Hybrid Language Compiler Development: Inline IA-32 Assembler Language Structures As a Part of C language

Nail Sharipov
Moscow State Technical University named after N.E. Bauman, Moscow, Russia
sharipovn@gmail.com

Abstract. Trial version of C compiler was made for the exploration of system software developing benefits (particularly in operating system developing), given by the high-level C language.

1. INTRODUCTION

During distributed operating system basics research and development by means of assembler (IA-32 platform) the following problems were encountered:

- necessity of manual memory allocation (for example, for functions' local variables)
- absence of type match control and vague implementation of type in general
- function call notation absence significantly exaggerate the source code with necessity to take care about non-automotive variable passing while invoking a particular function

Using foreign high-level language compilers (for example, GNU C Compiler) with inline assembler leads to the following problems:

- some microprocessor instructions are not implemented yet or avoided
- the ways of using C variables inside of assembler block is illegible
- assembler conventions fairly distinctive versus compiler developers/distributors
- compiler binding to particular formats of executable file made it complicated to get appropriate binary code for the goals of distributed operating system research

As a result of necessity to develop sophisticated system software and for an operating system architecture research, the **necessity** of hybrid language design (C with inline Assembler) and compiler development has been appeared. Development **topicality** becomes apparent in following:

- distributed operating system development requires full control of code generation, thus, it is rather crucial to make own means of development, such as, C compiler with inline Assembler
- Studying the high- and low-level languages syntactical models confluence and its influence on code generation gives sharp notion about UNIX-like systems as handy programming environment – translator of high level languages into machine instructions
- C language using for the further research is considerably increase the rate of an operating system development [2]

Project **novelty** consists of the following:

- practical implementation of theoretical knowledge in discrete mathematics
- code generation research in distributed operating system development perspective innovates traditional approaches in language translation

Goals of this research are:

- literature exploration and hybrid language mathematical model design
- trial compiler development which is approximate to ANSI C standard with inline Assembler implementation
- define all places in mathematical model and implementation where the hybrid nature of compiler appears
- porting previous assembler-version programs to the new language
- appreciation of further technical and economical ways in research

Desirable use case model of the research stage includes (Fig. 1):

- ability to specify the name of the source code file with .c extension
- ability to specify an absolute address of the code segment of the program

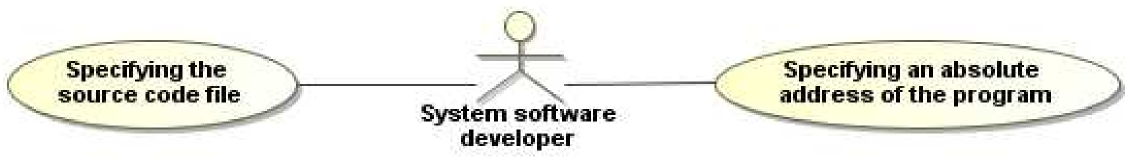

Specifying the source code file
System software developer
Specifying an absolute address of the program

Figure 1. Use-case model of the compiler includes ability to specify the name of the source code file with .c extension and ability to specify an absolute address of the code segment of the program

2. MATERIALS AND METHODS

2.1. Use case implementation. To achieve the desirable use case model (Fig. 1) the console command line method of interaction was chosen and the following command line format was designed (Fig. 2). It consists of a name of the compiler "cc.exe", proper program name extended with ".c" and optional absolute offset (address) of code segment in memory.

```
cc.exe program_name.c [offset]
```

Figure 2. Command line format consists of a name of the compiler "cc.exe", proper program name extended with ".c" and optional absolute offset (address) of code segment in memory

For example, "`cc.exe loader.c 7C00`"command will be treated as follow:

- source code file name: `loader.c`
- initial offset of code segment: `7C00`

If file name is not specified, the input line "`cc.exe test.c 7C00`" will be accepted as default.

2.2. Technical implementation of the hybrid compiler. Phases of compilation. The process of the compiler development splits on several tasks of analysis, which is appropriate to call "phases of compilation". Major phases, which present practically in all compilers, are:

- lexical analysis
- syntactical analysis
- semantic analysis
- code generation

According to the goals of this research the main goal of the implementation is to distinguish confines between the syntax of C language and Assembler language on every phase of compilation. To illustrate the models and implementation the follow example of source code will be used (Fig. 3)

```
short a;
short b;

a = b + 0x0001;
__asm
{
   MOV AX, a
}
```

Figure 3. Example of source code to illustrate the phases of compilation

2.2.1 Lexical analysis. The standard incoming data for this phase is a source code. Typically it is an array of symbols – a part of some text data, which does not have any sense. The task of lexical analysis is to separate out special units, which was called "lexeme". Practically, they may be presented by key words, variable identifiers, constant values, immediate value. The following activity diagram shows general algorithm of lexical analysis applied in the compiler (Fig. 4)

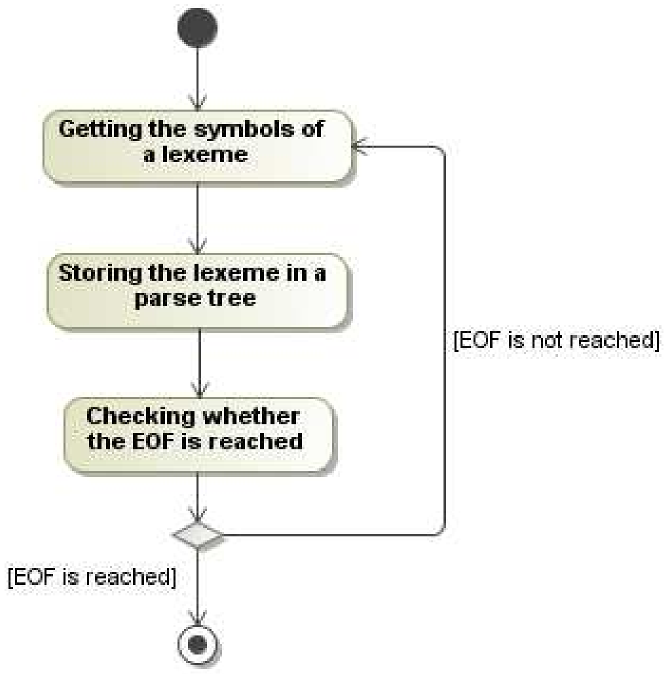

Getting the symbols of a lexeme
Storing the lexeme in a parse tree
[EOF is not reached]
Checking whether the EOF is reached
[EOF is reached]

Figure 4. Activity diagram of general lexical analysis algorithm

The result of such algorithm is the Lexeme List. More detailed decomposition is shown in activity diagram in Figure 5.

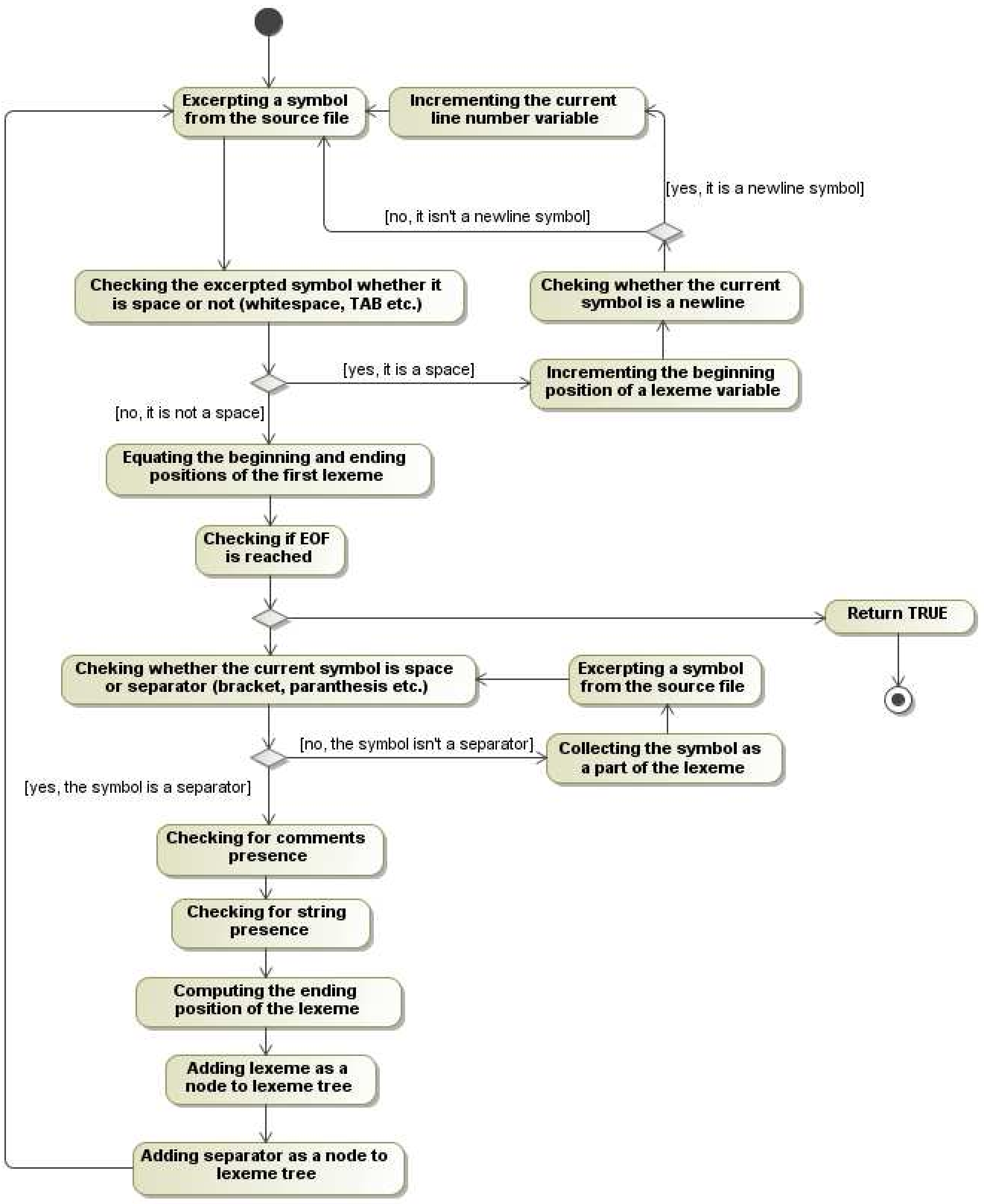

Excerpting a symbol from the source file
Incrementing the current line number variable
[yes, it is a newline symbol]
[no, it isn't a newline symbol]
Checking the excerpted symbol whether it is space or not (whitespace, TAB etc.)
Cheking whether the current symbol is a newline
[yes, it is a space]
Incrementing the beginning position of a lexeme variable
[no, it is not a space]
Equating the beginning and ending positions of the first lexeme
Checking if EOF is reached
Return TRUE
Cheking whether the current symbol is space or separator (bracket, paranthesis etc.)
Excerpting a symbol from the source file
[no, the symbol isn't a separator]
Collecting the symbol as a part of the lexeme
[yes, the symbol is a separator]
Checking for comments presence
Checking for string presence
Computing the ending position of the lexeme
Adding lexeme as a node to lexeme tree
Adding separator as a node to lexeme tree

Figure 5. Next level of decomposition of Lexical Analysis algorithm

Figure 6 shows a structure of a Lexeme List node (TLexemeListItem) which was applied in the compiler.

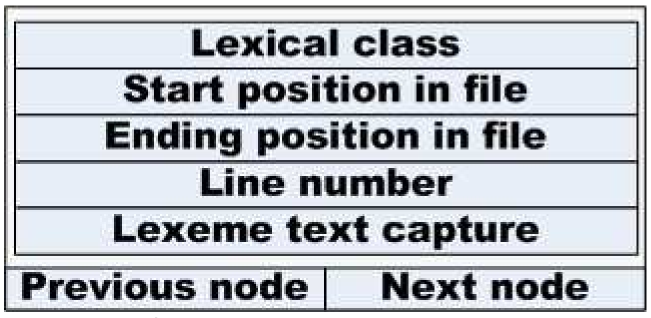


Figure 6. A structure of a Lexeme List node which was applied in the compiler.

Figure 7 shows the implementation of the structure of the Lexeme List Node, which was designed with two structures involved: TLexeme and TLexemeListItem. (Please, refer to Appendix B, Figure 1-10 for full source code of the compiler.)

```
typedef struct tagLexeme {
    UINT    uLexClass;
    UINT    uPosBeg;
    UINT    uPosEnd;
    UINT    uLineNum;
    char      *cpTextFrag;
} TLexeme, *TLexemePtr;

typedef struct tagLexemeListItem {
    struct tagLexemeListItem *pPrevLexemeInstance;
    struct tagLexemeListItem *pNextLexemeInstance;
    TLexeme                   LexemeInstance;
} TLexemeListItem, *TLexemeListItemPtr;
```

Figure 7. The implementation of the structure of the Lexeme List Node: two structures TLexeme and TLexemeListItem

For the most lexeme types there will be enough to use just a numeric constant to distinguish them. For the example above (Fig. 3) the following Lexeme List is created (Fig. 8)

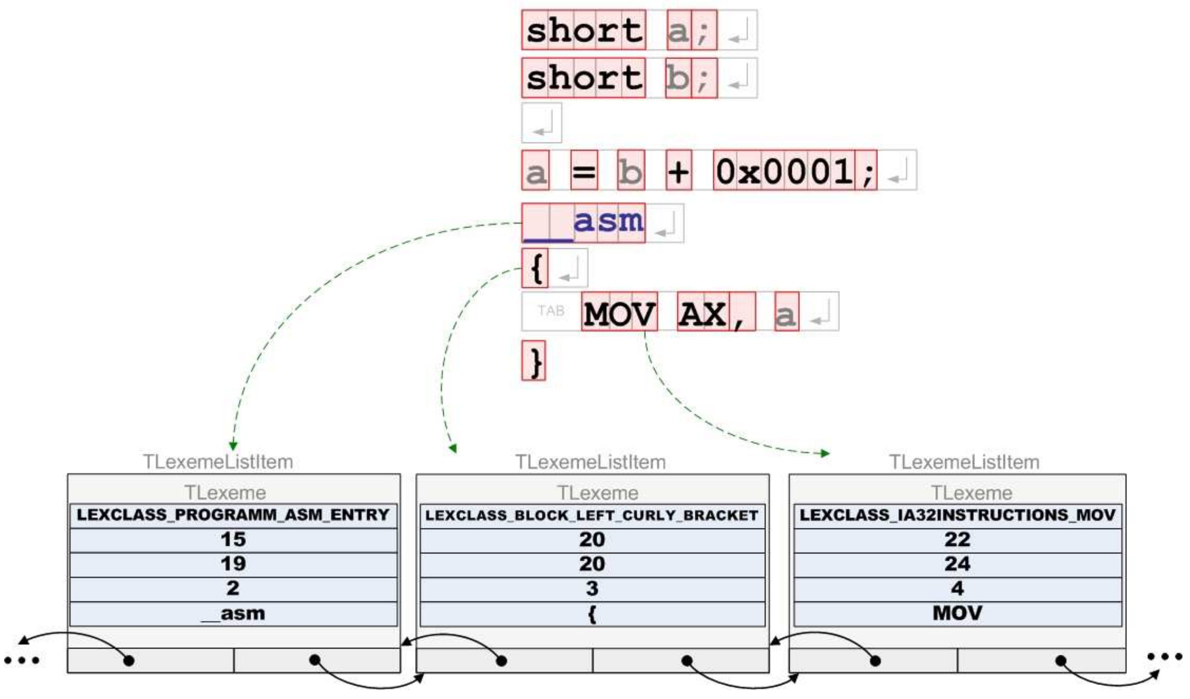


Figure 8. Example of Lexeme List Generation

For other types it was necessary to create additional structures with lexeme data. For example, identifiers had to be described with the Representation Table for future needs of identifier match analysis.

The influence of hybrid nature of language is slightly appeared. It is only the additional set of Lexeme Classes (refer to the Appendix A, Table 1).

Figure 9 shows a structure of a Representation Table node which was applied in the compiler.

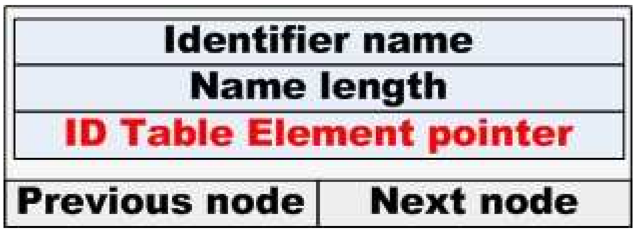


Figure 9. A structure of a Representation Table node which was applied in the compiler.

Figure 10 shows the implementation of the structure of the Representation Table Node, which was designed with two structures involved: TRTItemContent and TRTItem.

```
typedef struct tagRTItemContent {
    char          *cpIdName;
    UINT          uNameLength;
    TIdTablePtr   pIdTableElem;
} TRTItemContent, *TRTItemContentPtr;

typedef struct tagRTItem {
    TRTItemContent    RTItemContent;
    struct tagRTItem  *pPrevRTItem;
    struct tagRTItem  *pNextRTItem;
} TRTItem, *TRTItemPtr;
```

Figure 10. The implementation of the structure of the Representation Table Node: two structures TRTItemContent and TRTItem

Each unique identifier, that is, has a unique name through the source code, has only one entry in the Representation Table. ID Table Element pointer is a special field for future needs of identifier match analysis. Figure 11 illustrates the generation of the Representation Table during lexical analysis.

The results of Lexical Analysis phase, that is, the Lexeme List and the Representation Table are passed to the next phase, syntactical analysis.

2.2.2 Syntactical analysis. Incoming data for this phase is the Lexeme List and Representation Table, that is, the direct result of lexical analysis. The major steps involved with the syntactical analysis are:

- checking the sequence of lexemes in the Lexeme List according to special rules, which form the language syntax. It is essential to have syntax well-designed on this step
- generating the Parse Tree – special intermediate state of program representation, which is not exactly the target program, but which contain the important information about structure of future program, operation sequences and important relations (for example, the local variable and it’s future place in memory)
- the identifier match analysis take place on this stage

The following activity diagram (Fig. 12) shows general algorithm of syntactical analysis applied in the compiler. This algorithm was designed only for the first two major steps of syntactical analysis that is, checking the syntax and generating the Parse Tree. Identifier match analysis is more specific and distinctive, thus required separate sub-step described below involving the Representation Tree and Identifier Tree analysis.

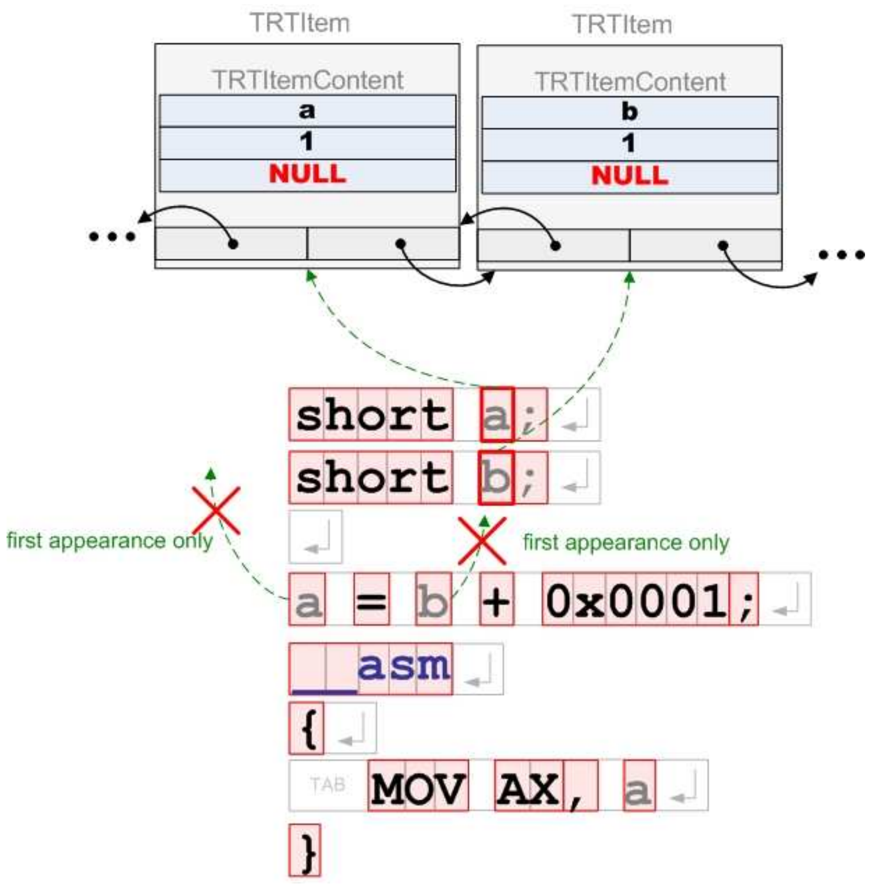


Figure 11. Example of Representation Table Generation

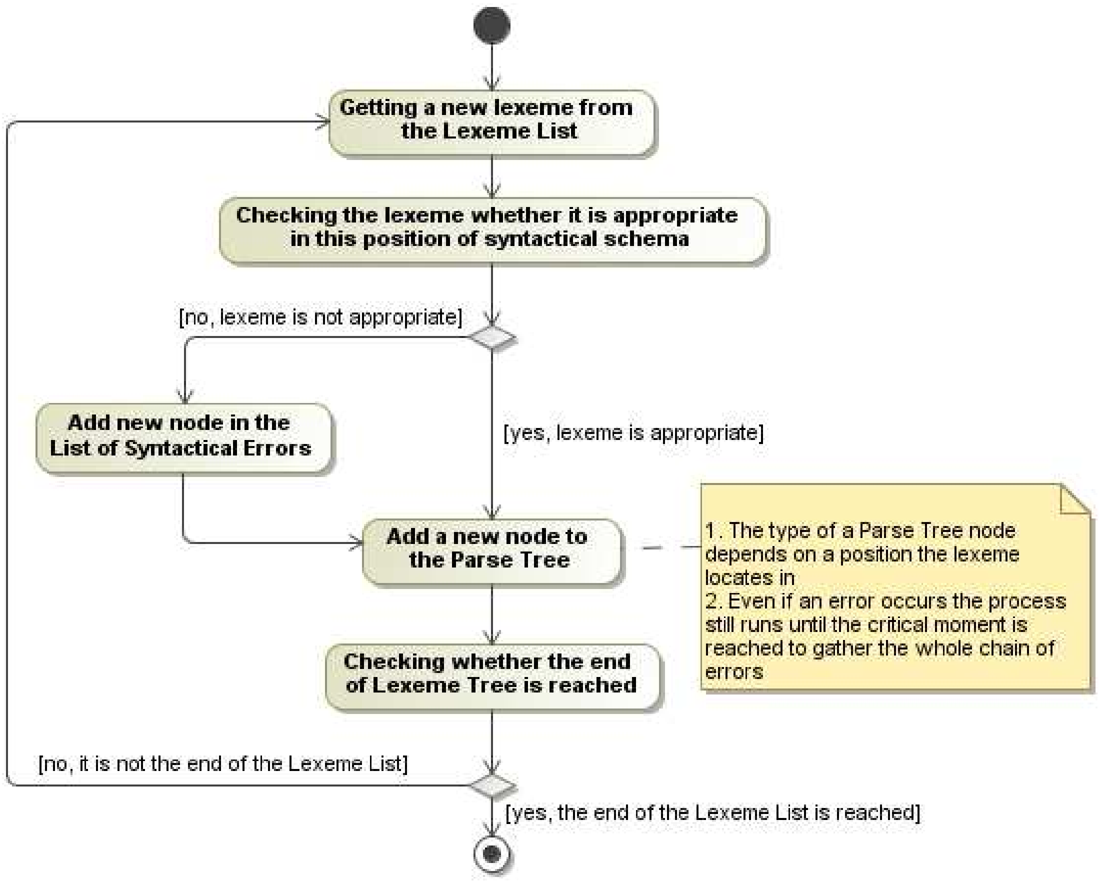

Getting a new lexeme from the Lexeme List
Checking the lexeme whether it is appropriate in this position of syntactical schema
[no, lexeme is not appropriate]
Add new node in the List of Syntactical Errors
[yes, lexeme is appropriate]
Add a new node to the Parse Tree
1. The type of a Parse Tree node depends on a position the lexeme locates in
2. Even if an error occurs the process still runs until the critical moment is reached to gather the whole chain of errors
Checking whether the end of Lexeme Tree is reached
[no, it is not the end of the Lexeme List]
[yes, the end of the Lexeme List is reached]

Figure 12. Activity diagram of general lexical analysis algorithm

One of the most popular syntax specification tool is the Extended Backus-Naur Form [7]. According to the [6] ISO/IEC 14977:1996(E) EBNF (Extended Backus-Naur Form), there is several rules of syntax description:

- Non-terminals are represented by simple words (there are no limits to description language)
- Terminals are represented by words in quotes. For example, “BEGIN”
- Vertical bar (|) – alternative representation
- Round brackets – grouping
- Square brackets – probable entry identification
- Curly brackets – probable repeating of symbol or group of symbols
- Equation is represented by “=” symbol
- Rule elements are separated by comma
- Rules are separated by semicolon

According to these rules, the syntax of trial compiler following (Table 1)

Table 1. The syntax of hybrid compiler (C language with inline IA-32 Assembler)

```
(* see lexIsHex() *)
              hex_digit =  “0” | “1” | “2” | “3” | “4” | “5” |
                           “6” | “7” | “8” | “9”| “A” | “B” |
                           “C” | “D” | “E” | “F”;

(* see lexIsOccurrenceHex() *)
           hex_constant =  “0x”, hex_digit, [{hex_digit}];

(* see synAnalysis() *)
       program_sentence =  function, {function};

               function =  function_declaration, block;

(* see synCheckNextNontermFuncDecl() *)
   function_declaration =  type_spec, identifier, “(”,
                           [func_arg_list], “)”;

(* see synCheckNextNontermType() *)
              type_spec =  “char” | “short”;

          func_arg_list =  type_spec, identifier_decl,
                           [{“,”, type_spec,
                           identifier_decl }];

(* see synCheckVarDeclaration() *)
        identifier_decl =  [{*}],
                           ((“(”,identifier_decl, “)”) |
                           identifier),
                           [{“[”, [hex_constant], “]”}];

(* see synCheckNextNontermBlock() *)
                  block =  “{”, [variable_decl],
                           [{implementation}], “}”;

(* see synCheckNextNontermVarDecl() *)
```

```
             variable_decl =  type_spec, identifier_decl,
                              [{“,” , identifier_decl}], “;”,
                              [{type_spec, identifier_decl,
                              [{“,” , identifier_decl}],
                              “;”}];

(* see synCheckNextNontermImplementation() *)
            implementation =  block | asm_inline |
                              return_point | construction_if |
                              construction_while | expression;

                asm_inline =  “__asm”, asm_block;

(* see synCheckNontermAsmBlock() *)
                 asm_block =  “{”, [{IA32instruction_MOV |
                              IA32instruction_INT |
                              IA32instruction_JMP}], “}”;

       IA32instruction_MOV =  “MOV”, (segment_register,
                              SREG_operand_group)
                              | ([segment_register, “:”],
                              (effective_memory_address,
                              EA_operand_group)
                              | (byte_register | word_register
                              | double_word_register), “,”,
                              register_operand_group);

(* see synCheckNextNontermSegmentRegister() *)
          segment_register =  “CS” | “DS” | “ES” | “SS” | “GS”
                              | “FS”;

(* see synAsmOperandGroup_SREG() *)
        SREG_operand_group =  “,”, (word_register,
                              identifier);

(* see synAddMemoryLocationOEGroup() *)
  effective_memory_address =  “[”, hex_constant
                              | (word_pointer_register, [“+”,
                              hex_constant])
                              | (word_base_register, [“+”,
                              word_pointer_register], [“+”,
                              hex_constant]);

(* see synCheckNextNontermWordPointerRegister () *)
     word_pointer_register =  “SI”, “DI”;

(* see synCheckNextNontermWordBaseRegister () *)
        word_base_register =  “BX”, “BP”;

          EA_operand_group =  “]”, “,”, hex_constant
                              | (byte_register | word_register
                              | double_word_register);
```

```
(* see synAsmOperandGroup_REGXX () *)
   register_operand_group =  (byte_register | word_register |
                             double_word_register)
                             | hex_constant
                             | identifier
                             | ([segment_register, “:”],
                             effective_memory_address, “]”)
                             | segment_register;

      IA32instruction_INT =  “INT”, hex_constant;

      IA32instruction_JMP =  “JMP”, hex_constant,
                             “:”,hex_constant;

(* see synCheckNextNontermImplementation() *)
             return_point =  “return”, expression, “;”;

          construction_if =  “if”, “(”, expression, “)”,
                             block, [“else”, block];

       construction_while =  “while”, “(”, expression, “)”,
                             block;

               expression =  expression, “=”,expression;

               expression =  expression, “+”, expression

               expression =  expression, “+”, expression

               expression =  hex_constant
                             | ({*},identifier,
                             [ (“[”, expression, “]”)
                             | (“(”, arguments_enumeration,
                             “)”)])
                             | (“(”, expression, “)”);
```

Hybrid nature of the compiler appeared in an additional non-terminal symbol “asm_inline”, which includes the implementation of several IA-32 Assembler basic instructions. Also the single variable environment concepts were included in the syntax model, that is, the ability to use variables defined and declared outside the assembler block within the instructions.

The other way to illustrate syntax is to use the syntactical diagram. According to [7] non-terminal symbols should be presented in the rectangles and terminal symbols in rounded-corner figures. The following figure (Fig. 13) shows an example of syntactical diagram of non-terminal symbol “segment_register”.

For the example above (Fig. 3) the process of syntax checking, that is, checking the sequence of lexemes in the Lexeme List according to special rules is illustrated on the figure 14.

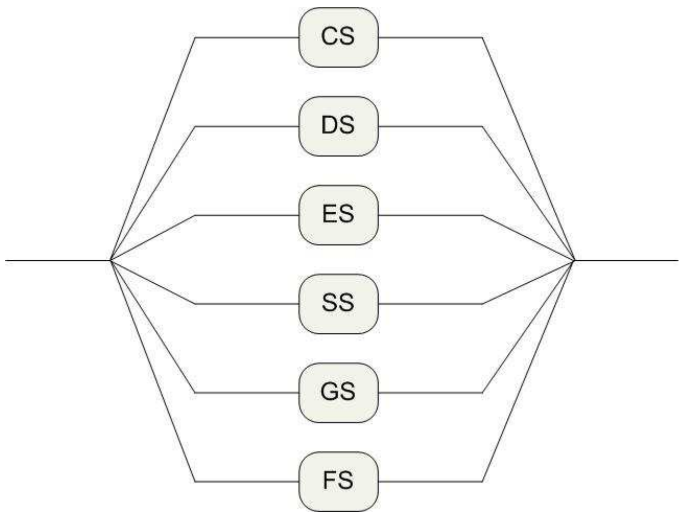


Figure 13. Syntactical diagram of non-terminal symbol “`segment_register`”.

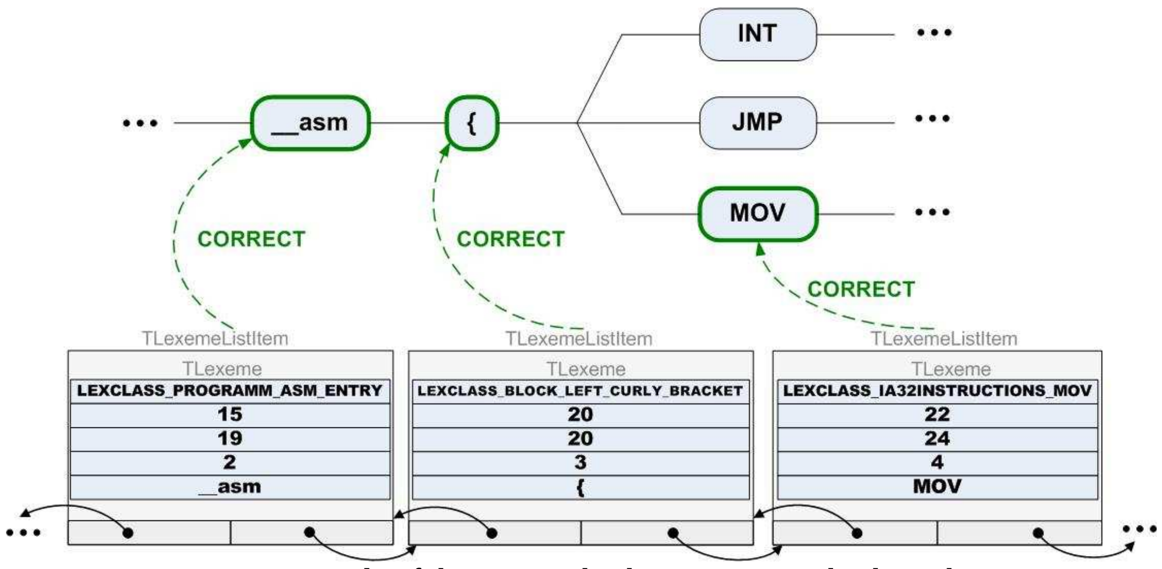


Figure 14. An example of the syntax checking process - checking the sequence of lexemes in the Lexeme List according to special rules

The next major process, which appears simultaneously with the syntax checking, is generating the Parse Tree. The Parse Tree is a list of elements, which are the intermediate representation of target program. There are four main node types of the Parse Tree, which were developed to represent the structures of hybrid language:

1. **Function Node** – a node of the Parse Tree, which represents functions. Due to the structural block nature of C language this node type was designed as a basic node, that is, other nodes will be the child-nodes of the Function Node. The function as an entity is identified by some identifier, so the function identifier is also the matter of identifier match analysis and the Function Node should refer to the Identifier Table. Figure 15 shows a structure of a Function Node which was applied in the compiler. Figure 16 shows the implementation of the structure of the Function Node, which was designed with two type involved: TFuncNode and TParseListNode.

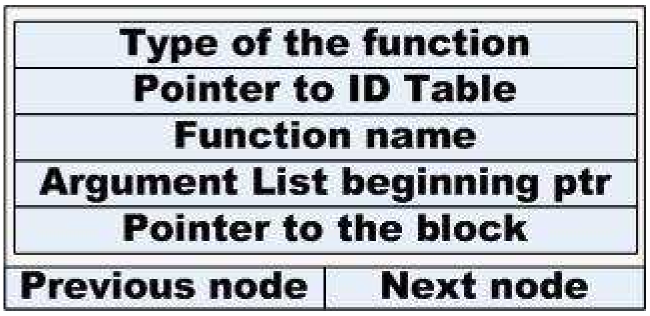


Figure 15. A structure of a Function Node, which was applied in the compiler

```
typedef struct tagFuncNode {
   UINT                 uType;
   TIdTablePtr          pFuncId;
   char                 *cpFuncName;
   TOperationElemPtr    pBegArgList;
   TBlockPtr            pFuncBlock;
} TFuncNode, *TFuncNodePtr;

typedef struct tagParseListNode {
   TFuncNode                 FuncNode;
   struct tagParseListNode  *pNextFuncNode;
   struct tagParseListNode  *pPrevFuncNode;
} TParseListNode, *TParseListNodePtr;
```

Figure 16. The implementation of the Function Node structure: two types TFuncNode and TParseListNode

2. **Block Node** – a node of the Parse Tree, which represents blocks, that is, code inside of the curly brackets. The Block Node was designed also to contain other blocks inside providing the whole functionality to implement local/global context of the program. For example, compiler was developed to detect local and global variables, to differ global and local variables with the same names on perpetual levels of nesting. Figure 17 shows a structure of a Block Node which was applied in the compiler. Figure 18 shows the implementation of the structure of the Block Node, which was designed with one type involved: TBlock

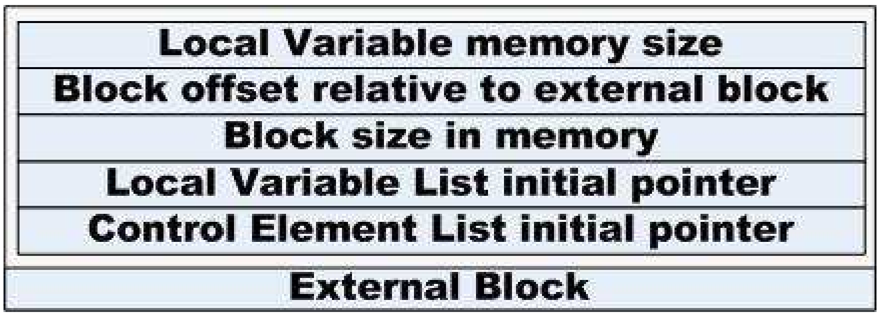

Local Variable memory size
Block offset relative to external block
Block size in memory
Local Variable List initial pointer
Control Element List initial pointer
External Block

Figure 17. A structure of a Block Node, which was applied in the compiler

```
typedef struct tagBlock{
    UINT                uLocVarSize;
    UINT                uBlockOffset;
    UINT                uBlockSize;
    TOperationElemPtr   pBegLocVarList;
    TContrElemPtr       pBegContrList;
    struct tagBlock     *pExternalBlock;
} TBlock, *TBlockPtr;
```

Figure 18. The implementation of the Block Node structure: type TBlock

3. **Control Element Node** – a node of the Parse Tree, which represents different type of actions, that is, the context of some operations. For example mathematical operations and assembler instructions are the operations with the some actions behind. This type of Parse Tree node was designed to represent the action itself, that is, what kind of data manipulation should be done. Due to the data-manipulating nature of the node the structure was designed to include the special pointer to a function which arranges data of Control Element to the proper instructions, depending on the type of Control Element, or, as it was called, a Semantic Class (refer to the Appendix A, Table 2). Figure 19 shows a structure of a Control Element Node which was applied in the compiler. Figure 20 shows the implementation of the structure of the Control Element Node, which was designed with two types involved: TContrElemHandlerPtr and TContrElem

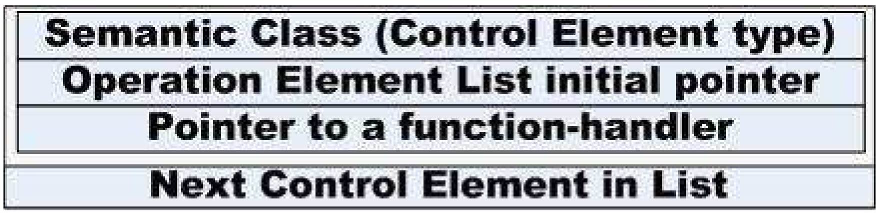

Semantic Class (Control Element type)
Operation Element List initial pointer
Pointer to a function-handler
Next Control Element in List

Figure 19. A structure of a Control Element Node, which was applied in the compiler

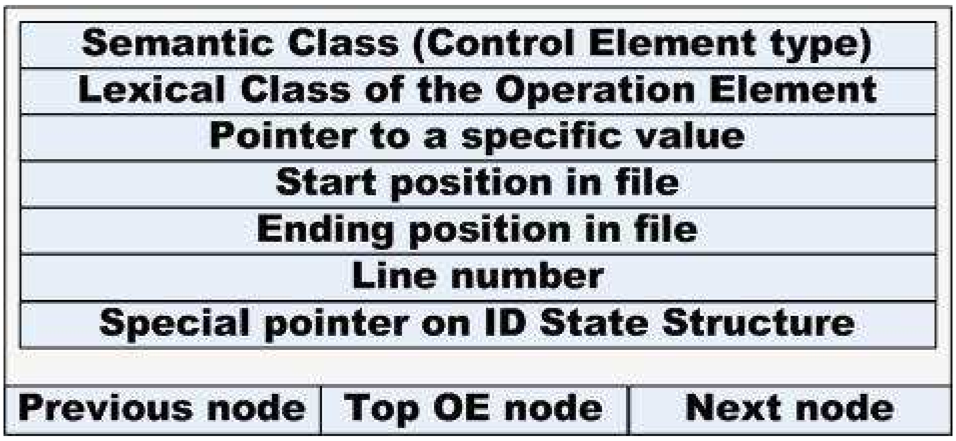
```
typedef int (* TContrElemHandlerPtr)(TOperationElemPtr);

typedef struct tagContrElem {
     UINT                      uSemClass;
     TOperationElemPtr         pBegArgList;
     TContrElemHandlerPtr      pContrElemHandler;
     struct tagContrElem       *pNextContrElem;
} TContrElem, *TContrElemPtr;
```

Figure 20. The implementation of the Control Element Node structure: two types TContrElemHandlerPtr and TContrElem

Please note, that function-handler, on which the **`pContrElemHandler`** was developed to point to a function, which has only one argument – the initial pointer to the list of operation elements, that is, the data behind the action, and the function returning value is always of the **`int`** type.

4. **Operation Element Node -** a node of the Parse Tree, which represents data in all kinds. For example, data for the mathematical operation or some data to process the assembler instruction. Usually, these data elements are the child-

nodes of the Control Element Node, but in some cases there are the child-nodes of the Function Node or Block Node. For example, in case of declaring the function arguments or local variables of the particular block, the Operation Element is used to contain the information about the variables. Figure 21 shows a structure of an Operation Element Node which was applied in the compiler. Figure 22 shows the implementation of the structure of the Operation Element Node, which was designed with two types involved: TOperationElem and TArgument.


