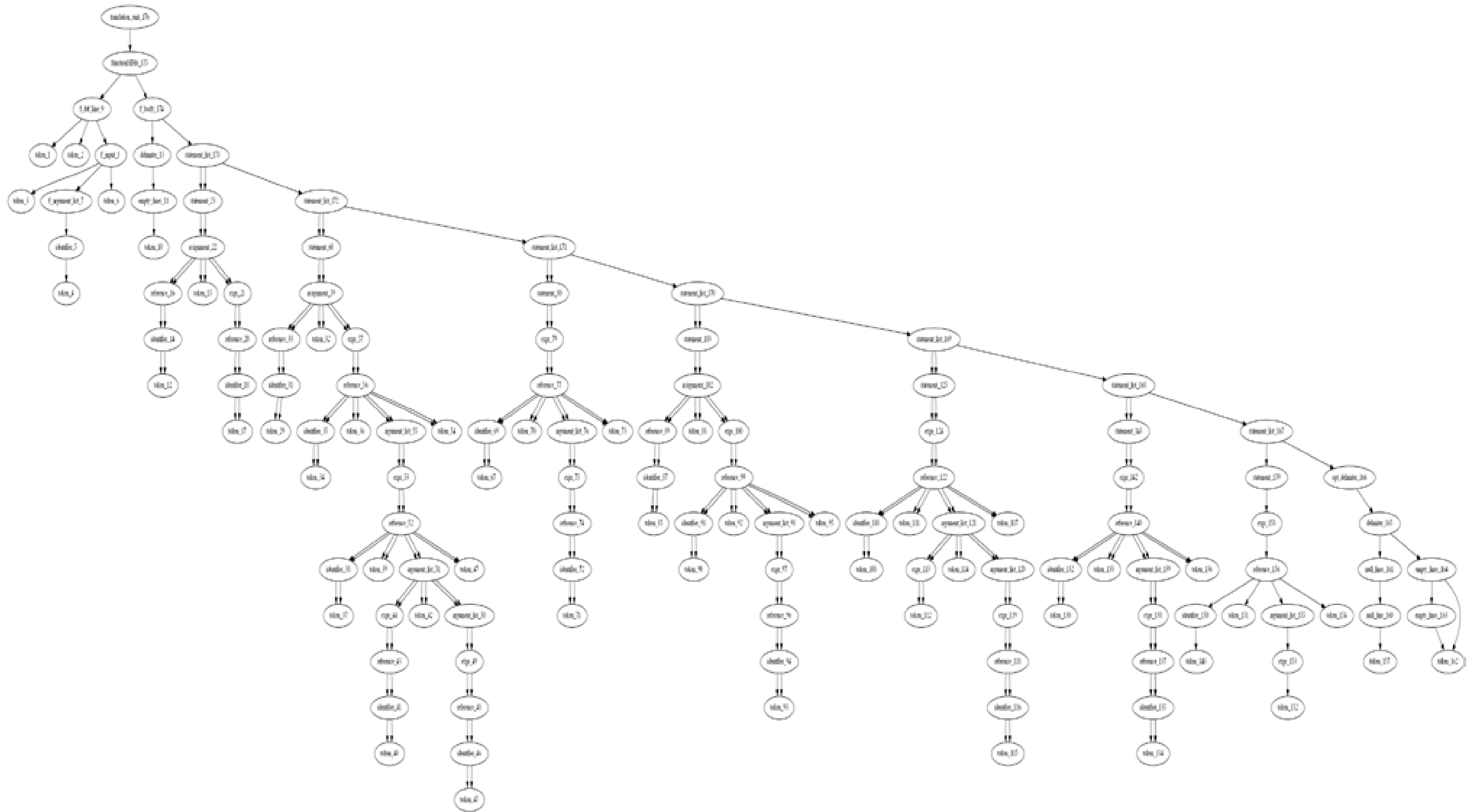

Figure 11:input23.m AST graphical representation

Appendix 2

Source code used in this project

File MATFE.l

```
%{
#include <stdlib.h>
#include <iostream>
#include <errno.h>
#include <limits.h>
#include <string>
#include "ASTDefines.h"
#include "ASTSyntaxElements.h"
#include "driver.h"
#include "MATFE.tab.h"

# undef yywrap
# define yywrap() 1

#define yyterminate() return token::END

      unsigned int SourceLine = 1;
    static int Depth;
    CTOKEN* newtoken;
%}

%option noyywrap batch debug
%s QuoteSC


%{
# define YY_USER_ACTION yylloc->columns (yyleng);
%}

HSPACE          [ \t]
HSPACES          {HSPACE}+
NEWLINE         \n|\r|\f
NEWLINES        {NEWLINE}+
ELLIPSIS        \.\.\.
CONTINUATION    {ELLIPSIS}[^\n\r\f]*{NEWLINE}?
COMMENT         \%[^\n\r\f]*{NEWLINE}?
IDENTIFIER      [a-zA-Z][_a-zA-Z0-9]*
DIGIT           [0-9]
INTEGER         {DIGIT}+
EXPONENT        [DdEe][+-]?{DIGIT}+
MANTISSA        ({DIGIT}+\.)|({DIGIT}*\.{DIGIT}+)
FLOATINGPOINT   {MANTISSA}{EXPONENT}?
DOUBLE          {INTEGER}{EXPONENT}|{FLOATINGPOINT}
NUMBER          {INTEGER}|{DOUBLE}
IMAGINARYUNIT   [IiJj]

%%
%{
yylloc->step ();
%}
%{
typedef MATFE::MATParserClass::token token;
%}
```

```
"for"                          {
                                CTOKEN* newtoken;
                                 newtoken = new CTOKEN(FOR_,
                                (MATFE::location *)&yylloc,
                                 TOKEN_FOR);
                                yylval->ast = newtoken;
                                return(token::FOR);
                               }

 "end"                         {
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(END_,
                                (MATFE::location *)&yylloc,
                                TOKEN_END);
                                yylval->ast = newtoken;
                                return(token::END);
                                }

"if"                            {
                                 CTOKEN* newtoken;
                                 newtoken = new CTOKEN(IF_,
                                 (MATFE::location *)&yylloc,
                                 TOKEN_IF);
                                 yylval->ast = newtoken;
                                 return(token::IF);
                                 }


"elseif"                         {
                                  CTOKEN* newtoken;
                                  newtoken = new CTOKEN(ELSEIF_,
                                  (MATFE::location *)&yylloc,
                                  TOKEN_ELSEIF);
                                  yylval->ast = newtoken;
                                  return(token::ELSEIF);
                                 }


"else"                               {
                                 CTOKEN* newtoken;
                                 newtoken = new CTOKEN(ELSE_,
                                 (MATFE::location *)&yylloc,
                                 TOKEN_ELSE);
                                 yylval->ast = newtoken;
                                 return(token::ELSE);
                                 }

"global"                               {
                                  CTOKEN* newtoken;
                                  newtoken = new CTOKEN(GLOBAL_,
                                  (MATFE::location *)&yylloc,
                                  TOKEN_GLOBAL);
                                  yylval->ast = newtoken;
                                  return(token::GLOBAL);
                                 }


"while"                          {
                                  CTOKEN* newtoken;
```

```
                              newtoken = new CTOKEN(WHILE_,
                              (MATFE::location *)&yylloc,
                              TOKEN_WHILE);
                              yylval->ast = newtoken;
                              return(token::WHILE);
                              }

"function"                    {
                               CTOKEN* newtoken;
                               newtoken = new CTOKEN(FUNCTION_,
                               (MATFE::location *)&yylloc,
                               TOKEN_FUNCTION);
                               yylval->ast = newtoken;
                               return(token::FUNCTION);
                               }


"return"                       {
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(RETURN_,
                                (MATFE::location *)&yylloc,
                                TOKEN_RETURN);
                                yylval->ast = newtoken;
                                return(token::RETURN);
                                }

<INITIAL>'[^'\n\r\f]*'/'        {
                                 yymore();
                                 }


,                               {
                                                 BEGIN(INITIAL);
                                  CTOKEN* newtoken;
                                  newtoken = new CTOKEN(COMMA_,
                                  (MATFE::location *)&yylloc,
                                  TOKEN_COMMA);
                                  yylval->ast = newtoken;
                                  return(token::COMMA);
                                  }

;                               {
                                                 BEGIN(INITIAL);
                                  CTOKEN* newtoken;
                                  newtoken = new CTOKEN(SEMICOLON_,
                                  (MATFE::location *)&yylloc,
                                  TOKEN_SEMICOLON);
                                  yylval->ast = newtoken;
                                  return(token::SEMICOLON);
                                  }


<INITIAL>'[^'\r\f\n]*'            {
                                   register int i, size;
                                   char* modified=0;

                                   const int length = yyleng-2;

                                   for (size = 0, i= 1; i <= length;
size++, i++)
                                   if (*(yytext+i) == '\'') i++;
                                   modified = (char*)calloc(size+1,1);
```

```
                                *(modified+size) = '\0';

                                for (size = 0, i = 1; i <= length;
size++, i++) {
                                *(modified+size) = *(yytext+i);
                                 if (*(yytext+ i) == '\'') i++;
                                 }

                                  yylval->text = modified;
                                  CTOKEN* newtoken;
                                  newtoken = new CTOKEN(TEXT_,
                                 (MATFE::location *)&yylloc,
                                  TOKEN_TEXT);
                                  yylval->ast = newtoken;
                                  return(token::TEXT);
                                 }


{NUMBER}{IMAGINARYUNIT}         {
                                 BEGIN(QuoteSC);
                                 *(yytext+yyleng-1) = '\0';
                                 yylval->imaginary = atof(yytext);
                                 CTOKEN* newtoken;
                                 newtoken = new CTOKEN(IMAGINARY_,
                                (MATFE::location *)&yylloc,
                                 TOKEN_IMAGINARY);
                                 yylval->ast = newtoken;
                                 return(token::IMAGINARY);
                                 }


{DIGIT}+/\.[*/\\^]               {
                                 CTOKEN* newtoken;
                                 newtoken = new CTOKEN(INTEGER_,
                                (MATFE::location *)&yylloc,
                                 TOKEN_INTEGER);
                                 yylval->ast = newtoken;
                                 return(token::INTEGER);
                                 }


{DIGIT}+/\.'                     {
                                 CTOKEN* newtoken;
                                 newtoken = new CTOKEN(INTEGER_,
                                (MATFE::location *)&yylloc,
                                 TOKEN_INTEGER);
                                 yylval->ast = newtoken;
                                 return(token::INTEGER);
                                 }


{INTEGER}                     {
                                 BEGIN(QuoteSC);
                                 CTOKEN* newtoken;
                                 newtoken = new CTOKEN(INTEGER_,
                                 (MATFE::location *)&yylloc,
                                 TOKEN_INTEGER);
                                 yylval->ast = newtoken;
                                 return(token::INTEGER);
                                 }

{DOUBLE}                        {
```

```
                                BEGIN(QuoteSC);
                               CTOKEN* newtoken;
                               newtoken = new CTOKEN(DOUBLE_,
                              (MATFE::location *)&yylloc,
                               TOKEN_DOUBLE);
                               yylval->ast = newtoken;
                               return(token::DOUBLE);
                              }


{HSPACES}                                    {
                              BEGIN(INITIAL);
                               }


{CONTINUATION}                                  {
                                SourceLine++;
                               }


{NEWLINES}                 {
                            BEGIN(INITIAL);
                            SourceLine += yyleng;
                            CTOKEN* newtoken;
                            newtoken = new CTOKEN(LINE_,
                            (MATFE::location *)&yylloc,
                            TOKEN_LINE);
                            yylval->ast = newtoken;
                            return(token::LINE);
                           }

{IDENTIFIER}               {

                                          BEGIN(QuoteSC);
                            CTOKEN* newtoken;
                            newtoken = new CTOKEN(IDENTIFIER_,
                           (MATFE::location *)&yylloc,
                            TOKEN_IDENTIFIER);
                            yylval->ast = newtoken;
                            return(token::IDENTIFIER);

                           }

{COMMENT}                  {
                                           BEGIN(INITIAL);
                                           SourceLine++;
                                           return(token::LINE);
                            }

  &                       {
                                         BEGIN(INITIAL);
                           CTOKEN* newtoken;
                           newtoken = new CTOKEN(AND_,
                           (MATFE::location *)&yylloc,
                           TOKEN_AND);
                           yylval->ast = newtoken;
                           return(token::AND);
                           }


  \|                     {
                                         BEGIN(INITIAL);
```

```
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(OR_,
                                (MATFE::location *)&yylloc,
                                TOKEN_OR);
                                yylval->ast = newtoken;
                                return(token::OR);
                                }


  >                           {
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(GTHAN_,
                                (MATFE::location *)&yylloc,
                                TOKEN_GTHAN);
                                yylval->ast = newtoken;
                                return(token::GTHAN);
                                }


 >=                           {
                                BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(GTANHE_,
                                (MATFE::location *)&yylloc,
                                TOKEN_GTHANE);
                                yylval->ast = newtoken;
                                return(token::GTHANE);
                              }

 \<                           {
                                BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                 newtoken = new CTOKEN(LTHAN_,
                                (MATFE::location *)&yylloc,
                                TOKEN_LTHAN);
                                yylval->ast = newtoken;
                                return(token::LTHAN);
                                }

\<=                           {
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(LTHANE_,
                                (MATFE::location *)&yylloc,
                                TOKEN_LTANHE);
                                yylval->ast = newtoken;
                                 return(token::LTHANE);
                              }

==                           {
                                BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                 newtoken = new CTOKEN(EQUAL_,
                                (MATFE::location *)&yylloc,
                                TOKEN_EQUAL);
                                yylval->ast = newtoken;
                                return(token::EQUAL);
                                }


~=                              {
```

```
                                BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                 newtoken = new CTOKEN(UNEQUAL_,
                                (MATFE::location *)&yylloc,
                                TOKEN_UNEQUAL);
                                yylval->ast = newtoken;
                                return(token::UNEQUAL);
                              }

:                              {
                                BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                 newtoken = new CTOKEN(COLONOPERATOR_,
                                (MATFE::location *)&yylloc,
                                TOKEN_COLONOPERATOR);
                                yylval->ast = newtoken;
                                return(token::COLONOPERATOR);
                               }



 =                             {
                                BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                 newtoken = new CTOKEN(ASSIGN_,
                                (MATFE::location *)&yylloc,
                                TOKEN_ASSIGN);
                                yylval->ast = newtoken;
                                return(token::ASSIGN);
                                }

 -                             {
                                BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                 newtoken = new CTOKEN(MINUS_,
                                (MATFE::location *)&yylloc,
                                TOKEN_MINUS);
                                yylval->ast = newtoken;
                                return(token::MINUS);
                               }


 \+                            {
                                BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                 newtoken = new CTOKEN(PLUS_,
                                (MATFE::location *)&yylloc,
                                TOKEN_PLUS);
                                yylval->ast = newtoken;
                                return(token::PLUS);
                                }

  \*                           {
                                              BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(BIMUL_,
                                (MATFE::location *)&yylloc,
                                TOKEN_BIMUL);
                                yylval->ast = newtoken;
                                return(token::BIMUL);
                                }
```

```
".*"                            {
                                            BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(EMUL_,
                                (MATFE::location *)&yylloc,
                                TOKEN_EMUL);
                                yylval->ast = newtoken;
                                return(token::EMUL);
                                }


\/                              {
                                            BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(BIDIV_,
                                (MATFE::location *)&yylloc,
                                TOKEN_BIDIV);
                                yylval->ast = newtoken;
                                return(token::BIDIV);
                                }

"./"                            {
                                            BEGIN(INITIAL);
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(EDIV_,
                                (MATFE::location *)&yylloc,
                                 TOKEN_EDIV);
                                 yylval->ast = newtoken;
                                 return(token::EDIV);
                                 }
\\                              {
                                             BEGIN(INITIAL);
                                 CTOKEN* newtoken;
                                 newtoken = new CTOKEN(BILEFTDIV_,
                                 (MATFE::location *)&yylloc,
                                 TOKEN_BILEFTDIV);
                                 yylval->ast = newtoken;
                                 return(token::BILEFTDIV);
                                 }
\.\\                             {
                                            BEGIN(INITIAL);
                                  CTOKEN* newtoken;
                                  newtoken = new CTOKEN(ELEFTDIV_,
                                  (MATFE::location *)&yylloc,
                                  TOKEN_ELEFTDIV);
                                  yylval->ast = newtoken;
                                  return(token::ELEFTDIV);
                                  }

{HSPACES}\.'                              {
                                                BEGIN(INITIAL);
                                    newtoken = new CTOKEN(LEXERROR_,
                                    (MATFE::location *)&yylloc,
                                    TOKEN_LEXERROR);
                                    yylval->ast = newtoken;
                                    return(token::LEXERROR);
                                  }
```

```
\.'                         {
                                         BEGIN(QuoteSC);
                              newtoken = new CTOKEN(TRANSPOSE_,
                              (MATFE::location *)&yylloc,
                              TOKEN_TRANSPOSE);
                              yylval->ast = newtoken;
                              return(token::TRANSPOSE);
                            }

\.^                         {
                             CTOKEN* newtoken;
                             newtoken = new CTOKEN(EPOWER_,
                             (MATFE::location *)&yylloc,
                             TOKEN_EPOWER);
                             yylval->ast = newtoken;
                             return(token::EPOWER);
                             }

\^                          {
                             CTOKEN* newtoken;
                             newtoken = new CTOKEN(POWER_,
                             (MATFE::location *)&yylloc,
                             TOKEN_POWER);
                             yylval->ast = newtoken;
                             return(token::POWER);
                             }

~                            {
                                          BEGIN(INITIAL);
                             CTOKEN* newtoken;
                             newtoken = new CTOKEN(NOT_,
                            (MATFE::location *)&yylloc,
                             TOKEN_NOT);
                             yylval->ast = newtoken;
                             return(token::NOT);
                            }

<QuoteSC>'                  {
                              newtoken = new CTOKEN(CTRANSPOSE_,
                              (MATFE::location *)&yylloc,
                              TOKEN_CTRANSPOSE);
                              yylval->ast = newtoken;
                              return(token::CTRANSPOSE);
                            }

<INITIAL>'                          {
                                           return(token::LEXERROR);
                            }


\[                          {
                             if(Depth)
                             {
                              Depth++;
                              CTOKEN* newtoken;
                              newtoken = new CTOKEN(LB_,
                              (MATFE::location *)&yylloc,
                               TOKEN_LB);
                               yylval->ast = newtoken;
```

```
                                return(token::LB);
                               }
                                int current=0, next=0;
                                char* buffer=0;
                                int level=1, length=0;
                                while(level &&
(current=yyinput())!=EOF) {

buffer=(char*)realloc(buffer,++length);
                                *(buffer+length-1)=current;
                                 if(current=='[') level++;
                                 if(current==']') level--;
                                }
                                 if(level) {
                                            CTOKEN* newtoken;
                                            newtoken = new
CTOKEN(LEXERROR_,
                                           (MATFE::location *)&yylloc,
                                            TOKEN_LEXERROR);
                                            yylval->ast = newtoken;
                                            return(token::LEXERROR);
                                            }
                                 while((current=yyinput())!=EOF) {

buffer=(char*)realloc(buffer,++length);
                                *(buffer+length-1)=current;
                                if (current!=' '&& current!='\t')
break;
                                   }
                                if((next=yyinput())!=EOF) {

buffer=(char*)realloc(buffer,++length);
                                 *(buffer+length-1)=next;
                                   }
                                 for( ;length > 0; length-- )
                                 unput(*(buffer+length-1));
                                 free(buffer);
                                 Depth=1;
                                 if(current=='=' && next!='=') {
                                 CTOKEN* newtoken;
                                 newtoken = new CTOKEN(LD_,
                                 (MATFE::location *)&yylloc,
                                 TOKEN_LD);
                                 yylval->ast = newtoken;
                                 return(token::LD);
                                }
                              else {
                             return(token::LB);
                                }
                            }


\]/{HSPACE}*=[^=]            {
                              BEGIN(INITIAL);
                              Depth--;
                              CTOKEN* newtoken;
                              newtoken = new CTOKEN(RD_,
                              (MATFE::location*)&yylloc,
                              TOKEN_RD);
                              yylval->ast = newtoken;
                              return(token::RD);
                              }
```

```
\]                          {
                             BEGIN(QuoteSC);
                             Depth--;
                             CTOKEN* newtoken;
                             newtoken = new CTOKEN(RB_,
                             (MATFE::location *)&yylloc,
                             TOKEN_RB);
                             yylval->ast = newtoken;
                             return(token::RB);
                             }

\(                           {
                               BEGIN(INITIAL);
                               CTOKEN* newtoken;
                               newtoken = new CTOKEN(LPAREN_,
                              (MATFE::location *)&yylloc,
                               TOKEN_LPAREN);
                               yylval->ast = newtoken;
                               return(token::LPAREN);
                              }


 \)                            {
                                BEGIN(QuoteSC);
                                CTOKEN* newtoken;
                                newtoken = new CTOKEN(RPAREN_,
                                (MATFE::location *)&yylloc,
                                TOKEN_RPAREN);
                                yylval->ast = newtoken;
                                return(token::RPAREN);
                                }


.                                          {
                                                return(token::LEXERROR);

                              }


%%
```