Semantic Class (Control Element type)
Lexical Class of the Operation Element
Pointer to a specific value
Start position in file
Ending position in file
Line number
Special pointer on ID State Structure
Previous node
Top OE node
Next node
Figure 21. A structure of a Operation Element Node, which was applied in the compiler

```
typedef struct tagArgument {
   UINT                 uSemClass;
   UINT                 uLexClass;
   void                 *pvValue;
   UINT                 uBegPos;
   UINT                 uEndPos;
   UINT                 uStrNum;
   TIDCurStateListPtr   pIDCurState;
} TArgument, *TArgumentPtr;

typedef struct tagOperationElem {
   TArgument                 pArgument;
   struct tagOperationElem   *pNextOperationElem;
   struct tagOperationElem   *pPrevOperationElem;
   struct tagOperationElem   *pTopOperationElem;
} TOperationElem, *TOperationElemPtr;
```
Figure 22. The implementation of the Operation Element Node structure: two types TOperationElem and TArgument

Please note, that an identifier represents the data either and the Operation Element Node structure was developed to handle the process of identifier match analysis by means of special `pIDCurState` pointer, which points to a special ID State structure, which represents the state of identifier in the context of particular action. For example, if the variable is defined as a pointer and further dereferenced in some equation, the “dereferencing” state of particular Operation Element of the variable in Control Element of equation is presented in the ID State structure.

Figure 23 shows an example of schematic structure of the Parse Tree.

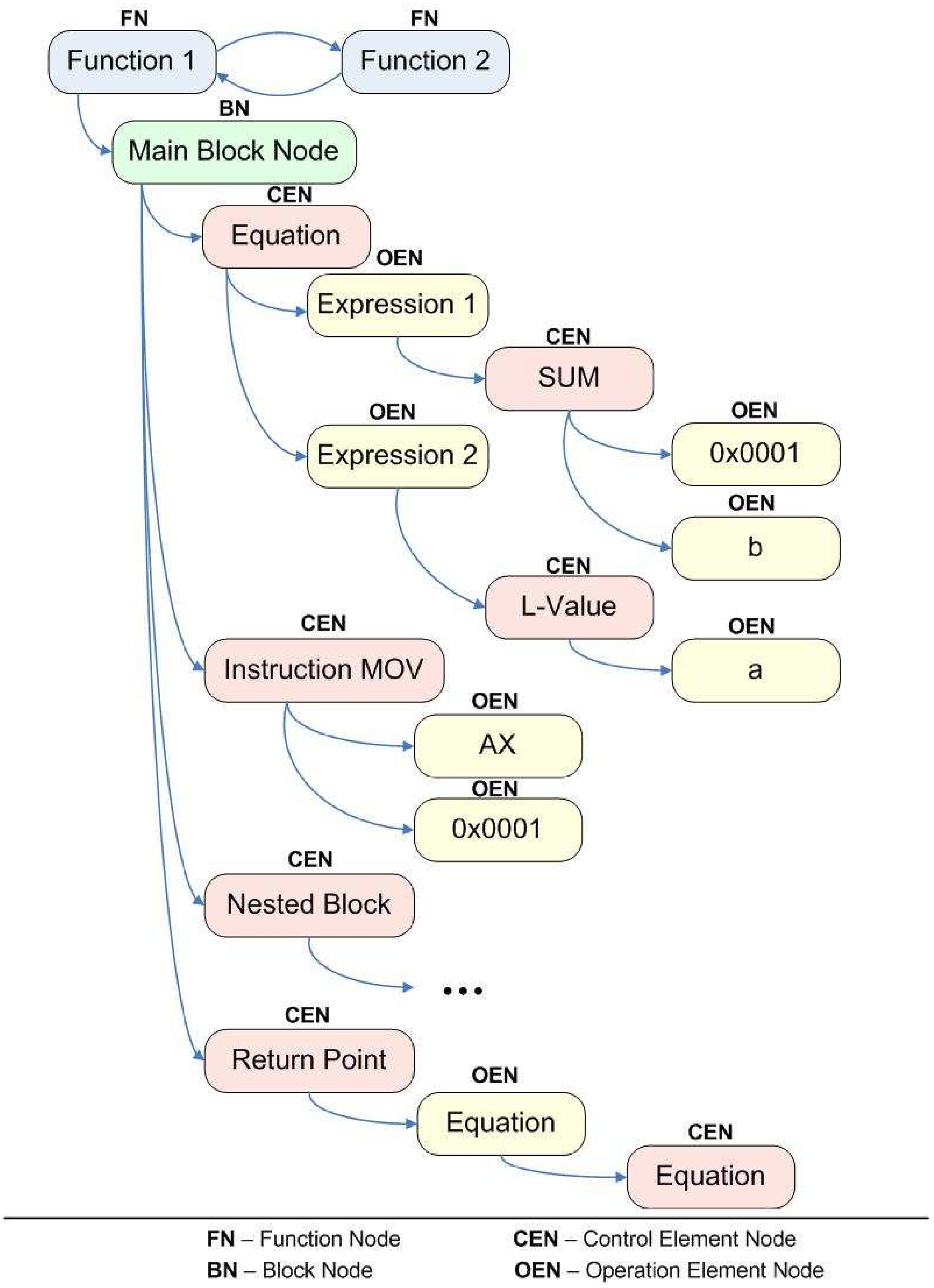
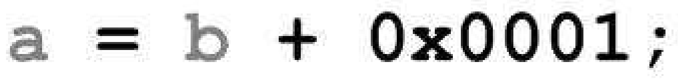


Figure 23. An example of schematic structure of the Parse Tree.

2.2.3. Semantic analysis. The main tasks of semantic analysis are:

- arithmetic operations type checking
- function call checking: arguments count comparison, type matching etc.
- type conversion

An example of arithmetic operations and type conversion could be presented in the equation case. In example of the source code above (Fig. 3) the equation is presented (Fig. 24).



```
a = b + 0x0001;
```

Figure 24. An example of equation excerpted from the source code example

During Syntactical analysis this equation is transformed into the following Parse Tree branch (Fig. 25):

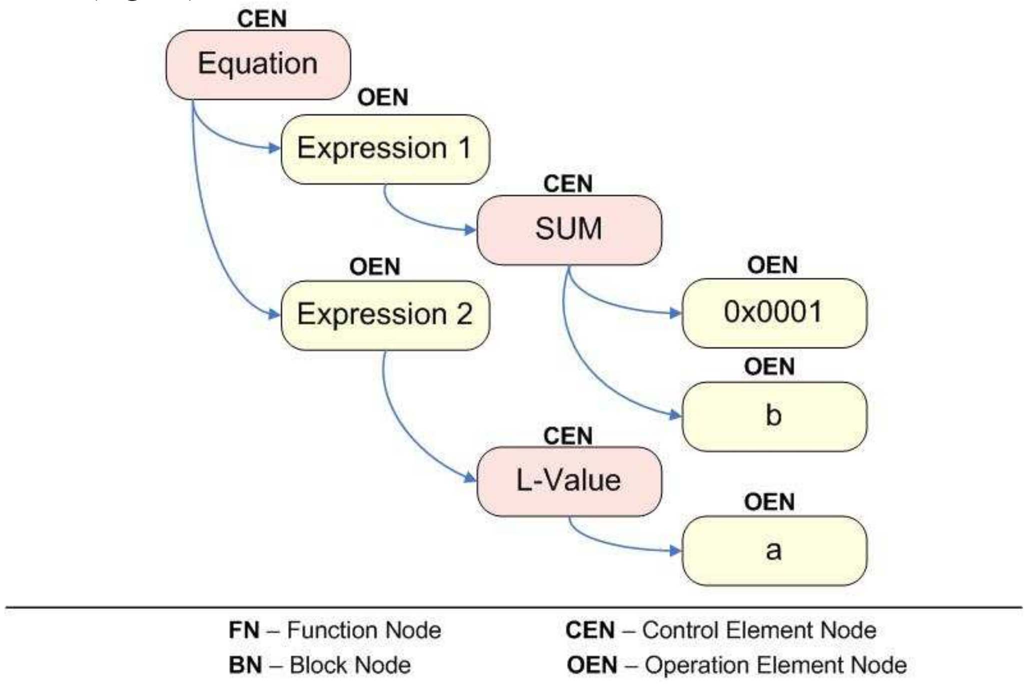


Figure 25. A Parse Tree branch represents the equation

The process of Semantic Analysis includes type checking via the whole Parse Tree. For the particular example above (Fig. 25) during the Semantic Analysis the type of the result of “SUM” Control Element is determined, the same type is considered as a type of “Expression 1” Operation Element Node then. The same way the “Expression 2” Operation Element Node is processed, that is, the type of the “Expression 2” will be considered as the type of “L-Value” Control Element Node, which is in turn considered as the type of variable a. If both expressions on both sides of the equation have the same type, the Semantic Analysis passes the equation, otherwise the process of the type conversion takes place or a user gets notification (Fig. 26)

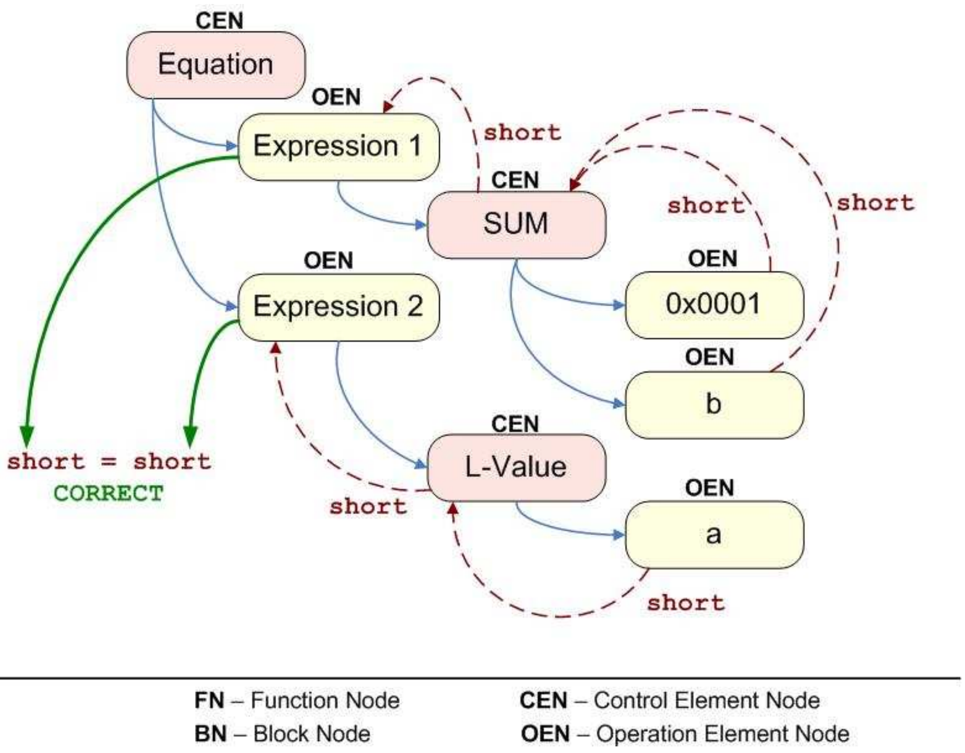


Figure 26. An example of type checking during the Semantic Analysis

2.2.4. Code generation. There are a few major steps, which compiler has to do before the process of code generation:

- the main program parts size calculation (e.g. functions size)
- functions absolute address calculation
- function linking

After all necessary operations, the instructions of target language are substituted instead of conventional operators of the Parse Tree, and the whole result data is put into a result file. Following target code will be generated by the compiler (Fig. 27) (in our case target language – IA-32 instructions [3][4]):

```
FFB60100
680100
58
5B
03C3
50
58
88860000
```

Figure 27. The result of code generation for the equation

which is conform to the next assembler instructions (Fig. 28):

```
push    word ptr [bp+0001]
push    0001
pop     ax
pop     bx
add     ax,bx
push    ax
pop     ax
mov     [bp],al
```

Figure 28. “Assembler instructions, which is conform to the code generated during Code Generation Phase for the example of equation”

Here [BP+0001] – “b” variable address and [BP] – “a” variable address

2.2.5. C compiler realization results. Hybrid language compiler (C language with inline IA-32 Assembler) was developed with the following properties:

Compiler implementation language: C

Compiler target language: IA-32 instructions

Compiler executable file format: EXE

Executable file size: 224 Kb

Source code line count: 7555

Source code file size: 230 Kb

The following component diagram shows the source code organization of the compiler (Fig. 29)

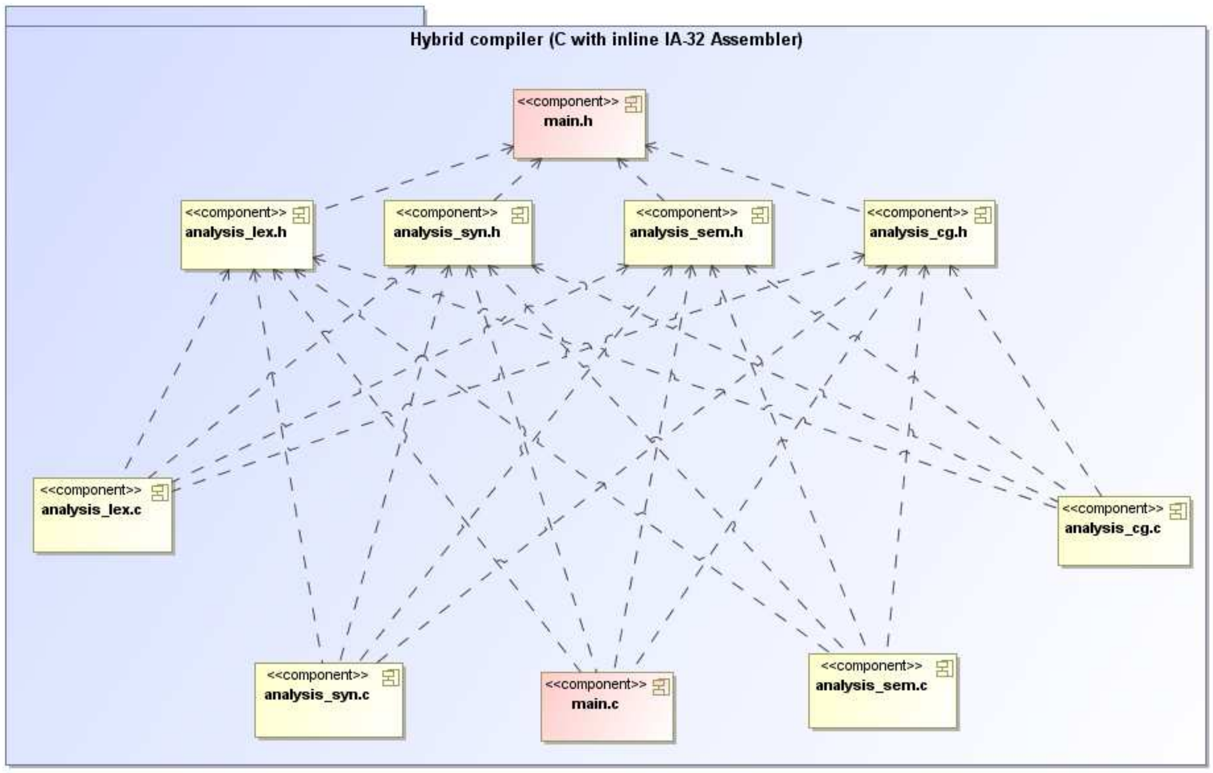

Hybrid compiler (C with inline IA-32 Assembler)
<<component>> main.h
<<component>> analysis_lex.h
<<component>> analysis_syn.h
<<component>> analysis_sem.h
<<component>> analysis_cg.h
<<component>> analysis_lex.c
<<component>> analysis_cg.c
<<component>> analysis_syn.c
<<component>> main.c
<<component>> analysis_sem.c

Figure 29. Component diagram of the source code organization of the compiler

Please, refer to Appendix B, Figure 1-10 for full source code of the compiler.

Current version of compiler was designed and developed with the following facilities support:

- ability to specify an absolute offset of program in address space
- inline assembler with the following instructions support (Table 2):

Table 2. Inline IA-32 Assembler Instructions support included in the current version of compiler

Instruction mnemonic	Description
INT imm8	Interrupt vector number specified by immediate byte
JMP ptr16:16	Jump far, absolute, address given in operand
MOV r/m8,r8	Move r8 to r/m8
MOV r/m16,r16	Move r16 to r/m16
MOV r8,r/m8	Move r/m8 to r8
MOV r16,r/m16	Move r/m16 to r16
MOV Sreg,r/m16	Move r/m16 to segment register
MOV r8,imm8	Move imm8 to r8

MOV r16,imm16	Move imm16 to r16
MOV r/m8,imm8	Move imm8 to r/m8
MOV r/m16,imm16	Move imm16 to r/m16

where [3][4][5]:

r8 - One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH,BPL, SPL, DIL and SIL; or one of the byte registers (R8L - R15L) available when using REX.R and 64-bit mode

r16 - one of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode

r/m8 - a byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL) or a byte from memory. Byte registers R8L - R15L are available using REX.R in 64-bit mode

r/m16 - a word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W-R15W are available using REX.R in 64-bit mode

imm8 - an immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value

imm16 - an immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive

- data types were implemented: short and char (because of 16-bit addressing mode implementation only)
- the single variable environment, that is, an ability to use variables defined and declare outside the assembler block in the assembler block was implemented for the following assembler instructions:
  MOV Sreg,r/m16
  MOV r8,r/m8
  MOV r16,r/m16
- the following forms of effective address were included:

Table 3. Formats of effective addresses included into the hybrid language model

Sreg: [BX+SI]	Sreg: [BX+SI+disp16]
Sreg: [BX+DI]	Sreg: [BX+DI+disp16]
Sreg: [BP+SI]	Sreg: [BP+SI+disp16]
Sreg: [BP+DI]	Sreg: [BP+DI+disp16]
Sreg: [SI]	Sreg: [SI+disp16]
Sreg: [DI]	Sreg: [DI+disp16]
Sreg: [disp16]	Sreg: [BP+disp16]
Sreg: [BX]	Sreg: [BX+disp16]

- arithmetic operation available: + , - , * , / (NOTE: the result of using division operation will be integral number, without reminder)
- there is an address arithmetic ability now in array and pointer using ability

- __cdecl function call convention
- recursive function call
- flexibility of compiler mathematical model gives an opportunity for the further easy improvement

2.2.6. Compiler using experiment. For example, there is a necessity to create an IBM PC-compatible loader with further BMP-file output on the screen. The standard 320x200x256 video mode is used. The whole project consists of two programs: initial boot program and program for parsing and printing of the BMP-file. The programs listing following (Fig. 30 and Fig. 31):

```
//---------------------------------------------------------
// TITLE:             First loader
// DESCRIPTION:       Initial boot program
// AUTHOR:            Sharipov R Nail
// DATE:              2008/05/31
// COMPILING:  cc.exe loader.c 7C00
//---------------------------------------------------------
//      @brief: reads disk sectors
//
//      @in:          cSectNum - number of sectors to read (1-128 dec.)
//                    sStartSect - first sector
//                    cHeadNum - head number  (0-15 dec.)
//                    cDriveNum - drive number (0=A:, 1=2nd floppy, 80h=drive 0,
//                                   81h=drive 1)
//                    sESValue - ES value
//                    sBXValue - BX value


char biosReadDiskSectors(char cSectNum, short sStartSect, char cHeadNum,
                                     char cDriveNum, short sESValue, short sBXValue)
{
        __asm
        {
                MOV AH,0x02          // function number
                MOV AL,cSectNum      // number of sectors to read  (1-64 dec.)
                                     // CH - track/cylinder number  (0-1023 dec., see
                                     // below)
                MOV CX,sStartSect    // CL - start sector number  (1-17 dec.)
                MOV DH,cHeadNum      // head number  (0-15 dec.)
                MOV DL,cDriveNum     // drive number (0=A:, 1=2nd floppy,80h=drive 0,
                                                   // 81h=drive 1)
                MOV ES,sESValue      //
                MOV BX,sBXValue      // ES:BX = pointer to buffer
                INT 0x13             // BIOS Service Interrupt
        }

//        |F|E|D|C|B|A|9|8|7|6|5-0|  CX
//         | | | | | | | | | |`-----  sector number
//         | | | | | | | | `--------   high order 2 bits of track/cylinder
//         `-----------------------    low order 8 bits of track/cyl number

        return 0x1;
}

//      @brief: reads disk sectors from diskette
//
//      @in:   sSectNum - number of sectors to read (1-128 dec.)
//                    sMemAddr - memory address where to read
char readSectorsFromDiskette(short sSectNum, short sMemAddr)
{
        short sIterationCount;
        char  cRemainder;
        short sESValue, sStartSect;

        sStartSect = 0x2; // the first is boot sector, so we start to read from
                             //the 2nd

        sESValue = sMemAddr/0x10;

        sIterationCount = sSectNum/0x40; // we will read by 64 sectors
        cRemainder = sSectNum - sIterationCount*0x40;

        while (sIterationCount)
        {
                biosReadDiskSectors(0x40, sStartSect, 0x0, 0x0, sESValue, 0x0);
                sStartSect = sStartSect + 0x40;
                sESValue = sESValue + 0x800; // 0x8000 - size of 64d sectors, so
                                                // we increase ES on 0x800
                sIterationCount = sIterationCount - 0x1;
        }

        return 0x1;
}
void main (  )
{
        readSectorsFromDiskette(0x40,0x7E00); // read 64d sectors (32 Kb),
                                                // starting from 7E00

        // Jump to the first byte of the loaded program
        __asm
        {
                JMP 0x0000:0x7E00
        }
}
```

Figure 30. initial boot program

This program reads 64 sectors from diskette and put them after 0x7E00 address. Then it jumps to the first read byte on 0x7E00 address.

```
//-----------------------------------------------------------------
// TITLE:               BMP-file screen printing
// DESCRIPTION:         Program for parsing and printing of the BMP-file
// AUTHOR:              Sharipov R Nail

// DATE:                2008/05/31

// COMPILING:   cc.exe program.c 7E00
//-----------------------------------------------------------------


//      @brief: sets desired video mode
//
//      @in:    cVideoMode      = 0x03 - text mode (720x400, 16 colors)
//                              = 0x0D - graphical mode (320x200,  16 colors)
//                              = 0x0E - graphical mode (640x200,  16 colors)
//                              = 0x10 - graphical mode (640x350,  16 colors)
//                              = 0x11 - graphical mode (640x480,   2 colors)
//                              = 0x12 - graphical mode (640x480,  16 colors)
//                              = 0x13 - graphical mode (320x200, 256 colors)
short setVideoMode(char cVideoMode)
{
        __asm
        {
                MOV AH,0x00
                MOV AL,cVideoMode
                INT 0x10
        }
        return 0x1;
}

//      @brief: adjusts the palette to the BMP-file, which is started from
//                      sBMPaddr address value
//
//      @in:    sBMPaddr - absolute address value, from which the BMP-file
//                      location is started
short set256RGBPalette(short sBMPaddr)
{
        char    *cpBMPPalette, *cpRGBTable ;
        short   sOffset1, sOffset2, sTableAddr, sCounter;

        sTableAddr = 0x7000; // the beginning address of the new RGB palette
                                // table

        cpBMPPalette = sBMPaddr + 0x36;         // the beginning address of BMP-file
                                                // palette
        cpRGBTable = sTableAddr;                // the beginning address of the RGB
                                                // palette table

        sOffset1 = 0x0;         // offset from the new RGB table beginning
                                // (for the INT 10/1012 interrupt)
        sOffset2 = 0x0;       // offset from the palette table of BMP-file
        sCounter = 0x100; // repeat loop 256d times

        while(sCounter)
        {
        // each iteration sets one color register
        // BMP-file palette tint sequence : Blue Green Red + 1 clear byte (0x0)
        // RGB palette table tint sequence: Red Green Blue
                cpRGBTable[sOffset1] = (cpBMPPalette[sOffset2+0x2]*0x3F)/0xFF;
                cpRGBTable[sOffset1+0x1] = (cpBMPPalette[sOffset2+0x1]*0x3F)/0xFF;
                cpRGBTable[sOffset1+0x2] = (cpBMPPalette[sOffset2]*0x3F)/0xFF;

                sOffset1 = sOffset1 + 0x3; // 3 tints <-> 3 bytes
                sOffset2 = sOffset2 + 0x4; // 3 tints <-> 3 bytes + 1 byte
                sCounter = sCounter - 0x1; // decrement counter
        }
        __asm
        {
                MOV AX,0x0000
                MOV ES,AX

                MOV AH,0x10                   // function number
                MOV AL,0x12                   // sub-function number
                MOV BX,0x0000         // first color register to set
                MOV CX,0x0100         // color registers count (256d)
                MOV DX,sTableAddr             // ES:DX point to the beginning of RGB
                                              //palette table now
                INT 0x10                      // BIOS Service Interrupt
        }

        return 0x1;
}

//      @brief: draws the BMP-file, which is started from sBMPaddr address value
//
//      @in:    sBMPaddr -  absolute address value, from which the BMP-file
//                          location is started
short drawBMP (short sBMPaddr)
{
        char    *pcBitmap, cPixel;
        short   sValueBX, sOffset, sReminder;
        short   sHorCounter,sVertCounter, *psHeight, *psWidth;

        // ES = video buffer beginning
        __asm
        {
                MOV AX,0xA000
                MOV ES,AX
        }

        // BMP-file parsing
        psHeight = sBMPaddr + 0x16;   // pointer to the picture height byte
        psWidth  = sBMPaddr + 0x12;   // pointer to the picture width byte
        pcBitmap = sBMPaddr + 0x435;  // image data beginning

        sVertCounter = *psHeight;     // sVertCounter = image height
        sHorCounter = *psWidth;               // sHorCounter = image width

        sValueBX = 0xA0 - sHorCounter/0x2;    // BX - offset in video buffer, so
                                              // now sValueBX has the value for
                                              // screen centered image output

        // Specific calculation of useless bytes in image bitmap
        if(sHorCounter - (sHorCounter/0x4)*0x4)
        {
                sReminder = (sHorCounter/0x4 + 0x1)*0x4 - sHorCounter;
        }
        else
        {
                sReminder = 0x0;
        }

        // As a result of reverse containing of image bitmap
        // we have to begin from the end of the bitmap to
        // have the right picture orientation

        sOffset = sVertCounter*(sHorCounter + sReminder);
        while(sVertCounter)
        {
                sValueBX = sValueBX + sHorCounter;
                sOffset = sOffset - sReminder;

                while(sHorCounter)
                {
                        // take one byte from the bitmap
                        cPixel = pcBitmap[sOffset];
                        __asm
                        {
                                MOV AH,cPixel
                                MOV BX,sValueBX
                                MOV ES:[BX],AH // color byte sending to the video buffer
                        }
                        sOffset = sOffset - 0x1;
                        sValueBX = sValueBX - 0x1;
                        sHorCounter = sHorCounter - 0x1;
                }
                sHorCounter = *psWidth;
                sValueBX = sValueBX + 0x140; // + 320d to begin the next pixel line
                sVertCounter = sVertCounter - 0x1;
        }
        return 0x1;
}
void main ()
{
        setVideoMode(0x13);            // setting 320x200 256 color mode
        set256RGBPalette( 0x8200 );    // BMP-file address
        drawBMP( 0x8200 );             // BMP-file address

        // perpetual loop
        while(0x1)
        {}
}
```

Figure 31. Program for parsing and printing of the BMP-file

This program parses BMP-file, which starts from 0x8200 of absolute address space, and will print it on the 320x200x256 screen. The result of hybrid compiler work is two binary files “loader.bin” and “program.bin”. To simulate the IBM PC-compatible work, the virtual machine VMware5.0 was used. A single binary floppy image was created with two binary files, which is used for boot. In this single file binary the data of loader.bin was put with 0 offset, binary data of program.bin was put with 0x200 offset and binary data of test BMP-file was put with 0x600 offset. Results are the following (Fig. 31, 32):

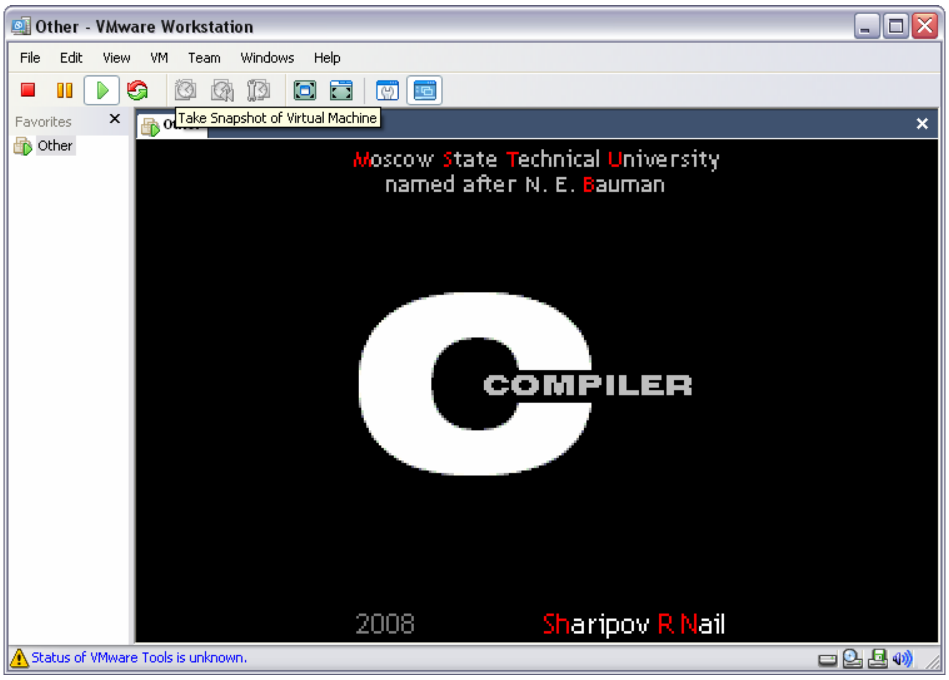

Other - VMware Workstation
File Edit View VM Team Windows Help
Take Snapshot of Virtual Machine
Favorites
Other
Moscow State Technical University
named after N. E. Bauman
COMPILER
2008
Sharipov R Nail
Status of VMware Tools is unknown.

Figure 32. First image output

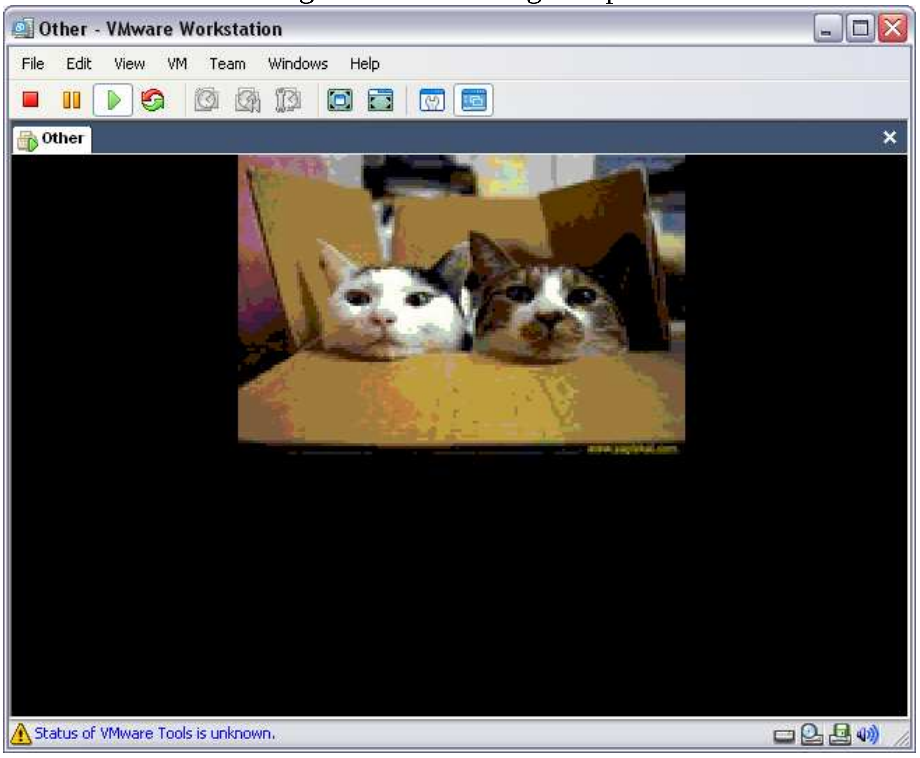

Other - VMware Workstation
File Edit View VM Team Windows Help
Other
Status of VMware Tools is unknown.

Figure 33. Second image output

3. RESULTS

As positive results of hybrid compiler development the following were highlighted:

- All necessary phases were designed and successfully implemented
- Programming style on hybrid language compiler is more clear and sound
- It took less time to develop program using high-level structures of C language while the area of application is still might be very low system software development due to inline IA-32 Assembler

- As the design of the compiler is opened and understandable there is an ability to avoid any mistakes, which might appear during any phase of the translation process
- Auxiliary structures, such as comments, are not the part of the language syntax since they are not included into the syntax model. Thus, there is an ability to include additional structures or macros in compiler implementation omitting any harm to the language

As negative results of hybrid compiler development the following were highlighted:

- There are still unnecessary code elements in executable due to imperfect compilation model
- The syntax model includes too much non-terminal symbols, which in the source code of the compiler appears as additional functions increase the time of compilation

4. DISCUSSION

To make deep comprehensive operating system development, it is mandatory to understand the process of creation and running the programs in every specific detail. To achieve these goals, it is compulsory the different ways of research to be done.

Technical research includes:

- Compiler theory improvement
- Exokernel theory developing
- Filesystem theory improvement
- Security policy developing

After hybrid language compiler had been developed, the set of new abilities is appeared:

- It is possible to make research in low-level system software development, particularly, in distributed operating systems development
- It is possible to add new features to the design and implementation of the compiler to achieve necessary goals in code generation process

As a technical prospective it could be highlighted the necessity to decrease the amount of non-terminal symbols in syntax model to increase the speed of compilation in general.

5. REFERENCES

1. Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language, Third Edition
2. W. Stallings, Operating system, fourth edition
3. Intel® 64 and IA-32 Architectures, Software Developer’s Manual, Volume 2A: Instruction Set Reference, A-M
4. Intel® 64 and IA-32 Architectures, Software Developer’s Manual, Volume 2B: Instruction Set Reference, N-Z
5. Intel® 64 and IA-32 Architectures Software Developer’s, Manual Volume 1: Basic Architecture
6. ISO/IEC 14977:1996
7. Wood, D.: Theory of Computation: Library of Congress Cataloging-in-Publication Data: John Wiley & Sons, Inc. (1987)

APPENDIX I. Constants were used in source code of the Hybrid Language Compiler

Table 1. Lexical Class Constants applied in the compiler

Lexical Class (Constant Name in the Source Code)	**Appearance**
LEXCLASS_IA32INSTRUCTIONS_JMP	JMP
LEXCLASS_IA32INSTRUCTIONS_INT	INT
LEXCLASS_COMPARISON_GREATER	>
LEXCLASS_COMPARISON_LESS	<
LEXCLASS_BLOCK_LEFT_SQAURE_BRACKET	[
LEXCLASS_BLOCK_RIGHT_SQAURE_BRACKET	]
LEXCLASS_PUNCTUATION_COLON	:
LEXCLASS_TYPE_VOID	void
LEXCLASS_TYPE_INT	int
LEXCLASS_TYPE_SHORT	short
LEXCLASS_TYPE_STRUCT	struct
LEXCLASS_TYPE_CHAR	char
LEXCLASS_PROGRAMM_ASM_ENTRY	__asm
LEXCLASS_CONSTRUCTIONS_IF	if
LEXCLASS_CONSTRUCTIONS_ELSE	else
LEXCLASS_CONSTRUCTIONS_WHILE	while
LEXCLASS_PROGRAMM_RETURN_POINT	return
LEXCLASS_BLOCK_LEFT_CURLY_BRACKET	{
LEXCLASS_IA32INSTRUCTIONS_MOV	MOV
LEXCLASS_IA32INSTRUCTIONS_MOV	PUSH
LEXCLASS_IA32REGISTERS_EAX	EAX
LEXCLASS_IA32REGISTERS_AX	AX
LEXCLASS_IA32REGISTERS_AH	AH
LEXCLASS_IA32REGISTERS_AL	AL
LEXCLASS_IA32REGISTERS_EBX	EBX
LEXCLASS_IA32REGISTERS_BX	BX
LEXCLASS_IA32REGISTERS_BH	BH
LEXCLASS_IA32REGISTERS_BL	BL
LEXCLASS_IA32REGISTERS_ECX	ECX
LEXCLASS_IA32REGISTERS_CX	CX
LEXCLASS_IA32REGISTERS_CH	CH
LEXCLASS_IA32REGISTERS_CL	CL
LEXCLASS_IA32REGISTERS_EDX	EDX
LEXCLASS_IA32REGISTERS_DX	DX
LEXCLASS_IA32REGISTERS_DH	DH
LEXCLASS_IA32REGISTERS_DL	DL
LEXCLASS_IA32REGISTERS_CS	CS
LEXCLASS_IA32REGISTERS_DS	DS
LEXCLASS_IA32REGISTERS_ES	ES
LEXCLASS_IA32REGISTERS_SS	SS
LEXCLASS_IA32REGISTERS_SI	SI
LEXCLASS_IA32REGISTERS_DI	DI
LEXCLASS_IA32REGISTERS_BP	BP
LEXCLASS_IA32REGISTERS_SP	SP
LEXCLASS_BLOCK_RIGHT_CURLY_BRACKET	}
LEXCLASS_BLOCK_LEFT_ROUND_BRACKET	(

LEXCLASS_BLOCK_RIGHT_ROUND_BRACKET	)
LEXCLASS_PUNCTUATION_SEMICOLUMN	;
LEXCLASS_COMPARISON_EQUAL	=
LEXCLASS_OPERATION_SLASH	/
LEXCLASS_OPERATION_ASTERISK	*
LEXCLASS_OPERATION_PLUS	+
LEXCLASS_OPERATION_MINUS	-
LEXCLASS_PUNCTUATION_COMMA	,
LEXCLASS_RT_NUMERIC_CONSTANT	0xXXXX[*]
LEXCLASS_RT_STRING_CONSTANT	“string” [**]
LEXCLASS_RT_IDENTIFIER	any other case

Table 2. Semantic Class Constants applied in the compiler

Semantic Class (Constant Name in the Source Code)	**Value**
SEMCLASS_UNKNOWN	1024
SEMCLASS_LVALUE	1025
SEMCLASS_MEM8	1026
SEMCLASS_MEM16	1027
SEMCLASS_MEM32	1028
SEMCLASS_IMM8	1029
SEMCLASS_IMM16	1030
SEMCLASS_IMM32	1031
SEMCLASS_REG8	1032
SEMCLASS_REG16	1033
SEMCLASS_REG32	1034
SEMCLASS_SYSREG	1035
SEMCLASS_VAR8	1036
SEMCLASS_VAR16	1037
SEMCLASS_VAR32	1038
SEMCLASS_INSTRUCTION_Sreg_GPReg16	1039
SEMCLASS_INSTRUCTION_Sreg_MEM16	1040
SEMCLASS_INSTRUCTION_MEM8_GPReg8	1041
SEMCLASS_INSTRUCTION_MEM16_GPReg16	1042
SEMCLASS_INSTRUCTION_MEM32_GPReg32	1043
SEMCLASS_INSTRUCTION_MEM8_IMM8	1044
SEMCLASS_INSTRUCTION_MEM16_IMM16	1045
SEMCLASS_INSTRUCTION_MEM32_IMM32	1046
SEMCLASS_INSTRUCTION_REG8_REG8	1047
SEMCLASS_INSTRUCTION_REG16_REG16	1048
SEMCLASS_INSTRUCTION_REG32_REG32	1049
SEMCLASS_INSTRUCTION_REG8_IMM8	1050
SEMCLASS_INSTRUCTION_REG16_IMM16	1051
SEMCLASS_INSTRUCTION_REG32_IMM32	1052
SEMCLASS_INSTRUCTION_PTR16_16	1053
SEMCLASS_INSTRUCTION_IMM8	1054
SEMCLASS_INSTRUCTION_IMM16	1055
SEMCLASS_INSTRUCTION_IMM32	1056
SEMCLASS_INSTRUCTION_REG8_MEM8	1057
SEMCLASS_INSTRUCTION_REG16_MEM16	1058
SEMCLASS_INSTRUCTION_REG32_MEM32	1059

SEMCLASS_PREFIX	1060
SEMCLASS_INSTRUCTION	1061
SEMCLASS_CONSTANT	1062
SEMCLASS_FUNCTION	1063
SEMCLASS_OPERATION	1064
SEMCLASS_FUNCARG	1065
SEMCLASS_FUNCREFTOIDTABLE	1066
SEMCLASS_FUNCRETURN	1067
SEMCLASS_EXPRESSION	1068
SEMCLASS_EQUATION	1069
SEMCLASS_BLOCK	1070
SEMCLASS_CONSTRUCTION_ELSE	1071
SEMCLASS_CONSTRUCTION_IF	1072
SEMCLASS_CONSTRUCTION_WHILE	1073