File MATFE.y

```
%skeleton "lalr1.cc"
%require "2.4.1"
%defines
%define namespace "MATFE"
%define parser_class_name "MATParserClass"



%code requires {
      #include <string.h>
      #include <stdlib.h>
      #include <fstream>
      #include "ASTDefines.h"
      #include "ASTSyntaxElements.h"
       class MATFE_driver;
       class CASTSyntaxElement;
```

```
       extern int yylex(void);


#define YY_DECL \
      MATFE::MATParserClass::token_type  \
       yylex(MATFE::MATParserClass::semantic_type* yylval, \
             MATFE::MATParserClass::location_type* yylloc, \
             MATFE_driver& driver)
}


%parse-param {MATFE_driver& driver}
%lex-param {MATFE_driver& driver}
%locations

%initial-action
{
@$.begin.filename = @$.end.filename = &driver.file;
};

%debug
%verbose



%union{
    char* text;
      CASTSyntaxElement *ast;
      double imaginary;
};

%code {
#include "driver.h"
      }

%token <ast> LEXERROR
%token <ast> LINE
%token <ast> LB
%token <ast> RB
%token <ast> LD
%token <ast> RD
%token <ast> FOR
%token <ast> END  0  "end of file"
%token <ast> IF
%token <ast> ELSEIF
%token <ast> ELSE
%token <ast> WHILE
%token <ast> RETURN
%token <ast> TEXT
%token <ast> IDENTIFIER
%token <ast> INTEGER
%token <ast> DOUBLE
%token <ast> IMAGINARY
%token <ast> GLOBAL
%token <ast> ASSIGN
%token <ast> LPAREN
%token <ast> RPAREN
%nonassoc MINOR_PREC
%left <ast> AND OR
%left <ast> LTHAN LTHANE GTHAN GTHANE EQUAL UNEQUAL
%left <ast> COLONOPERATOR
```

```
%left <ast> PLUS MINUS
%left <ast> BIMUL EMUL BIDIV EDIV ELEFTDIV BILEFTDIV
%nonassoc <ast> NOT
%nonassoc <ast> TRANSPOSE CTRANSPOSE
%left <ast> EPOWER POWER
%nonassoc UPLUS UMINUS
%nonassoc LPAREN



%token <ast> COMMA
%token <ast> SEMICOLON
%token <ast> FUNCTION


%type <ast> translation_unit
%type <ast> scriptMFile
%type <ast> opt_delimiter
%type <ast> delimiter
%type <ast> null_lines
%type <ast> null_line
%type <ast> empty_lines
%type <ast> statement_list
%type <ast> statement
%type <ast> command_form
%type <ast> text_list
%type <ast> expr
%type <ast> reference
%type <ast> identifier
%type <ast> argument_list
%type <ast> matrix
%type <ast> rows
%type <ast> row
%type <ast> row_with_commas
%type <ast> colon_expr
%type <ast> assignment
%type <ast> s_assignee_matrix
%type <ast> m_assignee_matrix
%type <ast> reference_list
%type <ast> for_command
%type <ast> for_cmd_list
%type <ast> if_command
%type <ast> if_cmd_list
%type <ast> opt_else
%type <ast> global_command
%type <ast> global_decl_list
%type <ast> while_command
%type <ast> while_cmd_list
%type <ast> return_command
%type <ast> delimited_input
%type <ast> delimited_list
%type <ast> functionMFile
%type <ast> f_def_line
%type <ast> f_output
%type <ast> f_input
%type <ast> f_argument_list
%type <ast> f_body
%type <ast> parse_error

%start translation_unit

%%
```

```
translation_unit   :scriptMFile
                             {
                              $$ = CTransUnit::instance = new
CTransUnit((CScriptMFile*)$1,
                      (MATFE::location *)&yyloc,
                      TranslationUnit__ScriptMFile);
                             }
                             | functionMFile
                              {
                              $$ = CTransUnit::instance = new
CTransUnit((CFunctionMFile*)$1,
                      (MATFE::location *)&yyloc,
                      TranslationUnit__FunctionMFile);

                             }
                             | parse_error
                             {
                              $$ = CTransUnit::instance = new
CTransUnit((CParseError*)$1,
                      (MATFE::location *)&yyloc,
                       TranslationUnit__ParseError);
                             }
                             ;

scriptMFile           : opt_delimiter
                     {
                      $$ = new CScriptMFile((COptDelimiter*)$1,
                      (MATFE::location *)&yyloc,
                       ScriptMFile__OptDelimiter);
                       cout << "ScriptMFile Bison(1st rule)"<<endl;
                       cout << "---------------------------" << endl;
                     }

                     | opt_delimiter statement_list
                     {
                      $$ = new CScriptMFile((COptDelimiter*)$1,
                      (CStatementList*)$2,
                      (MATFE::location *)&yyloc,
                      ScriptMFile__OptDelimiter_StatementList);
                      cout << "ScriptMFile Bison(2nd rule)"<<endl;
                      cout << "---------------------------" << endl;
                     }
                     ;


opt_delimiter          : %prec MINOR_PREC
                     {
                     $$ = new COptDelimiter(
                      (MATFE::location *)&yyloc,
                       OptDelimiter__Delimiter);
                       cout <<"Opt_delimiter stage(1st rule)" << endl;
                       cout << "-----------------------------" <<
endl;
                     }

                     |delimiter
                               {
                               $$ = new COptDelimiter((CDelimiter*)$1,
                      (MATFE::location *)&yyloc,
                       OptDelimiter__Delimiter);
```

```
                          cout <<"Opt_delimiter stage (2ndrule)" << endl;
                          cout << "----------------------------" << endl;
                        }
                                  ;


delimiter                  : null_lines
                                  {
                                   $$ = new CDelimiter((CNullLines*)$1,
                         (MATFE::location *)&yyloc,
                          Delimiter__NullLines);
                          cout << "Delimiter stage(null_lines)" << endl;
                          cout << "---------------------------" << endl;
                         }
                               | empty_lines
                                  {
                                   $$ = new CDelimiter((CEmptyLines*)$1,
                         (MATFE::location *)&yyloc,
                          Delimiter__EmptyLines);
                          cout << "Delimiter  stage(empty_lines)" <<
endl;
                          cout << "-----------------------------" <<
endl;
                         }

                               | null_lines empty_lines
                                  {
                         $$ = new CDelimiter((CNullLines*)$1,
                         (CEmptyLines*)$2,
                         (MATFE::location *)&yyloc,
                         Delimiter__NullLines_EmptyLines);
                         cout << "Delimiter stage(null_lines
empty_lines)" << endl;
                          cout << "-------------------------------------
" << endl;
                        }
                              ;


null_lines                 : null_line
                               {
                                   $$ = new CNullLines((CNullLine*)$1,
                         (MATFE::location *)&yyloc,
                          NullLines__NullLine);
                         cout <<"Null lines stage(1st rule)" << endl;
                         cout << "------------------------" << endl;
                         }
                                  | null_lines null_line
                                  {
                                   $$ = new CNullLines((CNullLines*)$1,
                                   (CNullLine*)$2,
                         (MATFE::location *)&yyloc,
                          NullLines__NullLines_NullLine);
                          cout <<"Null lines stage(2nd rule)" << endl;
                          cout << "------------------------" << endl;
                         }
                                  ;

 null_line               : COMMA
                                  {
                                   $$ = new CNullLine((CTOKEN*)$1,
                         (MATFE::location *)&yyloc,
```

```
                         NullLine__COMMA);
                         cout <<"Null line stage(1nd rule)" << endl;
                         cout << "------------------------" << endl;
                        }
                                  | SEMICOLON
                                  {
                                   $$ = new CNullLine((CTOKEN*)$1,
                        (MATFE::location *)&yyloc,
                         NullLine__SEMICOLON);
                        cout <<"Null lines stage(2nd rule)" << endl;
                        cout << "------------------------" << endl;
                                  }
                                  | empty_lines COMMA
                              {
                                   $$ = new CNullLine((CEmptyLines*)$1,
                                   (CTOKEN*)$2,
                        (MATFE::location *)&yyloc,
                         NullLine__EmptyLines_COMMA);
                         cout <<"Null lines stage(3th rule)" << endl;
                         cout << "------------------------" << endl;
                                  }
                                  | empty_lines SEMICOLON
                                  {
                                   $$ = new CNullLine((CEmptyLines*)$1,
                                   (CTOKEN*)$2,
                        (MATFE::location *)&yyloc,
                         NullLine__EmptyLines_SEMICOLON);
                         cout <<"Null lines stage(4th rule)" << endl;
                         cout << "------------------------" << endl;
                                  }
                                  ;

empty_lines                 : LINE
                        {
                         $$ = new CEmptyLines((CTOKEN*)$1,
                        (MATFE::location *)&yyloc,
                         EmptyLines__LINE);
                         cout <<"Empty lines stage(1st rule)" << endl;
                         cout << "------------------------" << endl;
                        }
                            | empty_lines LINE
                                  {
                                   $$ = new CEmptyLines((CEmptyLines*)$1,
                                   (CTOKEN*)$2,
                        (MATFE::location *)&yyloc,
                         EmptyLines__EmptyLines_LINE);
                          cout <<"Empty lines stage(2nd rule)" << endl;
                         cout << "------------------------" << endl;
                                  }
                                ;

statement_list              : statement opt_delimiter
                                  {

                                  $$ = new
CStatementList((CStatement*)$1,
                                   (COptDelimiter*)$2,
                        (MATFE::location *)&yyloc,
                        StatementList__Statement_OptDelimiter);
                        cout <<"Statement_list stage (1st rule)" <<
endl;
```

```
                        cout << "------------------------------" <<
endl;

                                 }
                                 | statement delimiter statement_list
                                 {
                                  $$ = new
CStatementList((CStatement*)$1,
                                  (CDelimiter*)$2, (CStatementList*)$3,
                        (MATFE::location *)&yyloc,

StatementList__Statement_Delimiter_StatementList);
                        cout <<"Statement_list stage (2nd rule)" <<
endl;
                        cout << "------------------------------" <<
endl;
                                 }
                                 ;

statement                  : command_form
                                 {
                                  $$ = new CStatement((CCommandForm*)$1,
                        (MATFE::location *)&yyloc,
                         Statement__CommandForm);
                         cout << "Statement stage(1st rule)"<<endl;
                         cout << "------------------" << endl;
                                 }
                                 | expr
                                  {

                                   $$ = new CStatement((CExpression*)$1,
                        (MATFE::location *)&yyloc,
                         Statement__Expr);
                         cout << "Statement stage(2st rule)"<<endl;
                         cout << "------------------" << endl;

                                  }
                                 | assignment
                        {
                                   $$ = new CStatement((CAssignment*)$1,
                        (MATFE::location *)&yyloc,
                         Statement__Assignment);
                         cout << "Statement stage(3rd rule)"<<endl;
                         cout << "------------------" << endl;
                                  }
                                 | for_command
                                  {
                        $$ = new CStatement((CForCommand*)$1,
                        (MATFE::location *)&yyloc,
                         Statement__ForCommand);
                         cout << "Statement stage(4th rule)"<<endl;
                         cout << "------------------" << endl;
                                   }
                                 | if_command
                                    {
                        $$ = new CStatement((CIfCommand*)$1,
                        (MATFE::location *)&yyloc,
                         Statement__IfCommand);
                         cout << "Statement stage(5th rule)"<<endl;
                         cout << "------------------" << endl;
                                   }
                                 | global_command
```

```
                                   {
                      $$ = new CStatement((CIfCommand*)$1,
                      (MATFE::location *)&yyloc,
                       Statement__GlobalCommand);
                       cout << "Statement stage(6th rule)"<<endl;
                       cout << "------------------" << endl;
                                  }
                                | while_command
                                 {
                                  $$ = new
CStatement((CWhileCommand*)$1,
                      (MATFE::location *)&yyloc,
                       Statement__WhileCommand);
                       cout << "Statement stage(7th rule)"<<endl;
                       cout << "------------------" << endl;
                                     }
                                | return_command
                                 {
                                  $$ = new
CStatement((CReturnCommand*)$1,
                      (MATFE::location *)&yyloc,
                       Statement__ReturnCommand);
                       cout << "Statement stage(8th rule)"<<endl;
                       cout << "------------------" << endl;
                                     }
                               ;

command_form              : identifier text_list
                               {
                                $$ = new
CCommandForm((CIdentifier*)$1,
                                 (CTextList*)$2,
                                 (MATFE::location *)&yyloc,
                       CommandForm__Identifier_TextList);
                       cout << "Command_form stage"<<endl;
                       cout << "------------------" << endl;
                               }
                               ;

text_list                 : TEXT
                               {
                                $$ = new CTextList((CTOKEN*)$1,
                      (MATFE::location *)&yyloc,
                       TextList__TEXT);
                       cout << "Text_list stage(1st rule)"<<endl;
                       cout << "--------------------" << endl;

                               }
                               | text_list TEXT
                               {
                               $$ = new CTextList((CTextList*)$1,
                                (CTOKEN*)$2,
                      (MATFE::location *)&yyloc,
                       TextList__TEXT);
                       cout << "Text_list stage(2nd rule)"<<endl;
                       cout << "--------------------" << endl;
                               }
                               ;

   expr                      : INTEGER
                              {
```

```
           $$ = new CExpression((CTOKEN*)$1,
 (MATFE::location *)&yyloc,
  Expr__INTEGER);
 cout << "Expr stage(1st rule)"<<endl;
 cout << "---------------------" << endl;

         }
         | DOUBLE
         {

           $$ = new CExpression((CTOKEN*)$1,
 (MATFE::location *)&yyloc,
  Expr__DOUBLE);
  cout << "Expr stage(2nd rule)"<<endl;
 cout << "----------------------" << endl;
         }
         | IMAGINARY
         {

           $$ = new CExpression((CTOKEN*)$1,
 (MATFE::location *)&yyloc,
  Expr__IMAGINARY);
  cout << "Expr stage(3rd rule)"<<endl;
 cout << "----------------------" << endl;
         }
         | TEXT
         {

           $$ = new CExpression((CTOKEN*)$1,
 (MATFE::location *)&yyloc,
  Expr__TEXT);
  cout << "Expr stage(4th rule)"<<endl;
 cout << "----------------------" << endl;
         }
         | LPAREN expr RPAREN
         {

          $$ = new CExpression((CTOKEN*)$1,
           (CExpression*)$2,(CTOKEN*)$3,
(MATFE::location *)&yyloc,
Expr__LPAREN_Expr_RPAREN);
cout << "Expr stage(5th rule)"<<endl;
cout << "----------" << endl;
         }
         | reference
         {
          $$ = new CExpression((CReference*)$1,
(MATFE::location *)&yyloc,
Expr__Reference);
cout << "Expr stage(6th rule)"<<endl;
cout << "---------------------" << endl;
         }
         | matrix
         {
          $$ = new CExpression((CMatrix*)$1,
(MATFE::location *)&yyloc,
Expr__Matrix);
cout << "Expr stage(7th rule)"<<endl;
cout << "----------" << endl;
         }
         | expr EPOWER expr
         {
```

```
          $$ = new CExpression((CExpression*)$1,
          (CTOKEN*)$2, (CExpression*)$3,
(MATFE::location *)&yyloc,
Expr__Expr_EPOWER_Expr);
cout << "Expr stage(8th rule)"<<endl;
cout << "--------------------" << endl;
         }
         | expr POWER expr
         {
          $$ = new CExpression((CExpression*)$1,
          (CTOKEN*)$2, (CExpression*)$3,
(MATFE::location *)&yyloc,
Expr__Expr_POWER_Expr);
cout << "Expr stage(9th rule)"<<endl;
cout << "--------------------" << endl;
         }
         | expr TRANSPOSE
         {

           $$ = new CExpression((CExpression*)$1,
          (CTOKEN*)$2,
(MATFE::location *)&yyloc,
Expr__Expr_TRANSPOSE);
cout << "Expr stage(10th rule)"<<endl;
cout << "---------------------" << endl;
         }
         | expr CTRANSPOSE
         {

          $$ = new CExpression((CExpression*)$1,
          (CTOKEN*)$2,
(MATFE::location *)&yyloc,
Expr__Expr_CTRANSPOSE);
cout << "Expr stage(11th rule)"<<endl;
cout << "---------------------" << endl;
         }
         | NOT expr
         {
          $$ = new CExpression((CTOKEN*)$1,
          (CExpression*)$2,
(MATFE::location *)&yyloc,
Expr__NOT_Expr);
cout << "Expr stage(12th rule)"<<endl;
cout << "---------------------" << endl;
         }
         | PLUS expr %prec UPLUS
         {
          $$ = new CExpression((CTOKEN*)$1,
          (CExpression*)$2,
(MATFE::location *)&yyloc,
Expr__PLUS_Expr);
cout << "Expr stage(13th rule)"<<endl;
cout << "---------------------" << endl;
         }
         | MINUS expr %prec UMINUS
         {
          $$ = new CExpression((CTOKEN*)$1,
          (CExpression*)$2,
(MATFE::location *)&yyloc,
Expr__MINUS_Expr);
cout << "Expr stage(14th rule)"<<endl;
cout << "---------------------" << endl;
```

```
          }
          |  expr BIMUL expr
          {
           $$ = new CExpression((CExpression*)$1,
           (CTOKEN*)$2,(CExpression*)$3,
(MATFE::location *)&yyloc,
Expr__Expr_BIMUL_Expr);
cout << "Expr stage(15th rule)"<<endl;
cout << "---------------------" << endl;
          }
          |  expr BIDIV expr
          {
           $$ = new CExpression((CExpression*)$1,
           (CTOKEN*)$2,(CExpression*)$3,
(MATFE::location *)&yyloc,
Expr__Expr_BIDIV_Expr);
cout << "Expr stage(16th rule)"<<endl;
cout << "---------------------" << endl;
          }
          |  expr BILEFTDIV expr
          {
           $$ = new CExpression((CExpression*)$1,
           (CTOKEN*)$2,(CExpression*)$3,
(MATFE::location *)&yyloc,
Expr__Expr_BILEFTDIV_Expr);
cout << "Expr stage(17th rule)"<<endl;
cout << "---------------------" << endl;
          }
          |  expr EMUL expr
          {
           $$ = new CExpression((CExpression*)$1,
           (CTOKEN*)$2,(CExpression*)$3,
(MATFE::location *)&yyloc,
Expr__Expr_EMUL_Expr);
cout << "Expr stage(18th rule)"<<endl;
cout << "----------------------" << endl;
          }
          |  expr EDIV expr
          {
           $$ = new CExpression((CExpression*)$1,
           (CTOKEN*)$2,(CExpression*)$3,
(MATFE::location *)&yyloc,
Expr__Expr_EDIV_Expr);
cout << "Expr stage(19th rule)"<<endl;
cout << "---------------------" << endl;
          }
          |  expr ELEFTDIV expr
          {
           $$ = new CExpression((CExpression*)$1,
           (CTOKEN*)$2,(CExpression*)$3,
(MATFE::location *)&yyloc,
Expr__Expr_ELEFTDIV_Expr);
cout << "Expr stage(20th rule)"<<endl;
cout << "---------------------" << endl;
          }
          |  expr PLUS expr
          {
           $$ = new CExpression((CExpression*)$1,
           (CTOKEN*)$2,(CExpression*)$3,
(MATFE::location *)&yyloc,
Expr__Expr_PLUS_Expr);
cout << "Expr stage(21th rule)"<<endl;
```

```
                    cout << "---------------------" << endl;
                             }
                             |  expr MINUS expr
                             {
                              $$ = new CExpression((CExpression*)$1,
                              (CTOKEN*)$2,(CExpression*)$3,
                    (MATFE::location *)&yyloc,
                    Expr__Expr_MINUS_Expr);
                    cout << "Expr stage(22th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                             |  colon_expr  %prec MINOR_PREC
                             {
                              $$ = new
CExpression((CColonExpression*)$1,
                    (MATFE::location *)&yyloc,
                    Expr__ColonExpr);
                    cout << "Expr stage(23th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                             |  expr LTHAN expr
                             {
                              $$ = new CExpression((CExpression*)$1,
                              (CTOKEN*)$2,(CExpression*)$3,
                    (MATFE::location *)&yyloc,
                    Expr__Expr_LTHAN_Expr);
                    cout << "Expr stage(24th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                             |  expr LTHANE expr
                             {
                             $$ = new CExpression((CExpression*)$1,
                              (CTOKEN*)$2,(CExpression*)$3,
                    (MATFE::location *)&yyloc,
                    Expr__Expr_LTHANE_Expr);
                    cout << "Expr stage(25th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                             |  expr GTHAN expr
                             {
                             $$ = new CExpression((CExpression*)$1,
                              (CTOKEN*)$2,(CExpression*)$3,
                    (MATFE::location *)&yyloc,
                    Expr__Expr_GTHAN_Expr);
                    cout << "Expr stage(26th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                           |  expr GTHANE expr
                             {
                             $$ = new CExpression((CExpression*)$1,
                              (CTOKEN*)$2,(CExpression*)$3,
                    (MATFE::location *)&yyloc,
                    Expr__Expr_GTHANE_Expr);
                    cout << "Expr stage(27th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                             |  expr EQUAL expr
                             {
                             $$ = new CExpression((CExpression*)$1,
                              (CTOKEN*)$2,(CExpression*)$3,
                    (MATFE::location *)&yyloc,
                    Expr__Expr_EQUAL_Expr);
```

```
                    cout << "Expr stage(28th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                             |  expr UNEQUAL expr
                             {
                             $$ = new CExpression((CExpression*)$1,
                              (CTOKEN*)$2,(CExpression*)$3,
                    (MATFE::location *)&yyloc,
                    Expr__Expr_UNEQUAL_Expr);
                    cout << "Expr stage(29th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                             |  expr AND expr
                             {
                             $$ = new CExpression((CExpression*)$1,
                              (CTOKEN*)$2,(CExpression*)$3,
                    (MATFE::location *)&yyloc,
                    Expr__Expr_AND_Expr);
                    cout << "Expr stage(30th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                             |  expr OR expr
                             {
                             $$ = new CExpression((CExpression*)$1,
                              (CTOKEN*)$2,(CExpression*)$3,
                    (MATFE::location *)&yyloc,
                    Expr__Expr_OR_Expr);
                    cout << "Expr stage(31th rule)"<<endl;
                    cout << "---------------------" << endl;
                             }
                             ;


reference                  : identifier %prec MINOR_PREC
                             {
                             $$ = new CReference((CIdentifier *)$1,
                    (MATFE::location *)&yyloc,
                    Reference__Identifier);
                    cout << "Reference stage(1st rule)"<<endl;
                    cout << "-------------------------" << endl;
                             }
                             | identifier LPAREN  argument_list
RPAREN
                             {

                              $$ = new CReference((CIdentifier *)$1,

(CTOKEN*)$2,(CArgumentList*)$3,(CTOKEN*)$4,
                    (MATFE::location *)&yyloc,

Reference__Identifier_LPAREN_ArgumentList_RPAREN);
                    cout << "Reference stage(2nd rule)"<<endl;
                    cout << "------------------------" << endl;
                             }
                             ;

 identifier                : IDENTIFIER
                             {
                             $$ = new CIdentifier((CTOKEN*)$1,
                    (MATFE::location *)&yyloc,
                    Identifier__IDENTIFIER);
                    cout << "Identifier stage"<<endl;
```

```
                        cout << "---------------" << endl;
                                  }
                                  ;


argument_list               : COLONOPERATOR
                                  {
                                   $$ = new CArgumentList((CTOKEN*)$1,
                        (MATFE::location *)&yyloc,
                        ArgumentList__COLONOPERATOR);
                        cout << "argument_list stage(1st rule)"<<endl;
                        cout << "-----------------------------" << endl;
                                  }
                                  | expr
                                  {
                                   $$ = new
CArgumentList((CExpression*)$1,
                        (MATFE::location *)&yyloc,
                        ArgumentList__Expr);
                        cout << "argument_list stage(2nd rule)"<<endl;
                        cout << "-----------------------------" << endl;
                                  }
                                  | COLONOPERATOR COMMA argument_list
                                  {
                                   $$ = new CArgumentList((CTOKEN*)$1,
                                   (CTOKEN*)$2, (CArgumentList*)$3,
                        (MATFE::location *)&yyloc,
                        ArgumentList__COLONOPERATOR_COMMA_ArgumentList);
                        cout << "argument_list stage(3rd rule)"<<endl;
                        cout << "-----------------------------" << endl;
                                  }
                                  | expr COMMA argument_list
                                  {
                                   $$ = new
CArgumentList((CExpression*)$1,
                                   (CTOKEN*)$2,(CArgumentList*)$3,
                        (MATFE::location *)&yyloc,
                        ArgumentList__Expr_COMMA_ArgumentList);
                        cout << "argument_list stage(4rth rule)"<<endl;
                        cout << "-----------------------------" << endl;
                                  }
                                ;

matrix                            : LB rows RB
                                  {
                                   $$ = new CMatrix((CTOKEN*)$1,
                                   (CRows*)$2,(CTOKEN*)$3,
                        (MATFE::location *)&yyloc,
                        Matrix__LB_Rows_RB);
                        cout << "Matrix stage"<<endl;
                        cout << "------------" << endl;
                                  }
                                  ;

rows                          : %prec MINOR_PREC
                         {
                          $$ = new CRows(
                         (MATFE::location *)&yyloc,
                          Rows__);
                          cout <<"Rows stage(1st rule)" << endl;
                          cout << "-------------------" << endl;
                          }
```

```
| row
           {
          $$ = new CRows((CRow*)$1,
(MATFE::location *)&yyloc,
Rows__Row);
cout << "Rows stage(2nd rule)"<<endl;
cout << "- ------------------" << endl;
         }
          | rows SEMICOLON
         {
          $$ = new CRows((CRows*)$1,
          (CTOKEN*)$2,
(MATFE::location *)&yyloc,
Rows__Rows_SEMICOLON);
cout << "Rows stage(3rd rule)"<<endl;
cout << "- ------------------" << endl;
         }
         | rows SEMICOLON row
         {
          $$ = new CRows((CRows*)$1,
          (CTOKEN*)$2, (CRow*)$3,
(MATFE::location *)&yyloc,
Rows__Rows_SEMICOLON_Row);
cout << "Rows stage(4th rule)"<<endl;
cout << "- ------------------" << endl;
         }
         | rows LINE
         {
         $$ = new CRows((CRows*)$1,
          (CTOKEN*)$2,
(MATFE::location *)&yyloc,
Rows__Rows_LINE);
cout << "Rows stage(5th rule)"<<endl;
cout << "- ------------------" << endl;
         }
         | rows LINE row
         {
         $$ = new CRows((CRows*)$1,
          (CTOKEN*)$2,(CRow*)$3,
(MATFE::location *)&yyloc,
Rows__Rows_LINE_Row);
cout << "Rows stage(6th rule)"<<endl;
cout << "- ------------------" << endl;
         }
         ;

row                     : expr
         {
          $$ = new CRow((CExpression*)$1,
(MATFE::location *)&yyloc,
Row__Expr);
cout << "Row stage(1st rule)"<<endl;
cout << "- ------------------" << endl;
         }
         | row_with_commas
         {
          $$ = new CRow((CRowWithCommas*)$1,
(MATFE::location *)&yyloc,
Row__RowWithCommas);
cout << "Row stage(2nd rule)"<<endl;
cout << "- ------------------" << endl;
         }
```

```
                         | row_with_commas expr
                         {
                          $$ = new CRow((CRowWithCommas*)$1,
                          (CExpression*)$2,
                 (MATFE::location *)&yyloc,
                 Row__RowWithCommas_Expr);
                 cout << "Row stage(3rd rule)"<<endl;
                 cout << "- ------------------" << endl;
                         }
                         ;

row_with_commas      : expr COMMA
                         {
                          $$ = new
CRowWithCommas((CExpression*)$1,
                          (CTOKEN*)$2,
                 (MATFE::location *)&yyloc,
                 RowWithCommas__Expr_COMMA);
                 cout << "Row_with_commas(1st rule)"<<endl;
                 cout << "- ------------------" << endl;
                         }
                         | row_with_commas expr COMMA
                {
                 $$ = new CRowWithCommas((CRowWithCommas*)$1,
                          (CExpression*)$2,(CTOKEN*)$3,
                 (MATFE::location *)&yyloc,
                 RowWithCommas__RowWithCommas_Expr_COMMA);
                  cout << "Row_with_commas(2nd rule)"<<endl;
                 cout << "- ------------------" << endl;
                }
                ;



colon_expr          : expr COLONOPERATOR expr
                         {
                          $$ = new
CColonExpression((CExpression*)$1,
                          (CTOKEN*)$2,(CExpression*)$3,
                 (MATFE::location *)&yyloc,
                 ColonExpr__Expr_COLONOPERATOR_Expr);
                 cout <<" Colon expr stage(1st rule)" << endl;
                  cout << "- -----------------------" << endl;
                         }

                         | colon_expr COLONOPERATOR expr
                {
                $$ = new CColonExpression((CColonExpression*)$1,
                          (CTOKEN*)$2,(CExpression*)$3,
                 (MATFE::location *)&yyloc,
                 ColonExpr__ColonExpr_Expr_COLONOPERATOR_Expr);
                 cout <<" Colon expr stage(2nd rule)" << endl;
                  cout << "- -----------------------" << endl;
                  }
                ;

assignment               : reference ASSIGN expr
                         {
                         $$ = new CAssignment((CReference*)$1,
                          (CTOKEN*)$2,(CExpression*)$3,
                 (MATFE::location *)&yyloc,
                 Assignment__Reference_ASSIGN_Expr);
```

```
                     cout <<"Assignment stage(1st rule)" << endl;
                              }
                              | s_assignee_matrix ASSIGN expr
                              {
                               $$ = new
CAssignment((CSassigneeMatrix*)$1,
                               (CTOKEN*)$2,(CExpression*)$3,
                     (MATFE::location *)&yyloc,
                     Assignment__SAssigneeMatrix_ASSIGN_Expr);
                     cout <<"Assignment stage(2nd rule)" << endl;
                              }
                              | m_assignee_matrix ASSIGN reference
                              {
                              $$ = new
CAssignment((CMassigneeMatrix*)$1,
                               (CTOKEN*)$2,(CReference*)$3,
                     (MATFE::location *)&yyloc,
                     Assignment__MAssigneeMatrix_ASSIGN_Reference);
                     cout <<"Assignment stage(3rd rule)" << endl;
                              }
                              ;

s_assignee_matrix    : LD reference RD
                    {
                     $$ = new CSassigneeMatrix((CTOKEN*)$1,
                               (CReference*)$2, (CTOKEN*)$3,
                     (MATFE::location *)&yyloc,
                     SAssigneeMatrix__LD_Reference_RD);
                     cout <<"s_assignee_matrix stage" << endl;
                      cout << "- ------------------" << endl;
                    }

                              ;

m_assignee_matrix    : LD reference COMMA reference_list RD
                    {
                     $$ = new CMassigneeMatrix((CTOKEN*)$1,

(CReference*)$2,(CTOKEN*)$3,(CReferenceList*)$4,(CTOKEN*)$5,
                     (MATFE::location *)&yyloc,

MAssigneeMatrix__LD_Reference_COMMA_ReferenceList_RD);
                     cout <<"m_assignee_matrix stage" << endl;
                      cout << "- ------------------" << endl;
                    }

                              ;


reference_list           : reference
                    {
                     $$ = new CReferenceList((CReference*)$1,
                     (MATFE::location *)&yyloc,
                     ReferenceList__Reference);
                     cout << "reference_list stage(1st rule)"<< endl;
                     cout << "------------------------------" << endl;
                    }
                    | reference COMMA reference_list
                    {
                     $$ = new CReferenceList((CReference*)$1,
                     (CTOKEN*)$2, (CReferenceList*)$3,
                     (MATFE::location *)&yyloc,
```

```
                     ReferenceList__Reference_COMMA_ReferenceList);
                     cout << "reference_list stage(2nd rule)"<< endl;
                     cout << "------------------------------" << endl;
                    }
                    ;

for_command         : FOR for_cmd_list END
                              {
                               $$ = new CForCommand((CTOKEN*)$1,
                               (CForCmdList*)$2,(CTOKEN*)$3,
                     (MATFE::location *)&yyloc,
                     ForCommand__FOR_ForCmdList_END);
                      cout << "for_command stage"<< endl;
                     cout << "------------------" << endl;
                              }
                              ;

for_cmd_list        : identifier ASSIGN expr delimited_input
                    {
                     $$ = new CForCmdList((CIdentifier*)$1,

(CTOKEN*)$2,(CExpression*)$3,(CDelimitedInput*)$4,
                     (MATFE::location*)&yyloc,

ForCmdList__Identifier_ASSIGN_Expr_DelimitedInput);
                     cout << "for_cmd_list stage"<< endl;
                     cout << "------------------" << endl;
                    }
                    ;


if_command          : IF if_cmd_list END
                    {
                     $$ = new CIfCommand((CTOKEN*)$1,
                               (CIfCmdList*)$2, (CTOKEN*)$3,
                     (MATFE::location *)&yyloc,
                     IfCommand__IF_IfCmdList_END);
                     cout << "if_command stage"<< endl;
                     cout << "---------------" << endl;
                    }
                    ;

if_cmd_list         : expr delimited_input opt_else
                    {
                     $$ = new CIfCmdList((CExpression*)$1,
                               (CDelimitedInput*)$2, (COptElse*)$3,
                     (MATFE::location *)&yyloc,
                     IfCmdList__Expr_DelimitedInput_OptElse);
                     cout << "if_cmd_list stage"<< endl;
                     cout << "-----------------" << endl;
                    }
                    ;

opt_else            :
                    {
                     cout << "opt_else  stage(1st rule)"<< endl;
                     cout << "-------------------------" << endl;
                    }
                    | ELSE delimited_input
                    {
                     $$ = new COptElse((CTOKEN*)$1,
                               (CDelimitedInput*)$2,
```

```
                    (MATFE::location *)&yyloc,
                    OptElse__ELSE_DelimitedInput);
                    cout << "opt_else  stage(2nd rule)"<< endl;
                    cout << "-------------------------" << endl;
                   }
                   | ELSEIF expr delimited_input opt_else
                   {
                    $$ = new COptElse((CTOKEN*)$1,
(CExpression*)$2,(CDelimitedInput*)$3,(COptElse*)$4,
                    (MATFE::location *)&yyloc,
                    OptElse__ELSEIF_DelimitedInput_OptElse);
                    cout << "opt_else  stage(3rd rule)"<< endl;
                    cout << "-------------------------" << endl;
                   }
                   ;

global_command     : GLOBAL global_decl_list
                   {
                    $$ = new CGlobalCommand((CTOKEN*)$1,
                              (CGlobalDeclList*)$2,
                    (MATFE::location *)&yyloc,
                    GlobalCommand__GLOBAL_GlobalDeclList);
                    cout << "global_command stage"<< endl;
                    cout << "--------------------" << endl;
                   }
                   ;


global_decl_list   : IDENTIFIER
                   {
                    $$ = new CGlobalDeclList((CTOKEN*)$1,
                    (MATFE::location *)&yyloc,
                    GlobalDeclList__IDENTIFIER);
                    cout << "global_decl_list stage(1st rule)"<<
endl;
                    cout << "--------------------------------" <<
endl;
                   }
                   | global_decl_list IDENTIFIER
                   {
                    $$ = new CGlobalDeclList((CGlobalDeclList*)$1,
                    (CTOKEN*)$2,
                    (MATFE::location *)&yyloc,
                    GlobalDeclList__GlobalDeclList_IDENTIFIER);
                    cout << "global_decl_list stage(2nd rule)"<<
endl;
                    cout << "--------------------------------" <<
endl;
                   }
                   ;

while_command      : WHILE while_cmd_list END
                   {
                    $$ = new CWhileCommand((CTOKEN*)$1,
                    (CWhileCmdList*)$2, (CTOKEN*)$3,
                    (MATFE::location *)&yyloc,
                    WhileCommand__WHILE_WhileCmdList_END);
                    cout << "while_command stage"<< endl;
                    cout << "-------------------" << endl;
                   }
                   ;
```

```
while_cmd_list       : expr delimited_input
                     {
                      $$ = new CWhileCmdList((CExpression*)$1,
                      (CDelimitedInput*)$2,
                      (MATFE::location *)&yyloc,
                      WhileCmdList__Expr_DelimitedInput);
                      cout << "while_cmd_list  stage"<< endl;
                      cout << "-------------------" << endl;
                     }
                     ;

return_command           : RETURN
                              {
                              $$ = new CReturnCommand((CTOKEN*)$1,
                      (MATFE::location*)&yyloc,
                      ReturnCommand__RETURN);
                      cout << "return_command  stage"<< endl;
                      cout << "---------------------" << endl;
                              }
                              ;

delimited_input      : opt_delimiter
                     {
                      $$ = new CDelimitedInput((COptDelimiter*)$1,
                      (MATFE::location *)&yyloc,
                      DelimitedInput__OptDelimiter);
                      cout << "Delimited input stage(1st rule)"<< endl;
                      cout << "----------------------------" << endl;
                     }
                     | opt_delimiter delimited_list
                     {
                      $$ = new CDelimitedInput((COptDelimiter*)$1,
                      (CDelimitedList*)$2,
                      (MATFE::location *)&yyloc,
                      DelimitedInput__OptDelimiter_DelimitedList);
                      cout << "Delimited input stage(2nd rule)"<< endl;
                      cout << "----------------------------" << endl;
                     }
                     ;

delimited_list       : statement delimiter
                     {
                      $$ = new CDelimitedList((CStatement*)$1,
                      (CDelimiter*)$2,
                      (MATFE::location *)&yyloc,
                      DelimitedList__Statement_Delimiter);
                      cout << "Delimited list stage(1st rule)"<< endl;
                      cout << "----------------------------" << endl;
                     }
                     | statement delimiter delimited_list
                     {
                      $$ = new CDelimitedList((CStatement*)$1,
                      (CDelimiter*)$2, (CDelimitedList*)$3,
                      (MATFE::location *)&yyloc,

DelimitedList__Statement_Delimiter_DelimitedList);
                      cout << "Delimited list stage(2nd rule)"<< endl;
                      cout << "----------------------------" << endl;
                     }
                     ;
```

```
functionMFile        : empty_lines f_def_line f_body
                     {
                      $$ = new CFunctionMFile((CEmptyLines*)$1,
                      (CFDefLine*)$2, (CFBody*)$3,
                      (MATFE::location *)&yyloc,
                      FunctionMFile__EmptyLines_FDefLine_FBody);
                      cout << "FunctionMFile stage(1st rule)"<< endl;
                      cout << "-----------------------------" << endl;
                     }
                     | f_def_line f_body
                     {
                      $$ = new CFunctionMFile((CFDefLine*)$1,
                      (CFBody*)$2,
                      (MATFE::location *)&yyloc,
                      FunctionMFile__FDefLine_FBody);
                      cout << "FunctionMFile stage(2nd rule)"<< endl;
                      cout << "-----------------------------" << endl;
                     }
                     ;

f_def_line           : FUNCTION f_output ASSIGN IDENTIFIER f_input
                     {
                     $$ = new CFDefLine((CTOKEN*)$1,
                      (CFOutput*)$2, (CTOKEN*)$3,
                      (CTOKEN*)$4, (CFInput*)$5,
                      (MATFE::location *)&yyloc,

FDefLine__FUNCTION_FOutput_ASSIGN_IDENTIFIER_FInput);
                      cout << "f_def_line stage(1st rule)"<< endl;
                      cout << "--------------------------" << endl;
                     }
                     | FUNCTION IDENTIFIER f_input
                     {
                      $$ = new CFDefLine((CTOKEN*)$1,
                      (CTOKEN*)$2, (CFInput*)$3,
                      (MATFE::location *)&yyloc,
                      FDefLine__FUNCTION_IDENTIFIER_FInput);
                      cout << "f_def_line stage(2nd rule)"<< endl;
                      cout << "--------------------------" << endl;
                     }
                     ;

f_output             : identifier
                     {
                      $$ = new CFOutput((CIdentifier*)$1,
                      (MATFE::location *)&yyloc,
                      FOutput__Identifier);
                      cout << "f_output stage(1st rule)"<< endl;
                      cout << "------------------------" << endl;
                     }
                     | LD f_argument_list RD
                     {
                      $$ = new CFOutput((CTOKEN*)$1,
                      (CFArgumentList*)$2, (CTOKEN*)$3,
                      (MATFE::location *)&yyloc,
                        FOutput__LD_FArgumentList_RD);
                        cout << "f_output stage(2nd rule)"<< endl;
                      cout << "------------------------" << endl;
                     }
                     ;

f_input              :
```

```
                     {

                     }
                    | LPAREN RPAREN
                    {
                     $$ = new CFInput((CTOKEN*)$1,
                     (CTOKEN*)$2,
                     (MATFE::location *)&yyloc,
                       FInput__LPAREN_RPAREN);
                       cout << "f_input stage(1st rule)"<< endl;
                     cout << "-----------------------" << endl;
                    }
                    | LPAREN f_argument_list RPAREN
                    {
                     $$ = new CFInput((CTOKEN*)$1,
                     (CFArgumentList*)$2,(CTOKEN*)$3,
                     (MATFE::location *)&yyloc,
                       FInput__LPAREN_FArgumentList_RPAREN);
                       cout << "f_input stage(2nd rule)"<< endl;
                     cout << "-----------------------" << endl;
                    }
                    ;


f_argument_list     : identifier COMMA f_argument_list
                              {
                               $$ = new
CFArgumentList((CIdentifier*)$1,
                     (CTOKEN*)$2, (CFArgumentList*)$3,
                     (MATFE::location*)&yyloc,
                       FArgumentList__Identifier_COMMA_FArgumentList);
                       cout << "f_argument_list stage(1st rule)"<<
endl;
                     cout << "--------------------------------" <<
endl;
                              }
                              | identifier
                              {
                               $$ = new
CFArgumentList((CIdentifier*)$1,
                     (MATFE::location *)&yyloc,
                       FArgumentList__Identifier);
                       cout << "f_argument_list stage(2nd rule)"<<
endl;
                     cout << "--------------------------------" <<
endl;
                              }
                              ;

f_body              : delimiter statement_list
                    {
                     $$ = new CFBody((CDelimiter*)$1,
                     (CStatementList*)$2,
                     (MATFE::location *)&yyloc,
                       FBody__Delimiter_StatementList);
                       cout << "f_body stage(1st rule)"<< endl;
                     cout << "-----------------------" << endl;
                    }
                    | opt_delimiter
                    {
                     $$ = new CFBody((COptDelimiter*)$1,
                     (MATFE::location *)&yyloc,
```

```
                      FBody__OptDelimiter);
                      cout << "f_body stage(2nd rule)"<< endl;
                    cout << "-----------------------" << endl;
                   }
                   ;



parse_error          : LEXERROR
                      {
                      $$ = new CParseError((CTOKEN*)$1,
                      (MATFE::location *)&yyloc,
                        ParseError__LEXERROR);
                        cout << "parse_error"<< endl;
                      cout << "------------" << endl;
                      }
                    | error
                    {
                      $$ = new CParseError((MATFE::location *)&yyloc,
                        ParseError__LEXERROR);
                        cout << "parse_error"<< endl;
                      cout << "------------" << endl;
                    }
                    ;

%%

void MATFE::MATParserClass::error(const
MATFE::MATParserClass::location_type& l,
                                          const std::string& m)
{
      driver.error (l, m);
}
```