APPENDIX II. Source code of the Hybrid Language Compiler

```
#ifndef INCL_MAIN_H
#define INCL_MAIN_H

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

#define TRUE    1
#define FALSE   0
#define BEG_ADDRESS 0x7C00

enum lexClassList {
    LEXCLASS_UNKNOWN = 256,

      //lexClassSeparators
      //Block elements
      LEXCLASS_BLOCK_LEFT_SQAURE_BRACKET,
      LEXCLASS_BLOCK_RIGHT_SQAURE_BRACKET,
      LEXCLASS_BLOCK_LEFT_CURLY_BRACKET,
      LEXCLASS_BLOCK_RIGHT_CURLY_BRACKET,
      LEXCLASS_BLOCK_LEFT_ROUND_BRACKET,
      LEXCLASS_BLOCK_RIGHT_ROUND_BRACKET,
      // Punctuation marks
      LEXCLASS_PUNCTUATION_COMMA,
      LEXCLASS_PUNCTUATION_SEMICOLUMN,
      LEXCLASS_PUNCTUATION_COLON,
      // Comparison elements
      LEXCLASS_COMPARISON_GREATER,
      LEXCLASS_COMPARISON_LESS,
      LEXCLASS_COMPARISON_EQUAL,
      // Operation symbols
      LEXCLASS_OPERATION_MINUS,
      LEXCLASS_OPERATION_PLUS,
      LEXCLASS_OPERATION_ASTERISK,
      LEXCLASS_OPERATION_SLASH,

      //lexClassConstructions
      LEXCLASS_CONSTRUCTIONS_IF,
      LEXCLASS_CONSTRUCTIONS_ELSE,
      LEXCLASS_CONSTRUCTIONS_WHILE,
      //lexClassProgrammPoints
      LEXCLASS_PROGRAMM_ENTRY, //useless
      LEXCLASS_PROGRAMM_RETURN_POINT,
      LEXCLASS_PROGRAMM_ASM_ENTRY,
      //lexClassTypes
      LEXCLASS_TYPE_STRUCT,
      LEXCLASS_TYPE_VOID,           /* 0 bytes */
      LEXCLASS_TYPE_CHAR,           /* 1 byte  */
      LEXCLASS_TYPE_SHORT,    /* 2 bytes */
      LEXCLASS_TYPE_INT,            /* 4 bytes */
      //lexClassReprTableDealt
      LEXCLASS_RT_STRING_CONSTANT,
      LEXCLASS_RT_NUMERIC_CONSTANT,
      LEXCLASS_RT_IDENTIFIER,
      //lexClassIA32Registers
      LEXCLASS_IA32REGISTERS_EAX,
      LEXCLASS_IA32REGISTERS_AX,
      LEXCLASS_IA32REGISTERS_AH,
      LEXCLASS_IA32REGISTERS_AL,
      LEXCLASS_IA32REGISTERS_EBX,
      LEXCLASS_IA32REGISTERS_BX,
      LEXCLASS_IA32REGISTERS_BH,
      LEXCLASS_IA32REGISTERS_BL,
      LEXCLASS_IA32REGISTERS_ECX,
      LEXCLASS_IA32REGISTERS_CX,
      LEXCLASS_IA32REGISTERS_CH,
      LEXCLASS_IA32REGISTERS_CL,
      LEXCLASS_IA32REGISTERS_EDX,
      LEXCLASS_IA32REGISTERS_ESI,
      LEXCLASS_IA32REGISTERS_EDI,
      LEXCLASS_IA32REGISTERS_EBP,
      LEXCLASS_IA32REGISTERS_ESP,
      LEXCLASS_IA32REGISTERS_DX,
      LEXCLASS_IA32REGISTERS_DH,
      LEXCLASS_IA32REGISTERS_DL,
      LEXCLASS_IA32REGISTERS_CS,
      LEXCLASS_IA32REGISTERS_DS,
      LEXCLASS_IA32REGISTERS_ES,
      LEXCLASS_IA32REGISTERS_SS,
      LEXCLASS_IA32REGISTERS_FS,
      LEXCLASS_IA32REGISTERS_GS,
      LEXCLASS_IA32REGISTERS_SI,
      LEXCLASS_IA32REGISTERS_DI,
      LEXCLASS_IA32REGISTERS_BP,
      LEXCLASS_IA32REGISTERS_SP,
      //lexClassIA32Instructions
      LEXCLASS_IA32INSTRUCTIONS_MOV,
      LEXCLASS_IA32INSTRUCTIONS_PUSH,
      LEXCLASS_IA32INSTRUCTIONS_INT,
      LEXCLASS_IA32INSTRUCTIONS_JMP,
      //lexClassAccessories
      SYNCLASS_ACCESSORIES_BEGINBLOCK,
      SYNCLASS_ACCESSORIES_ENDBLOCK
};

enum semClassList{
      SEMCLASS_UNKNOWN = 1024,
      SEMCLASS_LVALUE,
      SEMCLASS_MEM8,
      SEMCLASS_MEM16,
      SEMCLASS_MEM32,
      SEMCLASS_IMM8,
      SEMCLASS_IMM16,
      SEMCLASS_IMM32,
      SEMCLASS_REG8,
      SEMCLASS_REG16,
      SEMCLASS_REG32,
      SEMCLASS_SYSREG,
      SEMCLASS_VAR8,
      SEMCLASS_VAR16,
      SEMCLASS_VAR32,
      // Instruction available
      SEMCLASS_INSTRUCTION_Sreg_GPReg16,
      SEMCLASS_INSTRUCTION_Sreg_MEM16,
      SEMCLASS_INSTRUCTION_MEM8_GPReg8,
      SEMCLASS_INSTRUCTION_MEM16_GPReg16,
      SEMCLASS_INSTRUCTION_MEM32_GPReg32,
      SEMCLASS_INSTRUCTION_MEM8_IMM8,
      SEMCLASS_INSTRUCTION_MEM16_IMM16,
      SEMCLASS_INSTRUCTION_MEM32_IMM32,
      SEMCLASS_INSTRUCTION_REG8_REG8,
      SEMCLASS_INSTRUCTION_REG16_REG16,
      SEMCLASS_INSTRUCTION_REG32_REG32,
      SEMCLASS_INSTRUCTION_REG8_IMM8,
      SEMCLASS_INSTRUCTION_REG16_IMM16,
      SEMCLASS_INSTRUCTION_REG32_IMM32,
      SEMCLASS_INSTRUCTION_PTR16_16,
      SEMCLASS_INSTRUCTION_IMM8,
      SEMCLASS_INSTRUCTION_IMM16,
      SEMCLASS_INSTRUCTION_IMM32,
      SEMCLASS_INSTRUCTION_REG8_MEM8,
      SEMCLASS_INSTRUCTION_REG16_MEM16,
      SEMCLASS_INSTRUCTION_REG32_MEM32,
      SEMCLASS_PREFIX,
      SEMCLASS_INSTRUCTION,
      SEMCLASS_CONSTANT,
      SEMCLASS_FUNCTION,
      SEMCLASS_OPERATION,
      SEMCLASS_FUNCARG,
      SEMCLASS_FUNCREFTOIDTABLE,
      SEMCLASS_FUNCRETURN,
      SEMCLASS_EXPRESSION,
      SEMCLASS_EQUATION,
      SEMCLASS_BLOCK,
      SEMCLASS_CONSTRUCTION_ELSE,
      SEMCLASS_CONSTRUCTION_IF,
      SEMCLASS_CONSTRUCTION_WHILE
};

enum synDeclElemType{
      SYNDECLELEMTYPE_ROUND_BRACKETS = 2048,
      SYNDECLELEMTYPE_SQUARE_BRACKETS,
      SYNDECLELEMTYPE_ASTERISK
};

enum semOperationPriority{
      SEMOPERPRIORITY_EQUAL = 2148,
      SEMOPERPRIORITY_BRACKET,
      SEMOPERPRIORITY_PLUS,
      SEMOPERPRIORITY_MUL
};

enum synIdCurState{
      SYNIDCURSTATETYPE_INITIAL_VALUES = 2248,
      SYNIDCURSTATETYPE_DEREFERENCING,
      SYNIDCURSTATETYPE_INDEX_DEREFERENCING,
      SYNIDCURSTATETYPE_OPERATION
};

enum ModRMByteEffectiveAddress16{  /*    Mod   Reg/Opcode  R/M   */
      EA16MEMLOC_BX_SI  = 0x00,     /*    00    000         000   */
      EA16MEMLOC_BX_DI  = 0x01,     /*    00    000         001   */
      EA16MEMLOC_BP_SI  = 0x02,     /*    00    000         010   */
      EA16MEMLOC_BP_DI  = 0x03,     /*    00    000         011   */
      EA16MEMLOC_SI     = 0x04,     /*    00    000         100   */
      EA16MEMLOC_DI     = 0x05,     /*    00    000         101   */
      EA16MEMLOC_DISP16 = 0x06,     /*    00    000         110   */
      EA16MEMLOC_BX     = 0x07,     /*    00    000         111   */
      EA16MEMLOC_BX_SI_DISP8 = 0x40,/*    01    000         000   */
      EA16MEMLOC_BX_DI_DISP8 = 0x41,/*    01    000         001   */
      EA16MEMLOC_BP_SI_DISP8 = 0x42,/*    01    000         010   */
      EA16MEMLOC_BP_DI_DISP8 = 0x43,/*    01    000         011   */
      EA16MEMLOC_SI_DISP8    = 0x44,/*    01    000         100   */
      EA16MEMLOC_DI_DISP8    = 0x45,/*    01    000         101   */
      EA16MEMLOC_BP_DISP8    = 0x46,/*    01    000         110   */
      EA16MEMLOC_BX_DISP8    = 0x47,/*    01    000         111   */
      EA16MEMLOC_BX_SI_DISP16= 0x80,/*    10    000         000   */
      EA16MEMLOC_BX_DI_DISP16 = 0x81,/*   10    000         001 */
      EA16MEMLOC_BP_SI_DISP16 = 0x82,/*   10    000         010 */
      EA16MEMLOC_BP_DI_DISP16 = 0x83,/*   10    000         011 */
      EA16MEMLOC_SI_DISP16    = 0x84,/*   10    000         100 */
      EA16MEMLOC_DI_DISP16    = 0x85,/*   10    000         101 */
      EA16MEMLOC_BP_DISP16    = 0x86,/*   10    000         110 */
      EA16MEMLOC_BX_DISP16    = 0x87,/*   10    000         111 */
      EA16REG_EAX_AX_AL_MM0_XMM0= 0xC0,/* 11    000         000 */
      EA16REG_ECX_CX_CL_MM1_XMM1= 0xC1,/* 11    000         001 */
      EA16REG_EDX_DX_DL_MM2_XMM2= 0xC2,/* 11    000         010 */
      EA16REG_EBX_BX_BL_MM3_XMM3= 0xC3,/* 11    000         011 */
      EA16REG_ESP_SP_AH_MM4_XMM4= 0xC4,/* 11    000         100 */
      EA16REG_EBP_BP_CH_MM5_XMM5= 0xC5,/* 11    000         101 */
      EA16REG_ESI_SI_DH_MM6_XMM6= 0xC6,/* 11    000         110 */
      EA16REG_EDI_DI_BH_MM7_XMM7= 0xC7,/* 11    000         111 */
      ////////////////////////////////////////////////////////////
      /*    For logical purposes                            */
      ////////////////////////////////////////////////////////////
      EA16MEMLOCPART_BP
};

enum ModRMByteRegOpcode16{                       /* Mod Reg/Opcode R/M*/
      RO16REG_AL_AX_EAX_MM0_XMM0_0_000 = 0x00,/*00   000      000 */
      RO16REG_CL_CX_ECX_MM1_XMM1_1_001 = 0x08,/*00   001      000 */
      RO16REG_DL_DX_EDX_MM2_XMM2_2_010 = 0x10,/*00   010      000 */
      RO16REG_BL_BX_EBX_MM3_XMM3_3_011 = 0x18,/*00   011      000 */
      RO16REG_AH_SP_ESP_MM4_XMM4_4_100 = 0x20,/*00   100      000 */
      RO16REG_CH_BP_EBP_MM5_XMM5_5_101 = 0x28,/*00   101      000 */
      RO16REG_DH_SI_ESI_MM6_XMM6_6_110 = 0x30,/*00   110      000 */
      RO16REG_BH_DI_EDI_MM7_XMM7_7_111 = 0x38/* 00   111      000 */
};

enum opcodeJcc{
      JA_REL16          = 0x87,
      JB_REL16          = 0x82,
      JNZ_REL16         = 0x85,
      JZ_REL16          = 0x84
};

enum opcodeMOV{
      MOV_RM8_R8        = 0x88,
      MOV_RM16_R16      = 0x89,
      MOV_R8_RM8        = 0x8A,
      MOV_R16_RM16      = 0x8B,
      MOV_RM16_SREG     = 0x8C,
      MOV_SREG_RM16     = 0x8E,
      MOV_AL_MOFFS8     = 0xA0,
      MOV_AX_MOFFS16    = 0xA1,
      MOV_MOFFS8_AL     = 0xA2,
      MOV_MOFFS16_AX    = 0xA3,
      MOV_R8_IMM8       = 0xB0,
      MOV_R16_IMM16     = 0xB8,
      MOV_RM8_IMM8      = 0xC6,
      MOV_RM16_IMM16    = 0xC7
};

enum opcodeSUB{
      SUB_AL_IMM8       = 0x2C,
      SUB_AX_IMM16      = 0x2D,
      SUB_RM8_IMM8      = 0x80,
      SUB_RM16_IMM16    = 0x81,
      SUB_RM8_R8        = 0x28,
      SUB_RM16_R16      = 0x29,
      SUB_R8_RM8        = 0x2A,
      SUB_R16_RM16      = 0x2B
};

enum opcodePOP{
      POP_RM16          = 0x8F,
      POP_R16           = 0x58
};

enum opcodeINT{
      INT_IMM8          = 0xCD
};


enum opcodeADD{
      ADD_AL_IMM8       = 0x04,
      ADD_AX_IMM16      = 0x05,
      ADD_RM8_IMM8      = 0x80,
      ADD_RM16_IMM16    = 0x81,
      ADD_RM8_R8        = 0x00,
      ADD_RM16_R16      = 0x01,
      ADD_R8_RM8        = 0x02,
      ADD_R16_RM16      = 0x03
};

enum opcodePUSH{
      PUSH_RM16         = 0xFF,
      PUSH_R16          = 0x50,
      PUSH_IMM8         = 0x6A,
      PUSH_IMM16        = 0x68
};

enum opcodeRET{
      RET_NEAR          = 0xC3,
};

enum opcodeCALL{
      CALL_NEAR_RM16    = 0xFF,
};

enum opcodeMUL{
      MUL_RM8           = 0xF6,
      MUL_RM16          = 0xF7
};

enum opcodeTEST{
      TEST_RM16_R16     = 0x85
};

enum opcodeJMP{
      JMP_PTR_16_16     = 0xEA,
      JMP_REL16         = 0xE9
};

enum opcodeDIV{
      DIV_RM8           = 0xF6,
      DIV_RM16          = 0xF7
};

enum ReturnAddrSize{
      RETURNSIZE32 = 4,
      RETURNSIZE16 = 2
};

enum opcodeRegByteAddition{
      RB_AL,
      RB_CL,
      RB_DL,
      RB_BL,
      RB_AH,
      RB_CH,
      RB_DH,
      RB_BH,
};

enum opcodeRegWordAddition{
      RW_AX,
      RW_CX,
      RW_DX,
      RW_BX,
      RW_SP,
      RW_BP,
      RW_SI,
      RW_DI,
};

enum prefixSegment{
      prefCS            = 0x2E,
      prefSS            = 0x36,
      prefDS            = 0x3E,
      prefES            = 0x26,
      prefFS            = 0x64,
      prefGS            = 0x65
};

enum opcodeSreg{             /*    Mod   Reg/Opcode  R/M       */

      SREG_ES_0   = 0x00,     /*    00          000         000 */
      SREG_CS_1   = 0x08,     /*    00          001         000 */
      SREG_SS_2   = 0x10,     /*    00          010         000 */
      SREG_DS_3   = 0x18,     /*    00          011         000 */
      SREG_FS_4   = 0x20,     /*    00          100         000 */
      SREG_GS_5   = 0x28,     /*    00          101         000 */
};


typedef unsigned int    UINT;
typedef unsigned char   UCHAR;

typedef struct tagLexeme {
    UINT    uLexClass;       // Lexeme Class
    UINT    uPosBeg;         // Beginning position in the file
    UINT    uPosEnd;         // Ending position in the file
      UINT    uLineNum;      // Line number in the file where a
lexeme appeared
    char   *cpTextFrag;      // String capture of a lexeme
} TLexeme, *TLexemePtr;

typedef struct tagLexemeListItem {
    struct tagLexemeListItem * pPrevLexemeInstance;
    struct tagLexemeListItem * pNextLexemeInstance;
    TLexeme                    LexemeInstance; // Lexeme info
} TLexemeListItem, *TLexemeListItemPtr;

// Address arithmetic
typedef struct tagAAOperationElement{
      UINT  uDeclType;
      void  *pvValue;
} TAAOperElem, *TAAOperElemPtr;
```

```
typedef struct tagAAOperationElementList{
      TAAOperElem                         AAOperElem;
      struct tagAAOperationElementList    *pPrevAAElem;
      struct tagAAOperationElementList    *pNextAAElem;
} TAAOperationElementList, *TAAOperElemListPtr;

// Identifier table structures
typedef struct tagIdTable {
      UINT  uType;
      UINT  uPosBeg;
      UINT  uPosEnd;
      UINT  uLineNum;
      UINT  uStackOffset;
      TAAOperElemListPtr pBegIdDecl;
      struct tagBlock        *pBlock;
      struct tagIdTable       *pPrevTie;
      struct tagRTItemContent *pItemContent;
      struct tagOperationElem *pArgInstance;
      struct tagIdTable       *pNextListItem;
      struct tagIdTable       *pPrevListItem;
} TIdTable, *TIdTablePtr;

// Identifier current state

typedef struct tagIDCurState {
      UINT                   uStateType;
      TIdTablePtr            pIdTableElem;
      TAAOperElemListPtr     pCurAAElem;
      UINT                   uIndLvlCount;
      UINT                   uTotalIndLevelCnt;
      void                   *pvValue;
} TIDCurState, *TIDCurStatePtr;

typedef struct tagIDCurStateList {
      TIDCurStatePtr                      pCurIdState;
      struct tagIDCurStateList      *pNextCurStateElem;
      struct tagIDCurStateList      *pPrevCurStateElem;
} TIDCurStateList, *TIDCurStateListPtr;


// Identifier identification error structures
typedef struct tagIdError {
    struct tagIdError * pNextError;
    char *                   cpErrorMess;
      TLexeme             ErrorInstance;
} TIdError, *TIdErrorPtr;


// Representation table structures
typedef struct tagRTItemContent {
    char   *cpIdName;
    UINT    uNameLength;
      TIdTablePtr pIdTableElem;
} TRTItemContent, *TRTItemContentPtr;

typedef struct tagRTItem {
    TRTItemContent           RTItemContent;
      struct tagRTItem  *pPrevRTItem;
    struct tagRTItem    *pNextRTItem;
} TRTItem, *TRTItemPtr;

/*
* Parse list structures
```

```
*/

typedef struct tagArgument {
    UINT                      uSemClass;
      UINT                    uLexClass;
      void                    *pvValue;
    UINT                      uBegPos;
      UINT                    uEndPos;
      UINT                    uStrNum;
      TIDCurStateListPtr      pIDCurState;
} TArgument, *TArgumentPtr;

// Operation
typedef struct tagOperationElem {
    TArgument                             pArgument;
      struct tagOperationElem     *pNextOperationElem;
    struct tagOperationElem         *pPrevOperationElem;
      struct tagOperationElem     *pTopOperationElem;
} TOperationElem, *TOperationElemPtr;

typedef int (* TContrElemHandlerPtr)(TOperationElemPtr);

typedef struct tagContrElem {
      UINT                          uSemClass;
      TOperationElemPtr       pBegArgList;
      TContrElemHandlerPtr    pContrElemHandler;
      struct tagContrElem     *pNextContrElem;
} TContrElem, *TContrElemPtr;

typedef struct tagBlock{
      UINT                    uLocVarSize;      // Size of memory
which should be allocated for local variables
      UINT                    uBlockOffset;     // Block offset
      UINT                    uBlockSize;       // Block size
      TOperationElemPtr pBegLocVarList;   // Local variables list
      TContrElemPtr           pBegContrList;    //
      struct tagBlock         *pExternalBlock;// Block surrounded
with
} TBlock, *TBlockPtr;

typedef struct tagFuncNode {
    UINT                      uType;            // type of value to
be returned
      TIdTablePtr             pFuncId;    // pointer to function
identifier in ID Table
      char                    *cpFuncName;// function name
      TOperationElemPtr pBegArgList;// argument list
      TBlockPtr               pFuncBlock; // main function block
} TFuncNode, *TFuncNodePtr;

typedef struct tagParseListNode {
    TFuncNode                             FuncNode;
    struct tagParseListNode     *pNextFuncNode;
    struct tagParseListNode         *pPrevFuncNode;
} TParseListNode, *TParseListNodePtr;


// Code generation structures
typedef struct tagByteList {
    unsigned char       cByte;
      struct tagByteList      *pNextByte;
      struct tagByteList      *pPrevByte;
} TByteList, *TByteListPtr;
```

```
typedef struct tagInstruction {
    UINT                      uInstrType;
      UINT                    uCodeSize;
      TByteListPtr            pByteList;
} TInstruction, *TInstructionPtr;

typedef struct tagInstructionList {
    TInstruction                    Instruction;
      struct tagInstructionList     *pNextInstruction;
      struct tagInstructionList     *pPrevInstruction;
} TInstructionList, *TInstructionListPtr;

typedef struct tagFuncCode {
    TIdTablePtr               pFuncId;
      UINT                    uType;
      UINT                    uSize;
      TInstructionListPtr     pInstructionList;
} TFuncCode, *TFuncCodePtr;

typedef struct tagFuncList {
    TFuncCode                 Function;
      struct tagFuncList      *pNextFuncCode;
      struct tagFuncList      *pPrevFuncCode;
} TFuncList, *TFuncListPtr;

// Code linking

typedef struct tagCallLink {
    TInstructionListPtr pInstruction;
      TIdTablePtr             pFuncId;
      struct tagCallLink      *pNextCallLink;
      struct tagCallLink      *pPrevCallLink;
} TCallLink, *TCallLinkPtr;

extern char bIsLValue;
extern char bIsPointer;
extern UINT uProgrammOffset;
extern UINT uCurLineNum;
extern UINT uCurType;
extern UINT uCurSemClass;
extern UINT uCurTotalIndLevelCnt;

extern TIdTable    firstIdTableElem;
extern TIdTablePtr pCurIdTableElem;
extern TIdTablePtr pCurIdTableBegBlock;
extern TIdTablePtr pCurIdTableFillingElem;

extern TLexemeListItemPtr     pCurLexListItem;
extern TLexemeListItemPtr     pBegLexList;

extern TRTItemPtr             pCurRTItem;
extern TRTItemPtr             pBegRT;
extern TRTItemPtr             pEndRT;

extern TIdErrorPtr                  pIdError;
extern TIdErrorPtr                  pIdErrorBegList;

extern TParseListNodePtr      pBegParseListNode; // Pointer to the
first parse tree node (function tree)
extern TParseListNodePtr      pCurParseListNode; // Pointer to a
current parse tree node

extern TOperationElemPtr      pCurFuncArg;
extern TBlockPtr              pCurBlock;
extern TOperationElemPtr      pCurLocVar;

extern TContrElemPtr          pCurContrElem;
extern TOperationElemPtr      pCurContrElemArg;

extern TFuncListPtr           pBegFuncList;
extern TFuncListPtr           pCurFunction;
extern TInstructionListPtr    pCurInstruction;
extern TByteListPtr           pCurByte;

extern TCallLinkPtr           pBegCallLinkList;
extern TCallLinkPtr           pCurCallLinkList;

extern TCallLink              mainCallLink;
extern TOperationElemPtr      pCurCGOperElem;

extern UINT                   uTopLvlExprAsteriskCount;
extern UINT                   uDeclAsteriskCount;
extern TAAOperElemListPtr     pCurIdAAE;
extern TAAOperElemListPtr     pBegAAElement;
extern TLexemeListItemPtr  pIdNameLexListItem;

extern TIDCurStateListPtr     pCurIdStateListElem;
extern TIDCurStateListPtr     pBegIdStateListElem;

#endif //INCL_MAIN_H
```

Figure 1. main.h

```
#include "analysis_lex.h"
#include "analysis_syn.h"
#include "analysis_sem.h"
#include "analysis_cg.h"

/// Global variables

char bIsLValue  = 0;
char bIsPointer = 0;
UINT uProgrammOffset = 0x7C00;
UINT uCurLineNum  = 1;
UINT uCurType           = LEXCLASS_TYPE_VOID;
UINT uCurSemClass = SEMCLASS_LVALUE;
UINT uCurTotalIndLevelCnt = 0;

TIdTable    firstIdTableElem =
{SYNCLASS_ACCESSORIES_BEGINBLOCK, 0,0,0, 0, NULL, NULL,
NULL};
TIdTablePtr pCurIdTableElem =       &firstIdTableElem;
TIdTablePtr pCurIdTableBegBlock =   &firstIdTableElem;
TIdTablePtr pCurIdTableFillingElem = &firstIdTableElem;

TLexemeListItemPtr      pCurLexListItem         = NULL;
TLexemeListItemPtr      pBegLexList             = NULL;

TRTItemPtr              pCurRTItem              = NULL;
TRTItemPtr              pBegRT                        = NULL;
TRTItemPtr              pEndRT                        = NULL;

TIdErrorPtr             pIdError                = NULL;
TIdErrorPtr             pIdErrorBegList         = NULL;

TParseListNodePtr pBegParseListNode = NULL; // Pointer to the
first parse tree node (function tree)
```

```
TParseListNodePtr pCurParseListNode = NULL; // Pointer to a
current parse tree node

TOperationElemPtr pCurFuncArg             = NULL;
TBlockPtr              pCurBlock               = NULL;
TOperationElemPtr pCurLocVar              = NULL;

TContrElemPtr          pCurContrElem           = NULL;
TOperationElemPtr pCurContrElemArg  = NULL;

TFuncListPtr           pBegFuncList            = NULL;
TFuncListPtr           pCurFunction            = NULL;
TInstructionListPtr    pCurInstruction         = NULL;
TByteListPtr           pCurByte                = NULL;

TCallLinkPtr           pBegCallLinkList  = NULL;
TCallLinkPtr           pCurCallLinkList  = NULL;

TCallLink              mainCallLink = {NULL, NULL, NULL,
NULL};
TOperationElemPtr pCurCGOperElem;

UINT                   uTopLvlExprAsteriskCount = 0;
UINT                   uDeclAsteriskCount = 0;
TAAOperElemListPtr           pCurIdAAE = NULL;
TAAOperElemListPtr           pBegAAElement = NULL;
TLexemeListItemPtr  pIdNameLexListItem = NULL;

TIDCurStateListPtr           pCurIdStateListElem = NULL;
TIDCurStateListPtr           pBegIdStateListElem = NULL;

void main(int argc, char * argv[])
{
    FILE * pInFile;
    char * cpInFileName, * cpOutFileName;
    extern TLexemeListItemPtr pCurLexListItem;
      extern UINT uProgrammOffset;

      // if there is only one argument provided then:
    // input file name is considered as "test.c"
      // program offset equals zero
      if (argc == 1)
      {
            cpInFileName = "test.c";
            uProgrammOffset = 0;
      }

      // if there are only two arguments provided then:
    // input file name equals the second argument argv[1]
      // program offset equals zero
      if (argc == 2)
      {
            cpInFileName = argv[1];
            uProgrammOffset = 0;
      }

      // if there are only three arguments provided then:
    // input file name equals the second argument argv[1]
      // program offset equals string-to-long value of the
third argument argv[2]
      if (argc == 3)
      {
            cpInFileName = argv[1];
            uProgrammOffset = strtol(argv[2], NULL, 16);
      }

      // Allocating cpInFileName string length +3 bytes of
memory
      // for an output name and associating it with
cpOutFileName
      cpOutFileName = (char *)malloc(strlen(cpInFileName)+1);
      // Copying the input file name
      strcpy(cpOutFileName, cpInFileName);

      cpOutFileName[strcspn(cpInFileName,".")] = 0;
      strcat(cpOutFileName,".bin");
      // name.c has become now name.bin

      printf("%s \n", cpOutFileName);

      if ((pInFile = fopen(cpInFileName,"rb")) == NULL)
    {
        printf("%s \n", cpOutFileName);
            printf ("\n error occured(%d): ", errno);
    }
    else
    {
            // running the Lexical Analysis
            lexAnalysis( pInFile );

        lexPrintLexemeList( pCurLexListItem, FALSE );

            synAnalysis();

            if (!pIdErrorBegList)
                  semAnalisys();

            if (!pIdErrorBegList)
                  cgCodeCreating();

            if (!pIdErrorBegList)
                  cgFuncSizeDetection();

            if (!pIdErrorBegList)
                  cgFuncOffsetDetection();

            if (!pIdErrorBegList)
                  cgLinking();

//          if (!pIdErrorBegList)
//                cgPrintFuncCode(FALSE);
//          Escaping printing out the code yet

            if (!pIdErrorBegList)
                  cgBinFileGeneration(cpOutFileName);

            synPrintIdErrors();
            if (argc == 2)
            {
                  printf(" warning: you didn't specify
initial entry point for program linking \n");
                  printf("          (default = 0 was
accepted)\n");
            }
            fclose(pInFile);
      }
}
```

Figure 2. main.c

```
#ifndef INCL_ANALYSIS_LEX_H
#define INCL_ANALYSIS_LEX_H

#include "main.h"
/** LEXICAL ANALISYS **/
// General functions
int         lexAnalysis(FILE * pInFile);
int         lexAddLexemeListItem ( char * cOccurrence, UINT
uPosBeg, UINT uPosEnd);
int         lexRetLexClassByOccurence(char * cOccurrence);
int         lexPrintLexemeList ( TLexemeListItemPtr pLexeme,
char bIsScreenPrint );

// Functions for work with representation table
int         lexIsItemInRT(char *cpItem, UINT uPosBeg, UINT
uPosEnd);
int         lexAddRTItem (char *cpItem, UINT uPosBeg, UINT
uPosEnd);
void  lexPrintRT();

// Initial text analysis
int         lexIsHex(char cSymbol);
int         lexIsOccurrenceHex(char * cOccurrence);
int         lexIsNotSepSymb(char cSymbol);

#endif //INCL_ANALYSIS_LEX_H
```

Figure 3. analysis_lex.h

```
#include "analysis_lex.h"
#include "analysis_syn.h"
#include "analysis_sem.h"
#include "analysis_cg.h"

/*
*     Returns TRUE if the identifier is already in the RT,
otherwise returns FALSE
*
*     @param char *cpItem - String capture of a identifier
lexeme
*     @param UINT uPosBeg - Beginning position in the file
*     @param UINT uPosEnd - Ending position in the file
*
*   @author Nail Sharipov
*/
int lexIsItemInRT(char *cpItem, UINT uPosBeg, UINT uPosEnd)
{
    UINT uRes;

    pCurRTItem = pBegRT;

      //Searching the current Representation Tree
      while (pCurRTItem)
    {
        // Checking whether the Identifier Name in RT node is
equal to which
            // was provided

            // Checking whether the lengths are equal
            if ( (pCurRTItem->RTItemContent.uNameLength) ==
(uPosEnd - uPosBeg) )
        {
            // Checking whether the strings are equal
                  if(!(uRes = strncmp(pCurRTItem-
>RTItemContent.cpIdName, cpItem, uPosEnd - uPosBeg + 1)))
            {
                // if the lengths and strings pairs are equal
                        return TRUE;
            }
        }
        pCurRTItem = pCurRTItem->pNextRTItem;
    }

    return FALSE;
}

/*
*     Adds new item to the Representation Tree
*
*     @param char *cpItem - String capture of a identifier
lexeme
*     @param UINT uPosBeg - Beginning position in the file
*     @param UINT uPosEnd - Ending position in the file
*
*   @author Nail Sharipov
*/
int lexAddRTItem (char *cpItem, UINT uPosBeg, UINT uPosEnd)
{
    extern TRTItemPtr pBegRT, pEndRT;
    TRTItemPtr pNewRTItem;

    pNewRTItem = (TRTItemPtr)malloc(sizeof(TRTItem));

    pNewRTItem->RTItemContent.uNameLength = uPosEnd -
uPosBeg;
    pNewRTItem->RTItemContent.cpIdName = (char *)malloc(
(uPosEnd - uPosBeg) + 2 );

      // Copying and zero-ending a string capture
    strncpy(pNewRTItem->RTItemContent.cpIdName, cpItem,
(uPosEnd - uPosBeg) + 1);
    pNewRTItem->RTItemContent.cpIdName[(uPosEnd - uPosBeg) +
1] = 0;

    pNewRTItem->pPrevRTItem = pEndRT;

      pNewRTItem->pNextRTItem = NULL;
    pNewRTItem->RTItemContent.pIdTableElem = NULL;

    if (pEndRT)
        pEndRT->pNextRTItem = pNewRTItem;

    pEndRT = pNewRTItem;

    if (!pBegRT)
        pBegRT = pEndRT;

    return TRUE;
}

/*
*     Prints the Representation Table in stdout and
CompileInfo.txt
*
*   @author Nail Sharipov
*/
```

```
void lexPrintRT()
{
    extern TRTItemPtr pCurRTItem,pBegRT;
    FILE * pOutInfoFile;

    pOutInfoFile = fopen("CompileInfo.txt","a");

    printf("\n");
    fprintf(pOutInfoFile, "\n");

    fprintf(pOutInfoFile, " ***REPRESENTATION TABLE*** \n");
    printf(" ***REPRESENTATION TABLE*** \n");

    printf("\n");
    fprintf(pOutInfoFile, "\n");

    fprintf(pOutInfoFile, " Name length | IdName \n");
    printf(" Name length | IdName \n" );

    pCurRTItem = pBegRT;

    while (pCurRTItem)
    {
        printf ("%12X ", pCurRTItem-
>RTItemContent.uNameLength);
        fprintf(pOutInfoFile, "%12X ", pCurRTItem-
>RTItemContent.uNameLength);

        printf ("  ");
        fprintf(pOutInfoFile, "  " );

        fprintf(pOutInfoFile, "%s ", pCurRTItem-
>RTItemContent.cpIdName);

            if (pCurRTItem->RTItemContent.pIdTableElem)
                  printf ("%s   --->   %d", pCurRTItem-
>RTItemContent.cpIdName, pCurRTItem-
>RTItemContent.pIdTableElem->uType);
            else
                  printf ("%s ", pCurRTItem-
>RTItemContent.cpIdName);

        printf("\n");
        fprintf(pOutInfoFile, "\n");
        pCurRTItem = pCurRTItem->pNextRTItem;
    }
    fclose(pOutInfoFile);
}

/*
*     Returns TRUE if the symbol provided is hex-digit
*
*     @param char cSymbol - Symbol to check
*   @author Nail Sharipov
*/
int lexIsHex(char cSymbol)
{
    switch (cSymbol)
    {
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
```

```
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
    case 'A':
    case 'B':
    case 'C':
    case 'D':
    case 'E':
    case 'F':
        return TRUE;
        break;
    }
    return FALSE;
}

/*
*     Returns TRUE if the lexeme instance represents the hex
value
*     with "0x"-prefix notation. Otherwise returns FALSE
*
*     @param char *cpOccurrence - String capture of a lexeme
*   @author Nail Sharipov
*/
int lexIsOccurrenceHex(char * cOccurrence)
{
    UINT i;
      UINT size_t;

      size_t = strlen(cOccurrence);
    if (strncmp( cOccurrence, "0x", 2))
        return FALSE;

    if (size_t - 2 == 0)
            return FALSE;

      for ( i = 2; i < size_t; i++)
        if(!lexIsHex(cOccurrence[i]))
        {
            return FALSE;
        }

    return TRUE;
}

/*
*     Returns the lexeme class by the string appearance
(capture)
*     of a lexeme. By default returns LEXCLASS_RT_IDENTIFIER,
so
*     all the instances of non-recognized lexeme are
considered as
*     identifier case
*
*     @param char *cpOccurrence - String capture of a lexeme
*   @author Nail Sharipov
*/
int lexRetLexClassByOccurence(char * cOccurrence)
{
    if ( !strcmp(cOccurrence, "JMP") )
        return LEXCLASS_IA32INSTRUCTIONS_JMP;
      if ( !strcmp(cOccurrence, "INT") )
        return LEXCLASS_IA32INSTRUCTIONS_INT;
      if ( !strcmp(cOccurrence, ">") )
        return LEXCLASS_COMPARISON_GREATER;
      if ( !strcmp(cOccurrence, "<") )
        return LEXCLASS_COMPARISON_LESS;
      if ( !strcmp(cOccurrence, "[") )
        return LEXCLASS_BLOCK_LEFT_SQAURE_BRACKET;
      if ( !strcmp(cOccurrence, "]") )
        return LEXCLASS_BLOCK_RIGHT_SQAURE_BRACKET;
      if ( !strcmp(cOccurrence, ":") )
        return LEXCLASS_PUNCTUATION_COLON;
      if ( !strcmp(cOccurrence, "void") )
        return LEXCLASS_TYPE_VOID;
    if ( !strcmp(cOccurrence, "int"))
        return LEXCLASS_TYPE_INT;
    if ( !strcmp(cOccurrence, "short"))
        return LEXCLASS_TYPE_SHORT;
    if ( !strcmp(cOccurrence, "struct"))
        return LEXCLASS_TYPE_STRUCT;
    if ( !strcmp(cOccurrence, "char"))
        return LEXCLASS_TYPE_CHAR;
    if ( !strcmp(cOccurrence, "__asm") )
        return LEXCLASS_PROGRAMM_ASM_ENTRY;
      if ( !strcmp(cOccurrence, "if") )
        return LEXCLASS_CONSTRUCTIONS_IF;
      if ( !strcmp(cOccurrence, "else") )
        return LEXCLASS_CONSTRUCTIONS_ELSE;
      if ( !strcmp(cOccurrence, "while") )
        return LEXCLASS_CONSTRUCTIONS_WHILE;
      if ( !strcmp(cOccurrence, "return") )
        return LEXCLASS_PROGRAMM_RETURN_POINT;
    if ( !strcmp(cOccurrence, "{") )
        return LEXCLASS_BLOCK_LEFT_CURLY_BRACKET;
    if ( !strcmp(cOccurrence, "MOV"))
        return LEXCLASS_IA32INSTRUCTIONS_MOV;
      if ( !strcmp(cOccurrence, "PUSH"))
        return LEXCLASS_IA32INSTRUCTIONS_MOV;
      if ( !strcmp(cOccurrence, "EAX"))
            return LEXCLASS_IA32REGISTERS_EAX;
      if ( !strcmp(cOccurrence, "AX"))
            return LEXCLASS_IA32REGISTERS_AX;
      if ( !strcmp(cOccurrence, "AH"))
            return LEXCLASS_IA32REGISTERS_AH;
      if ( !strcmp(cOccurrence, "AL"))
            return LEXCLASS_IA32REGISTERS_AL;
      if ( !strcmp(cOccurrence, "EBX"))
            return LEXCLASS_IA32REGISTERS_EBX;
      if ( !strcmp(cOccurrence, "BX"))
            return LEXCLASS_IA32REGISTERS_BX;
      if ( !strcmp(cOccurrence, "BH"))
            return LEXCLASS_IA32REGISTERS_BH;
      if ( !strcmp(cOccurrence, "BL"))
            return LEXCLASS_IA32REGISTERS_BL;
      if ( !strcmp(cOccurrence, "ECX"))
            return LEXCLASS_IA32REGISTERS_ECX;
      if ( !strcmp(cOccurrence, "CX"))
            return LEXCLASS_IA32REGISTERS_CX;
      if ( !strcmp(cOccurrence, "CH"))
            return LEXCLASS_IA32REGISTERS_CH;
      if ( !strcmp(cOccurrence, "CL"))
            return LEXCLASS_IA32REGISTERS_CL;
      if ( !strcmp(cOccurrence, "EDX"))
            return LEXCLASS_IA32REGISTERS_EDX;
      if ( !strcmp(cOccurrence, "DX"))
            return LEXCLASS_IA32REGISTERS_DX;
      if ( !strcmp(cOccurrence, "DH"))
            return LEXCLASS_IA32REGISTERS_DH;
      if ( !strcmp(cOccurrence, "DL"))
            return LEXCLASS_IA32REGISTERS_DL;
      if ( !strcmp(cOccurrence, "CS"))
            return LEXCLASS_IA32REGISTERS_CS;
      if ( !strcmp(cOccurrence, "DS"))
            return LEXCLASS_IA32REGISTERS_DS;
      if ( !strcmp(cOccurrence, "ES"))
            return LEXCLASS_IA32REGISTERS_ES;
      if ( !strcmp(cOccurrence, "SS"))
            return LEXCLASS_IA32REGISTERS_SS;
      if ( !strcmp(cOccurrence, "SI"))
            return LEXCLASS_IA32REGISTERS_SI;
      if ( !strcmp(cOccurrence, "DI"))
            return LEXCLASS_IA32REGISTERS_DI;
      if ( !strcmp(cOccurrence, "BP"))
            return LEXCLASS_IA32REGISTERS_BP;
      if ( !strcmp(cOccurrence, "SP"))
            return LEXCLASS_IA32REGISTERS_SP;
    if ( !strcmp(cOccurrence, "}") )
        return LEXCLASS_BLOCK_RIGHT_CURLY_BRACKET;
    if ( !strcmp(cOccurrence, "(") )
        return LEXCLASS_BLOCK_LEFT_ROUND_BRACKET;
    if ( !strcmp(cOccurrence, ")") )
        return LEXCLASS_BLOCK_RIGHT_ROUND_BRACKET;
    if ( !strcmp(cOccurrence, ";") )
        return LEXCLASS_PUNCTUATION_SEMICOLUMN;
    if ( !strcmp(cOccurrence, "=") )
        return LEXCLASS_COMPARISON_EQUAL;
    if ( !strcmp(cOccurrence, "/") )
        return LEXCLASS_OPERATION_SLASH;
      if ( !strcmp(cOccurrence, "*") )
        return LEXCLASS_OPERATION_ASTERISK;
    if ( !strcmp(cOccurrence, "+") )
        return LEXCLASS_OPERATION_PLUS;
    if ( !strcmp(cOccurrence, "-") )
        return LEXCLASS_OPERATION_MINUS;
      if ( !strcmp(cOccurrence, ",") )
        return LEXCLASS_PUNCTUATION_COMMA;
    if ( lexIsOccurrenceHex(cOccurrence) )
        return LEXCLASS_RT_NUMERIC_CONSTANT;
    if ( cOccurrence[0] == '\"' )
        return LEXCLASS_RT_STRING_CONSTANT;
    return LEXCLASS_RT_IDENTIFIER;
}

/*
*     Adds a new node to the lexeme list
*
*     @param char *cpOccurrence - String capture of a lexeme
*     @param UINT uPosBeg - Beginning position in the file
*     @param UINT uPosEnd - Ending position in the file
*   @author Nail Sharipov
*/
int lexAddLexemeListItem ( char *cpOccurrence, UINT uPosBeg,
UINT uPosEnd)
{
    extern TLexemeListItemPtr   pCurLexListItem;
    extern UINT uCurLineNum;
      TLexemeListItemPtr          pNewLexListItem;

      // Allocating memory for TLexemeListItem list node
element
```

```
    pNewLexListItem =
(TLexemeListItemPtr)malloc(sizeof(TLexemeListItem));

      // Allocating memory for string capture of a lexeme
      pNewLexListItem->LexemeInstance.cpTextFrag = (char
*)malloc( (uPosEnd - uPosBeg) + 2 );

      // Copying string capture
    strncpy(pNewLexListItem->LexemeInstance.cpTextFrag,
cpOccurrence, (uPosEnd - uPosBeg) + 1);

      pNewLexListItem->LexemeInstance.cpTextFrag[(uPosEnd -
uPosBeg) + 1] = 0;

      // Determining the lexeme class by string capture
      pNewLexListItem->LexemeInstance.uLexClass =
lexRetLexClassByOccurence( pNewLexListItem-
>LexemeInstance.cpTextFrag );

      pNewLexListItem->LexemeInstance.uPosBeg = uPosBeg;
    pNewLexListItem->LexemeInstance.uPosEnd = uPosEnd;
      pNewLexListItem->LexemeInstance.uLineNum = uCurLineNum;

    if (pNewLexListItem->LexemeInstance.uLexClass ==
LEXCLASS_RT_STRING_CONSTANT )
    {
        // Making cpOccurrence to point to the first symbol
of a string constant
            // avoiding the quote symbol
            // E.g. For "string" it makes cpOccurrence to
point to (s)
            cpOccurrence += 1;
        uPosEnd -= 2;
    }


    if (pNewLexListItem->LexemeInstance.uLexClass ==
LEXCLASS_RT_NUMERIC_CONSTANT )
    {
        // Making cpOccurrence to point to the first symbol
of a numeric constant
            // avoiding the "0x"
            // E.g. For "string" it makes cpOccurrence to
point to (s)
            cpOccurrence += 2;
        uPosEnd -= 2;
    }

      if (pNewLexListItem->LexemeInstance.uLexClass ==
LEXCLASS_RT_IDENTIFIER )
    {
            // A case of LEXCLASS_RT_IDENTIFIER is special
for handling. It concerns the filling of
            // additional data structures, the Representation
Table and the Id Table, since the way
            // of using identifiers by users is one of the
most complicated to analyze

            // Checking whether the identifier is already in
the RT
            if (!lexIsItemInRT(cpOccurrence, uPosBeg,
uPosEnd))
        {
            lexAddRTItem(cpOccurrence, uPosBeg, uPosEnd);
        }
    }

    if (pCurLexListItem)
        pCurLexListItem->pNextLexemeInstance =
pNewLexListItem;
    else
        pCurLexListItem = pBegLexList = pNewLexListItem;

    pNewLexListItem->pPrevLexemeInstance = pCurLexListItem;
    pNewLexListItem->pNextLexemeInstance = NULL;
    pCurLexListItem = pNewLexListItem;

    return TRUE;
}

/*
*     Returns FALSE if cSymbol is separator, otherwise
returns cSymbol
*
*     @param char cSymbol - symbol to check
*   @author Nail Sharipov
*/
int lexIsNotSepSymb(char cSymbol)
{
    switch (cSymbol)
    {
    case 0x09:
      case 0x0D:
      case '/':
      case '[':
      case ']':
      case '+':
    case '-':
      case '*':
    case '{':
    case '}':
    case '(':
    case ')':
    case '=':
    case ':':
      case ',':
    case '.':
    case ';':
    case ' ':
    case '\"':
    case 0x0A:
        return FALSE;
        break;
    }
        return cSymbol;
}

/*
*     Implements the lexical analysis process.
*
*     @param FILE *pInFile - source code input file
descriptor
*   @author Nail Sharipov
*/
int lexAnalysis(FILE * pInFile)
{
    char    cTempCommemt = 0;
      char    cBuffChar = 0;
    UINT    uPosBeg = 0,uPosEnd=0;
    char    cOccurrence[100];

      // Excerpting a symbol from the source file
      // Checking the excerpted symbol whether it is space or
not (whitespace, TAB etc.)
    while ( isspace(cBuffChar = getc (pInFile)) )
      {
            // Checking whether the current symbol is a
newline
            if (cBuffChar == 0x0A)
                  ++uCurLineNum; //Incrementing the current
line number variable

            ++uPosBeg; //Incrementing the beginning position
of a lexeme variable
    }

      uPosEnd = uPosBeg; // Equating the beginning and ending
positions of the first lexeme

    while ( cBuffChar != EOF) // Checking if EOF is reached
    {
        int i = 0;

            // Checking whether the current symbol is space,
separator (bracket, parenthesis etc.) or EOF
            while ( lexIsNotSepSymb(cBuffChar) && cBuffChar
!= EOF)
        {
                  cOccurrence[i] = cBuffChar; // Collecting
the symbol as a part of the lexeme
            ++i;
                  cBuffChar = getc (pInFile); // Excerpting a
symbol from the source file
            }

            // Checking for comments presence
            // (C++ notation "//" for one comment line)
            if    ((cBuffChar == '/') && ( (cTempCommemt =
getc(pInFile)) == '/'))
            {
                  // while it is not the end of a line or
EOF, skipping all symbols,
                  // incrementing uPosBeg
                  while ((cBuffChar != 0x0A) && (cBuffChar !=
EOF))
                  {
                        cBuffChar = getc (pInFile);
                        ++uPosBeg;
                  }
                  uPosEnd = uPosBeg;
                  // Since the end of a line will be
approached anyway incrementing
                  // the current line number variable
                  ++uCurLineNum;
                  continue;
            }
            else
            {
                  if    (cBuffChar == '/')
                        ungetc(cTempCommemt,pInFile);

            }
            // Checking for string constant presence.
            // Since all string constants begin with quote
symbol the first appearance
            // of a lexeme will be processed here, not with
the code above.
        if ( cBuffChar == '\"' )
        {
            cOccurrence[i] = cBuffChar; // Collecting the
quote symbol as a part of the lexeme
            do
            {
                ++i;
                cBuffChar = getc (pInFile); // Excerpting a
symbol from the source file
                cOccurrence[i] = cBuffChar; // Collecting
symbols of a string constant as a part of the lexeme
            }
            while ( cBuffChar != '\"' ); // Until the
excerpted symbol is not the quote symbol
            i++;
            cBuffChar = getc (pInFile); // Excerpting the
next symbol from the source file
        }

        uPosEnd = uPosBeg + (i - 1); // Computing the ending
position of the lexeme

            // Adding lexeme as a node to lexeme tree
            if (i != 0 )
            lexAddLexemeListItem (cOccurrence, uPosBeg,
uPosEnd);

        // Add separator as a node to lexeme tree if it's not
newline, whitespace or TAB
            if (!(cBuffChar == 0x0D) && !(cBuffChar == 0x0A)
&& !(cBuffChar == ' ') && !(cBuffChar == 0x09))
            {
                  lexAddLexemeListItem (&cBuffChar, uPosBeg +
i, uPosBeg + i);
            }
        uPosBeg = uPosEnd + 2;

            // Excerpting a symbol from the source file
            // Checking the excerpted symbol whether it is
space or not (whitespace, TAB etc.)
            while ( isspace(cBuffChar = getc (pInFile)) )
        {
            // Checking whether the current symbol is a
newline
                  if (cBuffChar == 0x0A)
                        ++uCurLineNum;

                  ++uPosBeg;
        }
            // Equating the beginning and ending positions of
the first lexeme
        uPosEnd = uPosBeg;
    }

    return TRUE;
}

/*
```

```
*     Prints the Lexeme List in stdout and CompileInfo.txt
*
*   @author Nail Sharipov
*/
int lexPrintLexemeList (TLexemeListItemPtr pLexeme, char
bIsScreenPrint)
{
    FILE * pOutInfoFile;

      pLexeme = pBegLexList;
    pOutInfoFile = fopen("CompileInfo.txt","w");
    fprintf(pOutInfoFile, "Lexeme class | Beginning pos |
Ending pos | Text \n");
    if (bIsScreenPrint)
            printf("Lexeme class | Beginning pos | Ending pos
| Text \n" );

    while (pLexeme)
    {
        if (bIsScreenPrint)
            {
                  printf ("%12d ", pLexeme-
>LexemeInstance.uLexClass);
                  printf ("%15X ", pLexeme-
>LexemeInstance.uPosBeg);
                  printf ("%12X ", pLexeme-
>LexemeInstance.uPosEnd);
        }
        fprintf(pOutInfoFile, "%12d ", pLexeme-
>LexemeInstance.uLexClass );
        fprintf(pOutInfoFile, "%15X ", pLexeme-
>LexemeInstance.uPosBeg );
        fprintf(pOutInfoFile, "%12X ", pLexeme-
>LexemeInstance.uPosEnd);

        if (bIsScreenPrint)
                  printf ("  ");
        fprintf(pOutInfoFile, "  " );

            if (bIsScreenPrint)
                  printf (" \"%s\" ", pLexeme-
>LexemeInstance.cpTextFrag);
        fprintf(pOutInfoFile, " \"%s\" ", pLexeme-
>LexemeInstance.cpTextFrag);

            if (bIsScreenPrint)
                  printf("\n");
        fprintf(pOutInfoFile, "\n");
        pLexeme = pLexeme->pNextLexemeInstance;
    }
    fclose(pOutInfoFile);
    return TRUE;
}
```

Figure 4. analysis_lex.c

```
#ifndef INCL_ANALYSIS_SYN_H
#define INCL_ANALYSIS_SYN_H

#include "main.h"
/** SYNTACTICAL ANALYSIS **/
// General functions
int         synAnalysis();
int         synRetNextLCValue();
int         synGetNextLexemeClass();
void  synPrintIdErrors();
void  synError();
void  synIdError(TLexemeListItemPtr pErrorLexListItem, char *
cpErrMess);
void  synPrintIdTable();

// Next lex step viewing functions
int         synIsNextTermLexeme (UINT uVerifiableLexClass);
int         synIsNextNontermType();
int         synIsNextNontermFuncImpl();
int         synIsNextNontermInstruction();
int         synIsNextNontermSegmentRegister();
int         synIsNextNontermByteGPRegister();
int         synIsNextNontermWordGPRegister();
int         synIsNextNontermDWordGPRegister();
int         synIsNextNontermValue();
int         synIsNextNontermArgEnum();

// Next lex step checking functions
int         synCheckNextTermLexeme (UINT
uVerifiableLexClass);
int         synCheckNextNontermType();
int         synCheckNontermAsmBlock();
int         synCheckNextNontermImplementation();
int         synCheckNextNontermBlock();
int         synCheckNextNontermSegmentRegister();
int         synCheckNextNontermByteGPRegister();
int         synCheckNextNontermWordGPRegister();
int         synCheckNextNontermDWordGPRegister();
int         synCheckNextNontermInstruction();
int         synCheckNextNontermValue();
int         synCheckNextNontermArgEnum();
int         synCheckNextNontermTransVarBrac();
int         synCheckNextNontermTransVarMul(UINT uType);
int         synCheckNextNontermTransVarPlus(UINT uType);
int         synCheckNextNontermVarEqual();
int         synCheckNextNontermFuncDecl();
int         synCheckNextNontermVarDecl();
int         synCheckNextNontermEquation();

int         synCheckVarDeclaration();
int         synIdDeclaration();
int         synIdProperDeclaration();

//    Parse tree creating functions
int         synAddNewFuncNode (char * cpFuncName, UINT uType,
TIdTablePtr pFuncId);
int         synAddNewStructNode (char * cpStructName, UINT
uType);
int         synAddNewContrElemToCurBlock(UINT uSemClass,
TContrElemHandlerPtr pHandler);
int         synAddNewOperElemToContrElem(UINT uSemClass, UINT
uType, void * pvValue, UINT uBegPos, UINT uEndPos, UINT
uStrNum, TIDCurStateListPtr pIDCurState);
int         synAddNewFuncArg(UINT uSemClass, UINT uType, void
* pvValue, UINT uBegPos, UINT uEndPos, UINT uStrNum,
TIDCurStateListPtr pIDCurState);
int         synAddNewFuncLocVar(UINT uSemClass, UINT uType,
void * pvValue, UINT uBegPos, UINT uEndPos, UINT uStrNum);
int         synAddReturnContrElem();
int         synAddBlockContrElem();
TContrElemPtr
      synCreateNewContrElem(TContrElemHandlerPtr pHandler);
```

```
TOperationElemPtr synCreateNewOperElem(UINT uSemClass, UINT
uType, void * pvValue, UINT uBegPos, UINT uEndPos, UINT
uStrNum, TIDCurStateListPtr pIDCurState);

// Identifier identification functions (these part of
compilation was merged with syntactical analysis)
TRTItemPtr  synGetRTElemAddrByName( char * cpName );
TIdTablePtr synAddIdentTableElem(UINT uType, UINT uPosBeg,
UINT uPosEnd, UINT uLineNum, TRTItemContentPtr pItemContent,
TIdTablePtr pPrevTie, TOperationElemPtr pArgInstance,
TBlockPtr pBlock, TAAOperElemListPtr pBegIdDecl);
TRTItemPtr  synCheckIdInRT(TLexemeListItemPtr
pLexIdentifier);
int               synFillRT();
int               synUnFillRT();

#endif //INCL_ANALYSIS_SYN_H
```