File ASTSyntaxElements.h

```
#ifndef ASTELEMENTS_
#define ASTELEMENTS_

#include <cstdlib>
#include <string>
#include <vector>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <errno.h>
#include <limits.h>
#include "ASTDefines.h"


using namespace std;


namespace MATFE {
    class location;
}

class CASTSyntaxElement;

class CTransUnit;
class CScriptMFile;
class CFunctionMFile;
```

```
class CParseError;
class COptDelimiter;
class CDelimiter;
class CStatementList;
class CNullLines;
class CNullLine;
class CEmptyLines;
class CStatementList;
class CStatement;
class CCommandForm;
class CTextList;
class CExpression;
class CParenthesis;
class CReference;
class CIdentifier;
class CArgumentList;
class CMatrix;
class CBoxes1;
class CRows;
class CRow;
class CRowWithCommas;
class CColonExpression;
class CAssignment;
class CSassigneeMatrix;
class CMassigneeMatrix;
class CBoxes2;
class CReferenceList;
class CForCommand;
class CForCmdList;
class CIfCommand;
class CIfCmdList;
class COptElse;
class CGlobalCommand;
class CGlobalDeclList;
class CWhileCommand;
class CWhileCmdList;
class CReturnCommand;
class CDelimitedInput;
class CDelimitedList;
class CFDefLine;
class CFOutput;
class CFInput;
class CFArgumentList;
class CFBody;
class CParseError;


class CASTSyntaxElement {
public:

	CASTSyntaxElement(NONTERMINAL_PRODUCTION_RULE pr,
MATFE::location * loc);
	~CASTSyntaxElement();
	virtual int map_syntax_element(CASTSyntaxElement *, int m=-
1){return -1;}


	TERMINAL_CODE m_TerminalCode;
	NONTERMINAL_CODE m_NonTerminalCode;
	NONTERMINAL_PRODUCTION_RULE m_NonTerminal_ProductionCode;
	static unsigned int m_ObjectSerialNumberCounter;
	unsigned int m_NumberOfDescendants;
```

```
      unsigned int m_ObjectSerialNumber;
      vector <CASTSyntaxElement *> m_Descendants;
    char* graphstring;


protected: //Information for later stages
      MATFE::location *m_location;
};


class CTOKEN : public  CASTSyntaxElement{
public:
      CTOKEN(TERMINAL_CODE token, MATFE::location *loc,
       NONTERMINAL_PRODUCTION_RULE pr);
      ~CTOKEN();


protected:
      int map_syntax_element(CASTSyntaxElement *, int c=-1);

};


class CTransUnit:public CASTSyntaxElement{
public:

    static CTransUnit* instance;

      CTransUnit(CScriptMFile *, MATFE::location *,
            NONTERMINAL_PRODUCTION_RULE);

      CTransUnit(CFunctionMFile*, MATFE::location *,
            NONTERMINAL_PRODUCTION_RULE);

      CTransUnit(CParseError*, MATFE::location *,
            NONTERMINAL_PRODUCTION_RULE);



protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

class CScriptMFile : public CASTSyntaxElement{
public:
      CScriptMFile(COptDelimiter *, MATFE::location * loc,
            NONTERMINAL_PRODUCTION_RULE);

      CScriptMFile(COptDelimiter *, CStatementList *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


};

class COptDelimiter : public CASTSyntaxElement{
public:
        COptDelimiter(MATFE::location *,
```

```
            NONTERMINAL_PRODUCTION_RULE);
        COptDelimiter(CDelimiter *, MATFE::location *,
            NONTERMINAL_PRODUCTION_RULE);



protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};


class CDelimiter : public CASTSyntaxElement{
public:
      CDelimiter(CNullLines *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CDelimiter(CEmptyLines *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CDelimiter(CNullLines *, CEmptyLines *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

 protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

class CNullLines :public CASTSyntaxElement {
public:
      CNullLines(CNullLine *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

          CNullLines(CNullLines *, CNullLine *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);


 protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


};

class CNullLine : public CASTSyntaxElement{
public:
      CNullLine(CTOKEN *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);
      CNullLine(CEmptyLines *,CTOKEN *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);



protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


};

class CEmptyLines : public CASTSyntaxElement{
public:
      CEmptyLines(CTOKEN *LINE, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);
```

```
      CEmptyLines(CEmptyLines *, CTOKEN *LINE, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

      ;
};

class CStatementList : public CASTSyntaxElement{
public:
      CStatementList(CStatement *, COptDelimiter*,
            MATFE::location *, NONTERMINAL_PRODUCTION_RULE);
      CStatementList(CStatement *, CDelimiter*, CStatementList *,
            MATFE::location *, NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

class CStatement : public CASTSyntaxElement{
public:
      CStatement(CCommandForm *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CStatement(CExpression *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CStatement(CAssignment *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CStatement(CForCommand *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CStatement(CIfCommand *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CStatement(CGlobalCommand *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CStatement(CWhileCommand *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CStatement(CReturnCommand *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


};


class CCommandForm :public CASTSyntaxElement{
public:
      CCommandForm(CIdentifier *, CTextList *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

protected:
```

```
	virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

class CTextList : public CASTSyntaxElement{
public:
	CTextList(CTOKEN *TEXT, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);
	CTextList(CTextList *, CTOKEN *TEXT, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);


protected:
	virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

class CExpression : public CASTSyntaxElement{
public:
	CExpression(CTOKEN *,MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

	CExpression(CTOKEN *, CExpression*, CTOKEN *, MATFE::location *,
NONTERMINAL_PRODUCTION_RULE);

	CExpression(CReference *,MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

	CExpression(CMatrix *,MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

	CExpression(CExpression *,CTOKEN *, CExpression
*,MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

	CExpression(CExpression *,CTOKEN *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

	CExpression(CTOKEN *, CExpression *,MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

	CExpression(CColonExpression *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

 protected:
	virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);



};


   class CReference: public CASTSyntaxElement{
   public:
		CReference(CIdentifier *, MATFE::location *,
NONTERMINAL_PRODUCTION_RULE);
		CReference(CIdentifier *, CTOKEN *,  CArgumentList *,
CTOKEN *, MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

 protected:
	virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);
```

```
};
   class CIdentifier: public CASTSyntaxElement{
   public:
          CIdentifier(CTOKEN *IDENTIFIER, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

      class CArgumentList: public CASTSyntaxElement {
      public:
            CArgumentList(CTOKEN *COLONOPERATOR, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

            CArgumentList(CExpression *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

            CArgumentList(CTOKEN *COLONOPERATOR , CTOKEN *COMMA ,
CArgumentList *, MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

            CArgumentList(CExpression *, CTOKEN* COMMA, CArgumentList
*, MATFE::location *,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


};

      class CMatrix:public CASTSyntaxElement {
      public:
            CMatrix(CTOKEN* , CRows *, CTOKEN* ,
MATFE::location*,NONTERMINAL_PRODUCTION_RULE);

protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

      class CRows:public CASTSyntaxElement {
      public:
            CRows(MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

            CRows(CRow *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

            CRows(CRows *, CTOKEN *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

            CRows(CRows *, CTOKEN *, CRow *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);
```

```
};

      class CRow:public CASTSyntaxElement {
      public:
             CRow(CExpression *,MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

             CRow(CRowWithCommas *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

             CRow(CRowWithCommas *, CExpression *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

    class CRowWithCommas:public CASTSyntaxElement {
      public:
             CRowWithCommas(CExpression *, CTOKEN* COMMA,
MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

            CRowWithCommas(CRowWithCommas *,CExpression *, CTOKEN*
COMMA, MATFE::location *,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

    class CColonExpression: public CASTSyntaxElement {
    public:
         CColonExpression(CExpression *, CTOKEN* COLONOPERATOR,
CExpression *,MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

             CColonExpression(CColonExpression *, CTOKEN*
COLONOPERATOR, CExpression *,MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};


    class CAssignment: public CASTSyntaxElement {
    public:
         CAssignment(CReference *, CTOKEN * , CExpression
*,MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

          CAssignment(CSassigneeMatrix *,CTOKEN *, CExpression
*,MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

            CAssignment(CMassigneeMatrix *,CTOKEN *, CReference
*,MATFE::location *,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);
```

```
      };

   class CSassigneeMatrix: public CASTSyntaxElement {
     public:
             CSassigneeMatrix(CTOKEN*, CReference*,CTOKEN*,
               MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

   };

  class CMassigneeMatrix: public CASTSyntaxElement {
  public:
       CMassigneeMatrix(CTOKEN*,CReference*, CTOKEN*,
                                CReferenceList*, CTOKEN*,
MATFE::location *,

NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


   };

   class CReferenceList: public CASTSyntaxElement{
   public:
      CReferenceList(CReference *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

      CReferenceList(CReference *,CTOKEN*, CReferenceList*,
MATFE::location *,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};


  class CForCommand: public CASTSyntaxElement{
  public:
      CForCommand(CTOKEN*, CForCmdList*, CTOKEN*, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};


  class CForCmdList:public CASTSyntaxElement{
  public:
        CForCmdList(CIdentifier*, CTOKEN* ASSIGN, CExpression*,
CDelimitedInput*, MATFE::location *,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);
```

```
  };

  class CIfCommand: public CASTSyntaxElement{
  public:
      CIfCommand(CTOKEN*, CIfCmdList*, CTOKEN*, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

   };

  class CIfCmdList:public CASTSyntaxElement{
  public:
        CIfCmdList(CExpression *, CDelimitedInput*, COptElse*
,MATFE::location *,NONTERMINAL_PRODUCTION_RULE);


protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


  };

  class COptElse: public CASTSyntaxElement{
  public:
         COptElse(MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

        COptElse(CTOKEN* ELSE, CDelimitedInput *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

        COptElse(CTOKEN* ELSEIF, CExpression *, CDelimitedInput *,
COptElse*, MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

  };

  class CGlobalCommand: public CASTSyntaxElement{
  public:
      CGlobalCommand(CTOKEN* GLOBAL, CGlobalDeclList*, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);
protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};



  class CGlobalDeclList:public CASTSyntaxElement{
  public:
        CGlobalDeclList(CTOKEN* IDENTIFIER, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);
        CGlobalDeclList(CGlobalDeclList*, CTOKEN* IDENTIFIER
,MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};
```

```
  class CWhileCommand: public CASTSyntaxElement{
  public:
      CWhileCommand(CTOKEN* WHILE, CWhileCmdList*, CTOKEN* END,
MATFE::location *,NONTERMINAL_PRODUCTION_RULE);
      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

  class CWhileCmdList:public CASTSyntaxElement{
  public:
        CWhileCmdList(CExpression *, CDelimitedInput *,
MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};


  class CReturnCommand: public CASTSyntaxElement{
  public:
      CReturnCommand(CTOKEN*, MATFE::location *,
NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

   };

  class CDelimitedInput: public CASTSyntaxElement{
  public:
    CDelimitedInput(COptDelimiter *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

      CDelimitedInput(COptDelimiter *, CDelimitedList *,
MATFE::location *, NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

   };

  class CDelimitedList: public CASTSyntaxElement{
  public:
    CDelimitedList(CStatement *, CDelimiter *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

      CDelimitedList(CStatement *, CDelimiter *, CDelimitedList *,
MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

   };

  class CFunctionMFile : public CASTSyntaxElement{
public:
      CFunctionMFile(CEmptyLines *, CFDefLine *, CFBody *,
              MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CFunctionMFile(CFDefLine *, CFBody *,
```

```
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);
      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

};

class CFDefLine : public CASTSyntaxElement{
public:
      CFDefLine(CTOKEN* FUNCTION, CFOutput *, CTOKEN* ASSIGN, CTOKEN*
IDENTIFIER, CFInput *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CFDefLine(CTOKEN* FUNCTION, CTOKEN* IDENTIFIER, CFInput *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


};


  class CFOutput : public CASTSyntaxElement{
public:
      CFOutput(CIdentifier *,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CFOutput(CTOKEN* LD, CFArgumentList*, CTOKEN* RD,
            MATFE::location *,NONTERMINAL_PRODUCTION_RULE);
      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


 };


  class CFInput : public CASTSyntaxElement{
  public:
      CFInput(MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CFInput(CTOKEN* LD, CTOKEN* RD, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

      CFInput(CTOKEN* LD, CFArgumentList *, CTOKEN* RD,
MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

 };

 class CFArgumentList : public CASTSyntaxElement{
  public:
      CFArgumentList(CIdentifier *, CTOKEN *, CFArgumentList *,
MATFE::location *,NONTERMINAL_PRODUCTION_RULE);

      CFArgumentList(CIdentifier *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);
```

```
 };

 class CFBody : public CASTSyntaxElement{
 public:
      CFBody(CDelimiter *, CStatementList *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

      CFBody(COptDelimiter *, MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);

 };


 class CParseError : public CASTSyntaxElement{
 public:
       CParseError(CTOKEN* LEXERROR ,MATFE::location
*,NONTERMINAL_PRODUCTION_RULE);

       CParseError(MATFE::location *,NONTERMINAL_PRODUCTION_RULE);
      protected:
      virtual int map_syntax_element(CASTSyntaxElement *, int c=-1);


 };

#endif
```