Figure 5. analysis_syn.h

```
#include "analysis_lex.h"
#include "analysis_syn.h"
#include "analysis_sem.h"
#include "analysis_cg.h"

/*
*
*
*     @param char * cpFuncName
*     @param UINT uType
*     @param TIdTablePtr pFuncId
*
*   @author Nail Sharipov
*/
int synAddNewFuncNode (char * cpFuncName, UINT uType,
TIdTablePtr pFuncId)
{
      extern TParseListNodePtr      pCurParseListNode;
      extern TParseListNodePtr      pBegParseListNode;

      TParseListNodePtr             pNewFuncNode;

      pNewFuncNode =
(TParseListNodePtr)malloc(sizeof(TParseListNode));

      pNewFuncNode->pNextFuncNode           = NULL;
      pNewFuncNode->FuncNode.pBegArgList    = NULL;
      pNewFuncNode->FuncNode.pFuncBlock = NULL;
      pNewFuncNode->FuncNode.pFuncId = pFuncId;
      pNewFuncNode->pPrevFuncNode = pCurParseListNode;

      pNewFuncNode->FuncNode.cpFuncName = (char
*)malloc(strlen(cpFuncName)+1);
      strcpy( pNewFuncNode->FuncNode.cpFuncName, cpFuncName
);

      pNewFuncNode->FuncNode.uType = uType;

      if (pCurParseListNode)
      {
            pCurParseListNode->pNextFuncNode = pNewFuncNode;
            pCurParseListNode = pNewFuncNode;
      }
      if (!pBegParseListNode)
      {
            pBegParseListNode = pNewFuncNode;
            pCurParseListNode = pNewFuncNode;
      }

      pCurFuncArg = NULL;
      pCurLocVar = NULL;
      pCurBlock = NULL;


      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAddNewStructNode (char * cpStructName, UINT uType)
{
      extern TParseListNodePtr      pCurParseListNode;
      extern TParseListNodePtr      pBegParseListNode;

      TParseListNodePtr             pNewNode;

      pNewNode =
(TParseListNodePtr)malloc(sizeof(TParseListNode));

      pNewNode->pNextFuncNode           = NULL;
      pNewNode->FuncNode.pBegArgList        = NULL;
      pNewNode->FuncNode.pFuncBlock  = NULL;

      pNewNode->pPrevFuncNode = pCurParseListNode;

      pNewNode->FuncNode.cpFuncName = (char
*)malloc(strlen(cpStructName)+1);
      strcpy( pNewNode->FuncNode.cpFuncName, cpStructName );

      pNewNode->FuncNode.uType = uType;

      if (pCurParseListNode)
      {
            pCurParseListNode->pNextFuncNode = pNewNode;
            pCurParseListNode = pNewNode;
      }

      if (!pBegParseListNode)
      {
            pBegParseListNode = pNewNode;
            pCurParseListNode = pNewNode;
      }

      pCurFuncArg = NULL;
      pCurLocVar = NULL;
      pCurContrElem = NULL;


      return TRUE;
}
```

```
/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
TBlockPtr synCreateNewBlock()
{
      TBlockPtr   pNewBlock;

      pNewBlock = (TBlockPtr)malloc(sizeof(TBlock));
      pNewBlock->pBegContrList = NULL;
      pNewBlock->pBegLocVarList = NULL;
      pNewBlock->uBlockOffset = 0;
      pNewBlock->uBlockSize = 0;
      pNewBlock->uLocVarSize = 0;
      pNewBlock->pExternalBlock = NULL;

      return pNewBlock;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
TContrElemPtr synCreateNewContrElem(TContrElemHandlerPtr
pHandler)
{
      TContrElemPtr                       pNewContrElem;

      pNewContrElem =
(TContrElemPtr)malloc(sizeof(TContrElem));
      pNewContrElem->uSemClass = SEMCLASS_EQUATION;
      pNewContrElem->pBegArgList = NULL;
      pNewContrElem->pNextContrElem = NULL;
      pNewContrElem->pContrElemHandler = pHandler;

      return pNewContrElem;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
TOperationElemPtr synCreateNewOperElem(UINT uSemClass, UINT
uType, void * pvValue, UINT uBegPos, UINT uEndPos, UINT
uStrNum, TIDCurStateListPtr pIDCurState)
{
      TOperationElemPtr
      pNewContrElemArg;

      pNewContrElemArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
```

```
      pNewContrElemArg->pArgument.pvValue = pvValue;
      pNewContrElemArg->pArgument.uSemClass = uSemClass;
      pNewContrElemArg->pArgument.uLexClass = uType;
      pNewContrElemArg->pArgument.uBegPos = uBegPos;
      pNewContrElemArg->pArgument.uEndPos = uEndPos;
      pNewContrElemArg->pArgument.uStrNum = uStrNum;
      pNewContrElemArg->pArgument.pIDCurState = pIDCurState;

      pNewContrElemArg->pNextOperationElem = NULL;
      pNewContrElemArg->pPrevOperationElem = NULL;
      pNewContrElemArg->pTopOperationElem  = NULL;

      return pNewContrElemArg;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAddBlockContrElem()
{
      TBlockPtr              pCurBlockContainer = NULL,
pNewBlock = NULL;
      TContrElemPtr          pCurBlockCEContainer = NULL;
      TOperationElemPtr pCurCEArgContainer = NULL;
      TContrElemPtr          pNewCE = NULL;

      synAddNewContrElemToCurBlock(SEMCLASS_BLOCK,
&cgContrElemBlock);
      synAddNewOperElemToContrElem(SEMCLASS_BLOCK, 0,
pCurBlock,0,0,0, NULL);

      pNewBlock = synCreateNewBlock();
      synAddNewOperElemToContrElem(SEMCLASS_BLOCK, 0,
pNewBlock,0,0,0, NULL);

      pCurBlockContainer = pCurBlock;
      pCurBlockCEContainer = pCurContrElem;
      pCurCEArgContainer = pCurContrElemArg;

      pCurBlock = pNewBlock;
      pCurLocVar = NULL;
      pCurContrElem = NULL;
      pCurContrElemArg = NULL;

      synCheckNextNontermBlock();

      pCurBlock = pCurBlockContainer;
      pCurContrElem = pCurBlockCEContainer;
      pCurContrElemArg = pCurCEArgContainer;

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAddNewBlockToCurFuncNode()
{
      extern TParseListNodePtr      pCurParseListNode;
      extern TContrElemPtr          pCurContrElem;
      TBlockPtr                     pNewBlock;

      pNewBlock = synCreateNewBlock();
      pNewBlock->uBlockSize = 0;
      pNewBlock->uBlockOffset = 0;
      pNewBlock->pExternalBlock = NULL;

      pCurParseListNode->FuncNode.pFuncBlock = pNewBlock;
      pCurBlock = pNewBlock;

      pCurLocVar = NULL;
      pCurContrElem = NULL;
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAddNewLocVarToCurBlock(UINT uSemClass, UINT uType,
void * pvValue, UINT uBegPos, UINT uEndPos, UINT uStrNum,
TIDCurStateListPtr pIDCurState)
{
      extern TBlockPtr        pCurBlock;
      extern TOperationElemPtr      pCurLocVar;
      TOperationElemPtr             pNewLocVar;

      pNewLocVar = synCreateNewOperElem(uSemClass, uType,
pvValue, uBegPos, uEndPos, uStrNum, pIDCurState);

      if (pCurLocVar)
      {
            pCurLocVar->pNextOperationElem = pNewLocVar;
            pNewLocVar->pPrevOperationElem = pCurLocVar;
            pCurLocVar = pNewLocVar;
      }

      if (!pCurBlock->pBegLocVarList)
      {
            pCurBlock->pBegLocVarList = pNewLocVar;
            pCurLocVar = pNewLocVar;
      }

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
```

```
TContrElemPtr synAddNewBlockContrElemToCurContrElem()
{
      extern TContrElemPtr          pCurContrElem;
      TBlockPtr              pCurBlockContainer = NULL,
pNewBlock = NULL;
      TContrElemPtr          pCurBlockCEContainer = NULL;
      TOperationElemPtr pCurCEArgContainer = NULL;
      TContrElemPtr          pNewCE = NULL;


      pNewCE = synCreateNewContrElem(&cgContrElemBlock);
      pCurContrElem = pNewCE;
      pCurContrElemArg = NULL;
      synAddNewOperElemToContrElem(SEMCLASS_BLOCK, 0,
pCurBlock,0,0,0, NULL);

      pNewBlock = synCreateNewBlock();
      synAddNewOperElemToContrElem(SEMCLASS_BLOCK, 0,
pNewBlock,0,0,0, NULL);

      pCurBlockContainer = pCurBlock;
      pCurBlockCEContainer = pCurContrElem;
      pCurCEArgContainer = pCurContrElemArg;


      pCurBlock = pNewBlock;
      pCurLocVar = NULL;
      pCurContrElem = NULL;
      pCurContrElemArg = NULL;

      synCheckNextNontermBlock();

      pCurBlock = pCurBlockContainer;
      pCurContrElem = pCurBlockCEContainer;
      pCurContrElemArg = pCurCEArgContainer;

      return pNewCE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAddNewContrElemToCurBlock(UINT uSemClass,
TContrElemHandlerPtr pHandler)
{
      extern TParseListNodePtr      pCurParseListNode;
      extern TContrElemPtr          pCurContrElem;

      TContrElemPtr                       pNewContrElem;

      pNewContrElem = synCreateNewContrElem(pHandler);
      pNewContrElem->uSemClass = uSemClass;
      if (pCurContrElem)
      {
            pCurContrElem->pNextContrElem = pNewContrElem;
            pCurContrElem = pNewContrElem;
      }

      if(!pCurBlock->pBegContrList)
      {
            pCurBlock->pBegContrList = pNewContrElem;
            pCurContrElem = pNewContrElem;
      }

      pCurContrElemArg = NULL;
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAddNewOperElemToContrElem(UINT uSemClass, UINT uType,
void * pvValue, UINT uBegPos, UINT uEndPos, UINT uStrNum,
TIDCurStateListPtr pIDCurState)
{
      extern TContrElemPtr          pCurContrElem;
      extern TOperationElemPtr      pCurContrElemArg;
      TOperationElemPtr             pNewContrElemArg;

      pNewContrElemArg = synCreateNewOperElem(uSemClass,
uType, pvValue, uBegPos, uEndPos, uStrNum, pIDCurState);

      if (pCurContrElemArg)
      {
            pCurContrElemArg->pNextOperationElem =
pNewContrElemArg;
            pNewContrElemArg->pPrevOperationElem =
pCurContrElemArg;
            pCurContrElemArg = pNewContrElemArg;
      }

      if (!pCurContrElem->pBegArgList)
      {
            pCurContrElem->pBegArgList = pNewContrElemArg;
            pCurContrElemArg = pNewContrElemArg;
      }

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAddNewFuncArg(UINT uSemClass, UINT uType, void *
pvValue, UINT uBegPos, UINT uEndPos, UINT uStrNum,
TIDCurStateListPtr pIDCurState)
{
      extern TParseListNodePtr      pCurParseListNode;
      extern TOperationElemPtr      pCurFuncArg;

      TOperationElemPtr             pNewFuncArg;

      pNewFuncArg = synCreateNewOperElem(uSemClass, uType,
pvValue, uBegPos, uEndPos, uStrNum, pIDCurState);
      pNewFuncArg->pNextOperationElem = NULL;
      pNewFuncArg->pPrevOperationElem = NULL;

      if (pCurFuncArg)
      {
            pCurFuncArg->pNextOperationElem = pNewFuncArg;
            pNewFuncArg->pPrevOperationElem = pCurFuncArg;
            pCurFuncArg = pNewFuncArg;

      }

      if(!pCurParseListNode->FuncNode.pBegArgList)
      {
            pCurParseListNode->FuncNode.pBegArgList =
pNewFuncArg;
            pCurFuncArg = pNewFuncArg;

      }

      return TRUE;
}

/*
*
*
*     @param UINT uType - type of identifier
*     @param UINT uPosBeg - Beginning position in the file
*     @param UINT uPosEnd - Ending position in the file
*     @param UINT uLineNum - Line number
*     @param TRTItemContentPtr pItemContent - Pointer to the
RT Item Content
*     @param TIdTablePtr pPrevTie - Pointer to a previous
Identifier in the ID Table
*     @param TOperationElemPtr pArgInstance -
*     @param TBlockPtr pBlock - Pointer to a current block
being processed
*     @param TAAOperElemListPtr pBegIdDecl
*
*   @author Nail Sharipov
*/
TIdTablePtr synAddIdentTableElem(UINT uType, UINT uPosBeg,
UINT uPosEnd, UINT uLineNum,

TRTItemContentPtr pItemContent, TIdTablePtr pPrevTie,

TOperationElemPtr pArgInstance, TBlockPtr pBlock,

TAAOperElemListPtr pBegIdDecl)
{
    extern TIdTablePtr pCurIdTableElem;
    TIdTablePtr pNewIdTableElem;

    pNewIdTableElem = (TIdTablePtr)malloc(sizeof(TIdTable));

      pNewIdTableElem->pItemContent = pItemContent;

      pNewIdTableElem->pNextListItem = NULL;
      pNewIdTableElem->pPrevListItem = pCurIdTableElem;

      pNewIdTableElem->pPrevTie = pPrevTie;
    pNewIdTableElem->uType = uType;
      pNewIdTableElem->pArgInstance = pArgInstance;
      pNewIdTableElem->uPosBeg = uPosBeg;
      pNewIdTableElem->uPosEnd = uPosEnd;
      pNewIdTableElem->uLineNum = uLineNum;
      pNewIdTableElem->pBlock = pBlock;
      pNewIdTableElem->pBegIdDecl = pBegIdDecl;

      pCurIdTableElem->pNextListItem = pNewIdTableElem;
    pCurIdTableElem = pNewIdTableElem;

    if (uType == SYNCLASS_ACCESSORIES_BEGINBLOCK)
      {
            pCurIdTableBegBlock = pNewIdTableElem;
            pCurIdTableFillingElem = pCurIdTableBegBlock;
      }

      if (uType == SYNCLASS_ACCESSORIES_ENDBLOCK)
      {
            pNewIdTableElem->pPrevTie = pCurIdTableBegBlock;
            pCurIdTableFillingElem = pNewIdTableElem;
      }

    return pNewIdTableElem;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
void synIdError(TLexemeListItemPtr pErrorLexListItem, char *
cpErrMess)
{
      extern TIdErrorPtr pIdErrorBegList;
      extern TIdErrorPtr pIdError;
      TIdErrorPtr pNewIdError;

      pNewIdError = (TIdErrorPtr)malloc(sizeof(TIdError));
      pNewIdError->cpErrorMess = (char
*)malloc(strlen(cpErrMess)+1);
      strcpy(pNewIdError->cpErrorMess, cpErrMess);
      pNewIdError->ErrorInstance.cpTextFrag =
pErrorLexListItem->LexemeInstance.cpTextFrag;
      pNewIdError->ErrorInstance.uLexClass =
pErrorLexListItem->LexemeInstance.uLexClass;
      pNewIdError->ErrorInstance.uPosBeg = pErrorLexListItem-
>LexemeInstance.uPosBeg;
      pNewIdError->ErrorInstance.uPosEnd = pErrorLexListItem-
>LexemeInstance.uPosEnd;
      pNewIdError->ErrorInstance.uLineNum =
pErrorLexListItem->LexemeInstance.uLineNum;
      pNewIdError->pNextError = NULL;

      if (pIdError)
      {
            pIdError->pNextError = pNewIdError;
      }

      if (!pIdErrorBegList)
      {
            pIdErrorBegList= pNewIdError;
      }
      pIdError = pNewIdError;
```

```
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
TRTItemPtr synCheckIdInRT(TLexemeListItemPtr pLexIdentifier)
{
      TRTItemPtr pBuff;
    extern TRTItemPtr pBegRT;

    pBuff = pBegRT;

    while (pBuff)
    {
            if (!strcmp(pLexIdentifier-
>LexemeInstance.cpTextFrag, pBuff->RTItemContent.cpIdName))
            if (pBuff->RTItemContent.pIdTableElem)
            {
                  return pBuff;
            }
            else
            {
                  synIdError (pLexIdentifier, "unknown
identifier");
                  return NULL;
            }
        pBuff = pBuff->pNextRTItem;
    };
      return FALSE;

}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
void synPrintIdErrors()
{
      extern TIdErrorPtr pIdErrorBegList;
      TIdErrorPtr pTemp;
      FILE * pOutInfoFile;

    pOutInfoFile = fopen("CompileInfo.txt","a");

    printf("\n");
    fprintf(pOutInfoFile, "\n");
      printf(" *** ERRORS *** \n\n");
      fprintf(pOutInfoFile, " *** ERRORS *** \n\n");

      if(pIdErrorBegList)
            pTemp = pIdErrorBegList;
      else
      {
            printf("     no errors \n\n");
            fprintf(pOutInfoFile, "     no errors \n\n");
            fclose(pOutInfoFile);
            return;
      }

      while (pTemp)
      {
            printf(" (%3d ) error: \'%s\' : %s\n",
                  pTemp->ErrorInstance.uLineNum, (pTemp-
>ErrorInstance.cpTextFrag)?pTemp-
>ErrorInstance.cpTextFrag:"",
                  pTemp->cpErrorMess);
            fprintf(pOutInfoFile, " (%X %X) error: \'%s\' :
%s\n",
                  pTemp->ErrorInstance.uPosBeg, pTemp-
>ErrorInstance.uPosEnd, pTemp->ErrorInstance.cpTextFrag,
                  pTemp->cpErrorMess);
            pTemp = pTemp->pNextError;
      }
      fclose(pOutInfoFile);

}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
void synError()
{
    extern TLexemeListItemPtr pCurLexListItem;

      if (pCurLexListItem)
            synIdError(pCurLexListItem->pPrevLexemeInstance,
"syntax error");
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
// Just returns the next lexeme class without changing
pCurLexListItem pointer to the current lexeme
int synRetNextLCValue()
{
    extern TLexemeListItemPtr pCurLexListItem;

    if (pCurLexListItem)
        return pCurLexListItem->LexemeInstance.uLexClass;
    return LEXCLASS_UNKNOWN;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synGetNextLexemeClass()
{
    extern TLexemeListItemPtr pCurLexListItem;
    UINT uNextLexClass;

    if (pCurLexListItem)
    {
        uNextLexClass = pCurLexListItem-
>LexemeInstance.uLexClass;
        pCurLexListItem = pCurLexListItem-
>pNextLexemeInstance;
        return uNextLexClass;
    }
    return LEXCLASS_UNKNOWN;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextTermLexeme (UINT uVerifiableLexClass)
{
    extern TLexemeListItemPtr pCurLexListItem;
    UINT uNextLexClass;

    uNextLexClass = synRetNextLCValue();
    if ( uVerifiableLexClass == uNextLexClass )
        return uNextLexClass;

    return FALSE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextTermLexeme (UINT uVerifiableLexClass)
{
    UINT uNextLexClass;

    uNextLexClass = synGetNextLexemeClass();
    if ( uVerifiableLexClass == uNextLexClass )
        return uNextLexClass;

    synError();
    return FALSE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermType()
{
    UINT uLexClass;

    uLexClass = synRetNextLCValue();
    switch ( uLexClass )
    {
    case LEXCLASS_TYPE_VOID:
    //case LEXCLASS_TYPE_INT:
    case LEXCLASS_TYPE_SHORT:
    case LEXCLASS_TYPE_CHAR:
        return uLexClass;
        break;
    default:
        return FALSE;
        break;
    }

}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermType()
{
    UINT uLexClass;

    uLexClass = synGetNextLexemeClass();
    switch (uLexClass)
    {
    case LEXCLASS_TYPE_VOID:
    //case LEXCLASS_TYPE_INT:
    case LEXCLASS_TYPE_SHORT:
    case LEXCLASS_TYPE_CHAR:
        return uLexClass;
        break;
    default:
        synError();
        return FALSE;
        break;
    }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermFuncImpl()
{
    UINT uLexClass;

    uLexClass = synRetNextLCValue();
    switch (uLexClass)
    {
    case LEXCLASS_PROGRAMM_ASM_ENTRY:
    case LEXCLASS_RT_IDENTIFIER:
    case LEXCLASS_PROGRAMM_RETURN_POINT:
      case LEXCLASS_BLOCK_LEFT_CURLY_BRACKET:
      case LEXCLASS_OPERATION_ASTERISK:
      case LEXCLASS_CONSTRUCTIONS_IF:
      case LEXCLASS_CONSTRUCTIONS_WHILE:
            return uLexClass;
        break;
    default:
        return FALSE;
        break;
    }
}

/*
*     Returns the Representation Table Item pointer by
identifier
*     name. Returns FALSE In case there is no RT Item with
such name
*
*     @param char *cpName - Identifier name
*   @author Nail Sharipov
*/

TRTItemPtr synGetRTElemAddrByName( char * cpName )
{
    TRTItemPtr pBuff;
    extern TRTItemPtr pBegRT;

    pBuff = pBegRT;

    while (pBuff)
    {
        if (!strcmp(cpName, pBuff->RTItemContent.cpIdName))
            return pBuff;
        pBuff = pBuff->pNextRTItem;
    };
    return FALSE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synFillRT()
{
      extern TIdTablePtr pCurIdTableElem;
      extern TIdTablePtr pCurIdTableBegBlock;
      TIdTablePtr pCur;


      pCur = pCurIdTableFillingElem->pNextListItem;
      while ( pCur != pCurIdTableElem->pNextListItem )
      {
            pCur->pPrevTie = pCur->pItemContent-
>pIdTableElem;
            pCur->pItemContent->pIdTableElem = pCur;
            pCur = pCur->pNextListItem;
      }
      pCurIdTableFillingElem = pCurIdTableElem;

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
TIdTablePtr synInternBlocksPassing(TIdTablePtr pCur)
{

      while ( pCur->uType != SYNCLASS_ACCESSORIES_ENDBLOCK )
      {
            if ( pCur->uType ==
SYNCLASS_ACCESSORIES_BEGINBLOCK )
            {
                  pCur = pCur->pNextListItem;
                  pCur = synInternBlocksPassing(pCur);
            }
            else
                  pCur = pCur->pNextListItem;
      }

      return pCur;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synUnFillRT()
{
      extern TIdTablePtr pCurIdTableElem;
      extern TIdTablePtr pCurIdTableBegBlock;
      TIdTablePtr pCur;

      pCur = pCurIdTableBegBlock->pNextListItem;
      while ( pCur )
      {
            if ( pCur->uType ==
SYNCLASS_ACCESSORIES_BEGINBLOCK )
            {
                  pCur = pCur->pNextListItem;
                  pCur = synInternBlocksPassing(pCur);
            }
            else
            {
                  if ( pCur->uType !=
SYNCLASS_ACCESSORIES_ENDBLOCK )
                        pCur->pItemContent->pIdTableElem =
pCur->pPrevTie;
            }

            pCur = pCur->pNextListItem;
      }
      pCurIdTableBegBlock = pCurIdTableBegBlock->pPrevTie;

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermFuncDecl()
{
      extern TLexemeListItemPtr pCurLexListItem;

    UINT uType;
    TRTItemPtr pBuff;
      TIdTablePtr pCurFuncId;

      uType = synCheckNextNontermType();
      synCheckNextTermLexeme(LEXCLASS_RT_IDENTIFIER);

      pBuff = synGetRTElemAddrByName(pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.cpTextFrag);

      if (pBuff)
      {
            /*
            *     @param UINT uType - type of identifier
            *     @param UINT uPosBeg - Beginning position in
the file
            *     @param UINT uPosEnd - Ending position in
the file
            *     @param UINT uLineNum - Line number

            *     @param TRTItemContentPtr pItemContent -
Pointer to the RT Item Content
            *     @param TIdTablePtr pPrevTie -

            *     @param TOperationElemPtr pArgInstance -
            *     @param TBlockPtr pBlock - Pointer to a
current block being processed
            *     @param TAAOperElemListPtr pBegIdDecl
            */
            pCurFuncId = synAddIdentTableElem(uType,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,



      &(pBuff->RTItemContent),

      pBuff->RTItemContent.pIdTableElem,
      NULL,

      NULL,

      NULL);
      }

      /*
      *
      *
      *     @param char * cpFuncName
      *     @param UINT uType
      *     @param TIdTablePtr pFuncId
      *
      *   @author Nail Sharipov
      */
      synAddNewFuncNode(pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.cpTextFrag, uType, pCurFuncId);

      synFillRT();
      synAddIdentTableElem(SYNCLASS_ACCESSORIES_BEGINBLOCK,
0,0,0,NULL, pCurIdTableBegBlock, NULL, NULL, NULL);

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_ROUND_BRACKE
T);

    if (uType = synIsNextNontermType())
    {
        synCheckNextNontermType();
        synAddNewFuncArg(SEMCLASS_FUNCARG, uType, NULL,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

        synCheckVarDeclaration();
            if(pIdNameLexListItem)
            {
                  pBuff =
synGetRTElemAddrByName(pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.cpTextFrag);

                  synAddIdentTableElem(uType,
                        pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                        pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                        pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,
                        &(pBuff->RTItemContent), pBuff-
>RTItemContent.pIdTableElem,
                        pCurFuncArg, NULL, pBegAAElement);

                  pCurFuncArg->pArgument.pvValue =
pCurIdTableElem;
            }

        while
(synIsNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA))
        {
```

```
synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA);

            uType = synIsNextNontermType();
            synCheckNextNontermType();

                  synAddNewFuncArg(LEXCLASS_RT_IDENTIFIER,
uType, NULL,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);
            synCheckVarDeclaration();

                  if(pIdNameLexListItem)
                  {
                        pBuff =
synGetRTElemAddrByName(pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.cpTextFrag);

                        synAddIdentTableElem(uType,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,
                              &(pBuff->RTItemContent), pBuff-
>RTItemContent.pIdTableElem,
                              pCurFuncArg, NULL,
pBegAAElement);

                        pCurFuncArg->pArgument.pvValue =
pCurIdTableElem;
                  }
        }
    }

      pCurFuncId->pArgInstance = pCurParseListNode-
>FuncNode.pBegArgList;
      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_ROUND_BRACK
ET);
    return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckVarDeclaration()
{
      extern TAAOperElemListPtr     pCurIdAAE;
      extern TAAOperElemListPtr     pBegAAElement;
      extern TLexemeListItemPtr     pIdNameLexListItem;

      pIdNameLexListItem      = NULL;
      pCurIdAAE   = NULL;
      pBegAAElement     = NULL;

      synIdDeclaration();
      if (!pIdNameLexListItem)
      {
            semError("Identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"needs name specification");
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCreateNewAAOperElement(UINT uAAOEType, void *pvValue)
{
      extern TAAOperElemListPtr     pCurIdAAE;
      extern TAAOperElemListPtr     pBegAAElement;
      TAAOperElemListPtr                  pNewAAOE;

      pNewAAOE =
(TAAOperElemListPtr)malloc(sizeof(TAAOperationElementList));
      pNewAAOE->AAOperElem.pvValue = pvValue;
      pNewAAOE->AAOperElem.uDeclType = uAAOEType;
      pNewAAOE->pNextAAElem = NULL;
      pNewAAOE->pPrevAAElem = NULL;

      if (pCurIdAAE)
      {
            pCurIdAAE->pNextAAElem = pNewAAOE;
            pNewAAOE->pPrevAAElem = pCurIdAAE;
            pCurIdAAE = pNewAAOE;
      }

      if (!pBegAAElement)
      {
            pBegAAElement = pNewAAOE;
            pCurIdAAE = pNewAAOE;
      }

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIdDeclaration()
{
      UINT uAsteriskCount = 0;
      void *pvValue;
      for (uAsteriskCount = 0;
synIsNextTermLexeme(LEXCLASS_OPERATION_ASTERISK);
uAsteriskCount++)

      synCheckNextTermLexeme(LEXCLASS_OPERATION_ASTERISK);

      synIdProperDeclaration();

      if (uAsteriskCount) // parasitic round brackets
elimination
      {
            pvValue = (UINT *)malloc(sizeof(UINT));
            *((UINT *)pvValue) = uAsteriskCount;

      synCreateNewAAOperElement(SYNDECLELEMTYPE_ASTERISK,
pvValue);
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIdProperDeclaration()
{
      char  *cValue;
      void  *pvValue    = NULL;
      extern TLexemeListItemPtr pCurLexListItem;
      extern TLexemeListItemPtr pIdNameLexListItem;

      if
(synIsNextTermLexeme(LEXCLASS_BLOCK_LEFT_ROUND_BRACKET))
      {

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_ROUND_BRACKE
T);
            synIdDeclaration();

      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_ROUND_BRACK
ET);
      }
      else

      if(synCheckNextTermLexeme(LEXCLASS_RT_IDENTIFIER))
            {
                  pIdNameLexListItem = pCurLexListItem;
            }

      // array declaration
      while(synIsNextTermLexeme(LEXCLASS_BLOCK_LEFT_SQAURE_BR
ACKET))
      {

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_SQAURE_BRACK
ET);
            if(synIsNextNontermValue())
            {
                  synCheckNextNontermValue();
                  pvValue = (UINT *)malloc(sizeof(UINT));
                  cValue = &pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.cpTextFrag[2];
                  *((UINT*)pvValue) = strtol(cValue, NULL,
16);
            }

      synCreateNewAAOperElement(SYNDECLELEMTYPE_SQUARE_BRACKE
TS, pvValue);

      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_SQAURE_BRAC
KET);
      }
      /* TODO: put here function pointer declaration ability
("ANSI C" D. Ritchie, p.161 ) */

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
void synPrintAAOEList()
{
      extern TAAOperElemListPtr     pBegAAElement;
      TAAOperElemListPtr pTempAAOE;
      pTempAAOE = pBegAAElement;

      while(pTempAAOE)
      {
            printf("%d ---- %d \n", pTempAAOE-
>AAOperElem.uDeclType, *((UINT*)pTempAAOE-
>AAOperElem.pvValue));
            pTempAAOE = pTempAAOE->pNextAAElem;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermVarDecl()
{

    UINT uType;
    TRTItemPtr pBuff;

      if (uType = synIsNextNontermType())
      {
            synCheckNextNontermType();

            synAddNewLocVarToCurBlock(LEXCLASS_RT_IDENTIFIER,
uType, NULL,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

            synCheckVarDeclaration();
            if(pIdNameLexListItem)
            {
                  pBuff =
synGetRTElemAddrByName(pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.cpTextFrag);

                  synAddIdentTableElem(uType,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,
                              &(pBuff->RTItemContent), pBuff-
>RTItemContent.pIdTableElem,
                              pCurLocVar, pCurBlock,
pBegAAElement);
            }
            pCurLocVar->pArgument.pvValue = pCurIdTableElem;

            while
(synIsNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA))
            {

      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA);


      synAddNewLocVarToCurBlock(LEXCLASS_RT_IDENTIFIER,
uType, NULL,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

                  synCheckVarDeclaration();

                  if(pIdNameLexListItem)
                  {
                        pBuff =
synGetRTElemAddrByName(pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.cpTextFrag);

                        synAddIdentTableElem(uType,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,
                              &(pBuff->RTItemContent), pBuff-
>RTItemContent.pIdTableElem,
                                    pCurLocVar, pCurBlock,
pBegAAElement);

                        pCurLocVar->pArgument.pvValue =
pCurIdTableElem;
```

```
                  }
            }
    }

    synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_SEMICOLUMN);
    while ( synIsNextNontermType() )
    {
        uType = synIsNextNontermType();
            synCheckNextNontermType();
            synAddNewLocVarToCurBlock(LEXCLASS_RT_IDENTIFIER,
uType, NULL,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

            synCheckVarDeclaration();
            if(pIdNameLexListItem)
            {
                  pBuff =
synGetRTElemAddrByName(pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.cpTextFrag);

                  synAddIdentTableElem(uType,
                        pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                        pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                        pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,
                        &(pBuff->RTItemContent), pBuff-
>RTItemContent.pIdTableElem,
                                    pCurLocVar, pCurBlock,
pBegAAElement);

                  pCurLocVar->pArgument.pvValue =
pCurIdTableElem;
            }
        while
(synIsNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA))
        {

synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA);



      synAddNewLocVarToCurBlock(LEXCLASS_RT_IDENTIFIER,
uType, NULL,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

            synCheckVarDeclaration();
                  if(pIdNameLexListItem)
                  {
                        pBuff =
synGetRTElemAddrByName(pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.cpTextFrag);
```

```
                        synAddIdentTableElem(uType,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                              pIdNameLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,
                              &(pBuff->RTItemContent), pBuff-
>RTItemContent.pIdTableElem,
                              pCurLocVar, pCurBlock,
pBegAAElement);

                        pCurLocVar->pArgument.pvValue =
pCurIdTableElem;
                  }
        }

synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_SEMICOLUMN);

    }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermArgEnum()
{
    switch (synRetNextLCValue())
    {
    case LEXCLASS_RT_NUMERIC_CONSTANT:
    case LEXCLASS_RT_STRING_CONSTANT:
    case LEXCLASS_OPERATION_ASTERISK:
    case LEXCLASS_RT_IDENTIFIER:
        return TRUE;
        break;
    default:
        return FALSE;
        break;
    }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermValue()
{
    UINT uType;
      switch (uType = synGetNextLexemeClass())
    {
    case LEXCLASS_RT_NUMERIC_CONSTANT:
    case LEXCLASS_RT_STRING_CONSTANT:
        uType = semRetValueType(pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd -
```

```
pCurLexListItem->pPrevLexemeInstance->LexemeInstance.uPosBeg
- 1);
            if (uType == LEXCLASS_UNKNOWN)
                  synIdError(pCurLexListItem-
>pPrevLexemeInstance, "value out of range");
            return uType;
        break;
    default:
        synError();
        return FALSE;
        break;
    }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermValue()
{
    switch (synRetNextLCValue())
    {
    case LEXCLASS_RT_NUMERIC_CONSTANT:
    case LEXCLASS_RT_STRING_CONSTANT:
        return TRUE;
        break;
    default:
        return FALSE;
        break;
    }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermArgEnum()
{
      TContrElemPtr pCurCEContainer = NULL, pNewContrElem =
NULL;
      TOperationElemPtr pCurCEArgContainer = NULL;
      UINT uType = LEXCLASS_TYPE_VOID;

      pNewContrElem =
synCreateNewContrElem(&cgContrElemEquation);

      synAddNewOperElemToContrElem(SEMCLASS_FUNCARG,
LEXCLASS_TYPE_VOID, pNewContrElem,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      pCurCEContainer = pCurContrElem;
      pCurCEArgContainer = pCurContrElemArg;

      pCurContrElem = pNewContrElem;
      pCurContrElemArg = NULL;

      uType = synCheckNextNontermEquation();

      pCurContrElem = pCurCEContainer;
      pCurContrElemArg = pCurCEArgContainer;

      pCurCEArgContainer->pArgument.uLexClass = uType;
      while (synIsNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA))
    {

      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA);
            pNewContrElem =
synCreateNewContrElem(&cgContrElemEquation);

            synAddNewOperElemToContrElem(SEMCLASS_FUNCARG,
LEXCLASS_TYPE_SHORT, pNewContrElem,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

            pCurCEContainer = pCurContrElem;
            pCurCEArgContainer = pCurContrElemArg;

            pCurContrElem = pNewContrElem;
            pCurContrElemArg = NULL;

            uType = synCheckNextNontermEquation();

            pCurContrElem = pCurCEContainer;
            pCurContrElemArg = pCurCEArgContainer;

            pCurCEArgContainer->pArgument.uLexClass = uType;
      }
    return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
TIDCurStatePtr synCreateNewIdCurStateElem(UINT uStateType,
TIdTablePtr pIdTableElem, TAAOperElemListPtr pCurAAElem, void
*pvValue, UINT uIndLvlCount)
{
      TIDCurStatePtr pNewIdStateElem;

      pNewIdStateElem =
(TIDCurStatePtr)malloc(sizeof(TIDCurState));
      pNewIdStateElem->uStateType = uStateType;
      pNewIdStateElem->pIdTableElem = pIdTableElem;
      pNewIdStateElem->pCurAAElem = pCurAAElem;
      pNewIdStateElem->pvValue = pvValue;
      pNewIdStateElem->uIndLvlCount = uIndLvlCount;
      return pNewIdStateElem;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAddNewIdCurStateElem(UINT uStateType, TIdTablePtr
pIdTableElem, TAAOperElemListPtr pCurAAElem, void *pvValue,
UINT uIndLvlCount)
{
      extern TIDCurStateListPtr pCurIdStateListElem;
      TIDCurStateListPtr pNewCurStateListElem;

      pNewCurStateListElem =
(TIDCurStateListPtr)malloc(sizeof(TIDCurStateList));
      pNewCurStateListElem->pCurIdState =
synCreateNewIdCurStateElem(uStateType, pIdTableElem,
pCurAAElem, pvValue, uIndLvlCount);

      pNewCurStateListElem->pNextCurStateElem = NULL;
      pNewCurStateListElem->pPrevCurStateElem = NULL;
      if (pCurIdStateListElem)
      {
            pCurIdStateListElem->pNextCurStateElem =
pNewCurStateListElem;
            pNewCurStateListElem->pPrevCurStateElem =
pCurIdStateListElem;
            pCurIdStateListElem = pNewCurStateListElem;
      }

      if (!pBegIdStateListElem)
      {
            pBegIdStateListElem = pNewCurStateListElem;
            pCurIdStateListElem = pNewCurStateListElem;
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermTransVarBrac()
{
      extern char bIsPointer;
      extern TAAOperElemListPtr     pCurIdAAE;
      extern TIDCurStateListPtr     pCurIdStateListElem;
      extern UINT                   uCurSemClass;
      extern UINT                   uCurType;
      extern UINT                   uTopLvlExprAsteriskCount;
      extern UINT                   uDeclAsteriskCount;
      extern TAAOperElemListPtr pBegAAElement;

      TAAOperElemListPtr            pBegAAElementContainer =
NULL;
      TAAOperElemListPtr           pCurIdAAEContainer
      = NULL;
      TIDCurStateListPtr      pBegIdStateContainer    = NULL;
      TIDCurStateListPtr      pCurIdStateContainer    = NULL;
      TOperationElemPtr pCurCEArgContainer            = NULL;
      TContrElemPtr           pCurCEContainer                =
NULL;
      TContrElemPtr           pNewContrElem                  =
NULL;
      TRTItemPtr              pBuff;

      char  bPrevIsPointer = 0, bIsPointerContainer = 0 ;
      char  *cValue;
      void  *pvValue    = NULL;
      UINT  uSemClass   = 0;
      UINT  uType;
      UINT  uExspressionAsteriskCount = 0;
      UINT  uDeclACContainer = 0;
      UINT  uIndLvlCountAsterContainer = 0;
      UINT  uIndLvlCountSqBrcContainer = 0;
      UINT  uIndLvlCountAster = 0;
      UINT  uIndLvlCountSqBrc = 0;

      for (uExspressionAsteriskCount = 0;
synIsNextTermLexeme(LEXCLASS_OPERATION_ASTERISK);uExspression
AsteriskCount++)

      synCheckNextTermLexeme(LEXCLASS_OPERATION_ASTERISK);

      uIndLvlCountAster = uExspressionAsteriskCount;

      if (synIsNextNontermValue())
    {
        uType = synCheckNextNontermValue();

        if(!bIsPointer)
            {
                  switch(semRetTypeSizeByType(uType))
                  {
                  case 4:
                        uSemClass = SEMCLASS_IMM32;
                        break;
                  case 2:
                        uSemClass = SEMCLASS_IMM16;
                        break;
                  case 1:
                        uSemClass = SEMCLASS_IMM8;
                        break;
                  }
                  uCurSemClass = uSemClass;
            }

            pvValue = (UINT *)malloc(sizeof(UINT));
            cValue = &pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.cpTextFrag[2];

            *((UINT*)pvValue) = strtol(cValue, NULL, 16);
            synAddNewOperElemToContrElem(uSemClass, uType,
pvValue,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);
            if (uExspressionAsteriskCount > 0)
                  semError("*",
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"unexpected use of this operation");

            return uType;
    }
if ((synRetNextLCValue() ==
LEXCLASS_BLOCK_LEFT_ROUND_BRACKET) || (synRetNextLCValue() ==
LEXCLASS_RT_IDENTIFIER))
{

      switch(synRetNextLCValue())
    {
      case LEXCLASS_BLOCK_LEFT_ROUND_BRACKET:

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_ROUND_BRACKE
T);

                  pvValue = (int *)malloc(1);
                  *((UINT *)pvValue) =
SEMOPERPRIORITY_BRACKET;

      synAddNewOperElemToContrElem(SEMCLASS_OPERATION,
LEXCLASS_BLOCK_LEFT_ROUND_BRACKET, pvValue,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

                  if (pBegIdStateListElem)
                  {
                        pBegIdStateContainer =
pBegIdStateListElem;
                        pCurIdStateContainer =
pCurIdStateListElem;
                        pCurIdAAEContainer = pCurIdAAE;
                        bPrevIsPointer = bIsPointer;
                        pBegAAElementContainer =
pBegAAElement;
                  };
                  bIsPointer = FALSE;
                  pBegIdStateListElem = NULL;
                  pCurIdStateListElem = NULL;
                  pCurIdAAE = NULL;

                  uType =
synCheckNextNontermTransVarPlus(synCheckNextNontermTransVarMu
l(synCheckNextNontermTransVarBrac()));
                  pvValue = (int *)malloc(1);
                  *((UINT *)pvValue) =
SEMOPERPRIORITY_BRACKET;

      synAddNewOperElemToContrElem(SEMCLASS_OPERATION,
LEXCLASS_BLOCK_RIGHT_ROUND_BRACKET, pvValue,
                        pCurLexListItem->pPrevLexemeInstance-
```

```
>LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);


      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_ROUND_BRACK
ET);

                  if (pCurIdStateListElem)
                        pCurIdAAE = pCurIdStateListElem-
>pCurIdState->pCurAAElem;

                  while ( (uDeclAsteriskCount == 0) &&
pCurIdAAE && (pCurIdAAE->pNextAAElem) )
                  {
                        pCurIdAAE = pCurIdAAE->pNextAAElem;
                        switch (pCurIdAAE-
>AAOperElem.uDeclType)
                        {
                              case
SYNDECLELEMTYPE_SQUARE_BRACKETS:
                                    uDeclAsteriskCount = 1;
                                    break;
                              case SYNDECLELEMTYPE_ASTERISK:
                                    uDeclAsteriskCount =
*((UINT *)pCurIdAAE->AAOperElem.pvValue);
                                    break;
                        }
                  }


      while(synIsNextTermLexeme(LEXCLASS_BLOCK_LEFT_SQAURE_BR
ACKET))
                  {
                        uIndLvlCountSqBrc ++;
                        if (pCurIdAAE && ((pCurIdAAE-
>AAOperElem.uDeclType == SYNDECLELEMTYPE_SQUARE_BRACKETS) ||
(pCurIdAAE->AAOperElem.uDeclType ==
SYNDECLELEMTYPE_ASTERISK)))
                        {

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_SQAURE_BRACK
ET);

                              pNewContrElem =
synCreateNewContrElem(&cgContrElemEquation);
                              pCurCEContainer =
pCurContrElem;
                              pCurCEArgContainer =
pCurContrElemArg;

                              pCurContrElem = pNewContrElem;
                              pCurContrElemArg = NULL;


                              bIsPointerContainer =
bIsPointer;
                              synCheckNextNontermEquation();

                              if (bIsPointer)
                                    semError("[",
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,"illegal index,
indirection not allowed");
                              else
                                    bIsPointer =
bIsPointerContainer;

                              pCurContrElem =
pCurCEContainer;
                              pCurContrElemArg =
pCurCEArgContainer;


      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_SQAURE_BRAC
KET);

                              while ( (uDeclAsteriskCount ==
0) && (pCurIdAAE->pNextAAElem) )
                              {
                                    pCurIdAAE = pCurIdAAE-
>pNextAAElem;
                                    switch (pCurIdAAE-
>AAOperElem.uDeclType)
                                    {
                                    case
SYNDECLELEMTYPE_SQUARE_BRACKETS:
                                          uDeclAsteriskCount =
1;
                                          break;
                                    case
SYNDECLELEMTYPE_ASTERISK:
                                          uDeclAsteriskCount =
*((UINT *)pCurIdAAE->AAOperElem.pvValue);
                                          break;
                                    }
                              }
                              if (uDeclAsteriskCount == 0)
                                    semError("[",
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,"subscript
requires array or pointer type");
                              else
                                    uDeclAsteriskCount --;


      synAddNewIdCurStateElem(SYNIDCURSTATETYPE_INDEX_DEREFER
ENCING, NULL, pCurIdAAE, pNewContrElem, uIndLvlCountSqBrc);
                        }
                  }
                  break;
      case LEXCLASS_RT_IDENTIFIER:
            synCheckNextTermLexeme(LEXCLASS_RT_IDENTIFIER);
            pBuff = synCheckIdInRT(pCurLexListItem-
>pPrevLexemeInstance);
            switch (synRetNextLCValue())
                  {
                  case LEXCLASS_BLOCK_LEFT_ROUND_BRACKET:
```

```
      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_ROUND_BRACKE
T);
                        if (pBuff)
                        {
                              pNewContrElem =
synCreateNewContrElem(&cgContrElemFuncCall);
                              uType = pBuff-
>RTItemContent.pIdTableElem->uType;

      synAddNewOperElemToContrElem(SEMCLASS_FUNCTION, uType,
pNewContrElem,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum, NULL);


      synAddNewOperElemToContrElem(SEMCLASS_FUNCREFTOIDTABLE,
pBuff->RTItemContent.pIdTableElem->uType,
                                    pBuff-
>RTItemContent.pIdTableElem,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum, NULL);

                              pCurCEContainer =
pCurContrElem;
                              pCurCEArgContainer =
pCurContrElemArg;
                              pCurContrElem = pNewContrElem;
                              pCurContrElemArg = NULL;


      synAddNewOperElemToContrElem(SEMCLASS_FUNCREFTOIDTABLE,
pBuff->RTItemContent.pIdTableElem->uType,
                                    pBuff-
>RTItemContent.pIdTableElem,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum, NULL);
                        }

                        if ( synIsNextNontermArgEnum() )
                              synCheckNextNontermArgEnum();

                        pCurContrElem = pCurCEContainer;
                        pCurContrElemArg =
pCurCEArgContainer;

      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_ROUND_BRACK
ET);
                        break;
                  default:
                        if (pBuff)
                        if (pBuff-
>RTItemContent.pIdTableElem->pArgInstance)
                        {
                              uType = pBuff-
>RTItemContent.pIdTableElem->pArgInstance-
>pArgument.uLexClass;

                              if (pBegIdStateListElem)
                              {
                                    pBegIdStateContainer =
pBegIdStateListElem;
                                    pCurIdStateContainer =
pCurIdStateListElem;
                                    pCurIdAAEContainer =
pCurIdAAE;
                                    uDeclACContainer =
uDeclAsteriskCount;
                                    bPrevIsPointer =
bIsPointer;
                                    pBegAAElementContainer =
pBegAAElement;
                              }

                              pBegIdStateListElem = NULL;
                              pCurIdStateListElem = NULL;
                              pCurIdAAE = NULL;
                              uDeclAsteriskCount = 0;
                              bIsPointer = FALSE;
                              pBegAAElement = NULL;

                              pBegAAElement = pBuff-
>RTItemContent.pIdTableElem->pBegIdDecl;
                              pCurIdAAE = pBegAAElement;

                              while ( (uDeclAsteriskCount ==
0) && pCurIdAAE )
                              {
                                    switch (pCurIdAAE-
>AAOperElem.uDeclType)
                                    {
                                    case
SYNDECLELEMTYPE_SQUARE_BRACKETS:
                                          uDeclAsteriskCount =
1;
                                          break;
                                    case
SYNDECLELEMTYPE_ASTERISK:
                                          uDeclAsteriskCount =
*((UINT *)pCurIdAAE->AAOperElem.pvValue);
                                          break;
                                    }
                              }

                              pvValue = (UINT
*)malloc(sizeof(UINT));
                              *((UINT *)pvValue) =
uDeclAsteriskCount;

      synAddNewIdCurStateElem(SYNIDCURSTATETYPE_INITIAL_VALUE
S, pBuff->RTItemContent.pIdTableElem, pCurIdAAE, pvValue, 0);
                              bIsPointer = TRUE;


      while(synIsNextTermLexeme(LEXCLASS_BLOCK_LEFT_SQAURE_BR
ACKET))
```

```
                              {
                                    uIndLvlCountSqBrc ++;


      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_SQAURE_BRACK
ET);

                                    pNewContrElem =
synCreateNewContrElem(&cgContrElemEquation);

                                    pCurCEContainer =
pCurContrElem;
                                    pCurCEArgContainer =
pCurContrElemArg;

                                    pCurContrElem =
pNewContrElem;
                                    pCurContrElemArg = NULL;

                                    bIsPointerContainer =
bIsPointer;

      synCheckNextNontermEquation();

                                    if (bIsPointer)
                                          semError("[",

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,
                                                "illegal
index, indirection not allowed");
                                    else
                                          bIsPointer =
bIsPointerContainer;

                                    pCurContrElem =
pCurCEContainer;
                                    pCurContrElemArg =
pCurCEArgContainer;


      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_SQAURE_BRAC
KET);

                                    while ( pCurIdAAE &&
(uDeclAsteriskCount == 0) && (pCurIdAAE->pNextAAElem) )
                                    {
                                          pCurIdAAE =
pCurIdAAE->pNextAAElem;
                                          switch (pCurIdAAE-
>AAOperElem.uDeclType)
                                          {
                                          case
SYNDECLELEMTYPE_SQUARE_BRACKETS:

      uDeclAsteriskCount = 1;
                                                break;
                                          case
SYNDECLELEMTYPE_ASTERISK:

      uDeclAsteriskCount = *((UINT *)pCurIdAAE-
>AAOperElem.pvValue);
                                                break;
                                          }
                                    }

                                    if (uDeclAsteriskCount ==
0)
                                          semError("[",

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,
                                                "subscript
requires array or pointer type");
                                    else
                                          uDeclAsteriskCount -
-;


      synAddNewIdCurStateElem(SYNIDCURSTATETYPE_INDEX_DEREFER
ENCING, NULL, pCurIdAAE, pNewContrElem, uIndLvlCountSqBrc);

                              }

                              uSemClass = SEMCLASS_MEM16;

      synAddNewOperElemToContrElem(uSemClass, uType,
pBegIdStateListElem,
                                          pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                          pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                                          pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum, NULL);
                        }
                        else
                              synIdError(pCurLexListItem-
>pPrevLexemeInstance, "unacceptable use of this element");
                        break;
                  }
                  break;
      }

      uIndLvlCountAster = uExspressionAsteriskCount;
      while (uExspressionAsteriskCount>0)
      {
            while ( pCurIdAAE && (uDeclAsteriskCount == 0) &&
(pCurIdAAE->pNextAAElem))
            {
                  pCurIdAAE = pCurIdAAE->pNextAAElem;
                  switch (pCurIdAAE->AAOperElem.uDeclType)
                  {
                  case SYNDECLELEMTYPE_SQUARE_BRACKETS:
                        uDeclAsteriskCount = 1;
                        break;
                  case SYNDECLELEMTYPE_ASTERISK:
                        uDeclAsteriskCount = *((UINT
*)pCurIdAAE->AAOperElem.pvValue);
                        break;
                  }
            }
            if (uDeclAsteriskCount == 0)
                  break;
            uDeclAsteriskCount--;
            uExspressionAsteriskCount--;
      }
      if (uIndLvlCountAster)
      {
            pvValue = (UINT *)malloc(sizeof(UINT));
            *((UINT *)pvValue) = uDeclAsteriskCount;