File ASTSyntaxElements.cpp

```
#include "ASTSyntaxElements.h"

#include <iostream>
#include <fstream>
#include "driver.h"

using namespace std;

unsigned int CASTSyntaxElement::m_ObjectSerialNumberCounter=0;
CTransUnit* CTransUnit::instance = 0;

CASTSyntaxElement::CASTSyntaxElement(NONTERMINAL_PRODUCTION_RULE pr,
MATFE::location * loc) {


     m_NonTerminal_ProductionCode=pr;
     m_location=loc;
     m_ObjectSerialNumberCounter++;
     m_ObjectSerialNumber=m_ObjectSerialNumberCounter;

}


CASTSyntaxElement::~CASTSyntaxElement() {
  m_ObjectSerialNumberCounter--;
 }
```

```
CTOKEN::CTOKEN(TERMINAL_CODE ct,MATFE::location *loc,
       NONTERMINAL_PRODUCTION_RULE pr)
    :CASTSyntaxElement(pr,loc){
    m_NonTerminalCode= token_;
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=1;
}


int CTOKEN::map_syntax_element(CASTSyntaxElement *el, int m){
      switch(m_NonTerminal_ProductionCode){
            case Token__TOKEN:
                  //TOKEN
                  if ( el->m_NonTerminalCode == token_ ){
                        return 0;
                      }
                  break;

      default:
                  cout << endl << "Error in Token::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
                  ;
      }
return -1;
}



//------------------------------End of CTOKEN-------------------------
------------------------------------

CTransUnit::CTransUnit(CScriptMFile* ScriptMFile,
                                              MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
      m_NonTerminalCode= translation_unit_;
      int position = map_syntax_element(ScriptMFile);
      cout << "CCScriptMFile pos:" << position <<endl;
      m_Descendants.push_back(ScriptMFile);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      cout << "Graphstring is:" << graphstring << endl;
      cout << "Productions Tree ready" <<endl;
      cout <<"We reached translation unit!" << endl;
      CTransUnit::instance = this;

}


CTransUnit::CTransUnit(CFunctionMFile* FunctionMFile,
                                              MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
      m_NonTerminalCode= translation_unit_;
      int position = map_syntax_element(FunctionMFile);
      cout << "CFunctionMFile pos:" << position <<endl;
      m_Descendants.push_back(FunctionMFile);
      m_NumberOfDescendants=1;
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      cout << "Productions Tree ready" <<endl;
```

```
	cout <<"We reached translation unit!" << endl;
	CTransUnit::instance = this;
}


CTransUnit::CTransUnit(CParseError* ParseError,
	                                          MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

	m_NonTerminalCode= translation_unit_;
	m_Descendants.push_back(ParseError);
	m_NumberOfDescendants=1;
	this->graphstring = nonterminals[this->m_NonTerminalCode];
	cout <<"We reached the error trans-unit!" << endl;
	exit(1);
}



int CTransUnit::map_syntax_element(CASTSyntaxElement *el, int m){
	cout <<"CTransUnit Stage" << endl;
	cout <<"-------------" << endl;
    cout << "In TransUnit::map_syntax_element() -- " << endl;
    cout << "NTPC : " << m_NonTerminal_ProductionCode << endl;
    cout <<  "el->m_NonTerminalCode : " << el->m_NonTerminalCode <<
endl;
	switch(m_NonTerminal_ProductionCode){
		case TranslationUnit__ScriptMFile:
			//TranslationUnit:ScriptMFile
			if ( el->m_NonTerminalCode == scriptMFile_ ){
				return 0;
			    }
			break;
	    case TranslationUnit__FunctionMFile:
			//TranslationUnit:FunctionMFile
			if ( el->m_NonTerminalCode == functionMFile_){
				return 0;
			    }
			break;
	    case TranslationUnit__ParseError:
			//TranslationUnit:ParseError
			if ( el->m_NonTerminalCode == parse_error_ ){
				return 0;
			    }
		break;
	default:
			cout << endl << "Error in
TranslationUnit::map_syntax_element"
				<< endl << "System Exits !!!";
				exit(0);


	}
return -1;
}


//------------------------------End of CTransUnit---------------------
---------------------------------------
```

```
CScriptMFile::CScriptMFile(COptDelimiter* opdel,
                                             MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
      m_NonTerminalCode = scriptMFile_;
      int position = map_syntax_element(opdel);
      cout << "COptDelimiter pos:" << position <<endl;
      m_Descendants.push_back(opdel);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      cout << graphstring << endl;
      m_NumberOfDescendants=1;
}

CScriptMFile::CScriptMFile(COptDelimiter* opdel, CStatementList*
lstat,
                                             MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)

:CASTSyntaxElement(pr,loc){

   m_NonTerminalCode = scriptMFile_;
   m_Descendants.push_back(opdel);
   int position1 = map_syntax_element(lstat);
   cout << "CStatementList pos:" << position1 <<endl;
   m_Descendants.push_back(lstat);
   this->graphstring = nonterminals[this->m_NonTerminalCode];
   cout << graphstring << endl;
   m_NumberOfDescendants=2;
}


int CScriptMFile::map_syntax_element(CASTSyntaxElement *el, int m){
       cout <<"CScriptMFile Stage" << endl;
       cout <<"-------------" << endl;
       cout << "In CScriptMFile::map_syntax_element() -- " << endl;
     cout << "NTPC : " << m_NonTerminal_ProductionCode << endl;
     cout <<"opt_delimiter_: " << opt_delimiter_ << endl;
       cout <<"statement_list_: " << statement_list_ << endl;
     cout <<  "el->m_NonTerminalCode : " << el->m_NonTerminalCode  <<
endl;
      switch(m_NonTerminal_ProductionCode){
            case ScriptMFile__OptDelimiter:
                  //OptDelimiter
                  if ( el->m_NonTerminalCode == opt_delimiter_ ){
                        return 0;
                      }
                  break;
            case ScriptMFile__OptDelimiter_StatementList:
                  //OptDelimiter_StatementList
                  if ( el->m_NonTerminalCode == opt_delimiter_){
                        return 0;
                      }
                  else if (el->m_NonTerminalCode == statement_list_){
                return 1;
                      }
      break;
      default:
                  cout << endl << "Error in
ScriptMfile::map_syntax_element"
```

```
                        << endl << "System Exits !!!";
                        exit(0);
                  ;
      }
return -1;
}


//-------------------------------End of ScriptMfile -------------------
-----------------------------------------


COptDelimiter::COptDelimiter( MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

      m_NonTerminalCode = opt_delimiter_;
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=0;
}

COptDelimiter::COptDelimiter(CDelimiter* del,
                                              MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

      m_NonTerminalCode = opt_delimiter_;
      int position =map_syntax_element(del);
      cout << "Opt pos:" << position << endl;
      m_Descendants.push_back(del);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=1;

}


int COptDelimiter::map_syntax_element(CASTSyntaxElement *el, int m){
      cout <<"COptDelimiter Stage" << endl;
      cout <<"-------------" << endl;
     cout << "In COptDelimiter::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "delimiter_: " << delimiter_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
            case OptDelimiter__Delimiter:
                  //Delimiter
                  if ( el->m_NonTerminalCode == delimiter_ ){
                        return 0;
                      }
      break;
      default:
                  cout << endl << "Error in
Optdelimiter::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
                  ;
      }
return -1;
}
```

```
//---------------------------------------End of OptDelimiter----------
-----------------------------------------------------------

CDelimiter::CDelimiter(CNullLines* nls,
                                              MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

      m_NonTerminalCode = delimiter_;
      int position = map_syntax_element(nls);
      cout <<"Del.Position" <<position <<endl;
      m_Descendants.push_back(nls);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=1;

}

CDelimiter::CDelimiter(CEmptyLines* els,
                                              MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

      m_NonTerminalCode = delimiter_;
      int position = map_syntax_element(els);
      cout <<"Del.Position:" <<position <<endl;
      m_Descendants.push_back(els);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=1;

};

CDelimiter::CDelimiter(CNullLines* nls, CEmptyLines* els,
                                              MATFE::location *loc,

NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

      m_NonTerminalCode = delimiter_;
      int position = map_syntax_element(nls);
    cout << "CNullLines pos" << position << endl;
      m_Descendants.push_back(nls);
    m_Descendants.push_back(els);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=2;
}


int CDelimiter::map_syntax_element(CASTSyntaxElement *el, int m){
      cout <<"CDelimiter" << endl;
      cout <<"-------------" << endl;
     cout << "In CDelimiter::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "empty_lines_: " << empty_lines_ << endl <<
        "el->m_NonTerminalCode: " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
            case Delimiter__NullLines:
                  //NullLines
                  if ( el->m_NonTerminalCode == null_lines_){
```

```
                        return 0;
                    }
                break;
            case Delimiter__EmptyLines:
                //EmptyLines
                if ( el->m_NonTerminalCode == empty_lines_){
                        return 0;
                    }
                break;
            case Delimiter__NullLines_EmptyLines:
                //NullLines_EmptyLines
                if ( el->m_NonTerminalCode == null_lines_){
                        return 0;
                    }
        else if (el->m_NonTerminalCode == empty_lines_){
                return 1;
                    }
      break;
      default:
                cout << endl << "Error in
Delimiter::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
                ;
      }
return -1;
}


//----------------------------------------End of Delimiter-------------
-------------------------------------------------------


CNullLines::CNullLines(CNullLine* nl,
                                  MATFE::location *loc,
                                NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

      m_NonTerminalCode = null_lines_;
      int position = map_syntax_element(nl);
    cout << "CNullLine pos" << position << endl;
      m_Descendants.push_back(nl);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=1;
}

CNullLines::CNullLines(CNullLines* nls, CNullLine* nl,
                                  MATFE::location *loc,
                                NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

      m_NonTerminalCode = null_lines_;
      int position = map_syntax_element(nls);
    cout << "CNullLines pos" << position << endl;
      m_Descendants.push_back(nls);
      m_Descendants.push_back(nl);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=2;

}

int CNullLines::map_syntax_element(CASTSyntaxElement *el, int m){
```

```
        cout <<"CNullLines Stage" << endl;
        cout <<"-----------" << endl;
      cout << "In CNullLines::map_syntax_element() -- " << endl <<
          "NTPC : " << m_NonTerminal_ProductionCode << endl <<
          "null_line_: " << null_line_ << endl <<
          "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;

        switch(m_NonTerminal_ProductionCode){
              case NullLines__NullLine:
                    //NullLine
                    if ( el->m_NonTerminalCode == null_line_){
                          return 0;
                        }
                    break;
              case NullLines__NullLines_NullLine:
                    //NullLines_NullLine
                    if ( el->m_NonTerminalCode == null_lines_){
                          return 0;
                        }
               else if (el->m_NonTerminalCode == null_line_){
                  return 1;
                        }
        break;
        default:
                    cout << endl << "Error in
NullLines::map_syntax_element"
                          << endl << "System Exits !!!";
                          exit(0);
                    ;
        }
return -1;
}


//-----------------------------------------End of NullLines-------------
-------------------------------------------------------


CNullLine::CNullLine(CTOKEN* token,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

        m_NonTerminalCode = null_line_;
        int position = map_syntax_element(token);
      cout << "CTOKEN pos" << position << endl;
        m_Descendants.push_back(token);
        this->graphstring = nonterminals[this->m_NonTerminalCode];
        m_NumberOfDescendants=1;
}


CNullLine::CNullLine(CEmptyLines* els, CTOKEN* token,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
        m_NonTerminalCode = null_line_;
      int position = map_syntax_element(els);
        cout << "CEmptyLines pos:" << position << endl;
        m_Descendants.push_back(els);
        m_Descendants.push_back(token);
        this->graphstring = nonterminals[this->m_NonTerminalCode];
```

```
	m_NumberOfDescendants=2;
}

int CNullLine::map_syntax_element(CASTSyntaxElement *el, int m){
  cout <<"CNullLine Stage" << endl;
  cout <<"-----------" << endl;
  cout << "In CNullLine::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "token_ : " << token_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;

	switch(m_NonTerminal_ProductionCode){
		case NullLine__COMMA:
			if ( el->m_NonTerminalCode == token_ ){

						return 0;
			}
	  break;
		case NullLine__SEMICOLON:
			if ( el->m_NonTerminalCode == token_ ){

						return 0;
			}
	  break;
		case NullLine__EmptyLines_COMMA:
			if ( el->m_NonTerminalCode == empty_lines_ ){
				return 0;
			}
			else if ( el->m_NonTerminalCode == token_ ){
				return 1;
			}
		break;
		case NullLine__EmptyLines_SEMICOLON:
			if ( el->m_NonTerminalCode == empty_lines_ ){
				return 0;
			}
			else if ( el->m_NonTerminalCode == token_ ){
				return 1;
			}
		break;
		default:
			cout << endl << "Error in
NullLine::map_syntax_element"
				<< endl << "System Exits !!!";
				exit(0);
	}
return -1;
}


//---------------------------------------End of NullLine--------------
-----------------------------------------------------

CEmptyLines::CEmptyLines(CTOKEN* token,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
	m_NonTerminalCode = empty_lines_;
	m_Descendants.push_back(token);
	this->graphstring = nonterminals[this->m_NonTerminalCode];
	m_NumberOfDescendants=1;
```

```
}

CEmptyLines::CEmptyLines(CEmptyLines* els, CTOKEN* token,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
      m_NonTerminalCode = empty_lines_;
      int position = map_syntax_element(els);
    cout << "CEmptyLines pos" << position << endl;
    m_Descendants.push_back(els);
      m_Descendants.push_back(token);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=2;
}

int CEmptyLines::map_syntax_element(CASTSyntaxElement *el, int m){
  //CTOKEN *ct;
    cout << "In CEmptyLines::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "empty_lines_: " << empty_lines_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
            case EmptyLines__LINE:
                  //LINE
                  if ( el->m_NonTerminalCode == token_ ){
                              return 0;
                  }
        break;
            case EmptyLines__EmptyLines_LINE:
                  //EmptyLines_LINE
                  if ( el->m_NonTerminalCode == empty_lines_ ){
                        return 0;
                  }
                  else if ( el->m_NonTerminalCode == token_ ){

                                    return 1;
                  }
            break;
            default:
                  cout << endl << "Error in
EmptyLines::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//---------------------------------------End of EmptyLines------------
-------------------------------------------------------


CStatementList::CStatementList(CStatement* stat, COptDelimiter*
optdel,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

  m_NonTerminalCode = statement_list_;
  int position = map_syntax_element(stat);
  cout << "CStatementList pos" << position << endl;
  m_Descendants.push_back(stat);
```

```
  m_Descendants.push_back(optdel);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;

}


CStatementList::CStatementList(CStatement* stat, CDelimiter* delm,
CStatementList* statl,
                                   MATFE::location *loc,
                                 NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

  m_NonTerminalCode = statement_list_;
  m_Descendants.push_back(stat);
  m_Descendants.push_back(stat);
  m_Descendants.push_back(statl);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;

}

int CStatementList::map_syntax_element(CASTSyntaxElement *el, int m){
          cout << "CStatementList::map_syntax_element() -- " << endl
<<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "statement_ : " << statement_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl <<
            "CStatementList::map_syntax_element() -- " << endl <<
        "opt_delimiter_: " << opt_delimiter_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;


      switch(m_NonTerminal_ProductionCode){
            case StatementList__Statement_OptDelimiter:
                  //Statement_OptDelimiter
                  if ( el->m_NonTerminalCode == statement_ ){
                              return 0;
                                   }
                  else if ( el->m_NonTerminalCode == opt_delimiter_ ){
                              return 1;
                  }
            break;
            case StatementList__Statement_Delimiter_StatementList:
                  //Statement_Delimiter_StatementList
                  if ( el->m_NonTerminalCode == statement_){
                        return 0;
                  }
                  else if (el->m_NonTerminalCode == delimiter_){
                        return 1;
                  }
                  else if (el->m_NonTerminalCode == statement_list_){
                        return 2;
                  }
            break;
            default:
                  cout << endl << "Error in
StatementList::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
```

```
}


//--------------------------------------End of StatementList---------
----------------------------------------------------------

CStatement::CStatement(CCommandForm* cmd,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

  m_NonTerminalCode = statement_;
  int position = map_syntax_element(cmd);
  cout << "CStatement pos" << position << endl;
  m_Descendants.push_back(cmd);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;

}

CStatement::CStatement(CExpression* exp,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = statement_;
  int position = map_syntax_element(exp);
  cout << "CStatement pos" << position << endl;
  m_Descendants.push_back(exp);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;


}

CStatement::CStatement(CAssignment* asg,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = statement_;
  int position = map_syntax_element(asg);
  cout << "CStatement pos" << position << endl;
  m_Descendants.push_back(asg);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}


CStatement::CStatement(CForCommand* fcm,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = statement_;
  int position = map_syntax_element(fcm);
  cout << "CStatement pos" << position << endl;
  m_Descendants.push_back(fcm);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CStatement::CStatement(CIfCommand* icm,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
```

```
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = statement_;
  int position = map_syntax_element(icm);
  cout << "CStatement pos" << position << endl;
  m_Descendants.push_back(icm);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CStatement::CStatement(CGlobalCommand* gcm,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = statement_;
  int position = map_syntax_element(gcm);
  cout << "CStatement pos" << position << endl;
  m_Descendants.push_back(gcm);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CStatement::CStatement(CWhileCommand* wcm,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = statement_;
  int position = map_syntax_element(wcm);
  cout << "CStatement pos" << position << endl;
  m_Descendants.push_back(wcm);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CStatement::CStatement(CReturnCommand* rcm,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = statement_;
  int position = map_syntax_element(rcm);
  cout << "CStatement pos" << position << endl;
  m_Descendants.push_back(rcm);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

int CStatement::map_syntax_element(CASTSyntaxElement *el, int m){
    cout<<"CStatement stage" << endl;
      cout <<"-----------" << endl;
      cout << "In CStatement::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "statement_ : " << statement_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;

      switch(m_NonTerminal_ProductionCode){
            case Statement__CommandForm:
                  //CommandForm
                  if ( el->m_NonTerminalCode == command_form_ ){
                              return 0;
                                    }
            break;
            case Statement__Expr:
                  //Expr
```

```
                  if ( el->m_NonTerminalCode == expr_){
                        return 0;
                  }
            break;
            case Statement__Assignment:
                  //Expr
                  if ( el->m_NonTerminalCode == assignment_){
                        return 0;
                  }
            break;
            case Statement__ForCommand:
                  //ForCommand
                  if ( el->m_NonTerminalCode == for_command_){
                        return 0;
                  }
            break;
            case Statement__IfCommand:
                  //IfCommand
                  if ( el->m_NonTerminalCode == if_command_){
                        return 0;
                  }
            break;
        case Statement__GlobalCommand:
                  //GlobalCommand
                  if ( el->m_NonTerminalCode == global_command_){
                        return 0;
                  }
            break;
            case Statement__WhileCommand:
                  //WhileCommand
                  if ( el->m_NonTerminalCode == while_command_){
                        return 0;
                  }
            break;
            case Statement__ReturnCommand:
                  //ReturnCommand
                  if ( el->m_NonTerminalCode == return_command_){
                        return 0;
                  }
            break;
            default:
                  cout << endl << "Error in
Statement::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//---------------------------------------End of Statement-------------
-------------------------------------------------------

CCommandForm::CCommandForm(CIdentifier* id, CTextList* tl,
                                    MATFE::location *loc,
                                  NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = command_form_;
  int position = map_syntax_element(id);
  cout << "CCommandForm pos" << position << endl;
  m_Descendants.push_back(id);
  m_Descendants.push_back(tl);
```

```
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

int CCommandForm::map_syntax_element(CASTSyntaxElement *el, int m){
      cout << "In CCommandForm::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "command_form_: " << command_form_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;

      switch(m_NonTerminal_ProductionCode){
            case CommandForm__Identifier_TextList:
                  //Identifier_TextList
                  if ( el->m_NonTerminalCode == identifier_ ){
                              return 0;
                                   }
                  else if ( el->m_NonTerminalCode == text_list_ ){
                              return 1;
                  }
            break;
            default:
                  cout << endl << "Error in
CommandForm::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//---------------------------------------End of CommandForm-----------
---------------------------------------------------------


 CTextList::CTextList(CTOKEN* token,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = text_list_;
  int position = map_syntax_element(token);
  cout << "CTextList pos" << position << endl;
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
};


CTextList::CTextList(CTextList* tl, CTOKEN* token,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = text_list_;
  int position = map_syntax_element(tl);
  cout << "CTextList pos" << position << endl;
  m_Descendants.push_back(tl);
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

int CTextList::map_syntax_element(CASTSyntaxElement *el, int m){
cout << "In CTextList::map_syntax_element() -- " << endl <<
```

```
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "text_list_ : " << text_list_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
            case TextList__TEXT:
                  //TEXT
                  if ( el->m_NonTerminalCode == token_){
                        return 0;
                      }
                  break;
            case TextList__TextList_TEXT:
                  //TextList_TEXT
                  if ( el->m_NonTerminalCode == text_list_){
                        return 0;
                      }
             else if (el->m_NonTerminalCode == token_){
                return 1;
                      }
      break;
      default:
                  cout << endl << "Error in
TextList::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
                  ;
      }
return -1;
}


//---------------------------------------End of Textlist--------------
-----------------------------------------------------

CExpression::CExpression(CTOKEN* token,
                   MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
   m_NonTerminalCode = expr_;
   int position = map_syntax_element(token);
   cout << "CTOKEN pos" << position << endl;
   m_Descendants.push_back(token);
   this->graphstring = nonterminals[this->m_NonTerminalCode];
   m_NumberOfDescendants=1;
}


CExpression::CExpression(CTOKEN * token, CExpression* exp,
                   MATFE::location *loc, NONTERMINAL_PRODUCTION_RULE
pr)
 :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = expr_;
  int position = map_syntax_element(token);
  cout << "CTOKEN pos" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

CExpression::CExpression(CTOKEN * token, CExpression* exp, CTOKEN *
token1,
```

```
                   MATFE::location *loc, NONTERMINAL_PRODUCTION_RULE
pr)
 :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = expr_;
  int position = map_syntax_element(token);
  cout << "CTOKEN pos" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp);
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

CExpression::CExpression(CReference* ref,
                   MATFE::location *loc, NONTERMINAL_PRODUCTION_RULE
pr)
 :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = expr_;
  int position = map_syntax_element(ref);
  cout << "CExpression pos" << position << endl;
  m_Descendants.push_back(ref);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CExpression::CExpression(CMatrix* mat,
                   MATFE::location *loc, NONTERMINAL_PRODUCTION_RULE
pr)
 :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = expr_;
  int position = map_syntax_element(mat);
  cout << "CMatrix pos" << position << endl;
  m_Descendants.push_back(mat);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}


CExpression::CExpression(CExpression * exp, CTOKEN* token,
CExpression* exp1,
                   MATFE::location *loc, NONTERMINAL_PRODUCTION_RULE
pr)
 :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = expr_;
  int position = map_syntax_element(exp);
  cout << "CExpression pos" << position << endl;
  m_Descendants.push_back(exp);
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}


CExpression::CExpression(CColonExpression* col,
                   MATFE::location *loc, NONTERMINAL_PRODUCTION_RULE
pr)
 :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = expr_;
  int position = map_syntax_element(col);
  cout << "CColonExpression pos" << position << endl;
  m_Descendants.push_back(col);
```

```
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CExpression::CExpression(CExpression* exp, CTOKEN* token,
                   MATFE::location *loc, NONTERMINAL_PRODUCTION_RULE
pr)
 :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = expr_;
  int position = map_syntax_element(exp);
  cout << "CExpression pos" << position << endl;
  m_Descendants.push_back(exp);
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}



int CExpression::map_syntax_element(CASTSyntaxElement *el, int m){
cout <<"CExpression Stage" << endl;
cout <<"-----------" << endl;
cout << "In CExpression::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "token_ : " << token_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
        switch(m_NonTerminal_ProductionCode){
            case Expr__INTEGER:
                  if ( el->m_NonTerminalCode == token_ ){
                        return 0;
                  }
        break;
            case Expr__DOUBLE:
                  if ( el->m_NonTerminalCode == token_ ){
                        return 0;
                  }
        break;
            case Expr__IMAGINARY:
                  if ( el->m_NonTerminalCode == token_ ){
                        return 0;
                  }
        break;
            case Expr__TEXT:
                  if ( el->m_NonTerminalCode == token_){
                        return 0;
                  }
        break;
        case Expr__LPAREN_Expr_RPAREN:
                  //LPAREN_Expr
                  if ( el->m_NonTerminalCode == token_){
                        return 0;}
                 else if (el->m_NonTerminalCode = expr_){
                              return 1;}
                else if (el->m_NonTerminalCode == token_){
                        return 1;}
      break;
            case Expr__Reference:
                  //Reference
                  if ( el->m_NonTerminalCode == reference_){
                        return 0;
                      }
             break;
```

```
  case Expr__Matrix:
          //Matrix
          if ( el->m_NonTerminalCode == matrix_){
                return 0;
              }
break;
case Expr__ColonExpr:
          //ColonExpr
          if ( el->m_NonTerminalCode == colon_expr_){
                return 0;
              }
break;
    case Expr__Expr_EPOWER_Expr:
          if ( el->m_NonTerminalCode == expr_) {
                return 0;}

          else if(el->m_NonTerminalCode == token_){
                return 1;}

       else if(el->m_NonTerminalCode == expr_){
             return 2;}
    break;
    case Expr__Expr_POWER_Expr:
          if ( el->m_NonTerminalCode == expr_) {
                return 0;}

          else if(el->m_NonTerminalCode == token_){
                return 1;}

           else if(el->m_NonTerminalCode == expr_){
             return 2;}
    break;
    case Expr__Expr_BIMUL_Expr:
          if ( el->m_NonTerminalCode == expr_) {
                return 0;}

          else if(el->m_NonTerminalCode == token_){
                return 1;}

           else if(el->m_NonTerminalCode == expr_){
             return 2;}

    break;
    case Expr__Expr_BIDIV_Expr:
          if ( el->m_NonTerminalCode == expr_) {