      synAddNewIdCurStateElem(SYNIDCURSTATETYPE_DEREFERENCING
, NULL, pCurIdAAE, pvValue, uIndLvlCountAster);
      }

      if (uExspressionAsteriskCount > 0)
      {
            semError("*",
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"illegal indirection");
      }
      else
      {
            if ((pCurIdAAE && !pCurIdAAE->pNextAAElem &&
uDeclAsteriskCount == 0) || !pBegAAElement)
            {
                  bIsPointer = FALSE;
            }
      }

      if (bPrevIsPointer && bIsPointer)
      {
            semError("expression",
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"cannot operate with two or more
pointers");
      }
      else
            if (pCurIdStateContainer)
            {
                  pBegIdStateListElem = pBegIdStateContainer;
                  pCurIdStateListElem = pCurIdStateContainer;
                  pCurIdAAE = pCurIdAAEContainer;
                  uDeclAsteriskCount = uDeclACContainer ;
                  bIsPointer = bPrevIsPointer;
                  pBegAAElement = pBegAAElementContainer;
            }
      return uType;
}

      synError();
      return FALSE;
```

```
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermTransVarMul(UINT uType)
{
    UINT uLexClass = 0;
      UINT uSemClass = 0;
      int * pvValue;

      if (uType > uCurType)
      {
            uCurType = uType;
      }

      if ( synIsNextTermLexeme(LEXCLASS_OPERATION_ASTERISK)
|| synIsNextTermLexeme(LEXCLASS_OPERATION_SLASH) )
      {
            if (synIsNextTermLexeme(
LEXCLASS_OPERATION_ASTERISK ))
                  uLexClass =
synCheckNextTermLexeme(LEXCLASS_OPERATION_ASTERISK);
            if (synIsNextTermLexeme( LEXCLASS_OPERATION_SLASH
))
                  uLexClass =
synCheckNextTermLexeme(LEXCLASS_OPERATION_SLASH);

            switch(semRetTypeSizeByType(uType))
            {
            case 4:
                  uSemClass = SEMCLASS_IMM32;
                  break;
            case 2:
                  uSemClass = SEMCLASS_IMM16;
                  break;
            case 1:
                  uSemClass = SEMCLASS_IMM8;
                  break;
            }
            uCurSemClass = uSemClass;

            pvValue = (int*)malloc(sizeof(int));
            *pvValue = uDeclAsteriskCount;
            if(pCurIdStateListElem)

      synAddNewIdCurStateElem(SYNIDCURSTATETYPE_OPERATION,
NULL, pCurIdAAE, pvValue, 0);

            synAddNewOperElemToContrElem(SEMCLASS_OPERATION,
uLexClass, pCurIdStateListElem,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      synCheckNextNontermTransVarMul(synCheckNextNontermTrans
VarBrac());
      }
    return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermTransVarPlus(UINT uType)
{
    UINT uLexClass = 0;
      UINT uSemClass = 0;
      int * pvValue  = NULL;
      int * pvValue2 = NULL;
      TIDCurStateListPtr pPlusCurStateElem;

      if (uType > uCurType)
      {
            uCurType = uType;
      }
      if (synIsNextTermLexeme( LEXCLASS_OPERATION_PLUS ) ||
synIsNextTermLexeme( LEXCLASS_OPERATION_MINUS ))
    {
        if (synIsNextTermLexeme( LEXCLASS_OPERATION_PLUS ))
                  uLexClass =
synCheckNextTermLexeme(LEXCLASS_OPERATION_PLUS);
            if (synIsNextTermLexeme( LEXCLASS_OPERATION_MINUS
))
                  uLexClass =
synCheckNextTermLexeme(LEXCLASS_OPERATION_MINUS);

            if(!bIsPointer)
            {
                  switch(semRetTypeSizeByType(uType))
                  {
                  case 4:
                        uSemClass = SEMCLASS_IMM32;
                        break;
                  case 2:
                        uSemClass = SEMCLASS_IMM16;
                        break;
                  case 1:
                        uSemClass = SEMCLASS_IMM8;
                        break;
                  }
                  uCurSemClass = uSemClass;
            }

            pvValue = (int*)malloc(sizeof(int));
            *pvValue = uDeclAsteriskCount;

            if(pCurIdStateListElem)

      synAddNewIdCurStateElem(SYNIDCURSTATETYPE_OPERATION,
NULL, pCurIdAAE, pvValue, 0);

            synAddNewOperElemToContrElem(SEMCLASS_OPERATION,
uLexClass, pPlusCurStateElem,
                  pCurLexListItem->pPrevLexemeInstance-
```

```
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);


      synCheckNextNontermTransVarPlus(synCheckNextNontermTran
sVarMul(synCheckNextNontermTransVarBrac()));
      }
    return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermVarEqual()
{
      extern char bIsPointer;
      char bIsEquationPointerType = 0;
      extern TIDCurStateListPtr pBegIdStateListElem;
      TIDCurStateListPtr pTempIdSLE = NULL;
      TContrElemPtr pCurCEContainer = NULL, pNewContrElem =
NULL;
      TOperationElemPtr pCurCEArgContainer = NULL;
      UINT uType;

      pBegIdStateListElem = NULL;
      pCurIdStateListElem = NULL;
      uCurSemClass = SEMCLASS_MEM16;
      uCurType = LEXCLASS_TYPE_VOID;

      pNewContrElem =
synCreateNewContrElem(&cgContrElemExpression);
      if (pCurLexListItem)
      synAddNewOperElemToContrElem(SEMCLASS_UNKNOWN,
LEXCLASS_UNKNOWN, pNewContrElem,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      pCurCEContainer = pCurContrElem;
      pCurCEArgContainer = pCurContrElemArg;

      pCurContrElem = pNewContrElem;
      pCurContrElemArg = NULL;

      bIsPointer = 0;

      uType =
synCheckNextNontermTransVarPlus(synCheckNextNontermTransVarMu
l(synCheckNextNontermTransVarBrac()));
      pCurContrElem = pCurCEContainer;
      pCurContrElemArg = pCurCEArgContainer;
      if (pCurContrElemArg)
      {
            pCurContrElemArg->pArgument.uLexClass = uType;
            pCurContrElemArg->pArgument.uSemClass =
uCurSemClass;
            pCurContrElemArg->pArgument.pIDCurState =
pBegIdStateListElem;
      }
      pTempIdSLE = pBegIdStateListElem;

      if(bIsPointer)
            bIsEquationPointerType = TRUE;

      uCurSemClass = SEMCLASS_MEM16;
      uType = uCurType;
      uCurType = LEXCLASS_TYPE_VOID;

      if (synIsNextTermLexeme( LEXCLASS_COMPARISON_EQUAL ))
      {

      synCheckNextTermLexeme(LEXCLASS_COMPARISON_EQUAL);
            pBegIdStateListElem = NULL;

            synCheckNextNontermVarEqual();
            if(bIsPointer)
                  bIsEquationPointerType = TRUE;
      }
      bIsPointer = bIsEquationPointerType;
      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermEquation()
{
      char  bPrevIsPointer = 0;
      UINT uType;
      TAAOperElemListPtr            pCurIdAAEContainer
      = NULL;
      TIDCurStateListPtr      pBegIdStateContainer    = NULL;
      TIDCurStateListPtr      pCurIdStateContainer    = NULL;
      UINT                    uDeclACContainer = 0;

      pBegIdStateContainer = pBegIdStateListElem;
      pCurIdStateContainer = pCurIdStateListElem;
      pCurIdAAEContainer = pCurIdAAE;
      uDeclACContainer = uDeclAsteriskCount;

      pBegIdStateListElem = NULL;
      pCurIdStateListElem = NULL;
      pCurIdAAE = NULL;
      uDeclAsteriskCount = 0;
      bIsPointer = 0;

      uType = synCheckNextNontermVarEqual();

      pBegIdStateListElem = pBegIdStateContainer;
      pCurIdStateListElem = pCurIdStateContainer;
      pCurIdAAE = pCurIdAAEContainer;
      uDeclAsteriskCount = uDeclACContainer;
      return uType;
};

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAddReturnContrElem()
{
      TContrElemPtr pCurCEContainer = NULL, pNewContrElem =
NULL;
      TOperationElemPtr pCurCEArgContainer = NULL;
      UINT uType;

      pNewContrElem =
synCreateNewContrElem(&cgContrElemEquation);
      synAddNewOperElemToContrElem(SEMCLASS_EQUATION,
LEXCLASS_TYPE_VOID, pNewContrElem,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      pCurCEContainer = pCurContrElem;
      pCurCEArgContainer = pCurContrElemArg;

      pCurContrElem = pNewContrElem;
      pCurContrElemArg = NULL;

      uType = synCheckNextNontermEquation();
      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_SEMICOLUMN)
;
      pCurContrElem = pCurCEContainer;
      pCurContrElemArg = pCurCEArgContainer;

      pCurContrElemArg->pArgument.uLexClass = uType;
      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermInstruction()
{
      switch(synRetNextLCValue())
      {
      case LEXCLASS_IA32INSTRUCTIONS_MOV:
      case LEXCLASS_IA32INSTRUCTIONS_PUSH:
      case LEXCLASS_IA32INSTRUCTIONS_JMP:
      case LEXCLASS_IA32INSTRUCTIONS_INT:
            return TRUE;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermSegmentRegister()
{
      switch(synRetNextLCValue())
      {
      case LEXCLASS_IA32REGISTERS_CS:
      case LEXCLASS_IA32REGISTERS_DS:
      case LEXCLASS_IA32REGISTERS_ES:
      case LEXCLASS_IA32REGISTERS_SS:
      case LEXCLASS_IA32REGISTERS_GS:
      case LEXCLASS_IA32REGISTERS_FS:
            return TRUE;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermByteGPRegister()
{
      switch(synRetNextLCValue())
      {
      case LEXCLASS_IA32REGISTERS_AH:
      case LEXCLASS_IA32REGISTERS_AL:
      case LEXCLASS_IA32REGISTERS_BH:
      case LEXCLASS_IA32REGISTERS_BL:
      case LEXCLASS_IA32REGISTERS_CH:
      case LEXCLASS_IA32REGISTERS_CL:
      case LEXCLASS_IA32REGISTERS_DH:
      case LEXCLASS_IA32REGISTERS_DL:
            return TRUE;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
```

```
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermWordBaseRegister()
{
      switch(synRetNextLCValue())
      {
      case LEXCLASS_IA32REGISTERS_BX:
      case LEXCLASS_IA32REGISTERS_BP:
            return TRUE;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermWordPointerRegister()
{
      switch(synRetNextLCValue())
      {
      case LEXCLASS_IA32REGISTERS_SI:
      case LEXCLASS_IA32REGISTERS_DI:
            return TRUE;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermWordGPRegister()
{
      switch(synRetNextLCValue())
      {
      case LEXCLASS_IA32REGISTERS_AX:
      case LEXCLASS_IA32REGISTERS_BX:
      case LEXCLASS_IA32REGISTERS_CX:
      case LEXCLASS_IA32REGISTERS_DX:
      case LEXCLASS_IA32REGISTERS_SI:
      case LEXCLASS_IA32REGISTERS_DI:
      case LEXCLASS_IA32REGISTERS_BP:
      case LEXCLASS_IA32REGISTERS_SP:
            return TRUE;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synIsNextNontermDWordGPRegister()
{
      switch(synRetNextLCValue())
      {
      case LEXCLASS_IA32REGISTERS_EAX:
      case LEXCLASS_IA32REGISTERS_EBX:
      case LEXCLASS_IA32REGISTERS_ECX:
      case LEXCLASS_IA32REGISTERS_EDX:
      case LEXCLASS_IA32REGISTERS_ESI:
      case LEXCLASS_IA32REGISTERS_EDI:
      case LEXCLASS_IA32REGISTERS_EBP:
      case LEXCLASS_IA32REGISTERS_ESP:
            return TRUE;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*     Checks whether a lexeme is a proper Segment Register
(SREG)
*     If it's not casts an syntactical error.
*     Moves the Lexeme List pointer forward.
*
*   @author Nail Sharipov
*/
int synCheckNextNontermSegmentRegister()
{
      UINT uType;
      switch (uType = synGetNextLexemeClass())
      {
      case LEXCLASS_IA32REGISTERS_CS:
      case LEXCLASS_IA32REGISTERS_DS:
      case LEXCLASS_IA32REGISTERS_ES:
      case LEXCLASS_IA32REGISTERS_SS:
      case LEXCLASS_IA32REGISTERS_GS:
      case LEXCLASS_IA32REGISTERS_FS:
        return uType;
        break;
    default:
        synError();
            return FALSE;
            break;
      }
}

/*
*     Checks whether a lexeme is a proper
*     byte (8-bit long) General Purpose Register
*     If it's not casts an syntactical error.
*     Moves the Lexeme List pointer forward.
*
*   @author Nail Sharipov
*/
int synCheckNextNontermByteGPRegister()
{
    UINT uType;
      switch (uType = synGetNextLexemeClass())
      {
      case LEXCLASS_IA32REGISTERS_AH:
      case LEXCLASS_IA32REGISTERS_AL:
      case LEXCLASS_IA32REGISTERS_BH:
      case LEXCLASS_IA32REGISTERS_BL:
      case LEXCLASS_IA32REGISTERS_CH:
      case LEXCLASS_IA32REGISTERS_CL:
      case LEXCLASS_IA32REGISTERS_DH:
      case LEXCLASS_IA32REGISTERS_DL:
        return uType;
        break;
    default:
        synError();
            return FALSE;
            break;
      }
}

/*
*     Checks whether a lexeme is a proper
*     word (16-bit long) General Purpose Register
*     If it's not casts an syntactical error.
*     Moves the Lexeme List pointer forward.
*
*   @author Nail Sharipov
*/
int synCheckNextNontermWordGPRegister()
{
      UINT uType;
      switch (uType = synGetNextLexemeClass())
      {
      case LEXCLASS_IA32REGISTERS_AX:
      case LEXCLASS_IA32REGISTERS_BX:
      case LEXCLASS_IA32REGISTERS_CX:
      case LEXCLASS_IA32REGISTERS_DX:
      case LEXCLASS_IA32REGISTERS_SI:
      case LEXCLASS_IA32REGISTERS_DI:
      case LEXCLASS_IA32REGISTERS_BP:
      case LEXCLASS_IA32REGISTERS_SP:
        return uType;
        break;
    default:
        synError();
            return FALSE;
            break;
      }
}

/*
*     Checks whether a lexeme is a proper
*     word (16-bit long) Pointer Register
*     If it's not casts an syntactical error.
*     Moves the Lexeme List pointer forward.
*
*   @author Nail Sharipov
```

```
*/
int synCheckNextNontermWordPointerRegister()
{
      UINT uType;
      switch (uType = synGetNextLexemeClass())
      {
      case LEXCLASS_IA32REGISTERS_SI:
      case LEXCLASS_IA32REGISTERS_DI:
            return uType;
        break;
    default:
        synError();
            return FALSE;
            break;
      }
}

/*
*     Checks whether a lexeme is a proper
*     word (16-bit long) Base Register to
*     specify the address (BX, BP)
*     If it's not casts an syntactical error.
*     Moves the Lexeme List pointer forward.
*
*   @author Nail Sharipov
*/
int synCheckNextNontermWordBaseRegister()
{
      UINT uType;
      switch (uType = synGetNextLexemeClass())
      {
      case LEXCLASS_IA32REGISTERS_BX:
      case LEXCLASS_IA32REGISTERS_BP:
            return uType;
        break;
    default:
        synError();
            return FALSE;
            break;
      }
}

/*
*     Checks whether a lexeme is a proper
*     double-word (32-bit long) General Purpose Register.
*     If it's not casts an syntactical error.
*     Moves the Lexeme List pointer forward.
*
*   @author Nail Sharipov
*/
int synCheckNextNontermDWordGPRegister()
{
      UINT uType;
      switch (uType = synGetNextLexemeClass())
      {
      case LEXCLASS_IA32REGISTERS_EAX:
      case LEXCLASS_IA32REGISTERS_EBX:
      case LEXCLASS_IA32REGISTERS_ECX:
      case LEXCLASS_IA32REGISTERS_EDX:
      case LEXCLASS_IA32REGISTERS_ESI:
      case LEXCLASS_IA32REGISTERS_EDI:
      case LEXCLASS_IA32REGISTERS_EBP:
      case LEXCLASS_IA32REGISTERS_ESP:
        return uType;
        break;
    default:
        synError();
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermInstruction()
{
    UINT uType;
      switch (uType = synGetNextLexemeClass())
    {
      case LEXCLASS_IA32INSTRUCTIONS_MOV:
      case LEXCLASS_IA32INSTRUCTIONS_PUSH:
        return uType;
        break;
    default:
        synError();
        return FALSE;
        break;
    }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
// Sreg:[...
int synAddPrefixSregOEGroup(UINT uType)
{
      TOperationElemPtr pTempOE = NULL;

      void * pvValue;
      UINT uSemClass = 0;

      pTempOE =
(TOperationElemPtr)malloc(sizeof(TOperationElem));

      pvValue = (int*)malloc(1);
      *((int *)pvValue) = semRetSregPrefixByType(uType);

      pTempOE->pNextOperationElem = pCurContrElem-
>pBegArgList;
      pCurContrElem->pBegArgList->pPrevOperationElem =
pTempOE;
      pCurContrElem->pBegArgList = pTempOE;


      pTempOE->pArgument.uSemClass = SEMCLASS_PREFIX;
      pTempOE->pArgument.pvValue = pvValue;
      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
// Sreg
int synAddSregOEGroup(UINT uType)
{
      void * pvValue;
      UINT uSemClass = 0;

      pvValue = (int*)malloc(1);
      *((int *)pvValue) = semRetSregOpcodeByType(uType);
      uSemClass = SEMCLASS_SYSREG;
      synAddNewOperElemToContrElem(uSemClass, uType, pvValue,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
//   R16,
int synAddGPRegXXRegOpcodeOEGroup(UINT uType)
{
      void * pvValue;
      UINT uSemClass = 0;

      pvValue = (int*)malloc(1);
      *((int *)pvValue) =
semRetRegRegisterOpcodeByType(uType);
      switch(semRetTypeSizeByType(uType))
      {
      case 4:
            uSemClass = SEMCLASS_REG32;
            break;
      case 2:
            uSemClass = SEMCLASS_REG16;
            break;
      case 1:
            uSemClass = SEMCLASS_REG8;
            break;
      default:
            uSemClass = SEMCLASS_UNKNOWN;
            semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, 0,"unknown type");
            break;
      }

      synAddNewOperElemToContrElem(uSemClass, uType, pvValue,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
//   ,R16
int synAddGPRegXXEffAddrOEGroup(UINT uType)
{
      void * pvValue;
      UINT uSemClass = 0;

      pvValue = (int*)malloc(1);
      *((int *)pvValue) = semRetRegEffectivAddrByType(uType);
      switch(semRetTypeSizeByType(uType))
      {
      case 4:
            uSemClass = SEMCLASS_REG32;
            break;
      case 2:
            uSemClass = SEMCLASS_REG16;
            break;
      case 1:
            uSemClass = SEMCLASS_REG8;
            break;
      default:
            uSemClass = SEMCLASS_UNKNOWN;
            semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, 0,"unknown type");
            break;
      }
      synAddNewOperElemToContrElem(uSemClass, uType, pvValue,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      return uType;
}

/*
*
*
```

```
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
//   RXX,
int synAddGPRegXXRWCodeOEGroup(UINT uType)
{
      void * pvValue;
      UINT uSemClass = 0;

      pvValue = (int*)malloc(1);
      *((int *)pvValue) = semRetRegRWByType(uType);
      switch(semRetTypeSizeByType(uType))
      {
      case 4:
            uSemClass = SEMCLASS_REG32;
            break;
      case 2:
            uSemClass = SEMCLASS_REG16;
            break;
      case 1:
            uSemClass = SEMCLASS_REG8;
            break;
      default:
            uSemClass = SEMCLASS_UNKNOWN;
            semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, 0,"unknown type");
            break;
      }
      synAddNewOperElemToContrElem(uSemClass, uType, pvValue,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
//  ,VarXX
int synAddVarXXOEGroup(UINT uType, TRTItemPtr pBuff)
{

      void * pvValue;
      UINT uSemClass = 0;


      switch(semRetTypeSizeByType(uType))
      {
      case 4:
            uSemClass = SEMCLASS_VAR32;
            break;
      case 2:
            uSemClass = SEMCLASS_VAR16;
            break;
      case 1:
            uSemClass = SEMCLASS_VAR8;
            break;
      default:
            uSemClass = SEMCLASS_UNKNOWN;
            semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, 0,"unknown type");
            break;
      }

      pvValue = malloc(sizeof(TIdTablePtr));
      pvValue = pBuff->RTItemContent.pIdTableElem;

      synAddNewOperElemToContrElem(uSemClass, uType, pvValue,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
//  ,0x0000
int synAddImmOEGroup(UINT uType)
{
      char * cValue;
      void * pvValue;
      UINT uSemClass = 0;

      switch(semRetTypeSizeByType(uType))
      {
      case 4:
            uSemClass = SEMCLASS_IMM32;
            break;
      case 2:
            uSemClass = SEMCLASS_IMM16;
            break;
      case 1:
            uSemClass = SEMCLASS_IMM8;
            break;
      default:
            uSemClass = SEMCLASS_UNKNOWN;
            semError("immediate value",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"incorrect");
            break;
      }

      pvValue = (UINT *)malloc(sizeof(UINT));
      cValue = &pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.cpTextFrag[2];

      *((UINT*)pvValue) = strtol(cValue, NULL, 16);
      synAddNewOperElemToContrElem(uSemClass, uType, pvValue,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
// MOV [Disp16],
int synAdd_Disp16_MemAddrOEGroup(UINT uType)
{
      char * cValue;
      void * pvValue;
      UINT uSemClass = 0;

      switch(semRetTypeSizeByType(uType))
      {
      case 4:
            //uSemClass = EA16MEMLOC_DISP32;
            semError("adrress",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"there is no 32-bit address mode
support yet");
            break;
      case 2:
            uSemClass = EA16MEMLOC_DISP16;
            break;
      case 1:
            //uSemClass = EA16MEMLOC_DISP8;
            semError("adrress",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"must not be less, than 16-bit
value");
            break;
      default:
            uSemClass = SEMCLASS_UNKNOWN;
            semError("adrress",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"out of range");
            break;
      };

      pvValue = (UINT *)malloc(sizeof(UINT));
      cValue = &pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.cpTextFrag[2];

      *((UINT*)pvValue) = strtol(cValue, NULL, 16);

      synAddNewOperElemToContrElem(uSemClass, uType, pvValue,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
// MOV [Base],
int synAdd_Base_MemAddrOEGroup(UINT uBaseType)
{
      UINT uType = 0;
      void * pvValue;
      UINT uSemClass = 0;

      pvValue = (int*)malloc(1);
      switch (uBaseType)
      {
      case LEXCLASS_IA32REGISTERS_BX:
            uSemClass = EA16MEMLOC_BX;
            break;
      case LEXCLASS_IA32REGISTERS_BP:
            uSemClass = EA16MEMLOCPART_BP;
            break;
      }

      synAddNewOperElemToContrElem(uSemClass, uType, pvValue,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
// MOV [Index],
int synAdd_Index_MemAddrOEGroup(UINT uIndexType)
{
      UINT uType = 0;
      void * pvValue;
      UINT uSemClass = 0;

      pvValue = (int*)malloc(1);
      switch (uIndexType)
      {
      case LEXCLASS_IA32REGISTERS_SI:
            uSemClass = EA16MEMLOC_SI;
            break;
      case LEXCLASS_IA32REGISTERS_DI:
            uSemClass = EA16MEMLOC_DI;
            break;
      }
      synAddNewOperElemToContrElem(uSemClass, uType, pvValue,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
            pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
// [Index+Disp16],
// [Index],
int synAdd_Index_Disp16_MemAddrOEGroup(UINT uType)
{
      UINT uSemClass = 0;
      UINT uIndexType = 0;

      uSemClass = SEMCLASS_REG16;
      uIndexType = pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLexClass;

      if (synIsNextTermLexeme(LEXCLASS_OPERATION_PLUS))
      {
            synCheckNextTermLexeme(LEXCLASS_OPERATION_PLUS);
            synAdd_Index_MemAddrOEGroup(uIndexType);

            uType = synCheckNextNontermValue();
            synAdd_Disp16_MemAddrOEGroup(uType);
      }
      else
      {
            synAdd_Index_MemAddrOEGroup(uIndexType);
      }
      return uType;
}
```

```
/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
// [Base+Index+Disp16],
// [Base+Index],
// [Base+Disp16],
// [Base],
int synAdd_Base_Index_Disp16_MemAddrOEGroup(UINT uType)
{
      UINT uSemClass = 0;
      UINT uBaseType = 0;

      uSemClass = SEMCLASS_REG16;
      uBaseType = pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLexClass;
      if (synIsNextTermLexeme(LEXCLASS_OPERATION_PLUS))
      {
            synCheckNextTermLexeme(LEXCLASS_OPERATION_PLUS);

            synAdd_Base_MemAddrOEGroup(uBaseType);
            if (synIsNextNontermValue())
            {
                  uType = synCheckNextNontermValue();
                  synAdd_Disp16_MemAddrOEGroup(uType);
            };

            if (synIsNextNontermWordPointerRegister())
            {
                  uType =
synCheckNextNontermWordPointerRegister();
                  synAdd_Index_Disp16_MemAddrOEGroup(uType);
            };
      }
      else
      {
            if (uBaseType == LEXCLASS_IA32REGISTERS_BP )
            {
                  semError("BP", pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                       pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                                       pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,
                                       "unacceptable use of
base register in addressing arithmetic");
            }
            synAdd_Base_MemAddrOEGroup(uBaseType);
      }
      return uType;
}

/*
*
*
*   @author Nail Sharipov
*/
int synAddMemoryLocationOEGroup()
{
      UINT uType;
      if (synIsNextNontermValue())
      {
            uType = synCheckNextNontermValue();
            synAdd_Disp16_MemAddrOEGroup(uType);
      };
      if (synIsNextNontermWordPointerRegister())
      {

            uType = synCheckNextNontermWordPointerRegister();
            synAdd_Index_Disp16_MemAddrOEGroup(uType);
      }
      if (synIsNextNontermWordBaseRegister())
      {
            uType = synCheckNextNontermWordBaseRegister();
            synAdd_Base_Index_Disp16_MemAddrOEGroup(uType);
      }

      return uType;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAsmOperandGroup_SREG(UINT uType, TOperationElemPtr
pOpcode)
{
      TRTItemPtr pBuff;
      UINT uSemClass = 0;

      synAddSregOEGroup(uType);
      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA);

      // MOV Sreg, wordGPReg
      if(synIsNextNontermWordGPRegister())
      {
            uType = synCheckNextNontermWordGPRegister();
            synAddGPRegXXEffAddrOEGroup(uType);
            pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_Sreg_GPReg16;
            return TRUE;
      }

      if (synIsNextNontermByteGPRegister())
      {
            synCheckNextNontermByteGPRegister();
            semError("MOV",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, 0,
                  "operand size mismatch");
            return FALSE;
      }

      if (synIsNextNontermDWordGPRegister())
      {
            synCheckNextNontermDWordGPRegister();
            semError("MOV",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, 0,
                  "operand size mismatch");
            return FALSE;
      }
      // MOV Sreg, Mem16
      if (synIsNextTermLexeme(LEXCLASS_RT_IDENTIFIER))
      {
            synCheckNextTermLexeme(LEXCLASS_RT_IDENTIFIER);
            pBuff = synCheckIdInRT(pCurLexListItem-
>pPrevLexemeInstance);
            uSemClass = SEMCLASS_VAR16;

            uType = pBuff->RTItemContent.pIdTableElem-
>pArgInstance->pArgument.uLexClass;
            if (semRetTypeSizeByType(uType) !=4 )
            {
                  synAddVarXXOEGroup(uType, pBuff);
                  pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_Sreg_MEM16;
                  return TRUE;
            }
            else
            {
                  //if not MEM16
                  semError("MOV",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, 0,
                        "operand size mismatch");
            }
      }

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAsmOperandGroup_MEMXX(UINT uType, TOperationElemPtr
pOpcode)
{
      UINT uType2 = 0;
      UINT uSemClass = 0;

      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_SQAURE_BRAC
KET);
      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA);

      // MOV memXX, immXX
      if(synIsNextNontermValue())
      {

      uType2 = synCheckNextNontermValue();
      synAddImmOEGroup(uType2);
      switch(semRetTypeSizeByType(uType2))
      {
      case 4:
            pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_MEM32_IMM32;
            break;
      case 2:
            pOpcode->pArgument.uSemClass = 
SEMCLASS_INSTRUCTION_MEM16_IMM16;
            break;
      case 1:
            pOpcode->pArgument.uSemClass = 
SEMCLASS_INSTRUCTION_MEM8_IMM8;
            break;
      default:
            uSemClass = SEMCLASS_UNKNOWN;
            semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                          pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd, 0,"unknown 
type");

            break;
      }
      return TRUE;
      }

      // MOV memXX, GPRegXX
      if (synIsNextNontermByteGPRegister())
      {
            uType2 = synCheckNextNontermByteGPRegister();
            synAddGPRegXXRegOpcodeOEGroup(uType2);
            pOpcode->pArgument.uSemClass = 
SEMCLASS_INSTRUCTION_MEM8_GPReg8;
            return TRUE;
      }


      if(synIsNextNontermWordGPRegister())
      {
            uType2 = synCheckNextNontermWordGPRegister();
            synAddGPRegXXRegOpcodeOEGroup(uType2);
            pOpcode->pArgument.uSemClass = 
SEMCLASS_INSTRUCTION_MEM16_GPReg16;
            return TRUE;
      }

      if (synIsNextNontermDWordGPRegister())
      {
            uType2 = synCheckNextNontermDWordGPRegister();
            synAddGPRegXXRegOpcodeOEGroup(uType2);
            pOpcode->pArgument.uSemClass = 
SEMCLASS_INSTRUCTION_MEM32_GPReg32;
            return TRUE;
      }

      synError();
      return FALSE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
```

```
int synAsmOperandGroup_IMMXX(TOperationElemPtr pOpcode)
{
      UINT uType2 = 0;
      UINT uSemClass = 0;
      TRTItemPtr pBuff;

      if(synIsNextNontermValue())
      {
            uType2 = synCheckNextNontermValue();

            synAddImmOEGroup(uType2);
            switch(semRetTypeSizeByType(uType2))
            {
            case 4:
                  pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_IMM32;
                  break;
            case 2:
                  pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_IMM16;
                  break;
            case 1:
                  pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_IMM8;
                  break;
            default:
                  uSemClass = SEMCLASS_UNKNOWN;
                  semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                          pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd, 0,"unknown
type");
                  break;
            }
      return TRUE;
      }
      return FALSE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAsmOperandGroup_PTR16_16(TOperationElemPtr pOpcode)
{
      UINT uType1 = 0, uType2 = 0;
      UINT uSemClass = 0;


      if(synIsNextNontermValue())
      {
            uType1 = synCheckNextNontermValue();

            synAddImmOEGroup(uType1);
            switch(semRetTypeSizeByType(uType1))
            {
            case 4:
                  semError("PTR 16:16",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"immediate value to large for this
addressing mode");
                  break;
            case 2:
            case 1:
                  break;
            default:
                  uSemClass = SEMCLASS_UNKNOWN;
                  semError("PTR 16:16",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"incorrect immediate value");
                  break;
            }


      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COLON);

            uType2 = synCheckNextNontermValue();

            synAddImmOEGroup(uType2);
            switch(semRetTypeSizeByType(uType2))
            {
            case 4:
                  semError("PTR 16:16",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"immediate value to large for this
addressing mode");
                  break;
            case 2:
            case 1:
                  break;
            default:
                  uSemClass = SEMCLASS_UNKNOWN;
                  semError("PTR 16:16",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"incorrect immediate value");
                  break;
            }
            pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_PTR16_16;
            return TRUE;
      }
      return FALSE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
```

```
int synAsmOperandGroup_REGXX(UINT uType, TOperationElemPtr
pOpcode)
{
      UINT uType2 = 0;
      UINT uSemClass = 0;
      TRTItemPtr pBuff;

      if (synIsNextNontermWordGPRegister() ||
synIsNextNontermByteGPRegister() ||
synIsNextNontermDWordGPRegister())
      {

            if (synIsNextNontermDWordGPRegister())
                  uType2 =
synCheckNextNontermDWordGPRegister();

            if (synIsNextNontermWordGPRegister())
                  uType2 =
synCheckNextNontermWordGPRegister();

            if (synIsNextNontermByteGPRegister())
                  uType2 =
synCheckNextNontermByteGPRegister();

            if ( semRetTypeSizeByType(uType2) !=
semRetTypeSizeByType(uType) )
                  semError ("MOV",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,
                  "register size mismatch");
            else
            {
                  synAddGPRegXXRegOpcodeOEGroup(uType);
                  synAddGPRegXXEffAddrOEGroup(uType2);

                  switch(semRetTypeSizeByType(uType2))
                  {
                  case 4:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG32_REG32;
                        break;
                  case 2:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG16_REG16;
                        break;
                  case 1:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG8_REG8;
                        break;
                  default:
                        uSemClass = SEMCLASS_UNKNOWN;

      semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                              pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd,
                              pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uLineNum,"unknown
type");
                        break;
                  }
            }
            return TRUE;
      }

      // MOV Rxx, immX
      if(synIsNextNontermValue())
      {
            uType2 = synCheckNextNontermValue();

            if ( semRetTypeSizeByType(uType2) !=
semRetTypeSizeByType(uType) )
                  semError ("MOV",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd, 0,
                                    "immediate value out of
range");
            else
            {
                  synAddGPRegXXRWCodeOEGroup(uType);
                  synAddImmOEGroup(uType2);
                  switch(semRetTypeSizeByType(uType2))
                  {
                  case 4:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG32_IMM32;
                        break;
                  case 2:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG16_IMM16;
                        break;
                  case 1:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG8_IMM8;
                        break;
                  default:
                        uSemClass = SEMCLASS_UNKNOWN;

      semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, 0,"unknown type");
                        break;
                  }
                  return TRUE;
            }
      return TRUE;
      }

      // MOV Rxx, VARxx
      if (synIsNextTermLexeme(LEXCLASS_RT_IDENTIFIER))
      {
            synCheckNextTermLexeme(LEXCLASS_RT_IDENTIFIER);
            pBuff = synCheckIdInRT(pCurLexListItem-
>pPrevLexemeInstance);
            if (pBuff)
                  uType2 = pBuff->RTItemContent.pIdTableElem-
>pArgInstance->pArgument.uLexClass;
            else
                  return TRUE;

            if ( semRetTypeSizeByType(uType2) !=
semRetTypeSizeByType(uType) )
```

```
                  semError ("MOV",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,
                                    pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosEnd, 0,
                                    "bad variable type for
this register");
            else
            {
                  synAddGPRegXXRegOpcodeOEGroup(uType);
                  synAddVarXXOEGroup(uType2, pBuff);
                  switch(semRetTypeSizeByType(uType2))
                  {
                  case 4:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG32_MEM32;
                        break;
                  case 2:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG16_MEM16;
                        break;
                  case 1:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG8_MEM8;
                        break;
                  default:
                        uSemClass = SEMCLASS_UNKNOWN;

      semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd, 0,"unknown type");
                        break;
                  }
                  return TRUE;
            }
      return TRUE;
      }

      if (synIsNextNontermSegmentRegister())
      {
            pOpcode = pCurContrElemArg;
            synAddGPRegXXRWCodeOEGroup(uType);
            uType2 = synCheckNextNontermSegmentRegister();
            if
(synIsNextTermLexeme(LEXCLASS_PUNCTUATION_COLON)) // MOV
RXX,Sreg:[...
            {
                  synAddPrefixSregOEGroup(uType2);

      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COLON);

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_SQAURE_BRACK
ET);
                  uType2 = synAddMemoryLocationOEGroup();

      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_SQAURE_BRAC
KET);

                  switch(semRetTypeSizeByType(uType))
                  {
                  case 4:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG32_MEM32;
```

```
                        break;
                  case 2:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG16_MEM16;
                        break;
                  case 1:
                        pOpcode->pArgument.uSemClass =
SEMCLASS_INSTRUCTION_REG8_MEM8;
                        break;
                  default:
                        uSemClass = SEMCLASS_UNKNOWN;

      semError("identifier",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"unknown type");
                        break;
                  }
                  return TRUE;
            }
            else // MOV R16, Sreg
            {
                  synAddSregOEGroup(uType);
                  return TRUE;
            }
      }
      return TRUE;
}

/*
*     Checks non-terminal symbol "assembler block"
*
*   @author Nail Sharipov
*/
int synCheckNontermAsmBlock()
{
      TOperationElemPtr pOpcode;
      UINT uType = 0, uType2 = 0;
      UINT uSemClass = 0;
      TOperationElemPtr pTempOE = NULL;

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_CURLY_BRACKE
T);
      while (synIsNextNontermInstruction())
      {


      synAddNewContrElemToCurBlock(SEMCLASS_INSTRUCTION,
&cgContrElemInstruction);

            /*
            *     MOV instruction
            */
            if
(synIsNextTermLexeme(LEXCLASS_IA32INSTRUCTIONS_MOV))
            {
                  uType =
synCheckNextTermLexeme(LEXCLASS_IA32INSTRUCTIONS_MOV);

      synAddNewOperElemToContrElem(SEMCLASS_UNKNOWN, uType,
NULL,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);
                  pOpcode = pCurContrElemArg;

                  if (synIsNextNontermSegmentRegister())
                  {
                        uType =
synCheckNextNontermSegmentRegister();
                        if
(synIsNextTermLexeme(LEXCLASS_PUNCTUATION_COLON)) // MOV
Sreg:0xXXXX,0xXX
                        {
                              pOpcode = pCurContrElemArg;
                              synAddPrefixSregOEGroup(uType);

      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COLON);
                        }

                        else // MOV Sreg
                        {
                              synAsmOperandGroup_SREG(uType,
pOpcode);
                        }
                  }

                  // MOV memXX,
                  if
(synIsNextTermLexeme(LEXCLASS_BLOCK_LEFT_SQAURE_BRACKET))
                  {

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_SQAURE_BRACK
ET);
                        uType =
synAddMemoryLocationOEGroup();
                        synAsmOperandGroup_MEMXX(uType,
pOpcode);
                        continue;
                  }

                  // MOV RXX,
                  if (synIsNextNontermWordGPRegister() ||
synIsNextNontermByteGPRegister() ||
synIsNextNontermDWordGPRegister())
                  {

                        if
(synIsNextNontermDWordGPRegister())
                        {
                              uSemClass = SEMCLASS_REG32;
                              uType =
synCheckNextNontermDWordGPRegister();
                        }

                        if (synIsNextNontermWordGPRegister())
                        {
                              uSemClass = SEMCLASS_REG16;
                              uType =
synCheckNextNontermWordGPRegister();
                        }
                        if (synIsNextNontermByteGPRegister())
                        {
                              uSemClass = SEMCLASS_REG8;
                              uType =
synCheckNextNontermByteGPRegister();
                        }


      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_COMMA);
                        synAsmOperandGroup_REGXX(uType,
pOpcode);
                  }

                  continue;
            } // *** END *** MOV instruction

            /*
            *     INT instruction
            */
            if
(synIsNextTermLexeme(LEXCLASS_IA32INSTRUCTIONS_INT))
            {
                  uType =
synCheckNextTermLexeme(LEXCLASS_IA32INSTRUCTIONS_INT);

      synAddNewOperElemToContrElem(SEMCLASS_UNKNOWN, uType,
NULL,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);
                  pOpcode = pCurContrElemArg;

                  synAsmOperandGroup_IMMXX(pOpcode);

                  if (pOpcode->pArgument.uSemClass !=
SEMCLASS_INSTRUCTION_IMM8)
                        semError("INT",pCurLexListItem-
>pPrevLexemeInstance->LexemeInstance.uPosBeg,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,

      pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum,"constant is too large");
                  continue;
            }

            /*
            *     JMP instruction
            */
            if
(synIsNextTermLexeme(LEXCLASS_IA32INSTRUCTIONS_JMP))
            {
                  uType =
synCheckNextTermLexeme(LEXCLASS_IA32INSTRUCTIONS_JMP);

      synAddNewOperElemToContrElem(SEMCLASS_UNKNOWN, uType,
NULL,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);
                  pOpcode = pCurContrElemArg;

                  synAsmOperandGroup_PTR16_16(pOpcode);

                  continue;
            }
      }

      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_CURLY_BRACK
ET);
      return FALSE;
};

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermImplementation()
{
      TContrElemPtr pCurCEContainer = NULL, pNewContrElem =
NULL;
      TOperationElemPtr pCurCEArgContainer = NULL;
      UINT uType = 0;
    switch (synRetNextLCValue())
    {
    case LEXCLASS_BLOCK_LEFT_CURLY_BRACKET:
            synAddBlockContrElem();
            break;
      case LEXCLASS_PROGRAMM_ASM_ENTRY:

      synCheckNextTermLexeme(LEXCLASS_PROGRAMM_ASM_ENTRY);
            synCheckNontermAsmBlock();
            break;
    case LEXCLASS_PROGRAMM_RETURN_POINT:

      synCheckNextTermLexeme(LEXCLASS_PROGRAMM_RETURN_POINT);
//////////////////////////////////////////////////////////////
///////////////
            synAddNewContrElemToCurBlock(SEMCLASS_FUNCRETURN,
&cgContrElemFuncRet);
            uType = pCurParseListNode->FuncNode.uType;
            synAddNewOperElemToContrElem(SEMCLASS_FUNCRETURN,
uType, pCurParseListNode,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);
            synAddReturnContrElem();
//////////////////////////////////////////////////////////////
/////////////
//
      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_SEMICOLUMN)
;
            break;
      case LEXCLASS_CONSTRUCTIONS_IF:

      synCheckNextTermLexeme(LEXCLASS_CONSTRUCTIONS_IF);

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_ROUND_BRACKE
T);

      synAddNewContrElemToCurBlock(SEMCLASS_CONSTRUCTION_IF,
&cgContrElemConstructionIf);
            pNewContrElem =
synCreateNewContrElem(&cgContrElemEquation);
            synAddNewOperElemToContrElem(SEMCLASS_EQUATION,
uType, pNewContrElem,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

            pCurCEContainer = pCurContrElem;
            pCurCEArgContainer = pCurContrElemArg;

            pCurContrElem = pNewContrElem;
            pCurContrElemArg = NULL;

            synCheckNextNontermEquation();

            pCurContrElem = pCurCEContainer;
            pCurContrElemArg = pCurCEArgContainer;


      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_ROUND_BRACK
ET);

            pCurCEContainer = pCurContrElem;
            pCurCEArgContainer = pCurContrElemArg;

            pNewContrElem =
synAddNewBlockContrElemToCurContrElem();

            pCurContrElem = pCurCEContainer;
            pCurContrElemArg = pCurCEArgContainer;

            synAddNewOperElemToContrElem(SEMCLASS_BLOCK,
uType, pNewContrElem,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

            if
(synIsNextTermLexeme(LEXCLASS_CONSTRUCTIONS_ELSE))
            {

      synCheckNextTermLexeme(LEXCLASS_CONSTRUCTIONS_ELSE);

                  pCurCEContainer = pCurContrElem;
                  pCurCEArgContainer = pCurContrElemArg;

                  pNewContrElem =
synAddNewBlockContrElemToCurContrElem();
```

```
                  pCurContrElem = pCurCEContainer;
                  pCurContrElemArg = pCurCEArgContainer;


      synAddNewOperElemToContrElem(SEMCLASS_CONSTRUCTION_ELSE
, uType, pNewContrElem,
                        pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);
            }
            break;
      case LEXCLASS_CONSTRUCTIONS_WHILE:

      synCheckNextTermLexeme(LEXCLASS_CONSTRUCTIONS_WHILE);

      synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_ROUND_BRACKE
T);


      synAddNewContrElemToCurBlock(SEMCLASS_CONSTRUCTION_WHIL
E, &cgContrElemConstructionWhile);

            pNewContrElem =
synCreateNewContrElem(&cgContrElemEquation);

            synAddNewOperElemToContrElem(SEMCLASS_EQUATION,
uType, pNewContrElem,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);

            pCurCEContainer = pCurContrElem;
            pCurCEArgContainer = pCurContrElemArg;

            pCurContrElem = pNewContrElem;
            pCurContrElemArg = NULL;

            synCheckNextNontermEquation();

            pCurContrElem = pCurCEContainer;
            pCurContrElemArg = pCurCEArgContainer;


      synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_ROUND_BRACK
ET);

            pCurCEContainer = pCurContrElem;
            pCurCEArgContainer = pCurContrElemArg;

            pNewContrElem =
synAddNewBlockContrElemToCurContrElem();

            pCurContrElem = pCurCEContainer;
            pCurContrElemArg = pCurCEArgContainer;

            synAddNewOperElemToContrElem(SEMCLASS_BLOCK,
uType, pNewContrElem,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosBeg,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uPosEnd,
                  pCurLexListItem->pPrevLexemeInstance-
>LexemeInstance.uLineNum, NULL);
            break;
      default:
            synAddNewContrElemToCurBlock(SEMCLASS_EQUATION,
&cgContrElemEquation);
            synCheckNextNontermEquation();

      synCheckNextTermLexeme(LEXCLASS_PUNCTUATION_SEMICOLUMN)
;
        break;
    };
    return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synCheckNextNontermBlock()
{
      synFillRT();
      synAddIdentTableElem(SYNCLASS_ACCESSORIES_BEGINBLOCK,
0,0,0,NULL, pCurIdTableBegBlock, NULL, NULL, NULL);

synCheckNextTermLexeme(LEXCLASS_BLOCK_LEFT_CURLY_BRACKET);

    if ( synIsNextNontermType() )
    {
        synCheckNextNontermVarDecl();
    };

      synFillRT();

      if (
!synIsNextTermLexeme(LEXCLASS_BLOCK_RIGHT_CURLY_BRACKET) )
    {
        while
(!synIsNextTermLexeme(LEXCLASS_BLOCK_RIGHT_CURLY_BRACKET) &&
pCurLexListItem)
                  synCheckNextNontermImplementation();
      };

      synUnFillRT();
      synAddIdentTableElem(SYNCLASS_ACCESSORIES_ENDBLOCK,
0,0,0, NULL, pCurIdTableBegBlock,NULL, NULL, NULL);


synCheckNextTermLexeme(LEXCLASS_BLOCK_RIGHT_CURLY_BRACKET);

    return TRUE;
}

/*
*
*
```

```
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int synAnalysis()
{
      UINT uType;
    TRTItemPtr pBuff;
      UINT uLexClass;
    extern TIdTablePtr pCurIdTableBegBlock;

      pCurLexListItem = pBegLexList;
    while (pCurLexListItem)
    {
        uLexClass = synRetNextLCValue();
        switch(uLexClass)
        {
        case LEXCLASS_TYPE_VOID:
        case LEXCLASS_TYPE_SHORT:
            case LEXCLASS_TYPE_CHAR:
            synCheckNextNontermFuncDecl();
            synAddNewBlockToCurFuncNode();
            synCheckNextNontermBlock();

                  synUnFillRT();

      synAddIdentTableElem(SYNCLASS_ACCESSORIES_ENDBLOCK,
0,0,0, NULL, pCurIdTableBegBlock,NULL, NULL, NULL);
                  //lexPrintRT();

                  break;

        default:
            uLexClass = synGetNextLexemeClass();
            synError();
            break;
        }
    }
    synAddIdentTableElem(SYNCLASS_ACCESSORIES_ENDBLOCK,
0,0,0, NULL, pCurIdTableBegBlock,NULL, NULL, NULL);
      return TRUE;
};

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
void synPrintIdTable()
{
    extern TIdTable firstIdTableElem;
    TIdTablePtr     pTemp, pIdTableElem = &firstIdTableElem;
    FILE * pOutInfoFile;

    pOutInfoFile = fopen("CompileInfo.txt","a");

    printf("\n");
    fprintf(pOutInfoFile, "\n");
```

```
      printf(" *** IDENTIFIER TABLE *** \n\n");
    fprintf(pOutInfoFile, " *** IDENTIFIER TABLE *** \n\n");
      while (pIdTableElem)
    {
        if (pIdTableElem->pArgInstance)
            printf("  %d  --- %s :   %d", pIdTableElem-
>uType,

                  (pIdTableElem->uType ==
SYNCLASS_ACCESSORIES_BEGINBLOCK)?
                  "BEGIN BLOCK":((pIdTableElem->uType ==
SYNCLASS_ACCESSORIES_ENDBLOCK)?
                  "END BLOCK":pIdTableElem->pItemContent-
>cpIdName),

                  pIdTableElem->pArgInstance-
>pArgument.uSemClass);
            else
            printf("  %d  --- %s ", pIdTableElem->uType,
                  (pIdTableElem->uType ==
SYNCLASS_ACCESSORIES_BEGINBLOCK)?
                  "BEGIN BLOCK":((pIdTableElem->uType ==
SYNCLASS_ACCESSORIES_ENDBLOCK)?
                  "END BLOCK":pIdTableElem->pItemContent-
>cpIdName));
        fprintf(pOutInfoFile, "  %d  --- %s ", pIdTableElem-
>uType,
                  (pIdTableElem->uType ==
SYNCLASS_ACCESSORIES_BEGINBLOCK)?
                  "BEGIN BLOCK":((pIdTableElem->uType ==
SYNCLASS_ACCESSORIES_ENDBLOCK)?
                  "END BLOCK":pIdTableElem->pItemContent-
>cpIdName));

            if ((pIdTableElem->pPrevTie) && (pIdTableElem-
>uType == SYNCLASS_ACCESSORIES_BEGINBLOCK))
            {
                  pTemp = pIdTableElem->pPrevTie-
>pNextListItem;
                  printf(" ( ");
                  fprintf(pOutInfoFile, " ( ");
                  while (pTemp->pItemContent)
                  {
                        printf(" %d, ", pTemp->uType);
                        fprintf(pOutInfoFile, " %d, ", pTemp-
>uType);
                        pTemp = pTemp->pNextListItem;
                  }
                  printf(" ) \n");
                  fprintf(pOutInfoFile, " ) \n");
            }
            else
            if (pIdTableElem->uType ==
SYNCLASS_ACCESSORIES_ENDBLOCK)
            {
                  pTemp = pIdTableElem->pPrevTie-
>pNextListItem;
                  printf(" ( ");
                  fprintf(pOutInfoFile, " ( ");
                  while (pTemp->pItemContent)
                  {
                        printf(" %d, ", pTemp->uType);
                        fprintf(pOutInfoFile, " %d, ", pTemp-
>uType);
```

```
                        pTemp = pTemp->pNextListItem;
                  }
                  printf(" ) \n");
                  fprintf(pOutInfoFile, " ) \n");
            }
            else
            {
                  printf("\n");
                  fprintf(pOutInfoFile, "\n");
            }
            pIdTableElem = pIdTableElem->pNextListItem;
    }
      fclose(pOutInfoFile);
}
```