                return 0;}

          else if(el->m_NonTerminalCode == token_){
                return 1;}

           else if(el->m_NonTerminalCode == expr_){
             return 2;}

    break;
    case Expr__Expr_BILEFTDIV_Expr:
          if ( el->m_NonTerminalCode == expr_) {
                return 0;}

          else if(el->m_NonTerminalCode == token_){
                return 1;}
           else if(el->m_NonTerminalCode == expr_){
             return 2;}
```

```
break;
case Expr__Expr_EMUL_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}
       else if(el->m_NonTerminalCode == expr_){
         return 2;}

break;
case Expr__Expr_EDIV_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;   }
       else if(el->m_NonTerminalCode == expr_){
         return 2;}

break;
case Expr__Expr_ELEFTDIV_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
       return 1;
      }
       else if(el->m_NonTerminalCode == expr_){
         return 2;}
break;
case Expr__Expr_PLUS_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}

      else if(el->m_NonTerminalCode == expr_){
            return 2;}

break;
case Expr__Expr_MINUS_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}

      else if(el->m_NonTerminalCode == expr_){
            return 2;}
break;
case Expr__Expr_LTHAN_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}

      else if(el->m_NonTerminalCode == expr_){
            return 2;}
```

```
break;
case Expr__Expr_LTHANE_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}

      else if(el->m_NonTerminalCode == expr_){
            return 2;}
break;
case Expr__Expr_GTHAN_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}

      else if(el->m_NonTerminalCode == expr_){
            return 2;}
break;
case Expr__Expr_GTHANE_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}

      else if(el->m_NonTerminalCode == expr_){
            return 2;}
break;
case Expr__Expr_EQUAL_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}

      else if(el->m_NonTerminalCode == expr_){
            return 2;}
break;
case Expr__Expr_UNEQUAL_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}

      else if(el->m_NonTerminalCode == expr_){
            return 2;}
break;
case Expr__Expr_AND_Expr:
      if ( el->m_NonTerminalCode == expr_) {
            return 0;}

      else if(el->m_NonTerminalCode == token_){
            return 1;}

      else if(el->m_NonTerminalCode == expr_){
            return 2;}
break;
case Expr__Expr_OR_Expr:
```

```
                  if ( el->m_NonTerminalCode == expr_) {
                        return 0;}

                  else if(el->m_NonTerminalCode == token_){
                        return 1;}

                  else if(el->m_NonTerminalCode == expr_){
                        return 2;}
            break;
            case Expr__Expr_TRANSPOSE:
                  if ( el->m_NonTerminalCode == expr_){
                        return 0;
                      }
                  else if(el->m_NonTerminalCode == token_){
                        return 1;
                  }
        break;
            case Expr__Expr_CTRANSPOSE:
                  if ( el->m_NonTerminalCode == expr_){
                        return 0;
                      }
                  else if(el->m_NonTerminalCode == token_){

                 return 1;
                  }
        break;
          case Expr__NOT_Expr:
                  if ( el->m_NonTerminalCode == expr_){
                        return 0;
                      }
                  else if(el->m_NonTerminalCode == token_)
                        {
                        return 1;
                  }
        break;
            case Expr__PLUS_Expr:
                  if ( el->m_NonTerminalCode == expr_){
                        return 0;
                      }
                  else if(el->m_NonTerminalCode == token_)
                        {
                        return 1;
                  }
        break;
            case Expr__MINUS_Expr:
                  if ( el->m_NonTerminalCode == token_){
                        return 0;
                      }
                  else if(el->m_NonTerminalCode == expr_){
                              return 1;
                  }
        break;
            default:
                  cout << endl << "Error in
Expression::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }

   return -1;
}
```

```
//--------------------------------------End of Expression------------
-----------------------------------------------------

CReference::CReference(CIdentifier* id,
                                    MATFE::location *loc,
                                  NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

   m_NonTerminalCode = reference_;
   int position = map_syntax_element(id);
   cout << "CIdentifier pos:" << position << endl;
   m_Descendants.push_back(id);
   this->graphstring = nonterminals[this->m_NonTerminalCode];
   m_NumberOfDescendants=1;

}

CReference::CReference(CIdentifier *id, CTOKEN *token,
                                CArgumentList *arg, CTOKEN *token1,
                                MATFE::location *loc,
                                NONTERMINAL_PRODUCTION_RULE pr)
  :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = reference_;
  int position = map_syntax_element(id);
  cout << "CIdentifier pos:" << position << endl;
  m_Descendants.push_back(id);
  m_Descendants.push_back(token);
  m_Descendants.push_back(arg);
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=4;
}

int CReference::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "In CReference::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "identifier_: " << identifier_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
            case Reference__Identifier:
                  //Identifier
                  if ( el->m_NonTerminalCode == identifier_ ){
                        return 0;
                      }
        break;
            case Reference__Identifier_LPAREN_ArgumentList_RPAREN:
                  //Identifier_LPAREN_ArgumentList_RPAREN
                  if ( el->m_NonTerminalCode == identifier_){
                        return 0;
                  }
                  else if (el->m_NonTerminalCode == token_){
                              return 1; }
                  else if (el->m_NonTerminalCode == argument_list_){
                        return 2;}
                  else if (el->m_NonTerminalCode == token_){
                        return 3;
                  }
            break;
      default:
```

```
                    cout << endl << "Error in
Reference::map_syntax_element"
                          << endl << "System Exits !!!";
                          exit(0);
                    ;
      }
return -1;
}

//-----------------------------------------------End of Reference-----
------------------------------------------------

CIdentifier::CIdentifier(CTOKEN* token,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
    m_NonTerminalCode = identifier_;
      int position = map_syntax_element(token);
      cout << "CIdentifier pos:" << position << endl;
      m_Descendants.push_back(token);
      this->graphstring = nonterminals[this->m_NonTerminalCode];
      m_NumberOfDescendants=1;
}

int CIdentifier::map_syntax_element(CASTSyntaxElement *el, int m){
    cout <<"CIdentifier Stage" << endl;
    cout <<"-----------" << endl;
    cout << "In CIdentifier::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "token_ : " << token_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;

      switch(m_NonTerminal_ProductionCode){
            case Identifier__IDENTIFIER:
                  //IDENTIFIER

                  if ( el->m_NonTerminalCode == token_){
                        return 0;
                      }
      break;
      default:
                  cout << endl << "Error in
Identifier::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
                  ;
      }
return -1;
}



//-----------------------------------------------End of Identifier----
-------------------------------------------------

CArgumentList::CArgumentList(CTOKEN* token,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){

  m_NonTerminalCode = argument_list_;
  m_Descendants.push_back(token);
```

```
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;

}

CArgumentList::CArgumentList(CExpression* exp,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = argument_list_;
  int position = map_syntax_element(exp);
  cout << "CExpression pos:" << position << endl;
  m_Descendants.push_back(exp);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CArgumentList::CArgumentList(CTOKEN* token, CTOKEN* token1,
                                     CArgumentList* argl,
MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = argument_list_;
  int position = map_syntax_element(argl);
  cout <<"ArgumentList pos:" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(token1);
  m_Descendants.push_back(argl);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}


CArgumentList::CArgumentList(CExpression* exp, CTOKEN* token,
                                     CArgumentList* argl,
MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = argument_list_;
  int position = map_syntax_element(argl);
  cout <<"ArgumentList pos:" << position << endl;
  m_Descendants.push_back(exp);
  m_Descendants.push_back(token);
  m_Descendants.push_back(argl);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

int CArgumentList::map_syntax_element(CASTSyntaxElement *el, int m){
 cout <<"CArgumentList Stage" << endl;
      cout <<"-----------" << endl;
    cout << "CArgumentList::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "argument_list_: " << argument_list_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     ArgumentList__COLONOPERATOR:
                  //COLONOPERATOR
                  if ( el->m_NonTerminalCode == token_){
                                  return 0;
                  }
        break;
```

```
            case ArgumentList__Expr:
                  //Expr
                  if ( el->m_NonTerminalCode == expr_ ){
                                    return 0;
                  }
        break;

        case ArgumentList__COLONOPERATOR_COMMA_ArgumentList:
                  //COLONOPERATOR_COMMA_ArgumentList
                   if (el->m_NonTerminalCode == token_ ){
                               return 0; }
                   else if (el->m_NonTerminal_ProductionCode ==
token_){
                          return 1;
                          }
                   else if (el->m_NonTerminalCode == argument_list_){
                         return 2;}

        break;
            case ArgumentList__Expr_COMMA_ArgumentList:
                  //Expr_COMMA_ArgumentList
                  if ( el->m_NonTerminalCode == expr_){
                        return 0;
                      }
                  else if (el->m_NonTerminalCode == token_){
                        return 1;
                  }
                  else if (el->m_NonTerminalCode == argument_list_){
                        return 2;
                  }
        break;
            default:
                  cout << endl << "Error in
ArgumentList::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//---------------------------------------End of ArgumentList----------
---------------------------------------------------------

CMatrix::CMatrix(CTOKEN* token, CRows* rows, CTOKEN* token1,
                                     MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = matrix_;
  int position = map_syntax_element(rows);
  cout <<"CRows pos:" << position << endl;
  m_Descendants.push_back(rows);
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

int CMatrix::map_syntax_element(CASTSyntaxElement *el, int m){
      cout << "CMatrix::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "rows_ : " << rows_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
```

```
      switch(m_NonTerminal_ProductionCode){
         case     Matrix__LB_Rows_RB:
                  //LB_Rows
                  if ( el->m_NonTerminalCode == token_){
                        return 0;
                      }
                  else if (el->m_NonTerminalCode == rows_){
                        return 1;
                  }
                  else if (el->m_NonTerminalCode == token_){
                        return 2;
                  }
        break;
            default:
                  cout << endl << "Error in
Matrix::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//-----------------------------------------End of Matrix----------------
---------------------------------------------------


CRows::CRows(MATFE::location *loc,
           NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = rows_;
  m_NumberOfDescendants=0;
  this->graphstring = nonterminals[this->m_NonTerminalCode];
}



CRows::CRows(CRow* row,
                   MATFE::location *loc,
                 NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = rows_;
  int position =map_syntax_element(row);
  cout <<"CRow pos:"<<position<<endl;
  m_Descendants.push_back(row);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CRows::CRows(CRows* rows, CTOKEN* token,
                                    MATFE::location *loc,
                                  NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = rows_;
  int position =map_syntax_element(rows);
  cout <<"CRows pos:"<<position<<endl;
  m_Descendants.push_back(rows);
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}
```

```
CRows::CRows(CRows* rows, CTOKEN* token, CRow* row,
                                   MATFE::location *loc,
                                 NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = rows_;
  int position =map_syntax_element(row);
  cout <<"CRows pos:"<<position<<endl;
  m_Descendants.push_back(rows);
  m_Descendants.push_back(token);
  m_Descendants.push_back(row);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

int CRows::map_syntax_element(CASTSyntaxElement *el, int m){
cout << "CRows::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "row_ : " << row_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     Rows__Row:
                  //Row
                  if ( el->m_NonTerminalCode == row_ ){
                        return 0;}

        break;
            case Rows__Rows_SEMICOLON:
            //Rows_SEMICOLON
                  if ( el->m_NonTerminalCode == rows_ ){
                        return 0;}
                  else if (el->m_NonTerminalCode == token_){
                  return 1;
                        }
         break;
            case Rows__Rows_LINE:
            //Rows_LINE
                  if ( el->m_NonTerminalCode == rows_ ){
                        return 0;}
                  else if (el->m_NonTerminalCode == token_){
                  return 1;
                        }
         break;
            case Rows__Rows_SEMICOLON_Row:
              //Rows_SEMICOLON_Row
                  if ( el->m_NonTerminalCode == rows_ ){
                        return 0;}
            else if (el->m_NonTerminalCode == token_){
                  return 1;
                        }
            else if(el->m_NonTerminalCode == row_){
                  return 2;}
            break;
            case Rows__Rows_LINE_Row:
              //Rows_LINE_Row
                  if ( el->m_NonTerminalCode == rows_ ){
                        return 0;}
            else if (el->m_NonTerminalCode == token_){
                  return 1;
                        }
            else if(el->m_NonTerminalCode == row_){
                  return 2;}
            break;
```

```
            default:
                  cout << endl << "Error in Rows::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//---------------------------------------End of Rows------------------
-------------------------------------------------

CRow::CRow(CExpression* exp,
                   MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = row_;
  int position =map_syntax_element(exp);
  cout <<"CExpression pos:"<<position<<endl;
  m_Descendants.push_back(exp);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CRow::CRow(CRowWithCommas* rwc,
                 MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = row_;
  int position =map_syntax_element(rwc);
  cout <<"CRowWithCommas pos:"<<position<<endl;
  m_Descendants.push_back(rwc);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CRow::CRow(CRowWithCommas* rwc,CExpression* exp,
                    MATFE::location *loc,
                    NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = row_;
  int position =map_syntax_element(rwc);
  cout <<"CRowWithCommas pos:"<<position<<endl;
  m_Descendants.push_back(rwc);
  m_Descendants.push_back(exp);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

int CRow::map_syntax_element(CASTSyntaxElement *el, int m){
      cout << "CRows::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "expr_: " << expr_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     Row__Expr:
                  //Expr
                  if ( el->m_NonTerminalCode == expr_ ){
                        return 0;}

        break;
            case Row__RowWithCommas:
```

```
            //RowWithCommas
                  if ( el->m_NonTerminalCode == row_with_commas_ ){
                        return 0;}
        break;
            case Row__RowWithCommas_Expr:
              //RowWithCommas_Expr
                  if ( el->m_NonTerminalCode == row_with_commas_ ){
                        return 0;}
            else if (el->m_NonTerminalCode == expr_){
                           return 1;}
                     break;
            default:
                  cout << endl << "Error in Row::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//---------------------------------------End of Row--------------------
------------------------------------------------

CRowWithCommas::CRowWithCommas(CExpression* exp, CTOKEN* token,
                      MATFE::location *loc,
                      NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = row_with_commas_;
  int position =map_syntax_element(exp);
  cout <<"CExpression pos:"<<position<<endl;
  m_Descendants.push_back(exp);
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

CRowWithCommas::CRowWithCommas(CRowWithCommas* rwc, CExpression* exp,
                    CTOKEN* token, MATFE::location *loc,
                      NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = row_with_commas_;
  int position =map_syntax_element(rwc);
  cout <<"CRowWithCommas pos:"<<position<<endl;
  m_Descendants.push_back(rwc);
  m_Descendants.push_back(exp);
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}


int CRowWithCommas::map_syntax_element(CASTSyntaxElement *el, int m){
//CTOKEN* ct;
      cout <<"CRowWithCommas Stage" << endl;
      cout <<"-----------" << endl;
    cout << "In CRowWithCommas::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "exp_: " << null_line_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     RowWithCommas__Expr_COMMA:
                  //Expr_COMMA
```

```
                  if (el->m_NonTerminalCode == expr_ ){
                        return 0;}
                  else if(el->m_NonTerminalCode == token_){
                        return 1;}
        break;
            case RowWithCommas__RowWithCommas_Expr_COMMA:
            //RowWithCommas_Expr_COMMA
                  if ( el->m_NonTerminalCode == row_with_commas_ ){
                        return 0;}
                  else if(el->m_NonTerminalCode == expr_){
                        return 1;}
                  else if(el->m_NonTerminalCode == token_){
                        return 2;}
        break;
            default:
                  cout << endl << "Error in
RowWithCommas::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//----------------------------------------End of RowWithCommas---------
----------------------------------------------------------
CColonExpression::CColonExpression(CExpression* exp, CTOKEN* token,
                        CExpression* exp1, MATFE::location *loc,
                        NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = colon_expr_;
  int position =map_syntax_element(exp);
  cout <<"CExpression pos:"<<position<<endl;
  m_Descendants.push_back(exp);
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

CColonExpression::CColonExpression(CColonExpression* cexp, CTOKEN*
token,
                        CExpression* exp, MATFE::location *loc,
                        NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = colon_expr_;
  int position =map_syntax_element(cexp);
  cout <<"CColonExpression pos:"<<position<<endl;
  m_Descendants.push_back(cexp);
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}


int CColonExpression::map_syntax_element(CASTSyntaxElement *el, int
m){
cout << "In CColonExpression::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "expr_ : " << expr_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
```

```
      switch(m_NonTerminal_ProductionCode){
         case     ColonExpr__Expr_COLONOPERATOR_Expr:
                  //Expr_COLONOPERATOR_Expr
                  if ( el->m_NonTerminalCode == expr_){
                        return 0;
                  }
                  else if  (el->m_NonTerminalCode == token_){
                        return 1;}
              break;
            case ColonExpr__ColonExpr_Expr_COLONOPERATOR_Expr:
            //ColonExpr_Expr_COLONOPERATOR_Expr
                  if ( el->m_NonTerminalCode == colon_expr_ ){
                        return 0;}
                  else if(el->m_NonTerminalCode == token_){
                        return 1;}
                  else if(el->m_NonTerminalCode == expr_){
                        return 2;}
        break;
            default:
                  cout << endl << "Error in
ColonExpr::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//---------------------------------------End of ColonExpr-------------
------------------------------------------------------

 CAssignment::CAssignment(CReference *ref, CTOKEN* token,
                         CExpression* exp,MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = assignment_;
  m_Descendants.push_back(ref);
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

CAssignment::CAssignment(CSassigneeMatrix *csa, CTOKEN* token,
                         CExpression* exp,MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = assignment_;
  m_Descendants.push_back(csa);
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

CAssignment::CAssignment(CMassigneeMatrix *msa, CTOKEN* token,
                    CReference* ref,MATFE::location *loc,
                    NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = assignment_;
  int position = map_syntax_element(msa);
```

```
  cout << "CMassigneeMatrix pos" <<position << endl;
  m_Descendants.push_back(msa);
  m_Descendants.push_back(token);
  m_Descendants.push_back(ref);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
};

int CAssignment::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "In CAssignment::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "s_assignee_matrix_: " << s_assignee_matrix_  << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     Assignment__Reference_ASSIGN_Expr:
                  //Reference_ASSIGN_Expr
                  if ( el->m_NonTerminalCode == reference_ ){
                        return 0;}
                  else if ( el->m_NonTerminalCode == token_){
                        return 1;}
                  else if ( el->m_NonTerminalCode == expr_ ){
                        return 2;}
        break;
            case Assignment__SAssigneeMatrix_ASSIGN_Expr:
            //SAssigneeMatrix_ASSIGN_Exp
                  if ( el->m_NonTerminalCode == s_assignee_matrix_ ){
                        return 0;}
                  else if ( el->m_NonTerminalCode == token_){
                        return 1;}
                  else if ( el->m_NonTerminalCode == expr_ ){
                        return 2;}
        break;
            case Assignment__MAssigneeMatrix_ASSIGN_Reference:
            //MAssigneeMatrix_ASSIGN_Reference
                  if ( el->m_NonTerminalCode == m_assignee_matrix_ ){
                        return 0;}
                  else if ( el->m_NonTerminalCode == token_){
                        return 1;}
                  else if ( el->m_NonTerminalCode == reference_ ){
                        return 2;}
                  break;
            default:
                  cout << endl << "Error in
Assignment::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//---------------------------------------End of Assignment------------
-------------------------------------------------------

CSassigneeMatrix::CSassigneeMatrix(CTOKEN* token,
                                  CReference* ref, CTOKEN* token1,
                                 MATFE::location *loc,
                               NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = s_assignee_matrix_;
  m_Descendants.push_back(token);
  m_Descendants.push_back(ref);
```

```
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

int CSassigneeMatrix::map_syntax_element(CASTSyntaxElement *el, int
m){
    cout << "In CSassigneeMatrix::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "reference_ : " << reference_  << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     SAssigneeMatrix__LD_Reference_RD:
                  //Boxes2_LD_Reference_RD
                  if ( el->m_NonTerminalCode == token_){
                        return 0;}
                  else if ( el->m_NonTerminalCode == reference_ ){
                        return 1;}
                  else if ( el->m_NonTerminalCode == token_ ){
                        return 1;}
        break;
            default:
                  cout << endl << "Error in
SAssigneeMatrix::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//---------------------------------------End of SAssigneeMatrix-------
------------------------------------------------------------


CMassigneeMatrix::CMassigneeMatrix(CTOKEN* token,
                                      CReference* ref, CTOKEN* token1,
                                      CReferenceList* rfl, CTOKEN*
token2, MATFE::location *loc,
                                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = m_assignee_matrix_;
  int position = map_syntax_element(ref);
  cout << "CReference pos:" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(ref);
  m_Descendants.push_back(token1);
  m_Descendants.push_back(rfl);
  m_Descendants.push_back(token2);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=5;
}

int CMassigneeMatrix::map_syntax_element(CASTSyntaxElement *el, int
m){
       cout << "In CMassigneeMatrix::map_syntax_element() -- " << endl
<<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "reference_ : " << reference_  << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
      case MAssigneeMatrix__LD_Reference_COMMA_ReferenceList_RD:
```

```
            //LD_Reference_COMMA_ReferenceList_RD
        if ( el->m_NonTerminalCode == token_){
                        return 0;}
        else if (el->m_NonTerminal_ProductionCode == reference_){
                      return 1;}
        else if ( el->m_NonTerminalCode == token_ ){
                        return 2;}
        else if ( el->m_NonTerminalCode == reference_list_ ){
                        return 3;}
        else if ( el->m_NonTerminalCode == token_ ){
                        return 4;}
          break;
            default:
                  cout << endl << "Error in
MAssigneeMatrix::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//-----------------------------------------End of MAssigneeMatrix-------
------------------------------------------------------------


CReferenceList::CReferenceList(CReference *ref,
                        MATFE::location *loc,
                        NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = reference_list_;
  int position = map_syntax_element(ref);
  cout << "Reference pos:" << position << endl;
  m_Descendants.push_back(ref);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
};

CReferenceList::CReferenceList(CReference *ref, CTOKEN* token,
                      CReferenceList* cref, MATFE::location *loc,
                        NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = reference_list_;
  int position = map_syntax_element(ref);
  cout << "Reference pos:" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(cref);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}


int CReferenceList::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "CReferenceList::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "reference_ " << reference_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     ReferenceList__Reference:
                  //Reference
                  if ( el->m_NonTerminalCode == reference_ ){
                        return 0;}
```

```
        break;
            case ReferenceList__Reference_COMMA_ReferenceList:
            //Reference_COMMA_ReferenceList
                  if ( el->m_NonTerminalCode == reference_ ){
                        return 0;}
                  else if ( el->m_NonTerminalCode == token_){
                        return 1;}
                  else if ( el->m_NonTerminalCode == reference_list_
){
                        return 2;}
        break;
            default:
                  cout << endl << "Error in
ReferenceList::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
      }
return -1;
}


//--------------------------------------End of ReferenceList---------
----------------------------------------------------------
CForCommand::CForCommand(CTOKEN* token,
                   CForCmdList* cfcl, CTOKEN* token1, MATFE::location
*loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = for_command_;
  int position = map_syntax_element(cfcl);
  cout << "CForCmdList pos:" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(cfcl);
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
};

int CForCommand::map_syntax_element(CASTSyntaxElement *el, int m){
      switch(m_NonTerminal_ProductionCode){
         case     ForCommand__FOR_ForCmdList_END:
                  //FOR_ForCmdList_END
                  if (el->m_NonTerminalCode == token_){
                         return 0;
                        }
                        else if (el->m_NonTerminalCode ==
for_cmd_list_) {
                   return 1;
                      }
                        else if (el->m_NonTerminalCode == token_) {
                   return 2;
                      }
                  break;
               default:
                  cout << endl << "Error in
ForCommand::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}
```

```
//---------------------------------------End of ForCommand------------
-------------------------------------------------------

CForCmdList::CForCmdList(CIdentifier* id, CTOKEN* token,
                   CExpression* exp, CDelimitedInput* deli,
MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = for_cmd_list_;
  int position = map_syntax_element(deli);
  cout << "CDelimitedInput pos:" << position << endl;
  m_Descendants.push_back(id);
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp);
  m_Descendants.push_back(deli);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=4;
}

int CForCmdList::map_syntax_element(CASTSyntaxElement *el, int m){
      cout <<"CForCmdList Stage" << endl;
      cout <<"-------------" << endl;
     cout << "CForCmdList::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "delimited_input_: " << delimited_input_<< endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     ForCmdList__Identifier_ASSIGN_Expr_DelimitedInput:
                  //Identifier_ASSIGN_Expr_DelimitedInput
                  if (el->m_NonTerminalCode == identifier_ ){
                        return 0;
                      }
                  else if (el->m_NonTerminalCode == token_) {
                return 1;
                      }
                  else if (el->m_NonTerminalCode == expr_) {
                return 2;
                      }
                  else if (el->m_NonTerminalCode == delimited_input_)
{
                return 3;
                      }
                  break;
               default:
                  cout << endl << "Error in
CForCmdList::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}



//---------------------------------------End of CForCmdList-----------
--------------------------------------------------------

CIfCommand::CIfCommand(CTOKEN* token,
                   CIfCmdList* cicl, CTOKEN* token1, MATFE::location
*loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
```

```
  m_NonTerminalCode = if_command_;
  int position = map_syntax_element(cicl);
  cout << "CIfCommand pos:" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(cicl);
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

int CIfCommand::map_syntax_element(CASTSyntaxElement *el, int m){
      cout <<"CIfCommand Stage" << endl;
      cout <<"-------------" << endl;
     cout << "CIfCommand::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "if_cmd_list_: " << if_cmd_list_<< endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     IfCommand__IF_IfCmdList_END:
                  //IF_IfCmdList_END
                  if (el->m_NonTerminalCode == token_){
                        return 0; }
                        else if (el->m_NonTerminalCode ==
if_cmd_list_) {
                   return 1;
                      }
                        else if (el->m_NonTerminalCode == token_){
                              return 2;
                      }
                  break;
               default:
                  cout << endl << "Error in
IfCommand::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}


//---------------------------------------End of IfCommand-------------
-------------------------------------------------------

CIfCmdList::CIfCmdList(CExpression* exp, CDelimitedInput* deli,
                  COptElse* cop, MATFE::location *loc,
                  NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = if_cmd_list_;
  int position = map_syntax_element(cop);
  cout << "COptElse pos:" << position << endl;
  m_Descendants.push_back(exp);
  m_Descendants.push_back(deli);
  m_Descendants.push_back(cop);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

int CIfCmdList::map_syntax_element(CASTSyntaxElement *el, int m){
      cout <<"CIfCmdList Stage" << endl;
      cout <<"-------------" << endl;
     cout << "CIfCmdList::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
```

```
         "delimited_input_: " << delimited_input_<< endl <<
         "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
       switch(m_NonTerminal_ProductionCode){
          case     IfCmdList__Expr_DelimitedInput_OptElse:
                   //Expr_DelimitedInput_OptElse
                   if (el->m_NonTerminalCode == expr_ ){
                         return 0;
                       }
                   else if (el->m_NonTerminalCode == delimited_input_)
{
                 return 1;
                       }
                   else if (el->m_NonTerminalCode == opt_else_) {
                 return 2;
                       }
                   break;
                default:
                   cout << endl << "Error in
CIfCmdList::map_syntax_element"
                         << endl << "System Exits !!!";
                         exit(0);
        }
return -1;
}



//----------------------------------------End of CIfCmdList------------
--------------------------------------------------------
COptElse::COptElse(MATFE::location *loc,
           NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = opt_else_;
  m_NumberOfDescendants=0;
  this->graphstring = nonterminals[this->m_NonTerminalCode];
}


COptElse::COptElse(CTOKEN* token, CDelimitedInput* deli,
                  MATFE::location *loc,
                  NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = opt_else_;
  int position = map_syntax_element(deli);
  cout << "CDelimitedInput:" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(deli);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

COptElse::COptElse(CTOKEN* token, CExpression* exp, CDelimitedInput*
deli,
                  COptElse* copt, MATFE::location *loc,
                  NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = opt_else_;
  int position = map_syntax_element(copt);
  cout << "COptElse pos:" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(exp);
  m_Descendants.push_back(deli);
```

```
  m_Descendants.push_back(copt);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=4;
}

int COptElse::map_syntax_element(CASTSyntaxElement *el, int m){
      cout <<"COptElse Stage" << endl;
      cout <<"-----------" << endl;
    cout << "In COptElse::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "delimited_input_: " << delimited_input_<< endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     OptElse__ELSE_DelimitedInput:
                  //ELSE_DelimitedInput
                  if (el->m_NonTerminalCode == token_){
                        return 0;
                      }
                  else if (el->m_NonTerminalCode == delimited_input_)
{
                return 1;
                      }
                  break;
        case      OptElse__ELSEIF_DelimitedInput_OptElse:
                  //ELSEIF_DelimitedInput_OptElse
                  if (el->m_NonTerminalCode == token_ ){
                        return 0;
                      }
                  else if (el->m_NonTerminalCode == delimited_input_)
{
                return 1;
                      }
                  else if (el->m_NonTerminalCode == opt_else_) {
                return 2;
                      }
                  break;
               default:
                  cout << endl << "Error in
COptElse::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}


//-----------------------------------------End of COptElse--------------
-----------------------------------------------------

CGlobalCommand::CGlobalCommand(CTOKEN* token,
                   CGlobalDeclList* gdl, MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = global_command_;
  int position = map_syntax_element(gdl);
  cout << "CGlobalDeclList pos:" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(gdl);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}
```

```
int CGlobalCommand::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "In CGlobalCommand::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "global_decl_list_: " << global_decl_list_<< endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     GlobalCommand__GLOBAL_GlobalDeclList:
                  //GLOBAL_GlobalDeclList
                  if (el->m_NonTerminalCode == token_){
                        return 0;
                      }
                  else if (el->m_NonTerminalCode == global_decl_list_)
{
                   return 1;
                      }
                  break;
               default:
                  cout << endl << "Error in
GlobalCommand::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}

//-----------------------------------------End of GlobalCommand---------
-----------------------------------------------------------

CGlobalDeclList::CGlobalDeclList(CTOKEN * token,
                      MATFE::location *loc,
                      NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = global_decl_list_;
  int position = map_syntax_element(token);
  cout << "CTOKEN pos::" << position << endl;
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CGlobalDeclList::CGlobalDeclList(CGlobalDeclList* gdl, CTOKEN * token,
                      MATFE::location *loc,
                      NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = global_decl_list_;
  int position = map_syntax_element(token);
  cout << "CTOKEN pos::" << position << endl;
  m_Descendants.push_back(gdl);
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}
int CGlobalDeclList::map_syntax_element(CASTSyntaxElement *el, int m){
      cout << "In CGlobalCommand::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "token_: " << token_<< endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     GlobalDeclList__IDENTIFIER:
                  //IDENTIFIER
                  if (el->m_NonTerminalCode == token_ ){
                        return 0;
```

```
                    }
                break;
        case    GlobalDeclList__GlobalDeclList_IDENTIFIER:
                //GlobalDeclList_IDENTIFIER
                if (el->m_NonTerminalCode == global_decl_list_ ){
                    return 0;
                    }
                else if (el->m_NonTerminalCode == token_ ){
                    return 1;
                    }
                break;
            default:
                cout << endl << "Error in
CGlobalDeclList::map_syntax_element"
                    << endl << "System Exits !!!";
                    exit(0);
        }
return -1;
}



//---------------------------------------End of CGlobalDeclLis--------
-----------------------------------------------------------


CWhileCommand::CWhileCommand(CTOKEN* token,
                   CWhileCmdList* wcl, CTOKEN* token1, MATFE::location
*loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = while_command_;
  int position = map_syntax_element(wcl);
  cout << "CWhileCmdList pos::" << position << endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(wcl);
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

int CWhileCommand::map_syntax_element(CASTSyntaxElement *el, int m){
      cout << "In CWhileCommand::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "while_cmd_list_: " << while_cmd_list_<< endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case    WhileCommand__WHILE_WhileCmdList_END:
                //WHILE_WhileCmdList_END
                if (el->m_NonTerminalCode == token_){
                      return 0;
                      }
                      else if (el->m_NonTerminalCode ==
while_cmd_list_) {
                 return 1;
                    }
                      else if (el->m_NonTerminalCode == token_) {
                 return 2;
                    }
                break;
            default:
                cout << endl << "Error in
WhileCommand::map_syntax_element"
```

```
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}


//---------------------------------------End of WhileCommand----------
---------------------------------------------------------

CWhileCmdList::CWhileCmdList(CExpression* exp, CDelimitedInput* deli,
                     MATFE::location *loc,
                     NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = while_cmd_list_;
  int position = map_syntax_element(deli);
  cout << "CDelimitedInput pos::" << position << endl;
  m_Descendants.push_back(exp);
  m_Descendants.push_back(deli);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

int CWhileCmdList::map_syntax_element(CASTSyntaxElement *el, int m){
      cout << "CWhileCmdList::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "delimited_input_: " << delimited_input_<< endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     WhileCmdList__Expr_DelimitedInput:
                  //Expr_DelimitedInput
                  if (el->m_NonTerminalCode == expr_ ){
                        return 0;
                      }
                  else if (el->m_NonTerminalCode == delimited_input_)
{
                return 1;
                      }
                  break;
               default:
                  cout << endl << "Error in
CWhileCmdList::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}



//---------------------------------------End of CWhileCmdList---------
----------------------------------------------------------

CReturnCommand::CReturnCommand(CTOKEN* token,
                    MATFE::location *loc,
                   NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = return_command_;
  int position = map_syntax_element(token);
  cout << "CTOKEN pos::" << position << endl;
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
```

```
  m_NumberOfDescendants=1;
}

int CReturnCommand::map_syntax_element(CASTSyntaxElement *el, int m){
      cout << "CCReturnCommand::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "token_ : " << token_ << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     ReturnCommand__RETURN:
                  //RETURN
                  if (el->m_NonTerminalCode == token_ ){
                        return 0;}
                  break;
               default:
                  cout << endl << "Error in
ReturnCommand::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}

//-----------------------------------------End of ReturnCommand---------
-----------------------------------------------------------

CDelimitedInput::CDelimitedInput(COptDelimiter* opt,
                    MATFE::location *loc,
                    NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = delimited_input_;
  int position = map_syntax_element(opt);
  cout << "COptDelimiter pos::" << position << endl;
  m_Descendants.push_back(opt);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

CDelimitedInput::CDelimitedInput(COptDelimiter* opt,CDelimitedList*
cdl,
                    MATFE::location *loc,
                    NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = delimited_input_;
  int position = map_syntax_element(opt);
  cout << "COptDelimiter pos::" << position << endl;
  m_Descendants.push_back(opt);
  m_Descendants.push_back(cdl);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

int CDelimitedInput::map_syntax_element(CASTSyntaxElement *el, int m){
           cout << "CDelimitedInput::map_syntax_element() -- " << endl
<<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "opt_delimiter_: " << opt_delimiter_ << endl <<
        "delimited_list_: " << delimited_list_<< endl <<
        "el->m_NonTerminalCode:" << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     DelimitedInput__OptDelimiter:
                  //OptDelimiter
```

```
                  if (el->m_NonTerminalCode == opt_delimiter_ ){
                        return 0;}
                  break;
         case DelimitedInput__OptDelimiter_DelimitedList:
                  //OptDelimiter_DelimitedList
                  if (el->m_NonTerminalCode == opt_delimiter_ ){
                        return 0;}
                  else if (el->m_NonTerminalCode == delimited_list_){
                        return 1;}
                  break;
               default:
                  cout << endl << "Error in
DelimitedInput::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}


//---------------------------------------End of DelimitedInput--------
-----------------------------------------------------------

CDelimitedList::CDelimitedList(CStatement* stat,
                    CDelimiter* del, MATFE::location *loc,
                    NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = delimited_list_;
  int position = map_syntax_element(stat);
  cout << "CStatement pos::" << position << endl;
  m_Descendants.push_back(stat);
  m_Descendants.push_back(del);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

CDelimitedList::CDelimitedList(CStatement* stat,
                    CDelimiter* del, CDelimitedList* cdl,
                    MATFE::location *loc,
                    NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = delimited_list_;
  int position = map_syntax_element(stat);
  cout << "CStatement pos::" << position << endl;
  m_Descendants.push_back(stat);
  m_Descendants.push_back(del);
  m_Descendants.push_back(cdl);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

int CDelimitedList::map_syntax_element(CASTSyntaxElement *el, int m){
      cout << "CDelimitedList::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "statement_: " << statement_ << endl <<
         "delimiter_: " << delimiter_<< endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     DelimitedList__Statement_Delimiter:
                  //Statement_Delimiter
                  if (el->m_NonTerminalCode == statement_ ){
                        return 0;}
```

```
                  else if (el->m_NonTerminalCode == delimiter_ ){
                        return 1;}
                  break;
         case DelimitedList__Statement_Delimiter_DelimitedList:
                  //Statement_Delimiter_DelimitedList
                  if (el->m_NonTerminalCode == statement_ ){
                        return 0;}
                  if (el->m_NonTerminalCode == delimiter_ ){
                        return 1;}
                  else if (el->m_NonTerminalCode == delimited_list_){
                        return 2;}
                  break;
               default:
                  cout << endl << "Error in
DelimitedList::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}


//---------------------------------------End of DelimitedList---------
-----------------------------------------------------------
CFunctionMFile::CFunctionMFile(CEmptyLines* eml,
                      CFDefLine* fdline, CFBody* body,
                        MATFE::location *loc,
                      NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = functionMFile_;
  int position = map_syntax_element(eml);
  cout << "CEmptyLines pos:"<< position <<endl;
  m_Descendants.push_back(eml);
  m_Descendants.push_back(fdline);
  m_Descendants.push_back(body);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

CFunctionMFile::CFunctionMFile(CFDefLine* fdline,
                        CFBody* fbody,
                        MATFE::location *loc,
                      NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = functionMFile_;
  m_Descendants.push_back(fdline);
  m_Descendants.push_back(fbody);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}


int CFunctionMFile::map_syntax_element(CASTSyntaxElement *el, int m){
      cout <<"CFunctionMFile" << endl;
    cout <<"-----------" << endl;
    cout << "In FunctionMFile::map_syntax_element() -- " << endl <<
        "NTPC : " << m_NonTerminal_ProductionCode << endl <<
        "functionMFile_ : " << functionMFile_  << endl <<
        "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
      switch(m_NonTerminal_ProductionCode){
         case     FunctionMFile__EmptyLines_FDefLine_FBody:
                  //EmptyLines_FDefLine_FBody
```

```
                  if (el->m_NonTerminalCode == empty_lines_ ){
                        return 0;}
                  else if (el->m_NonTerminalCode == f_def_line_ ){
                        return 1;}
                  else if (el->m_NonTerminalCode == f_body_ ){
                        return 2;}
                  break;
         case     FunctionMFile__FDefLine_FBody:
                  //EmptyLines_FDefLine_FBody
                if (el->m_NonTerminalCode == f_def_line_ ){
                        return 0;}
                  else if (el->m_NonTerminalCode == f_body_ ){
                        return 1;}
                  break;
               default:
                  cout << endl << "Error in
FunctionMFile::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}


//-----------------------------------------End of FunctionMFile---------
----------------------------------------------------------


CFDefLine::CFDefLine(CTOKEN* token, CFOutput* cfo,
                      CTOKEN* token1, CTOKEN* token2,
                        CFInput* cfi, MATFE::location *loc,
                      NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_def_line_;
  int position = map_syntax_element(cfo);
  cout << "CFOutput pos:"<< position <<endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(cfo);
  m_Descendants.push_back(token1);
  m_Descendants.push_back(token2);
  m_Descendants.push_back(cfi);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=5;
}


CFDefLine::CFDefLine(CTOKEN* token, CTOKEN* token1,
                          CFInput* cfi, MATFE::location *loc,
                        NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_def_line_;
  int position = map_syntax_element(cfi);
  cout << "CFInput pos:"<< position <<endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(token1);
  m_Descendants.push_back(cfi);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
}

int CFDefLine::map_syntax_element(CASTSyntaxElement *el, int m){
cout << "CFDefLine::map_syntax_element() -- " << endl <<
"NTPC : " << m_NonTerminal_ProductionCode << endl <<
```

```
"f_output_ : " << f_output_ << endl <<
"el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
switch(m_NonTerminal_ProductionCode){
         case     FDefLine__FUNCTION_FOutput_ASSIGN_IDENTIFIER_FInput:
                  //FUNCTION_FOutput_ASSIGN_IDENTIFIER_FInput
                  if (el->m_NonTerminalCode == token_ ){
                        return 0;
                    }
                  else if (el->m_NonTerminalCode == f_output_ ){
                        return 1;}
                  else if (el->m_NonTerminalCode == token_){
                        return 2;}
                  else if (el->m_NonTerminalCode == token_){
                        return 3;}
                  else if (el->m_NonTerminalCode == f_input_ ){
                        return 4;}
                  break;
              case FDefLine__FUNCTION_IDENTIFIER_FInput:
                  //FUNCTION_IDENTIFIER_FInput
                if (el->m_NonTerminalCode == token_ ){
                        return 0;
                    }
                  else if (el->m_NonTerminalCode == token_){
                        return 1;}
                  else if (el->m_NonTerminalCode == f_input_){
                        return 2;}
                  break;
               default:
                  cout << endl << "Error in
FDefLine::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}



//--------------------------------------End of FDefLine--------------
-----------------------------------------------------
CFOutput::CFOutput(CIdentifier* id,
                            MATFE::location *loc,
                          NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_output_;
  int position = map_syntax_element(id);
  cout << "CFInput pos:"<< position <<endl;
  m_Descendants.push_back(id);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
};

CFOutput::CFOutput(CTOKEN* token,  CFArgumentList* cfa,
                            CTOKEN* token1, MATFE::location *loc,
                          NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_input_;
  m_Descendants.push_back(token);
  m_Descendants.push_back(cfa);
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
```

```
}

int CFOutput::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "In CFOutput::map_syntax_element() -- " << endl <<
    "NTPC : " << m_NonTerminal_ProductionCode << endl <<
    "identifier_ : " << identifier_ << endl <<
    "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
switch(m_NonTerminal_ProductionCode){
         case     FOutput__Identifier:
                  //Identifier
                  if (el->m_NonTerminalCode == identifier_ ){
                           return 0;}
                  break;
              case FOutput__LD_FArgumentList_RD:
                  //LD_FArgumentList_RD
                if (el->m_NonTerminalCode == token_ ){
               return 0;
                    }
                  else if (el->m_NonTerminalCode == f_argument_list_ )
{
                        return 1;
                              }
                  else if (el->m_NonTerminalCode == token_) {
                        return 2;
                              }
                  break;
               default:
                  cout << endl << "Error in
FOutput::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}


//-----------------------------------------End of FOutput---------------
----------------------------------------------------

CFInput::CFInput(MATFE::location *loc,
           NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_input_;
  m_NumberOfDescendants=0;
  this->graphstring = nonterminals[this->m_NonTerminalCode];
}


CFInput::CFInput(CTOKEN* token, CTOKEN* token1,
                            MATFE::location *loc,
                          NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_input_;
  int position = map_syntax_element(token);
  cout << "CTOKEN pos:"<< position <<endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
}

CFInput::CFInput(CTOKEN* token,  CFArgumentList* cfa,
```

```
                          CTOKEN* token1, MATFE::location *loc,
                        NONTERMINAL_PRODUCTION_RULE pr)
 :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_input_;
  int position = map_syntax_element(token);
  cout << "CTOKEN pos:"<< position <<endl;
  m_Descendants.push_back(token);
  m_Descendants.push_back(cfa);
  m_Descendants.push_back(token1);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;
};

int CFInput::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "CFInput::map_syntax_element() -- " << endl <<
    "NTPC : " << m_NonTerminal_ProductionCode << endl <<
    "token_ : " << token_ << endl <<
    "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
switch(m_NonTerminal_ProductionCode){
         case     FInput__LPAREN_RPAREN:
                  //LPAREN_RPAREN
                  if (el->m_NonTerminalCode == token_){
                                  return 0;
                  }
                  else if (el->m_NonTerminalCode == token_){
                                  return 1;
                  }
                  break;
              case FInput__LPAREN_FArgumentList_RPAREN:
                  //LPAREN_FArgumentList_RPAREN
                if (el->m_NonTerminalCode == token_ ){
                        return 0;
                    }
                  else if (el->m_NonTerminalCode == f_argument_list_
){
                        return 1;}
                  else if (el->m_NonTerminalCode == token_ ){
                        return 2;}
                  break;
               default:
                  cout << endl << "Error in
FInput::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}


//---------------------------------------End of FInput----------------
---------------------------------------------------

CFArgumentList::CFArgumentList(CIdentifier* id, CTOKEN* token,
                                          CFArgumentList* cfa,
                                    MATFE::location *loc,
                                  NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_argument_list_;
  int position = map_syntax_element(id);
  cout << "CIdentifier pos:"<< position <<endl;
  m_Descendants.push_back(id);
  m_Descendants.push_back(token);
```

```
  m_Descendants.push_back(cfa);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=3;

}

CFArgumentList::CFArgumentList(CIdentifier* id,
                                      MATFE::location *loc,
                                    NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_argument_list_;
  int position = map_syntax_element(id);
  cout << "CIdentifier pos:"<< position <<endl;
  m_Descendants.push_back(id);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;


}

int CFArgumentList::map_syntax_element(CASTSyntaxElement *el, int m){
      cout << "CFArgumentList::map_syntax_element() -- " << endl <<
    "NTPC : " << m_NonTerminal_ProductionCode << endl <<
    "identifier_ : " << identifier_ << endl <<
    "el->m_NonTerminalCode : " << el->m_NonTerminalCode << endl;
    switch(m_NonTerminal_ProductionCode){
         case     FArgumentList__Identifier_COMMA_FArgumentList:
                  //Identifier_COMMA_FArgumentList
                  if (el->m_NonTerminalCode == identifier_ ){
                                    return 0;}
                  else if (el->m_NonTerminalCode == token_){
                                    return 1;}
                  else if (el->m_NonTerminalCode == f_argument_list_
){
                                    return 2;}
            break;
          case FArgumentList__Identifier:
                  //Identifier
                  if (el->m_NonTerminalCode == identifier_ ){
                        return 0;}
                  break;
               default:
                  cout << endl << "Error in
FArgumentList::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}



//---------------------------------------End of ArgumentList----------
---------------------------------------------------------

CFBody::CFBody(CDelimiter* del, CStatementList* stl,
                                      MATFE::location *loc,
                                    NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_body_;
  int position = map_syntax_element(del);
  cout << "CDelimiter Pos:"<< position <<endl;
```

```
  m_Descendants.push_back(del);
  m_Descendants.push_back(stl);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=2;
};

CFBody::CFBody(COptDelimiter* odl,
                      MATFE::location *loc,
                     NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = f_body_;
  int position = map_syntax_element(odl);
  cout << "COptDelimiter pos:"<< position <<endl;
  m_Descendants.push_back(odl);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

int CFBody::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "CFBody::map_syntax_element() -- " << endl;
    cout << "NTPC: " << m_NonTerminal_ProductionCode << endl;
      cout <<"opt_delimiter_ :" << opt_delimiter_ << endl;
    cout <<  "el->m_NonTerminalCode:" << el->m_NonTerminalCode <<
endl;
    switch(m_NonTerminal_ProductionCode){
         case     FBody__Delimiter_StatementList:
                  //Delimiter_StatementList
                  if (el->m_NonTerminalCode == delimiter_ ){
                                  return 0;}
                  else if (el->m_NonTerminalCode == statement_list_ ){
                                  return 1;}
            break;
          case FBody__OptDelimiter:
                  //OptDelimiter
                  if (el->m_NonTerminalCode == opt_delimiter_ ){
                        return 0;}
                  break;
               default:
                  cout << endl << "Error in FBody::map_syntax_element"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}


//-----------------------------------------End of FBody------------------
--------------------------------------------------


CParseError::CParseError(CTOKEN* token,
                                  MATFE::location *loc,
                                NONTERMINAL_PRODUCTION_RULE pr)
:CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = parse_error_;
  m_Descendants.push_back(token);
  this->graphstring = nonterminals[this->m_NonTerminalCode];
  m_NumberOfDescendants=1;
}

 CParseError::CParseError(MATFE::location *
loc,NONTERMINAL_PRODUCTION_RULE pr)
```

```
 :CASTSyntaxElement(pr,loc){
  m_NonTerminalCode = parse_error_;
  }


int CParseError::map_syntax_element(CASTSyntaxElement *el, int m){
    cout << "CParseError::map_syntax_element() -- " << endl;
    cout << "NTPC : " << m_NonTerminal_ProductionCode << endl;
      cout <<"scriptMFile_ :" << scriptMFile_ << endl;
    cout <<  "el->m_NonTerminalCode : " << el->m_NonTerminalCode <<
endl;
switch(m_NonTerminal_ProductionCode){
         case     ParseError__LEXERROR:
                  //LEXERROR
                  if (el->m_NonTerminalCode == token_){
                                  return 0;}
            break;
               default:
                  cout << endl << "Lexical Error!"
                        << endl << "System Exits !!!";
                        exit(0);
        }
return -1;
}



//-----------------------------------------End of ParseError------------
-------------------------------------------------------

//-----------------------------------------END OF ASTSYNTAXELEMENTS-----
-------------------------------------------------------
```