Figure 6. analysis_syn.c

```
#ifndef INCL_ANALYSIS_SEM_H
#define INCL_ANALYSIS_SEM_H

#include "main.h"

/** SEMANTIC ANALYSIS **/
// General functions
void  semAnalisys();

// General functions
int         semRetValueType(UINT uValueLength);
int         semRetTypeSizeByType(UINT uType);
void  semError(char * cpTextFrag, UINT uPosBeg,UINT uPosEnd,
UINT uLineNum, char * cpErrMess);

int         semCheckContrElemFuncRet(TOperationElemPtr
pBegArgList);
int         semCheckContrElemFuncCall(TOperationElemPtr
pBegArgList);
int         semCheckContrElemExpression(TContrElemPtr
pContrElem);
int         semCheckContrElemEquation(TOperationElemPtr
pBegArgList);
int         semCheckContrElemBlock(TOperationElemPtr
pBegArgList);
int         semCheckBlock(TBlockPtr pBlock);

int         semRetSregOpcodeByType(UINT uType);
int         semRetSregPrefixByType(UINT uType);
int         semRetRegRegisterOpcodeByType(UINT uType);
int         semRetRegEffectivAddrByType(UINT uType);

#endif //INCL_ANALYSIS_SEM_H
```

Figure 7. analysis_sem.h

```
#include "analysis_lex.h"
#include "analysis_syn.h"
#include "analysis_sem.h"
#include "analysis_cg.h"

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semRetTypeSizeByType(UINT uType)
{
      switch      (uType)
      {
      case LEXCLASS_TYPE_INT:
      case LEXCLASS_IA32REGISTERS_EAX:
      case LEXCLASS_IA32REGISTERS_EBX:
      case LEXCLASS_IA32REGISTERS_ECX:
      case LEXCLASS_IA32REGISTERS_EDX:
      case LEXCLASS_IA32REGISTERS_ESI:
      case LEXCLASS_IA32REGISTERS_EDI:
      case LEXCLASS_IA32REGISTERS_ESP:
      case LEXCLASS_IA32REGISTERS_EBP:
            return 4;
            break;
      case LEXCLASS_TYPE_SHORT:
      case LEXCLASS_IA32REGISTERS_AX:
      case LEXCLASS_IA32REGISTERS_BX:
      case LEXCLASS_IA32REGISTERS_CX:
      case LEXCLASS_IA32REGISTERS_DX:
      case LEXCLASS_IA32REGISTERS_SI:
      case LEXCLASS_IA32REGISTERS_DI:
      case LEXCLASS_IA32REGISTERS_BP:
      case LEXCLASS_IA32REGISTERS_SP:
            return 2;
            break;
      case LEXCLASS_TYPE_CHAR:      // there are some problems
with char because of
            // impossibility 8-bit value from mem16 to push
into stack
      case LEXCLASS_IA32REGISTERS_AH:
      case LEXCLASS_IA32REGISTERS_AL:
      case LEXCLASS_IA32REGISTERS_BH:
      case LEXCLASS_IA32REGISTERS_BL:
      case LEXCLASS_IA32REGISTERS_CH:
      case LEXCLASS_IA32REGISTERS_CL:
      case LEXCLASS_IA32REGISTERS_DH:
      case LEXCLASS_IA32REGISTERS_DL:
            return 1;
            break;
      case LEXCLASS_TYPE_VOID:
            return 0;
            break;
      }
      return FALSE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
void semError(char * cpTextFrag, UINT uPosBeg,UINT uPosEnd,
UINT uLineNum, char * cpErrMess)
{
      extern TIdErrorPtr pIdErrorBegList;
      extern TIdErrorPtr pIdError;
      TIdErrorPtr pNewIdError;
      pNewIdError = (TIdErrorPtr)malloc(sizeof(TIdError));
      pNewIdError->cpErrorMess = (char
*)malloc(strlen(cpErrMess)+1);
      strcpy(pNewIdError->cpErrorMess, cpErrMess);
      pNewIdError->ErrorInstance.cpTextFrag = (char
*)malloc(strlen(cpTextFrag)+1);
      strcpy(pNewIdError->ErrorInstance.cpTextFrag,
cpTextFrag);
      pNewIdError->ErrorInstance.uPosBeg = uPosBeg;
      pNewIdError->ErrorInstance.uPosEnd = uPosEnd;
      pNewIdError->ErrorInstance.uLineNum = uLineNum;
      pNewIdError->pNextError = NULL;

      if (pIdError)
      {
            pIdError->pNextError = pNewIdError;
      }

      if (!pIdErrorBegList)
      {
            pIdErrorBegList= pNewIdError;
      }
      pIdError = pNewIdError;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semCheckContrElemFuncRet(TOperationElemPtr pBegArgList)
{
      TContrElemPtr       pTempCE;
      /*    TODO: uncomment this errors checking - VERY
IMPORTANT WARNINGS!!! */
//    if (pBegArgList->pArgument.uLexClass != pBegArgList-
>pNextOperationElem->pArgument.uLexClass)
//    {
//          semError ("return", pBegArgList-
>pArgument.uBegPos, pBegArgList->pArgument.uEndPos,
//                pBegArgList->pArgument.uStrNum,"function
and returning value type mismatch");
//    }
      pTempCE = (TContrElemPtr)pBegArgList-
>pNextOperationElem->pArgument.pvValue;
      semCheckContrElemEquation(pTempCE->pBegArgList);
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semCheckContrElemFuncCall(TOperationElemPtr pBegArgList)
{
      TOperationElemPtr pOEInDecl,pOEInCurFunc;
      TOperationElemPtr pBegCurFuncDeclArgList,
pBegCurFuncArgList;
      TContrElemPtr       pTempCE;
      TIdTablePtr         pTempIdTableElem;

      pTempIdTableElem = (TIdTablePtr)pBegArgList-
>pArgument.pvValue;
      pBegCurFuncDeclArgList = pTempIdTableElem-
>pArgInstance;
      pBegCurFuncArgList = pBegArgList->pNextOperationElem;
      pOEInCurFunc = pBegCurFuncArgList;
      pOEInDecl = pBegCurFuncDeclArgList;
      // pBegCurFuncDeclArgList - pointer to the beginning of
the argument list of current function declaration
      // pBegCurFuncArgList - pointer to the beginning of the
argument list of current function


      while(pOEInCurFunc)
      {
            pTempCE = (TContrElemPtr)(pOEInCurFunc-
>pArgument.pvValue);

            semCheckContrElemEquation(pTempCE->pBegArgList);

            /*    TODO: uncomment this errors checking - VERY
IMPORTANT WARNINGS!!! */
//          if (pOEInDecl && pOEInDecl->pArgument.uLexClass
!= pOEInCurFunc->pArgument.uLexClass)
//          {
//                semError(pTempIdTableElem->pItemContent-
>cpIdName, pOEInCurFunc->pArgument.uBegPos,pOEInCurFunc-
>pArgument.uEndPos,
//                      pOEInCurFunc->pArgument.uStrNum,
"formal and actual argument type mismatch");
//          }
            if ( pOEInDecl && !pOEInCurFunc-
>pNextOperationElem && pOEInDecl->pNextOperationElem )
            {
                  semError(pTempIdTableElem->pItemContent-
>cpIdName, pOEInCurFunc->pArgument.uBegPos,pOEInCurFunc-
>pArgument.uEndPos,
                        pOEInCurFunc->pArgument.uStrNum, "too
few actual parameters");
                  break;
            }
            if ( !pOEInDecl || pOEInCurFunc-
>pNextOperationElem && !pOEInDecl->pNextOperationElem )
            {
                  semError(pTempIdTableElem->pItemContent-
>cpIdName, pOEInCurFunc->pArgument.uBegPos,pOEInCurFunc-
>pArgument.uEndPos,
                        pOEInCurFunc->pArgument.uStrNum, "too
many actual parameters");
                  break;
            }
            pOEInCurFunc = pOEInCurFunc->pNextOperationElem;
            pOEInDecl = pOEInDecl->pNextOperationElem;

      }
      if ( !pOEInCurFunc && pOEInDecl)
      {
            semError(pTempIdTableElem->pItemContent-
>cpIdName, pTempIdTableElem->uPosBeg,
                  pTempIdTableElem->uPosEnd,
                  pTempIdTableElem->uLineNum, "too few actual
parameters");
      }

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semRetOperPriorityByLexClass(int uClass)
{
      switch(uClass)
      {
      case LEXCLASS_OPERATION_ASTERISK:
      case LEXCLASS_OPERATION_SLASH:
            return SEMOPERPRIORITY_MUL;
            break;
      case LEXCLASS_OPERATION_PLUS:
      case LEXCLASS_OPERATION_MINUS:
            return SEMOPERPRIORITY_PLUS;
            break;
      case LEXCLASS_COMPARISON_EQUAL:
            return SEMOPERPRIORITY_EQUAL;
            break;
      }
      return FALSE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semCheckContrElemExpression(TContrElemPtr pContrElem)
{
      TIDCurStatePtr          pTempCS;
      TOperationElemPtr pContrElemArg;
      TContrElemPtr           pTempCE;
      TOperationElemPtr pTempStack = NULL;
      TOperationElemPtr pNewArg = NULL;
      TOperationElemPtr pCurArg = NULL;
      TOperationElemPtr pBegNewList = NULL;
      TIdTablePtr             pTempIdElem;
      TIDCurStatePtr          pTempIdCurState;
      TAAOperElemListPtr            pTempAAE;

      //reverse Polish notation
      pContrElemArg = pContrElem->pBegArgList;

      while (pContrElemArg)
      {
            if (pContrElemArg->pArgument.uSemClass !=
SEMCLASS_OPERATION)
            {
                  if (pContrElemArg->pArgument.uSemClass ==
SEMCLASS_FUNCTION)
                  {
                        pTempCE =
(TContrElemPtr)pContrElemArg->pArgument.pvValue;
                        semCheckContrElemFuncCall(pTempCE-
>pBegArgList);
                  }

                  pNewArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
                  *pNewArg = *pContrElemArg;
                  if (pCurArg)
                  {
                        pCurArg->pNextOperationElem =
pNewArg;
                        pNewArg->pPrevOperationElem =
pCurArg;
                        pCurArg = pNewArg;
                  }

                  if (!pBegNewList)
                  {
                        pBegNewList = pNewArg;
                        pCurArg = pNewArg;
                  }
            }
            else
            {
                  if (!pTempStack)
                  {
                        pNewArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
                        *pNewArg = *pContrElemArg;
                        pNewArg->pPrevOperationElem = NULL;
                        pNewArg->pNextOperationElem = NULL;
                        pTempStack = pNewArg;
                  }
                  else
                  {
                        if (pContrElemArg-
>pArgument.uLexClass == LEXCLASS_BLOCK_RIGHT_ROUND_BRACKET)
                        {
                              while ( pTempStack-
>pArgument.uLexClass != LEXCLASS_BLOCK_LEFT_ROUND_BRACKET)
                              {
                                    pNewArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
                                    *pNewArg = *pTempStack;
                                    pCurArg-
>pNextOperationElem = pNewArg;
                                    pNewArg-
>pPrevOperationElem = pCurArg;
                                    pNewArg-
>pNextOperationElem = NULL;
                                    pCurArg = pNewArg;
                                    pTempStack = pTempStack-
>pPrevOperationElem;
                                    free(pTempStack-
>pNextOperationElem);
                              }
                              if(pTempStack-
>pPrevOperationElem)
                              {
                                    pTempStack = pTempStack-
```

```
>pPrevOperationElem;
                                    free(pTempStack-
>pNextOperationElem);
                                    pTempStack-
>pNextOperationElem = NULL;
                              }
                              else
                              {
                                    free(pTempStack);
                                    pTempStack = NULL;
                              }
                        }
                        else if (pContrElemArg-
>pArgument.uLexClass == LEXCLASS_BLOCK_LEFT_ROUND_BRACKET)
                        {
                              if (pTempStack)
                              {
                                    pNewArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
                                    *pNewArg = *pContrElemArg;
                                    pTempStack-
>pNextOperationElem = pNewArg;
                                    pNewArg-
>pPrevOperationElem = pTempStack;
                                    pNewArg-
>pNextOperationElem = NULL;
                                    pTempStack = pNewArg;
                              }
                              else
                              {
                                    pNewArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
                                    *pNewArg = *pContrElemArg;
                                    pNewArg-
>pPrevOperationElem = NULL;
                                    pNewArg-
>pNextOperationElem = NULL;
                                    pTempStack = pNewArg;
                              }
                        }
                        else
                        {
                              while ( pTempStack &&
(semRetOperPriorityByLexClass(pContrElemArg-
>pArgument.uLexClass) <
semRetOperPriorityByLexClass(pTempStack-
>pArgument.uLexClass)))
                              {
                                    pNewArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
                                    *pNewArg = *pTempStack;
                                    pCurArg-
>pNextOperationElem = pNewArg;
                                    pNewArg-
>pPrevOperationElem = pCurArg;
                                    pNewArg-
>pNextOperationElem = NULL;
                                    pCurArg = pNewArg;
                                    pTempStack = pTempStack-
>pPrevOperationElem;
                              }

                              if (pTempStack)
                              {
                                    pNewArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
                                    *pNewArg = *pContrElemArg;
                                    pTempStack-
>pNextOperationElem = pNewArg;
                                    pNewArg-
>pPrevOperationElem = pTempStack;
                                    pNewArg-
>pNextOperationElem = NULL;
                                    pTempStack = pNewArg;
                              }
                              else
                              {
                                    pNewArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
                                    *pNewArg = *pContrElemArg;
                                    pNewArg-
>pPrevOperationElem = NULL;
                                    pNewArg-
>pNextOperationElem = NULL;
                                    pTempStack = pNewArg;
                              }
                        }
                  }
            }
            pContrElemArg = pContrElemArg-
>pNextOperationElem;
      }

      while (pTempStack)
      {
            pNewArg =
(TOperationElemPtr)malloc(sizeof(TOperationElem));
            *pNewArg = *pTempStack;

            pCurArg->pNextOperationElem = pNewArg;
            pNewArg->pPrevOperationElem = pCurArg;
            pNewArg->pNextOperationElem = NULL;
            pCurArg = pNewArg;
            pTempStack = pTempStack->pPrevOperationElem;
      }

      pContrElem->pBegArgList = pBegNewList;
      pContrElemArg = pContrElem->pBegArgList;

      while (pContrElemArg)
      {
            if (
                   (pContrElemArg->pArgument.uSemClass !=
SEMCLASS_OPERATION) &&
                   (pContrElemArg->pArgument.uSemClass !=
SEMCLASS_FUNCTION)  &&
                   (pContrElemArg->pArgument.uSemClass !=
SEMCLASS_IMM8) &&
                   (pContrElemArg->pArgument.uSemClass !=
SEMCLASS_IMM16) &&
                   (pContrElemArg->pArgument.uSemClass !=
SEMCLASS_IMM32)
                   )
            {
                  pTempIdCurState =
(TIDCurStatePtr)pContrElemArg->pArgument.pvValue;
                  pTempIdElem = pTempIdCurState-
>pIdTableElem;
```

```
            }
            pContrElemArg = pContrElemArg-
>pNextOperationElem;
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semCheckContrElemEquation(TOperationElemPtr pBegArgList)
{
      TIDCurStateListPtr      pTempCurStateListElem = NULL,
pTempCurStateBegList = NULL, pEIContainer = NULL;
      TIdTablePtr             pTempIdTableElem = NULL,
pExprIdTableElem = NULL;
      TAAOperElemListPtr            pDeclTempAAE = NULL,
pTempAAE = NULL, pExprAAE = NULL;
      TOperationElemPtr pContrElemArg;
      TContrElemPtr           pTempCE;
      UINT                    uBaseSubLvl = 0, uTempSubLvl =
0, uExprIndLevel = 0, uTempIndLevel = 0;

      pContrElemArg = pBegArgList;
      while (pContrElemArg->pNextOperationElem)
            pContrElemArg = pContrElemArg-
>pNextOperationElem;

      if (pContrElemArg)
      {
            pTempCE = (TContrElemPtr)pContrElemArg-
>pArgument.pvValue;
            semCheckContrElemExpression(pTempCE);
            pTempCurStateBegList = pContrElemArg-
>pArgument.pIDCurState;
      }
      pTempCurStateListElem = pTempCurStateBegList;
      if (pTempCurStateListElem)
      {
            pTempAAE = pTempCurStateListElem->pCurIdState-
>pIdTableElem->pBegIdDecl;
            while(pTempAAE)
            {
                  switch(pTempAAE->AAOperElem.uDeclType)
                  {
                  case SYNDECLELEMTYPE_SQUARE_BRACKETS:
                        uExprIndLevel++;
                        break;
                  case SYNDECLELEMTYPE_ASTERISK:
                        uExprIndLevel = uExprIndLevel +
*((UINT *)pTempAAE->AAOperElem.pvValue);
                  }
                  pTempAAE = pTempAAE->pNextAAElem;
            }
      }
      if (pTempCurStateBegList)
            pTempCurStateBegList->pCurIdState-
>uTotalIndLevelCnt = uExprIndLevel;
```

```
      pTempCurStateListElem = pTempCurStateBegList;
      if (pTempCurStateListElem)
      {
            pTempAAE = pTempCurStateListElem->pCurIdState-
>pIdTableElem->pBegIdDecl;
            pTempCurStateListElem = pTempCurStateListElem-
>pNextCurStateElem;
            while (pTempCurStateListElem)
            {
                  if (pTempCurStateListElem->pCurIdState-
>uStateType == SYNIDCURSTATETYPE_INDEX_DEREFERENCING)
                  {
                        pTempCE = pTempCurStateListElem-
>pCurIdState->pvValue;
                        semCheckContrElemEquation(pTempCE-
>pBegArgList);
                  }
                  pTempCurStateListElem =
pTempCurStateListElem->pNextCurStateElem;
            }
            uTempIndLevel = 0;
            if(pTempCurStateListElem)
            while (pTempAAE && (pTempAAE !=
pTempCurStateListElem->pCurIdState->pCurAAElem))
            {
                  switch(pTempAAE->AAOperElem.uDeclType)
                  {
                  case SYNDECLELEMTYPE_SQUARE_BRACKETS:
                        uTempIndLevel++;
                        break;
                  case SYNDECLELEMTYPE_ASTERISK:
                        uTempIndLevel = uTempIndLevel +
*((UINT *)pTempAAE->AAOperElem.pvValue);
                        break;
                  }

                  pTempAAE = pTempAAE->pNextAAElem;
            }
            if(pTempCurStateListElem)
            switch(pTempCurStateListElem->pCurIdState-
>uStateType)
            {
            case SYNIDCURSTATETYPE_INDEX_DEREFERENCING:
                  uTempIndLevel++;
                  break;
            case SYNIDCURSTATETYPE_DEREFERENCING:
                  uTempIndLevel += *((UINT *)pTempAAE-
>AAOperElem.pvValue) - *((UINT *)pTempCurStateListElem-
>pCurIdState->pvValue);
                  break;
            }
      }
      uBaseSubLvl = uExprIndLevel - uTempIndLevel;
      pContrElemArg = pContrElemArg->pPrevOperationElem;

      while (pContrElemArg)
      {
            pTempCE = (TContrElemPtr)pContrElemArg-
>pArgument.pvValue;
            semCheckContrElemExpression(pTempCE);

            uExprIndLevel = 0;
            uTempIndLevel = 0;
            pTempCurStateBegList = pContrElemArg-
```

```
>pArgument.pIDCurState;

            pTempCurStateListElem = pTempCurStateBegList;

            if (pTempCurStateListElem)
            {
                  pTempAAE = pTempCurStateListElem-
>pCurIdState->pIdTableElem->pBegIdDecl;
                  while(pTempAAE)
                  {
                        switch(pTempAAE-
>AAOperElem.uDeclType)
                        {
                        case SYNDECLELEMTYPE_SQUARE_BRACKETS:
                              uExprIndLevel++;
                              break;
                        case SYNDECLELEMTYPE_ASTERISK:
                              uExprIndLevel = uExprIndLevel +
*((UINT *)pTempAAE->AAOperElem.pvValue);
                        }
                  pTempAAE = pTempAAE->pNextAAElem;
                  }
            }

            if (pTempCurStateBegList)
                  pTempCurStateBegList->pCurIdState-
>uTotalIndLevelCnt = uExprIndLevel;

            pTempCurStateListElem = pTempCurStateBegList;



            if (pTempCurStateListElem)
            {
                  pTempAAE = pTempCurStateListElem-
>pCurIdState->pIdTableElem->pBegIdDecl;

                  while (pTempCurStateListElem-
>pNextCurStateElem)
                  {
                        pTempCurStateListElem =
pTempCurStateListElem->pNextCurStateElem;
                        if (pTempCurStateListElem-
>pCurIdState->uStateType ==
SYNIDCURSTATETYPE_INDEX_DEREFERENCING)
                        {
                              pTempCE =
pTempCurStateListElem->pCurIdState->pvValue;

      semCheckContrElemEquation(pTempCE->pBegArgList);
                        }

                  }

                  uTempIndLevel = 0;


                  if(pTempCurStateListElem)
                  while (pTempAAE && (pTempAAE !=
pTempCurStateListElem->pCurIdState->pCurAAElem))
                  {
                        switch(pTempAAE-
>AAOperElem.uDeclType)
                        {
```

```
                        case SYNDECLELEMTYPE_SQUARE_BRACKETS:
                              uTempIndLevel++;
                              break;
                        case SYNDECLELEMTYPE_ASTERISK:
                              uTempIndLevel = uTempIndLevel +
*((UINT *)pTempAAE->AAOperElem.pvValue);
                              break;
                        }
                        pTempAAE = pTempAAE->pNextAAElem;
                  }

                  if(pTempCurStateListElem)
                  switch(pTempCurStateListElem->pCurIdState-
>uStateType)
                  {
                  case SYNIDCURSTATETYPE_INDEX_DEREFERENCING:
                        uTempIndLevel++;
                        break;
                  case SYNIDCURSTATETYPE_DEREFERENCING:
                        uTempIndLevel += *((UINT *)pTempAAE-
>AAOperElem.pvValue) - *((UINT *)pTempCurStateListElem-
>pCurIdState->pvValue);
                        break;
                  }

                  while(pTempAAE && pTempAAE->pNextAAElem)
                        pTempAAE = pTempAAE->pNextAAElem;
            }
            uTempSubLvl = uExprIndLevel - uTempIndLevel;


      /*    TODO: uncomment this errors checking - VERY
IMPORTANT WARNINGS!!! */

      //    if ((uTempSubLvl != uBaseSubLvl))
      //          semError("equation", pContrElemArg-
>pArgument.uBegPos, pContrElemArg->pArgument.uEndPos,
pContrElemArg->pArgument.uStrNum, "difference in levels of
indirection");
      //    if (!(uTempSubLvl>0 && pContrElemArg-
>pNextOperationElem->pArgument.uSemClass == SEMCLASS_IMM16) )
      //    if (!(uTempSubLvl>0 && pContrElemArg-
>pNextOperationElem->pArgument.uSemClass == SEMCLASS_MEM16) )
      //    if (pContrElemArg->pArgument.uLexClass !=
pContrElemArg->pNextOperationElem->pArgument.uLexClass)
      //          semError("equation", pContrElemArg-
>pArgument.uBegPos, pContrElemArg->pArgument.uEndPos,
pContrElemArg->pArgument.uStrNum, "type mismatch");


            if ((pContrElemArg->pArgument.uSemClass !=
SEMCLASS_MEM16) || (pTempCurStateListElem &&
pTempCurStateListElem->pCurIdState->uStateType ==
SYNIDCURSTATETYPE_OPERATION))
                  semError("=", pContrElemArg-
>pArgument.uBegPos, pContrElemArg->pArgument.uEndPos,
pContrElemArg->pArgument.uStrNum, "left operand must be l-
value");

            pContrElemArg = pContrElemArg-
>pPrevOperationElem;
      }
      return TRUE;
}
```

```
/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semRetValueType(UINT uValueLength)
{
      return (uValueLength > 8)?LEXCLASS_UNKNOWN:((
            uValueLength > 4)?LEXCLASS_TYPE_INT:((
            uValueLength > 2)?LEXCLASS_TYPE_SHORT:((
            uValueLength >
0)?LEXCLASS_TYPE_CHAR:LEXCLASS_UNKNOWN)));
};

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semRetSregOpcodeByType(UINT uType)
{
      switch (uType)
      {
      case LEXCLASS_IA32REGISTERS_CS:
            return SREG_CS_1;
            break;
      case LEXCLASS_IA32REGISTERS_ES:
            return SREG_ES_0;
            break;
      case LEXCLASS_IA32REGISTERS_FS:
            return SREG_FS_4;
            break;
      case LEXCLASS_IA32REGISTERS_GS:
            return SREG_GS_5;
            break;
      case LEXCLASS_IA32REGISTERS_SS:
            return SREG_SS_2;
            break;
      case LEXCLASS_IA32REGISTERS_DS:
            return SREG_DS_3;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semRetSregPrefixByType(UINT uType)
{
      switch (uType)
      {
      case LEXCLASS_IA32REGISTERS_CS:
            return prefCS;
            break;
      case LEXCLASS_IA32REGISTERS_ES:
            return prefES;
            break;
      case LEXCLASS_IA32REGISTERS_FS:
            return prefFS;
            break;
      case LEXCLASS_IA32REGISTERS_GS:
            return prefGS;
            break;
      case LEXCLASS_IA32REGISTERS_SS:
            return prefSS;
            break;
      case LEXCLASS_IA32REGISTERS_DS:
            return prefDS;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semRetRegRegisterOpcodeByType(UINT uType)
{
      switch (uType)
      {
      case LEXCLASS_IA32REGISTERS_EAX:
      case LEXCLASS_IA32REGISTERS_AX:
      case LEXCLASS_IA32REGISTERS_AL:
            return RO16REG_AL_AX_EAX_MM0_XMM0_0_000;
            break;
      case LEXCLASS_IA32REGISTERS_ECX:
      case LEXCLASS_IA32REGISTERS_CX:
      case LEXCLASS_IA32REGISTERS_CL:
            return RO16REG_CL_CX_ECX_MM1_XMM1_1_001;
            break;
      case LEXCLASS_IA32REGISTERS_EDX:
      case LEXCLASS_IA32REGISTERS_DX:
      case LEXCLASS_IA32REGISTERS_DL:
            return RO16REG_DL_DX_EDX_MM2_XMM2_2_010;
            break;
      case LEXCLASS_IA32REGISTERS_EBX:
      case LEXCLASS_IA32REGISTERS_BX:
      case LEXCLASS_IA32REGISTERS_BL:
            return RO16REG_BL_BX_EBX_MM3_XMM3_3_011;
            break;
      case LEXCLASS_IA32REGISTERS_ESP:
      case LEXCLASS_IA32REGISTERS_SP:
      case LEXCLASS_IA32REGISTERS_AH:
            return RO16REG_AH_SP_ESP_MM4_XMM4_4_100;
            break;
      case LEXCLASS_IA32REGISTERS_EBP:
```

```
      case LEXCLASS_IA32REGISTERS_BP:
      case LEXCLASS_IA32REGISTERS_CH:
            return RO16REG_CH_BP_EBP_MM5_XMM5_5_101;
            break;
      case LEXCLASS_IA32REGISTERS_ESI:
      case LEXCLASS_IA32REGISTERS_SI:
      case LEXCLASS_IA32REGISTERS_DH:
            return RO16REG_DH_SI_ESI_MM6_XMM6_6_110;
            break;
      case LEXCLASS_IA32REGISTERS_EDI:
      case LEXCLASS_IA32REGISTERS_DI:
      case LEXCLASS_IA32REGISTERS_BH:
            return RO16REG_BH_DI_EDI_MM7_XMM7_7_111;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semRetRegEffectivAddrByType(UINT uType)
{
      switch (uType)
      {
      case LEXCLASS_IA32REGISTERS_EAX:
      case LEXCLASS_IA32REGISTERS_AX:
      case LEXCLASS_IA32REGISTERS_AL:
            return EA16REG_EAX_AX_AL_MM0_XMM0;
            break;
      case LEXCLASS_IA32REGISTERS_ECX:
      case LEXCLASS_IA32REGISTERS_CX:
      case LEXCLASS_IA32REGISTERS_CL:
            return EA16REG_ECX_CX_CL_MM1_XMM1;
            break;
      case LEXCLASS_IA32REGISTERS_EDX:
      case LEXCLASS_IA32REGISTERS_DX:
      case LEXCLASS_IA32REGISTERS_DL:
            return EA16REG_EDX_DX_DL_MM2_XMM2;
            break;
      case LEXCLASS_IA32REGISTERS_EBX:
      case LEXCLASS_IA32REGISTERS_BX:
      case LEXCLASS_IA32REGISTERS_BL:
            return EA16REG_EBX_BX_BL_MM3_XMM3;
            break;
      case LEXCLASS_IA32REGISTERS_ESP:
      case LEXCLASS_IA32REGISTERS_SP:
      case LEXCLASS_IA32REGISTERS_AH:
            return EA16REG_ESP_SP_AH_MM4_XMM4;
            break;
      case LEXCLASS_IA32REGISTERS_EBP:
      case LEXCLASS_IA32REGISTERS_BP:
      case LEXCLASS_IA32REGISTERS_CH:
            return EA16REG_EBP_BP_CH_MM5_XMM5;
            break;
      case LEXCLASS_IA32REGISTERS_ESI:
      case LEXCLASS_IA32REGISTERS_SI:
```

```
      case LEXCLASS_IA32REGISTERS_DH:
            return EA16REG_ESI_SI_DH_MM6_XMM6;
            break;
      case LEXCLASS_IA32REGISTERS_EDI:
      case LEXCLASS_IA32REGISTERS_DI:
      case LEXCLASS_IA32REGISTERS_BH:
            return EA16REG_EDI_DI_BH_MM7_XMM7;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semRetRegRWByType(UINT uType)
{
      switch (uType)
      {
      case LEXCLASS_IA32REGISTERS_EAX:
      case LEXCLASS_IA32REGISTERS_AX:
      case LEXCLASS_IA32REGISTERS_AL:
            return RW_AX;
            break;
      case LEXCLASS_IA32REGISTERS_ECX:
      case LEXCLASS_IA32REGISTERS_CX:
      case LEXCLASS_IA32REGISTERS_CL:
            return RW_CX;
            break;
      case LEXCLASS_IA32REGISTERS_EDX:
      case LEXCLASS_IA32REGISTERS_DX:
      case LEXCLASS_IA32REGISTERS_DL:
            return RW_DX;
            break;
      case LEXCLASS_IA32REGISTERS_EBX:
      case LEXCLASS_IA32REGISTERS_BX:
      case LEXCLASS_IA32REGISTERS_BL:
            return RW_BX;
            break;
      case LEXCLASS_IA32REGISTERS_ESP:
      case LEXCLASS_IA32REGISTERS_SP:
      case LEXCLASS_IA32REGISTERS_AH:
            return RW_SP;
            break;
      case LEXCLASS_IA32REGISTERS_EBP:
      case LEXCLASS_IA32REGISTERS_BP:
      case LEXCLASS_IA32REGISTERS_CH:
            return RW_BP;
            break;
      case LEXCLASS_IA32REGISTERS_ESI:
      case LEXCLASS_IA32REGISTERS_SI:
      case LEXCLASS_IA32REGISTERS_DH:
            return RW_SI;
            break;
      case LEXCLASS_IA32REGISTERS_EDI:
      case LEXCLASS_IA32REGISTERS_DI:
      case LEXCLASS_IA32REGISTERS_BH:
            return RW_DI;
            break;
      default:
            return FALSE;
            break;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semCheckContrElemBlock(TOperationElemPtr pBegArgList)
{
      TOperationElemPtr pTempOE;
      TBlockPtr               pTempBlock;
      pTempOE = pBegArgList->pNextOperationElem;
      pTempBlock = pTempOE->pArgument.pvValue;

      pTempOE = pBegArgList;
      pTempBlock->pExternalBlock = pTempOE-
>pArgument.pvValue;
      semCheckBlock(pTempBlock);
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int semCheckBlock(TBlockPtr pBlock)
{
      unsigned int uStackOffset = 0;
      TOperationElemPtr pBlockLocVar = NULL, pTempOE = NULL;
      TContrElemPtr           pContrElem = NULL, pTempCE =
NULL;
      TIdTablePtr             pTempIdTableElem = NULL;

      pBlockLocVar = pBlock->pBegLocVarList;
      while (pBlockLocVar)
      {
            pTempIdTableElem = (TIdTablePtr)pBlockLocVar-
>pArgument.pvValue;
            pTempIdTableElem->uStackOffset = uStackOffset;

            if (pTempIdTableElem->pBegIdDecl)
            {
                  uStackOffset += 2; // for 16-bit address
mode
            }
            else
            {
                  uStackOffset +=
semRetTypeSizeByType(pBlockLocVar->pArgument.uLexClass);
            }
            pBlockLocVar = pBlockLocVar->pNextOperationElem;
      }
      uStackOffset = uStackOffset + 
semRetTypeSizeByType(LEXCLASS_IA32REGISTERS_BP);
      pBlock->uLocVarSize = uStackOffset;
      pContrElem = pBlock->pBegContrList;
      while (pContrElem)
      {
            switch (pContrElem->uSemClass)
            {
            case SEMCLASS_BLOCK:
                  semCheckContrElemBlock(pContrElem-
>pBegArgList);
                  break;
            case SEMCLASS_EQUATION:
                  semCheckContrElemEquation(pContrElem-
>pBegArgList);
                  break;
            case SEMCLASS_FUNCRETURN:
                  semCheckContrElemFuncRet(pContrElem-
>pBegArgList);
                  break;
            case SEMCLASS_CONSTRUCTION_IF:
                  // IF Condition checking
                  pTempOE = pContrElem->pBegArgList;
                  pTempCE = pTempOE->pArgument.pvValue;
                  semCheckContrElemEquation(pTempCE-
>pBegArgList);

                  // IF Block checking
                  pTempOE = pTempOE->pNextOperationElem;
                  pTempCE = pTempOE->pArgument.pvValue;
                  semCheckContrElemBlock(pTempCE-
>pBegArgList);

                  // ELSE Block checking
                  pTempOE = pTempOE->pNextOperationElem;
                  if (pTempOE && (pTempOE-
>pArgument.uSemClass == SEMCLASS_CONSTRUCTION_ELSE))
                  {
                        pTempCE = pTempOE->pArgument.pvValue;
                        semCheckContrElemBlock(pTempCE-
>pBegArgList);
                  }
                  break;
            case SEMCLASS_CONSTRUCTION_WHILE:
                  //printf("$$$$$\n");
                  // WHILE Condition checking
                  pTempOE = pContrElem->pBegArgList;
                  pTempCE = pTempOE->pArgument.pvValue;
                  semCheckContrElemEquation(pTempCE-
>pBegArgList);

                  // WHILE Block checking
                  pTempOE = pTempOE->pNextOperationElem;
                  pTempCE = pTempOE->pArgument.pvValue;
                  semCheckContrElemBlock(pTempCE-
>pBegArgList);

                  break;
            case SEMCLASS_INSTRUCTION:
                  break;
            }
            pContrElem = pContrElem->pNextContrElem;
      }

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
void semAnalisys()
{
      unsigned int uStackOffset = 0;
      extern TParseListNodePtr      pBegParseListNode;

      TParseListNodePtr pParseListNode = NULL;
      TOperationElemPtr pFuncArg = NULL;

      TOperationElemPtr pContrElemArg = NULL;
      TBlockPtr              pBlock;
      TIdTablePtr            pTempIdTableElem = NULL;

      pParseListNode = pBegParseListNode;
      while (pParseListNode)
      {
            uStackOffset = 0;
            pFuncArg = pParseListNode->FuncNode.pBegArgList;
            if (pParseListNode->FuncNode.uType !=
LEXCLASS_TYPE_STRUCT)
            {
                  uStackOffset += RETURNSIZE16;
                  while (pFuncArg)
                  {
                        pTempIdTableElem = pFuncArg-
>pArgument.pvValue;
                        pTempIdTableElem->uStackOffset =
uStackOffset;

                        if (pTempIdTableElem->pBegIdDecl)
                              uStackOffset += 2; // for 16-
bit address mode
                        else
                              uStackOffset +=
semRetTypeSizeByType(pFuncArg->pArgument.uLexClass);


                        pFuncArg = pFuncArg-
>pNextOperationElem;
                  }

                  pParseListNode->FuncNode.pFuncBlock-
>uLocVarSize = uStackOffset;
                  pBlock = pParseListNode-
>FuncNode.pFuncBlock;
                  if (pBlock)
                  {
                        semCheckBlock(pBlock);
                  }
            }
            pParseListNode = pParseListNode->pNextFuncNode;
      }
```

```
}
```

Figure 8. analysis_sem.c

```
#ifndef INCL_ANALYSIS_CG_H
#define INCL_ANALYSIS_CG_H

#include "main.h"

/** CODE GENERATION **/

// General functions
int         cgCodeCreating();
void  cgPrintFuncCode(char bIsScreenPrint);

// Code generation functions
int         cgContrElemFuncRet(TOperationElemPtr
pBegArgList);
int         cgContrElemFuncCall(TOperationElemPtr
pBegArgList);
int         cgContrElemExpression(TOperationElemPtr
pBegArgList);
int         cgContrElemEquation(TOperationElemPtr
pBegArgList);
int         cgContrElemInstruction(TOperationElemPtr
pBegArgList);
int         cgContrElemConstructionIf(TOperationElemPtr
pBegArgList);
int         cgContrElemConstructionWhile(TOperationElemPtr
pBegArgList);
int         cgContrElemBlock(TOperationElemPtr pBegArgList);

// Code structures filling functions
int         cgAddNewFunctionToList(UINT uType, TIdTablePtr
pFuncId);
int         cgAddNewInstructionToFunc(UINT uInstrType);
int         cgAddNewByteToCurInstruction(unsigned char cByte,
char bIncrement);

// Size and absolute address location determinative functions
int         cgFuncSizeDetection();
int         cgFuncOffsetDetection();

// Absolute address substitution
int         cgCreateCallLink( TInstructionListPtr
pInstruction, TIdTablePtr pFuncId );

// Uniting code functions
int         cgLinking();

// Output binary file creation
void  cgBinFileGeneration(char * cpInFileName);

#endif //INCL_ANALYSIS_CG_H
```