File Driver.h

```
#ifndef DRIVER_H
#define DRIVER_H

#include <string>
#include "MATFE.tab.h"
#include "ASTDefines.h"
#include "ASTSyntaxElements.h"
#include <iostream>
#include <fstream>


extern FILE* yyin;
extern  char* nonterminals[];
extern  char* graphstring;

YY_DECL;

class MATFE_driver
{
  private:
    void ASTGraphEmmiter(CASTSyntaxElement *parent, ofstream* out);

  public:
      MATFE_driver();
      virtual ~MATFE_driver();
      int result;
      std::string file;
```

```
      bool debug_mode;
    bool trace_scanning;
    bool trace_parsing;

      // Handling the scanner.
      void scan_begin();
      void scan_end();


      // Run the parser. Return 0 on success.
      int parse (const std::string& f);

      // Error handling.
      void error (const MATFE::location& l, const std::string& m);
      void error (const std::string& m);

      // debug
      void printInfo(const char* msg);
      void set_debug_mode(bool flag);

      string to_string();
    void ASTGraphEmmiter(CASTSyntaxElement *parent);

};
#endif // DRIVER_H

--------------------------------------------------------------------
```

File driver.cpp

```
#include "driver.h"
#include <cstdio>
#include <iostream>
#include <fstream>

using namespace std;


MATFE_driver::MATFE_driver()
: trace_scanning(false), trace_parsing(false){

}
MATFE_driver::~MATFE_driver (){
      this->debug_mode = false;
}
int MATFE_driver::parse (const std::string &f)
{
      file=f;
      scan_begin();
      printInfo("Creating parser object...");
      MATFE::MATParserClass parser(*this);
      parser.set_debug_level(trace_parsing);
      printInfo("Created.");
      printInfo("Initiating parsing...");
      int res = parser.parse();
      scan_end();
      ASTGraphEmmiter(CTransUnit::instance);
      return res;
}
void MATFE_driver::error (const MATFE::location& l, const std::string&
m){
```

```
      std::cerr <<"Driver error: "<< l << ": " << m << std::endl;
}

void MATFE_driver::scan_begin (){

      yyin = fopen(file.c_str(),"r");
}

void MATFE_driver::scan_end (){
    fclose(yyin);
}
void MATFE_driver::error (const std::string& m){
      std::cerr << m << std::endl;
}

void MATFE_driver::printInfo(const char* msg) {
    if (!this->debug_mode) return;
    cout << msg << endl;
}

void MATFE_driver::set_debug_mode(bool flag) {
    this->debug_mode = flag;
}

string MATFE_driver::to_string() {
    string s("");
    s = s + file;
      return s;
}



void MATFE_driver::ASTGraphEmmiter(CASTSyntaxElement *parent,
ofstream* out){
    for (unsigned int i=0; i < (parent->m_Descendants.size()); i++)
    {

        (*out)  << parent->graphstring << parent->m_ObjectSerialNumber
<< ' ';
        (*out) << "-> ";
            (*out)  << parent->m_Descendants[i]->graphstring  <<
parent->m_Descendants[i]->m_ObjectSerialNumber<<";" << endl;

            ASTGraphEmmiter(parent->m_Descendants[i], out);
    }
}

void MATFE_driver::ASTGraphEmmiter(CASTSyntaxElement *parent){
      ofstream graphout;
      graphout.open("mygraph.grv");
      graphout << "digraph G {" << endl;
      ASTGraphEmmiter(parent, &graphout);
      graphout << "}" << endl;
      graphout.close();
}
----------------------------------------------------------------
File main.cpp

#include <iostream>
#include "driver.h"
#include "MATFE.tab.h"
```

```
using namespace std;

int main (int argc, char *argv[]){
    MATFE_driver driver;
    driver.set_debug_mode(true);
    char** argv_init = argv;
    cout << "Happy scanning..." << endl;
    for (int i = 0; i < argc; i++)
        argv = argv_init+i;
        if (*argv == std::string ("-p")) {
            driver.trace_parsing = false;
        }
        else if (*argv == std::string ("-s")) {
            driver.trace_scanning = false;
        }
        else {
            string filename = string(*argv);
            cout << "Scanning file: "<< filename << endl;
                  if (!driver.parse(filename)) {
             std::cout << "OK. End of file" << std::endl;
            }
        }
    cout << "It was a happy scanning!" << endl;
    return 0;
}
```

References

[1] "Context-free Grammar for Natural Language constructs", Balla Sundura Raman L, Ishwar S, Sanjeeth Kumar Ravindranath, Indian nstitute f Information technology "Innovator" Towers, ITPL, 2003

[2] "The Design and Implementation of a Parser and Scanner for the MATLAB Language in the MATCH Compiler" Pramod Joisha, Abhay Kanhere, Prithviraj Banerjee, U.Nagaraj Shenoy, Alok Choudhary Northwestern University September 1999.

[3]"Lexical Analysis" Handout written by Maggie Johnson and Julie Zelenski, June 22, 2011

[4]"MATCH: A MATLAB Compiler for Configurable Computing Systems" P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. BachmannM. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare A. Nayak, S. Periyacheri, M. Walkden Electrical and Computer Engineering Northwestern University.

[5]"Compiler Construction using Flex and Bison" Anthony A. Aaby, Walla-Walla College, February 2004

[6] "A MATLAB Compilation Environment for Adaptive Computing Systems" P.Banerjee, Northwestern University Center for Parallel and Distributed Computing

[7] "Flex & Bison" John Levine, O'Reilly 2009

[8] "Modern Compiler Implementation in C" Andrew W. Appel, Princeton University with Jens Palsberg, Purdue University

[9] "Experiences in building C++ Front End", Fuqing Yang, Hong Mei, Wanghong Yuan, Qionq Wu, Yao Guo Department of Computer Science and Technology, Peking University

[10] "A Term Pattern-Match Compiler Inspired by Finite Automata Theory" Mikael Pettersson, Department of Computer Science, Linköping University, Sweden

[11] "An Executable Intermediate Representation for Retargetable Compilation and High-Level Code Optimization" Rainer Leupers, Oliver Wahlen, Manuel Hohenauer, Tim Koge Aachen University of Technology (RWTH) Integrated Signal Processing Systems - Peter Marwede University of Dortmun Dept. of Computer Science 12

[12] GNU/LINUX Application Programming Tim Jones 2nd edition

[13] "Compiling Pattern Matching to Good Decision Trees" Luc Maranget INRIA — France

[14] "Serializing C Intermediate Representations to Promote Efficiency and Portability" Jefrey A. Meister, Department of Computer Science, University of Maryland, College Park, MD 20742, USA May 13, 2008

[15] "Converting M-Files to Stand-Alone applications" Rafic Bachnac, Engineering Technology Texas A&M University-CC and Roger Lee Avionics Naval Air Systems Command

[16] “A Modular MATLAB Compilation Infrastructure Targeting Embedded Systems” Ricardo Nobre, Faculdado de Engenharia de Universidade do Porto (FEUP), Departemento de Engenharia Informatica

[17] “A Pattern Matching Compiler for Multiple Target Languages”, Pierre-Etienne Moreau, Christophe Ringeissen LORIA-INRIA, -Nancy and Marian Vittek Institut of Informatica Mlynska Bratislava

[18] “An Overview of a compiler for mapping MATLAB program onto FPGA’s “P.Banerjee Department of electrical and Computer engineering Northwestern University