Figure 9. analysis_cg.h

```
#include "analysis_lex.h"
#include "analysis_syn.h"
#include "analysis_sem.h"
#include "analysis_cg.h"

/*
*
```

```
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgFuncSizeDetection()
{
      extern TFuncListPtr pBegFuncList;
      TFuncListPtr           pTempFunction;
      TInstructionListPtr pTempInstruction;
      UINT                   uFuncSize;

      pTempFunction = pBegFuncList;

      while (pTempFunction)
      {
            uFuncSize = 0;
            pTempInstruction = pTempFunction-
>Function.pInstructionList;
            while (pTempInstruction)
            {
                  uFuncSize += pTempInstruction-
>Instruction.uCodeSize;
                  pTempInstruction = pTempInstruction-
>pNextInstruction;

            }
            pTempFunction->Function.uSize = uFuncSize;
            pTempFunction = pTempFunction->pNextFuncCode;
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgFuncOffsetDetection()
{
      extern UINT uProgrammOffset;
      extern TFuncListPtr pBegFuncList;
      TFuncListPtr           pTempFunction;
      UINT                   uFuncOffset;

      pTempFunction = pBegFuncList;

      uFuncOffset = uProgrammOffset; // +4 for main function
call instruction
      uFuncOffset += pTempFunction->Function.uSize;
      pTempFunction = pTempFunction->pNextFuncCode;
      while (pTempFunction)
      {
            pTempFunction->Function.pFuncId->uStackOffset =
uFuncOffset;
            uFuncOffset += pTempFunction->Function.uSize;
            pTempFunction = pTempFunction->pNextFuncCode;
      }
      return TRUE;
}
```

```
/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgLinking()
{
      extern TCallLinkPtr pBegCallLinkList;
      TCallLinkPtr pTempCallLink;
      unsigned char   ucByte;
      unsigned short    usWord;

      pCurInstruction = mainCallLink.pInstruction;
      pCurByte = NULL;
      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_IMM16 + RW_SI, FALSE);

      usWord = (unsigned short)mainCallLink.pFuncId-
>uStackOffset;
      ucByte = (unsigned char)usWord;
      cgAddNewByteToCurInstruction( ucByte, FALSE);
      ucByte = usWord >> 8;
      cgAddNewByteToCurInstruction( ucByte, FALSE);

      pTempCallLink = pBegCallLinkList;
      while (pTempCallLink)
      {
            pCurInstruction = pTempCallLink->pInstruction;
            pCurByte = NULL;
            cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_IMM16 + RW_SI, FALSE);

            usWord = (unsigned short)pTempCallLink->pFuncId-
>uStackOffset;
            ucByte = (unsigned char)usWord;
            cgAddNewByteToCurInstruction( ucByte, FALSE);
            ucByte = usWord >> 8;
            cgAddNewByteToCurInstruction( ucByte, FALSE);

            pTempCallLink = pTempCallLink->pNextCallLink;
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
void cgPrintFuncCode(char bIsScreenPrint)
{
      extern TFuncListPtr pBegFuncList;
      TFuncListPtr            pTempFunction;
      TInstructionListPtr pTempInstruction;
      TByteListPtr            pTempByte;
    FILE * pOutInfoFile;
    pOutInfoFile = fopen("CompileInfo.txt","a");

      pTempInstruction = pBegFuncList-
>Function.pInstructionList;

      fprintf(pOutInfoFile, "\n Function address: %X\n",
uProgrammOffset);
      fprintf(pOutInfoFile, " Instruction size | Code\n");
      if (bIsScreenPrint)
      {
            printf("\n Function address: %X\n",
uProgrammOffset);
            printf(" Instruction size | Code\n");
      }
      while (pTempInstruction)
      {
            if (bIsScreenPrint)
                  printf(" %16X   ",pTempInstruction-
>Instruction.uCodeSize );
            fprintf(pOutInfoFile, " %16X
",pTempInstruction->Instruction.uCodeSize );

            pTempByte = pTempInstruction-
>Instruction.pByteList;
            while(pTempByte)
            {
                  if (bIsScreenPrint)
                        printf("%X ",(unsigned
char)pTempByte->cByte);
                  fprintf(pOutInfoFile, "%X ",(unsigned
char)pTempByte->cByte);
                  pTempByte = pTempByte->pNextByte;
            }
            if (bIsScreenPrint)
                  printf("\n");
            fprintf(pOutInfoFile, "\n");
            pTempInstruction = pTempInstruction-
>pNextInstruction;
      }
      pTempFunction = pBegFuncList->pNextFuncCode;
      while (pTempFunction)
      {
            pTempInstruction = pTempFunction-
>Function.pInstructionList;
            if (bIsScreenPrint)
            {
                  printf("\n Function address: %X\n",
pTempFunction->Function.pFuncId->uStackOffset);
                  printf(" Instruction size | Code\n");
            }
            fprintf(pOutInfoFile, "\n Function address:
%X\n", pTempFunction->Function.pFuncId->uStackOffset);
            fprintf(pOutInfoFile, " Instruction size |
Code\n");
            while (pTempInstruction)
            {
                  if (bIsScreenPrint)
                        printf(" %16X   ",pTempInstruction-
>Instruction.uCodeSize );
                  fprintf(pOutInfoFile, " %16X
",pTempInstruction->Instruction.uCodeSize );

                  pTempByte = pTempInstruction-
>Instruction.pByteList;
```

```
                  while(pTempByte)
                  {
                        if (bIsScreenPrint)
                              printf("%X ",(unsigned
char)pTempByte->cByte);
                        fprintf(pOutInfoFile, "%X ",(unsigned
char)pTempByte->cByte);
                        pTempByte = pTempByte->pNextByte;
                  }
                  if (bIsScreenPrint)
                        printf("\n");
                  fprintf(pOutInfoFile, "\n");

                  pTempInstruction = pTempInstruction-
>pNextInstruction;
            }

            pTempFunction = pTempFunction->pNextFuncCode;
      }
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
void cgBinFileGeneration(char * cpInFileName)
{
      FILE * pBinOut;
      extern TFuncListPtr pBegFuncList;
      TFuncListPtr           pTempFunction;
      TInstructionListPtr pTempInstruction;
      TByteListPtr           pTempByte;

      pBinOut = fopen(cpInFileName, "wb");

      pTempInstruction = pBegFuncList-
>Function.pInstructionList;
      while (pTempInstruction)
      {
            pTempByte = pTempInstruction-
>Instruction.pByteList;
            while(pTempByte)
            {
                  putc((unsigned char)pTempByte-
>cByte,pBinOut);
                  pTempByte = pTempByte->pNextByte;
            }
            pTempInstruction = pTempInstruction-
>pNextInstruction;
      }
      pTempFunction = pBegFuncList->pNextFuncCode;
      while (pTempFunction)
      {
            pTempInstruction = pTempFunction-
>Function.pInstructionList;
            while (pTempInstruction)
            {
                  pTempByte = pTempInstruction-
>Instruction.pByteList;
                  while(pTempByte)
                  {
                        putc((unsigned char)pTempByte-
>cByte,pBinOut);
                        pTempByte = pTempByte->pNextByte;
                  }
                  pTempInstruction = pTempInstruction-
>pNextInstruction;
            }
            pTempFunction = pTempFunction->pNextFuncCode;
      }
      fclose(pBinOut);
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgCreateCallLink( TInstructionListPtr pInstruction,
TIdTablePtr pFuncId )
{
      extern TCallLinkPtr           pBegCallLinkList;
      extern TCallLinkPtr           pCurCallLinkList;

      TCallLinkPtr pNewCallLink;

      pNewCallLink = (TCallLinkPtr)malloc(sizeof(TCallLink));
      pNewCallLink->pFuncId = pFuncId;
      pNewCallLink->pInstruction = pInstruction;
      pNewCallLink->pNextCallLink = NULL;
      pNewCallLink->pPrevCallLink = NULL;

      if (pCurCallLinkList)
      {
            pCurCallLinkList->pNextCallLink = pNewCallLink;
            pNewCallLink->pPrevCallLink = pCurCallLinkList;
            pCurCallLinkList = pNewCallLink;
      }

      if (!pBegCallLinkList)
      {
            pBegCallLinkList = pNewCallLink;
            pCurCallLinkList = pNewCallLink;
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgAddNewFunctionToList(UINT uType, TIdTablePtr pFuncId)
{
      extern TFuncListPtr           pBegFuncList;
      extern TFuncListPtr           pCurFunction;
      TFuncListPtr pNewFunction;
      pNewFunction = (TFuncListPtr)malloc(sizeof(TFuncList));
      pNewFunction->Function.pInstructionList = NULL;
      pNewFunction->Function.uSize = 0;
      pNewFunction->Function.uType = uType;
      pNewFunction->Function.pFuncId = pFuncId;
      pNewFunction->pNextFuncCode = NULL;
      pNewFunction->pPrevFuncCode = NULL;
      pCurInstruction = NULL;
      if (pCurFunction)
      {
            pCurFunction->pNextFuncCode = pNewFunction;
            pNewFunction->pPrevFuncCode = pCurFunction;
            pCurFunction = pNewFunction;
      }

      if (!pBegFuncList)
      {
            pBegFuncList = pNewFunction;
            pCurFunction = pNewFunction;
      }

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgAddNewInstructionToFunc(UINT uInstrType)
{
      extern TFuncListPtr                 pCurFunction;
      extern TInstructionListPtr   pCurInstruction;

      TInstructionListPtr                 pNewInstruction;

      pNewInstruction =
(TInstructionListPtr)malloc(sizeof(TInstructionList));
      pNewInstruction->Instruction.pByteList = NULL;
      pNewInstruction->Instruction.uCodeSize = 0;
      pNewInstruction->Instruction.uInstrType = uInstrType;
      pNewInstruction->pNextInstruction = NULL;
      pNewInstruction->pPrevInstruction = NULL;
      pCurByte = NULL;
      if (pCurInstruction)
      {
            pCurInstruction->pNextInstruction =
pNewInstruction;
            pNewInstruction->pPrevInstruction =
pCurInstruction;
            pCurInstruction = pNewInstruction;
      }

      if (!pCurFunction->Function.pInstructionList)
      {
            pCurFunction->Function.pInstructionList =
pNewInstruction;
            pCurInstruction = pNewInstruction;
      }
      return TRUE;
}
```

```
/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgAddNewByteToInstruction(TInstructionListPtr
pInstruction, unsigned char cByte, char bIncrement)
{
      TByteListPtr                        pByte;
      TByteListPtr                        pNewByte;

      pNewByte = (TByteListPtr)malloc(sizeof(TByteList));
      pNewByte->cByte = cByte;
      pNewByte->pNextByte = NULL;
      pNewByte->pPrevByte = NULL;

      pByte = pInstruction->Instruction.pByteList;

      if (pByte)
      {
            while (pByte->pNextByte)
            {
                  pByte = pByte->pNextByte;
            }
            pByte->pNextByte = pNewByte;
            pNewByte->pPrevByte = pByte;
      }
      else
      {
            pInstruction->Instruction.pByteList = pByte =
pNewByte;
      }

      if (bIncrement)
            ++pInstruction->Instruction.uCodeSize;

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgAddNewByteToCurInstruction(unsigned char cByte, char
bIncrement)
{
      extern TInstructionListPtr    pCurInstruction;
      extern TByteListPtr                 pCurByte;
      TByteListPtr                        pNewByte;

      pNewByte = (TByteListPtr)malloc(sizeof(TByteList));
      pNewByte->cByte = cByte;
      pNewByte->pNextByte = NULL;
      pNewByte->pPrevByte = NULL;
      if (pCurByte)
      {
            pCurByte->pNextByte = pNewByte;
            pNewByte->pPrevByte = pCurByte;
            pCurByte = pNewByte;
      }

      if (!pCurInstruction->Instruction.pByteList)
      {
            pCurInstruction->Instruction.pByteList =
pNewByte;
            pCurByte = pNewByte;
      }
      if (bIncrement)
      {
            ++pCurInstruction->Instruction.uCodeSize;
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgGetCurByteOffset()
{
      UINT uOffset;
      extern TFuncListPtr     pCurFunction;
      TInstructionListPtr pTempInstr;
      pTempInstr = pCurFunction->Function.pInstructionList;
      uOffset = 0;
      while(pTempInstr)
      {
            uOffset += pTempInstr->Instruction.uCodeSize;
            pTempInstr = pTempInstr->pNextInstruction;
      }

      return uOffset;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgContrElemBlock(TOperationElemPtr pBegArgList)
{
      TOperationElemPtr pTempOE;
      TBlockPtr               pTempBlock;
      unsigned short    usWord = 0;
      unsigned short    uLocVarTypeSum = 0;
      unsigned char     ucByte = 0;
      extern TParseListNodePtr      pBegParseListNode;
      extern TParseListNodePtr      pCurParseListNode;

      TOperationElemPtr pLocVar = NULL;

      TContrElemPtr           pContrElem = NULL;
      TOperationElemPtr pContrElemArg = NULL;
      TIdTablePtr             pTempIdTableElem = NULL;

      pTempOE = pBegArgList->pNextOperationElem;
      pTempBlock = pTempOE->pArgument.pvValue;
      pCurBlock = pTempBlock;

      pTempOE = pBegArgList;
      pTempBlock = pTempOE->pArgument.pvValue;
      // pTempBlock = pointer to the external block
      pCurBlock->pExternalBlock = pTempBlock;

      pLocVar = pCurBlock->pBegLocVarList;

      pCurBlock->uBlockOffset = cgGetCurByteOffset();

      // Saving previous BP
      // PUSH BP
      cgAddNewInstructionToFunc( PUSH_R16 + RW_BP );
      cgAddNewByteToCurInstruction( (unsigned char)PUSH_R16 +
RW_BP, TRUE );
      if (pLocVar)
      {
            while (pLocVar->pNextOperationElem)
            {
                  pLocVar = pLocVar->pNextOperationElem;
            }
            // Subtracting the size of all the local
variables from SP
            // (allocating the memory)
            cgAddNewInstructionToFunc( SUB_RM16_IMM16 );
            cgAddNewByteToCurInstruction( (unsigned
char)SUB_RM16_IMM16, TRUE );
            cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_CH_BP_EBP_MM5_XMM5_5_101), TRUE );
            while (pLocVar)
            {
                  pTempIdTableElem = (TIdTablePtr)pLocVar-
>pArgument.pvValue;
                  if (pTempIdTableElem->pBegIdDecl)
                        uLocVarTypeSum += 2; // for 16-bit
address mode
                  else
                        uLocVarTypeSum +=
semRetTypeSizeByType(pLocVar->pArgument.uLexClass);

                  pLocVar = pLocVar->pPrevOperationElem;
            }
            ucByte = (unsigned char)uLocVarTypeSum;
            cgAddNewByteToCurInstruction( ucByte, TRUE );
            ucByte = uLocVarTypeSum >> 8;
            cgAddNewByteToCurInstruction( ucByte, TRUE );
      }
      // BP with SP equating
      cgAddNewInstructionToFunc( MOV_R16_RM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_RM16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_CH_BP_EBP_MM5_XMM5_5_101), TRUE );


      pContrElem = pCurBlock->pBegContrList;
      while (pContrElem)
      {
            pContrElem->pContrElemHandler(pContrElem-
>pBegArgList);
            pContrElem = pContrElem->pNextContrElem;
      }

      // ADD SP,uLocVarTypeSum
      cgAddNewInstructionToFunc( ADD_RM16_IMM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)ADD_RM16_IMM16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned
char)EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000, TRUE );

      ucByte = (unsigned char)uLocVarTypeSum;
      cgAddNewByteToCurInstruction( ucByte, TRUE );
      ucByte = uLocVarTypeSum >> 8;
      cgAddNewByteToCurInstruction( ucByte, TRUE );

      // Restoring previous BP value
      // POP BP
      cgAddNewInstructionToFunc( POP_R16 );
      cgAddNewByteToCurInstruction( (unsigned char)POP_R16 +
RW_BP, TRUE );

      pCurBlock->uBlockSize = cgGetCurByteOffset() -
pCurBlock->uBlockOffset;
      pCurBlock = pCurBlock->pExternalBlock;

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgContrElemFuncRet(TOperationElemPtr pBegArgList)
{
      unsigned char   ucByte = 0;
      unsigned short    usWord = 0;
      TOperationElemPtr pTempOE;
      TContrElemPtr pTempCE;
      TBlockPtr pTempBlock = NULL;
      TIdTablePtr             pTempIdTableElem = NULL;

      TParseListNodePtr pTempParseListNode = NULL;
      unsigned short uLocVarTypeSum = 0;

      pTempOE = pBegArgList;

      pTempParseListNode = (TParseListNodePtr)pTempOE-
>pArgument.pvValue;
      pTempBlock = pCurBlock;


      while(pTempBlock)
      {
            usWord = usWord + pTempBlock->uLocVarSize;
            pTempBlock = pTempBlock->pExternalBlock;
      }
      uLocVarTypeSum = usWord;

      usWord += RETURNSIZE16;
      pTempOE = pTempParseListNode->FuncNode.pBegArgList;
      while(pTempOE)
      {
            pTempIdTableElem = (TIdTablePtr)pTempOE-
>pArgument.pvValue;


            if (pTempIdTableElem->pBegIdDecl)
                  usWord += 2; // for 16-bit address mode
            else
                  usWord += semRetTypeSizeByType(pTempOE-
>pArgument.uLexClass);

            pTempOE = pTempOE->pNextOperationElem;
      }
      usWord +=
semRetTypeSizeByType(LEXCLASS_IA32REGISTERS_BP);// size of BP
      pTempOE = pBegArgList->pNextOperationElem;
      pTempCE = (TContrElemPtr)pTempOE->pArgument.pvValue;
      pTempCE->pContrElemHandler(pTempCE->pBegArgList);
      // now AX   = the result of return equation
      // usWord = offset to the return value memory location
      // uLocVarTypeSum = offset to the return address


      // Copying AX to the return value 16-bit memory
location
      // MOV [bp + usWord],AX
      cgAddNewInstructionToFunc( MOV_RM16_R16 );
      cgAddNewByteToCurInstruction( (unsigned
char)MOV_RM16_R16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned
char)(EA16MEMLOC_BP_DISP16 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );

      ucByte = (unsigned char)usWord;
      cgAddNewByteToCurInstruction( ucByte, TRUE);
      ucByte = usWord >> 8;
      cgAddNewByteToCurInstruction( ucByte, TRUE );

      // MOV SP,BP
      cgAddNewInstructionToFunc( MOV_R16_RM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_RM16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_EBP_BP_CH_MM5_XMM5 |
RO16REG_AH_SP_ESP_MM4_XMM4_4_100), TRUE );


      // ADD SP,uLocVarTypeSum
      cgAddNewInstructionToFunc( ADD_RM16_IMM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)ADD_RM16_IMM16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned
char)EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000, TRUE );

      ucByte = (unsigned char)uLocVarTypeSum;
      cgAddNewByteToCurInstruction( ucByte, TRUE );
      ucByte = uLocVarTypeSum >> 8;
      cgAddNewByteToCurInstruction( ucByte, TRUE );

      // RET
      cgAddNewInstructionToFunc( RET_NEAR );
      cgAddNewByteToCurInstruction( (unsigned char)RET_NEAR,
TRUE );

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgContrElemFuncCall(TOperationElemPtr pBegArgList)
{

      unsigned char   ucByte;
      unsigned short    usWord;
      TOperationElemPtr pArgDeclOE;
      TOperationElemPtr pTempOE;
      TContrElemPtr pTempCE;
      TIdTablePtr       pFuncIdTableElem = NULL,
pTempIdTableElem = NULL;

      cgAddNewInstructionToFunc( SUB_RM16_IMM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)SUB_RM16_IMM16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_CH_BP_EBP_MM5_XMM5_5_101), TRUE );

      usWord = 2; //(unsigned
short)semRetTypeSizeByType(*((UINT*)pBegArgList-
>pArgument.pvValue)); !!!!!!!!!!!!! do luchshih vremen
      //(perechital v Canade 2009-08-29, poradovalo :) )

      ucByte = (unsigned char)usWord;
      cgAddNewByteToCurInstruction( ucByte, TRUE );
      ucByte = usWord >> 8;
      cgAddNewByteToCurInstruction( ucByte, TRUE );

      // Saving previous function BP
      // PUSH BP
      cgAddNewInstructionToFunc( PUSH_R16 + RW_BP );
      cgAddNewByteToCurInstruction( (unsigned char)PUSH_R16 +
RW_BP, TRUE );

      pTempOE = pBegArgList;

      pFuncIdTableElem = pTempOE->pArgument.pvValue;
      pArgDeclOE = pFuncIdTableElem->pArgInstance;

      pTempOE = pBegArgList->pNextOperationElem;

      while (pArgDeclOE && pArgDeclOE->pNextOperationElem)
            pArgDeclOE = pArgDeclOE->pNextOperationElem;

      usWord = 0;
      if (pTempOE)
      {
            while (pTempOE->pNextOperationElem)
                  pTempOE = pTempOE->pNextOperationElem;
            while (pTempOE->pPrevOperationElem)
            {
                  pTempCE = (TContrElemPtr)pTempOE-
>pArgument.pvValue;
                  pTempCE->pContrElemHandler(pTempCE-
>pBegArgList);

                  pTempIdTableElem = pArgDeclOE-
>pArgument.pvValue;

                  if (pTempIdTableElem->pBegIdDecl)
                  {
                        usWord += 2; // for 16-bit address
mode
                        // PUSH AX
                        cgAddNewInstructionToFunc( PUSH_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)PUSH_R16 + RW_AX, TRUE );
                  }
                  else
                  {
                        switch
(semRetTypeSizeByType(pArgDeclOE->pArgument.uLexClass))
                        {
                        case 1:
                              // SUB SP, 0x0001
                              cgAddNewInstructionToFunc(
SUB_RM16_IMM16 );
                              cgAddNewByteToCurInstruction(
(unsigned char)SUB_RM16_IMM16, TRUE );
                              cgAddNewByteToCurInstruction(
(unsigned char)(EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_CH_BP_EBP_MM5_XMM5_5_101), TRUE );

      cgAddNewByteToCurInstruction(0x01, TRUE);

      cgAddNewByteToCurInstruction(0x00, TRUE);

                              // MOV SI,SP
                              cgAddNewInstructionToFunc(
MOV_R16_RM16 );
                              cgAddNewByteToCurInstruction(
(unsigned char)MOV_R16_RM16, TRUE );
                              cgAddNewByteToCurInstruction(
(unsigned char)(EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_DH_SI_ESI_MM6_XMM6_6_110), TRUE );

                              // MOV [SI],AL
                              cgAddNewInstructionToFunc(
MOV_RM8_R8 );

                              cgAddNewByteToCurInstruction(
(unsigned char)prefSS, TRUE ); // Prefix SS

                              cgAddNewByteToCurInstruction(
(unsigned char)MOV_RM8_R8, TRUE );
                              cgAddNewByteToCurInstruction(
(unsigned char)(EA16MEMLOC_SI |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );
```

```
                              break;
                        case 2:
                              // PUSH AX
                              cgAddNewInstructionToFunc(
PUSH_R16 + RW_AX );
                              cgAddNewByteToCurInstruction(
(unsigned char)PUSH_R16 + RW_AX, TRUE );
                              break;
                        }
                        usWord +=
semRetTypeSizeByType(pArgDeclOE->pArgument.uLexClass);

                  }
                  pArgDeclOE = pArgDeclOE-
>pPrevOperationElem;
                  pTempOE = pTempOE->pPrevOperationElem;
            }
      }
      pTempOE = pBegArgList;
      cgAddNewInstructionToFunc( MOV_R16_RM16 );
      pCurInstruction->Instruction.uCodeSize = 3;
      cgCreateCallLink(pCurInstruction, pTempOE-
>pArgument.pvValue);

      cgAddNewInstructionToFunc( CALL_NEAR_RM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)CALL_NEAR_RM16, TRUE);
      cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_ESI_SI_DH_MM6_XMM6 |
RO16REG_DL_DX_EDX_MM2_XMM2_2_010), TRUE);

      // Making SP to point to the memory, comprising the
previous BP value
      // ADD SP, usWord
      cgAddNewInstructionToFunc( ADD_RM16_IMM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)ADD_RM16_IMM16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned
char)EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000, TRUE );

      ucByte = (unsigned char)usWord;
      cgAddNewByteToCurInstruction( ucByte, TRUE );
      ucByte = usWord >> 8;
      cgAddNewByteToCurInstruction( ucByte, TRUE);

      // Restoring previous BP value
      // POP BP
      cgAddNewInstructionToFunc( POP_R16 );
      cgAddNewByteToCurInstruction( (unsigned char)POP_R16 +
RW_BP, TRUE );

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgContrElemExpression(TOperationElemPtr pBegArgList)
```

```
{
      unsigned char   ucByte;
      unsigned short   usWord;
      TOperationElemPtr pTempOE;
      TContrElemPtr pTempCE;
      TBlockPtr pTempBlock = NULL;
      TBlockPtr pTempCurBlock = NULL;
      TIdTablePtr pTempIdTable;
      TIDCurStateListPtr pTempCurState = NULL;
      UINT  uCounter = 0;
      UINT  uTotalIndLevelCnt = 0;
      UINT  uCurTtlIndLevelCntContainer = 0;

      pTempOE = pBegArgList;
      while (pTempOE)
      {
            if(pTempOE->pArgument.uSemClass ==
SEMCLASS_IMM16)
            {
                  cgAddNewInstructionToFunc( PUSH_IMM16 );
                  cgAddNewByteToCurInstruction( (unsigned
char)PUSH_IMM16, TRUE );

                  usWord = (unsigned short)*((UINT *)pTempOE-
>pArgument.pvValue);

                  ucByte = (unsigned char)usWord;
                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
                  ucByte = (unsigned char)(usWord>>8);

                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
            }

            if(pTempOE->pArgument.uSemClass == SEMCLASS_IMM8)
            {
                  cgAddNewInstructionToFunc( PUSH_IMM16 );
                  cgAddNewByteToCurInstruction( (unsigned
char)PUSH_IMM16, TRUE );
                  ucByte = (unsigned char)*((UINT *)pTempOE-
>pArgument.pvValue);
                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
                  cgAddNewByteToCurInstruction( 0, TRUE );
            }

            if(pTempOE->pArgument.uSemClass ==
SEMCLASS_OPERATION)
            {
                  switch(pTempOE->pArgument.uLexClass)
                  {
                  case LEXCLASS_OPERATION_SLASH:
                        // POP BX
                        cgAddNewInstructionToFunc( POP_R16 +
RW_BX );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_BX, TRUE );
                        // POP AX
                        cgAddNewInstructionToFunc( POP_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_AX, TRUE );
```

```
                        // MOV DX,0x0000
                        cgAddNewInstructionToFunc(
MOV_R16_IMM16 + RW_DX );
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_R16_IMM16 + RW_DX, TRUE );
                        cgAddNewByteToCurInstruction( 0x00,
TRUE );
                        cgAddNewByteToCurInstruction( 0x00,
TRUE );

                        // DIV BX
                        cgAddNewInstructionToFunc( DIV_RM16
);
                        cgAddNewByteToCurInstruction(
(unsigned char)DIV_RM16, TRUE );
                        cgAddNewByteToCurInstruction(
(unsigned char)(EA16REG_EBX_BX_BL_MM3_XMM3 |
RO16REG_DH_SI_ESI_MM6_XMM6_6_110), TRUE );

                        // PUSH AX
                        cgAddNewInstructionToFunc( PUSH_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)PUSH_R16 + RW_AX, TRUE );
                        break;
                  case LEXCLASS_OPERATION_ASTERISK:
                        // POP BX
                        cgAddNewInstructionToFunc( POP_R16 +
RW_BX );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_BX, TRUE );
                        // POP AX
                        cgAddNewInstructionToFunc( POP_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_AX, TRUE );

                        // MUL BX
                        cgAddNewInstructionToFunc( MUL_RM16
);
                        cgAddNewByteToCurInstruction(
(unsigned char)MUL_RM16, TRUE );
                        cgAddNewByteToCurInstruction(
(unsigned char)(EA16REG_EBX_BX_BL_MM3_XMM3 |
RO16REG_AH_SP_ESP_MM4_XMM4_4_100), TRUE );

                        // PUSH AX
                        cgAddNewInstructionToFunc( PUSH_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)PUSH_R16 + RW_AX, TRUE );
                        break;
                  case LEXCLASS_OPERATION_PLUS:
                        // POP AX
                        cgAddNewInstructionToFunc( POP_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_AX, TRUE );
                        // POP BX
                        cgAddNewInstructionToFunc( POP_R16 +
RW_BX );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_BX, TRUE );
                        // ADD AX,BX
                        cgAddNewInstructionToFunc(
ADD_R16_RM16 );
                        cgAddNewByteToCurInstruction(
(unsigned char)ADD_R16_RM16, TRUE );
                        cgAddNewByteToCurInstruction(
(unsigned char)(EA16REG_EBX_BX_BL_MM3_XMM3 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );
                        // PUSH AX
                        cgAddNewInstructionToFunc( PUSH_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)PUSH_R16 + RW_AX, TRUE );
                        break;
                  case LEXCLASS_OPERATION_MINUS:
                        // POP BX
                        cgAddNewInstructionToFunc( POP_R16 +
RW_BX );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_BX, TRUE );
                        // POP AX
                        cgAddNewInstructionToFunc( POP_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_AX, TRUE );
                        // SUB AX,BX
                        cgAddNewInstructionToFunc(
SUB_R16_RM16 );
                        cgAddNewByteToCurInstruction(
(unsigned char)SUB_R16_RM16, TRUE );
                        cgAddNewByteToCurInstruction(
(unsigned char)(EA16REG_EBX_BX_BL_MM3_XMM3 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );
                        // PUSH AX
                        cgAddNewInstructionToFunc( PUSH_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)PUSH_R16 + RW_AX, TRUE );
                        break;
                  }
            }

      ////////////////////////////////////////////////////////
            if(pTempOE->pArgument.uSemClass ==
SEMCLASS_MEM16)
            {
                  cgAddNewInstructionToFunc( PUSH_RM16 );
                  cgAddNewByteToCurInstruction( (unsigned
char)PUSH_RM16, TRUE );
                  cgAddNewByteToCurInstruction( (unsigned
char)(EA16MEMLOC_BP_DISP16 |
RO16REG_DH_SI_ESI_MM6_XMM6_6_110), TRUE );

                  usWord = 0;

                  pTempCurState =
(TIDCurStateListPtr)pTempOE->pArgument.pvValue;

                  uTotalIndLevelCnt = pTempCurState-
>pCurIdState->uTotalIndLevelCnt;
                  uCurTotalIndLevelCnt = uTotalIndLevelCnt;

                  pTempIdTable = pTempCurState->pCurIdState-
>pIdTableElem;
```

```
                  pTempCurBlock = pCurBlock;
                  pTempBlock = pTempIdTable->pBlock;

                  while(pTempCurBlock != pTempBlock)
                  {
                        usWord = usWord + pTempCurBlock-
>uLocVarSize;
                        pTempCurBlock = pTempCurBlock-
>pExternalBlock;
                  }

                  //usWord = usWord + *((unsigned short
*)pTempIdTable->pArgInstance->pArgument.pvValue);
///!!!!!!!!!! Change this

                  usWord = usWord + pTempIdTable-
>uStackOffset;

                  ucByte = (unsigned char)usWord;
                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
                  ucByte = (unsigned char)(usWord>>8);

                  cgAddNewByteToCurInstruction( ucByte, TRUE
);

                  if (uCurTotalIndLevelCnt == 0)
                  switch(semRetTypeSizeByType(pTempOE-
>pArgument.uLexClass))
                  {
                  case 1:
                        // POP AX
                        cgAddNewInstructionToFunc( POP_R16 );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_AX, TRUE );

                        // MOV AH,0x00
                        cgAddNewInstructionToFunc(
MOV_R8_IMM8 + RB_AH );
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_R8_IMM8 + RB_AH, TRUE );
                        cgAddNewByteToCurInstruction( 0x00,
TRUE );

                        // PUSH AX
                        cgAddNewInstructionToFunc( PUSH_R16 +
RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)PUSH_R16 + RW_AX, TRUE );

                        break;
                  case 2:
                        break;
                  }


                  pTempCurState = pTempCurState-
>pNextCurStateElem;
                  while(pTempCurState && (pTempCurState-
>pCurIdState->uStateType != SYNIDCURSTATETYPE_OPERATION))
                  {
                        switch (pTempCurState->pCurIdState-
>uStateType)
                        {
```

```
                        case
SYNIDCURSTATETYPE_INDEX_DEREFERENCING:
                              uCurTtlIndLevelCntContainer =
uCurTotalIndLevelCnt;
                              pTempCE =
(TContrElemPtr)pTempCurState->pCurIdState->pvValue;
                              pTempCE-
>pContrElemHandler(pTempCE->pBegArgList);
                              uCurTotalIndLevelCnt =
uCurTtlIndLevelCntContainer;
                              uCurTotalIndLevelCnt --;

                              // POP BX
                              cgAddNewInstructionToFunc(
POP_R16 );
                              cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_BX, TRUE );

                              // ADD AX,BX
                              cgAddNewInstructionToFunc(
ADD_R16_RM16 );
                              cgAddNewByteToCurInstruction(
(unsigned char)ADD_R16_RM16, TRUE );
                              cgAddNewByteToCurInstruction(
(unsigned char)(EA16REG_EBX_BX_BL_MM3_XMM3 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );

                              if (!bIsLValue)
                              {
                                    // MOV SI,AX
                                    cgAddNewInstructionToFunc(
MOV_R16_RM16 );

      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_RM16, TRUE );

      cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_EAX_AX_AL_MM0_XMM0 |
RO16REG_DH_SI_ESI_MM6_XMM6_6_110), TRUE );

                                    // MOV AX,[SI]
                                    cgAddNewInstructionToFunc(
MOV_R16_RM16 );

      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_RM16, TRUE );

      cgAddNewByteToCurInstruction( (unsigned
char)(EA16MEMLOC_SI | RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE
);

                                    if (uCurTotalIndLevelCnt
== 0)

      switch(semRetTypeSizeByType(pTempOE-
>pArgument.uLexClass))
                                    {
                                    case 1:
                                          // MOV AH,0x00

      cgAddNewInstructionToFunc( MOV_R8_IMM8 + RB_AH );

      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R8_IMM8 + RB_AH, TRUE );
```

```
      cgAddNewByteToCurInstruction( 0x00, TRUE );
                                    break;
                              case 2:
                                    break;
                              }
                        }

                        // PUSH AX
                        cgAddNewInstructionToFunc(
PUSH_R16 + RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)PUSH_R16 + RW_AX, TRUE );

                        break;
                  case SYNIDCURSTATETYPE_DEREFERENCING:
                        uCounter = pTempCurState-
>pCurIdState->uIndLvlCount;
                        // POP AX
                        cgAddNewInstructionToFunc(
POP_R16 );
                        cgAddNewByteToCurInstruction(
(unsigned char)POP_R16 + RW_AX, TRUE );

                        if (bIsLValue)
                              uCounter--;

                        while(uCounter > 0)
                        {
                              // MOV SI,AX
                              cgAddNewInstructionToFunc(
MOV_R16_RM16 );

      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_RM16, TRUE );

      cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_EAX_AX_AL_MM0_XMM0 |
RO16REG_DH_SI_ESI_MM6_XMM6_6_110), TRUE );

                              // MOV AX,[SI]
                              cgAddNewInstructionToFunc(
MOV_R16_RM16 );

      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_RM16, TRUE );

      cgAddNewByteToCurInstruction( (unsigned
char)(EA16MEMLOC_SI | RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE
);

                              uCurTotalIndLevelCnt--;
                              uCounter--;
                        }
                        uCounter = pTempCurState-
>pCurIdState->uIndLvlCount;

                        if (uCurTotalIndLevelCnt == 0)

      switch(semRetTypeSizeByType(pTempOE-
>pArgument.uLexClass))
                        {
                        case 1:
                              // MOV AH,0x00
                              cgAddNewInstructionToFunc(
MOV_R8_IMM8 + RB_AH );

      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R8_IMM8 + RB_AH, TRUE );


      cgAddNewByteToCurInstruction( 0x00, TRUE );
                              break;
                        case 2:
                              break;
                        }

                        // PUSH AX
                        cgAddNewInstructionToFunc(
PUSH_R16 + RW_AX );
                        cgAddNewByteToCurInstruction(
(unsigned char)PUSH_R16 + RW_AX, TRUE );

                        break;
                  }
                  pTempCurState = pTempCurState-
>pNextCurStateElem;
                  }

            }

            if(pTempOE->pArgument.uSemClass ==
SEMCLASS_FUNCTION)
            {
                  pTempCE = (TContrElemPtr)pTempOE-
>pArgument.pvValue;
                  pTempCE->pContrElemHandler(pTempCE-
>pBegArgList);
            }
            pTempOE = pTempOE->pNextOperationElem;
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgContrElemEquation(TOperationElemPtr pBegArgList)
{
      TBlockPtr pTempBlock = NULL;
      TBlockPtr pTempCurBlock = NULL;
      TOperationElemPtr pTempOE;
      TContrElemPtr pTempCE;
      unsigned char   ucByte;
      unsigned short    usWord = 0;
      TIdTablePtr pTempIdTable;
      TIDCurStateListPtr pTempCurState = NULL;
      UINT  uCounter = 0;
      UINT  uTotalIndLevelCnt = 0;
      extern  UINT uCurTotalIndLevelCnt;
      uCurTotalIndLevelCnt = 0;

      pTempOE = pBegArgList;
      while (pTempOE->pNextOperationElem)
            pTempOE = pTempOE->pNextOperationElem;

      if (pTempOE)
      {
            pTempCE = (TContrElemPtr)pTempOE-
>pArgument.pvValue;
            pTempCE->pContrElemHandler(pTempCE->pBegArgList);
      }

      if(!pTempOE->pPrevOperationElem)
      {
            // POP AX
            cgAddNewInstructionToFunc( POP_R16 );
            cgAddNewByteToCurInstruction( (unsigned
char)POP_R16 + RW_AX, TRUE );

            if (uCurTotalIndLevelCnt == 0)
                  switch(semRetTypeSizeByType(pTempOE-
>pArgument.uLexClass))
                  {
                  case 1:
                        // MOV AH,0x00
                        cgAddNewInstructionToFunc(
MOV_R8_IMM8 + RB_AH );
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_R8_IMM8 + RB_AH, TRUE );
                        cgAddNewByteToCurInstruction( 0x00,
TRUE );
                        break;
                  case 2:
                        break;
                  }
      }
      pTempOE = pTempOE->pPrevOperationElem;

      while (pTempOE)
      {
            pTempCE = (TContrElemPtr)pTempOE-
>pArgument.pvValue;

            pTempCurState = (TIDCurStateListPtr)pTempCE-
>pBegArgList->pArgument.pvValue;

            if (pTempCurState->pNextCurStateElem)
            {
                  // if there is an address arithmetic
                  uCurTotalIndLevelCnt = 0;
                  bIsLValue = 1;
                  pTempCE->pContrElemHandler(pTempCE-
>pBegArgList);
                  bIsLValue = 0;
                  // POP the 16-bit address, where the result
will put
                  // POP SI
                  cgAddNewInstructionToFunc( POP_R16 );
                  cgAddNewByteToCurInstruction( (unsigned
char)POP_R16 + RW_SI, TRUE );

                  // POP the result
                  // POP AX
                  cgAddNewInstructionToFunc( POP_R16 );
                  cgAddNewByteToCurInstruction( (unsigned
char)POP_R16 + RW_AX, TRUE );

                  if (uCurTotalIndLevelCnt == 0)
                  switch(semRetTypeSizeByType(pTempOE-
>pArgument.uLexClass))
                  {
                  case 1:
                        // MOV [SI],AL
                        cgAddNewInstructionToFunc( MOV_RM8_R8
);
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_RM8_R8, TRUE );
                        cgAddNewByteToCurInstruction(
(unsigned char)(EA16MEMLOC_SI |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );
                        break;
                  case 2:
                        // MOV [SI],AX
                        cgAddNewInstructionToFunc(
MOV_RM16_R16 );
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_RM16_R16, TRUE );
                        cgAddNewByteToCurInstruction(
(unsigned char)(EA16MEMLOC_SI |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );
                        break;
                  }
                  else
                  {
                        // MOV [SI],AX
                        cgAddNewInstructionToFunc(
MOV_RM16_R16 );
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_RM16_R16, TRUE );
                        cgAddNewByteToCurInstruction(
(unsigned char)(EA16MEMLOC_SI |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );
                  }

            }
            else
            {
            // if there is a variable name (without address
arithmetic),
            // than we just putting the AX value to the
address of this variable

                  usWord = 0;
                  pTempCurBlock = pCurBlock;
                  pTempIdTable = pTempCurState->pCurIdState-
>pIdTableElem;

                  uTotalIndLevelCnt = pTempCurState-
>pCurIdState->uTotalIndLevelCnt;

                  pTempBlock = pTempIdTable->pBlock;
                  while(pTempCurBlock != pTempBlock)
                  {
                        usWord = usWord + pTempCurBlock-
>uLocVarSize;
                        pTempCurBlock = pTempCurBlock-
>pExternalBlock;
```

```
                  }
                  usWord = usWord + pTempIdTable-
>uStackOffset;

                  // POP AX
                  cgAddNewInstructionToFunc( POP_R16 );
                  cgAddNewByteToCurInstruction( (unsigned
char)POP_R16 + RW_AX, TRUE );

                  if (uTotalIndLevelCnt > 0)
                  {
                        // Copying AX to the return value 16-
bit memory location (pointer type)
                        // MOV [bp + usWord],AX
                        cgAddNewInstructionToFunc(
MOV_RM16_R16 );
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_RM16_R16, TRUE );
                  }
                  else
                  {

      switch(semRetTypeSizeByType(pTempOE-
>pArgument.uLexClass))
                              {
                              case 1:
                                    // Copying AL to the
return value 8-bit memory location
                                    // MOV [bp + usWord],AL
                                    cgAddNewInstructionToFunc(
MOV_RM8_R8 );

      cgAddNewByteToCurInstruction( (unsigned
char)MOV_RM8_R8, TRUE );
                                    break;

                              case 2:
                                    // Copying AX to the
return value 16-bit memory location
                                    // MOV [bp + usWord],AX
                                    cgAddNewInstructionToFunc(
MOV_RM16_R16 );

      cgAddNewByteToCurInstruction( (unsigned
char)MOV_RM16_R16, TRUE );
                                    break;
                              }
                  }

                  cgAddNewByteToCurInstruction( (unsigned
char)(EA16MEMLOC_BP_DISP16 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );
                  ucByte = (unsigned char)usWord;
                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
                  ucByte = (unsigned char)(usWord>>8);

                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
            }
            pTempOE = pTempOE->pPrevOperationElem;
      }

      return TRUE;
```

```
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgContrElemConstructionWhile(TOperationElemPtr
pBegArgList)
{
      extern TInstructionListPtr pCurInstruction;
      TInstructionListPtr pJZInstruction = NULL,
pJMPInstruction = NULL;
      unsigned short    usWord = 0;
      unsigned char     ucByte = 0;

      TOperationElemPtr pTempOE;
      TContrElemPtr pTempCE;
      TBlockPtr pTempBlock;
      UINT uBegCondOffset = 0;
      UINT uEndCondOffset = 0;

      UINT uBegLoopOffset = 0;
      UINT uEndLoopOffset = 0;

      pTempOE = pBegArgList;

      uBegCondOffset = cgGetCurByteOffset();
      // Condition-expression implementation
      pTempCE = (TContrElemPtr)(pTempOE->pArgument.pvValue);
      pTempCE->pContrElemHandler(pTempCE->pBegArgList);
      pTempOE = pTempOE->pNextOperationElem;

      cgAddNewInstructionToFunc( TEST_RM16_R16 );
      cgAddNewByteToCurInstruction( (unsigned char)
TEST_RM16_R16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned char)
(EA16REG_EAX_AX_AL_MM0_XMM0 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );

      cgAddNewInstructionToFunc( JZ_REL16 );
      pJZInstruction = pCurInstruction;
      cgAddNewByteToCurInstruction( (unsigned char) 0x0F,
TRUE );
      cgAddNewByteToCurInstruction( (unsigned char) JZ_REL16,
TRUE );

      uEndCondOffset = cgGetCurByteOffset();
      // Block after if
      switch (pTempOE->pArgument.uSemClass)
      {
      case SEMCLASS_BLOCK:
            uBegLoopOffset = cgGetCurByteOffset();

            pTempCE = (TContrElemPtr)(pTempOE-
>pArgument.pvValue);
            pTempCE->pContrElemHandler(pTempCE->pBegArgList);
            pTempBlock = (TBlockPtr)(pTempCE->pBegArgList-
>pNextOperationElem->pArgument.pvValue);

            usWord = 0x0000 - (pTempBlock->uBlockSize + 3) -
```

```
(uEndCondOffset - uBegCondOffset + 2);
            // +2 because of the further 2 bytes adding for
pJZinstr

            cgAddNewInstructionToFunc( JMP_REL16 );
            cgAddNewByteToCurInstruction( (unsigned char)
JMP_REL16, TRUE );

            ucByte = (unsigned char)usWord;
            cgAddNewByteToCurInstruction(ucByte, TRUE );
            ucByte = (unsigned char)(usWord>>8);

            cgAddNewByteToCurInstruction(ucByte, TRUE );

            uEndLoopOffset = cgGetCurByteOffset();

            usWord = uEndLoopOffset -
uBegLoopOffset;//pTempBlock->uBlockSize + 3;

            ucByte = (unsigned char)usWord;
            cgAddNewByteToInstruction(pJZInstruction, ucByte,
TRUE );
            ucByte = (unsigned char)(usWord>>8);

            cgAddNewByteToInstruction(pJZInstruction, ucByte,
TRUE );

            pTempOE = pTempOE->pNextOperationElem;
            break;
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgContrElemConstructionIf(TOperationElemPtr pBegArgList)
{
      extern TInstructionListPtr pCurInstruction;
      TInstructionListPtr pJZInstruction = NULL,
pJMPInstruction = NULL;
      unsigned short   usWord = 0;
      unsigned char    ucByte = 0;

      TOperationElemPtr pTempOE;
      TContrElemPtr pTempCE;
      TBlockPtr pTempBlock;
      pTempOE = pBegArgList;

      // Condition-expression implementation
      pTempCE = (TContrElemPtr)(pTempOE->pArgument.pvValue);
      pTempCE->pContrElemHandler(pTempCE->pBegArgList);
      pTempOE = pTempOE->pNextOperationElem;

      cgAddNewInstructionToFunc( TEST_RM16_R16 );
      cgAddNewByteToCurInstruction( (unsigned char)
TEST_RM16_R16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned char)
(EA16REG_EAX_AX_AL_MM0_XMM0 |
RO16REG_AL_AX_EAX_MM0_XMM0_0_000), TRUE );

      cgAddNewInstructionToFunc( JZ_REL16 );
      pJZInstruction = pCurInstruction;
      cgAddNewByteToCurInstruction( (unsigned char) 0x0F,
TRUE );
      cgAddNewByteToCurInstruction( (unsigned char) JZ_REL16,
TRUE );

      // Block after if
      switch (pTempOE->pArgument.uSemClass)
      {
      case SEMCLASS_BLOCK:
            pTempCE = (TContrElemPtr)(pTempOE-
>pArgument.pvValue);
            pTempCE->pContrElemHandler(pTempCE->pBegArgList);

            cgAddNewInstructionToFunc( JMP_REL16 );
            pJMPInstruction = pCurInstruction;
            cgAddNewByteToCurInstruction( (unsigned char)
JMP_REL16, TRUE );


            pTempBlock = (TBlockPtr)(pTempCE->pBegArgList-
>pNextOperationElem->pArgument.pvValue);
            usWord = pTempBlock->uBlockSize + 3;

            ucByte = (unsigned char)usWord;
            cgAddNewByteToInstruction(pJZInstruction, ucByte,
TRUE );
            ucByte = (unsigned char)(usWord>>8);

            cgAddNewByteToInstruction(pJZInstruction, ucByte,
TRUE );

            pTempOE = pTempOE->pNextOperationElem;
            break;
      }
      usWord = 0;
      if ((pTempOE) && (pTempOE->pArgument.uSemClass ==
SEMCLASS_CONSTRUCTION_ELSE))
      {
            pTempCE = (TContrElemPtr)(pTempOE-
>pArgument.pvValue);
            pTempCE->pContrElemHandler(pTempCE->pBegArgList);

            pTempBlock = (TBlockPtr)(pTempCE->pBegArgList-
>pNextOperationElem->pArgument.pvValue);
            usWord = pTempBlock->uBlockSize;
            pTempOE = pTempOE->pNextOperationElem;
      }
      ucByte = (unsigned char)usWord;
      cgAddNewByteToInstruction(pJMPInstruction, ucByte, TRUE
);
      ucByte = (unsigned char)(usWord>>8);
      cgAddNewByteToInstruction(pJMPInstruction, ucByte, TRUE
);

      return TRUE;
}

/*
*
*
```

```
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgCodeCreating()
{
      unsigned short    usWord = 0;
      unsigned short    usTypeSum = 0;
      unsigned char     ucCode = 0;
      extern TParseListNodePtr      pBegParseListNode;
      extern TParseListNodePtr      pCurParseListNode;
      TOperationElemPtr pFuncLocVar = NULL;

      TContrElemPtr           pContrElem = NULL;
      TOperationElemPtr pContrElemArg = NULL;
      extern UINT             uProgrammOffset;
      TIdTablePtr pTempIdTableElem = NULL;

      pCurParseListNode = pBegParseListNode;

      cgAddNewFunctionToList(LEXCLASS_TYPE_VOID, NULL);

      /* INITIALIZATION BLOCK */
      /* CS = 0000h
       * SS = 0000h
       * SP = FFFFh
      */
      //far jump -> CS = 0000; IP = BEGADDRESS + 5 bytes (5
bytes = far jmp instruction)
      cgAddNewInstructionToFunc( JMP_PTR_16_16 );
      cgAddNewByteToCurInstruction( (unsigned
char)JMP_PTR_16_16, TRUE);
      usWord = uProgrammOffset + 5;

      ucCode = (unsigned char)usWord;
      cgAddNewByteToCurInstruction( ucCode, TRUE );
      ucCode = usWord >> 8;
      cgAddNewByteToCurInstruction( ucCode, TRUE );
      cgAddNewByteToCurInstruction( 0, TRUE );
      cgAddNewByteToCurInstruction( 0, TRUE );

      // MOV AX,0
      cgAddNewInstructionToFunc( MOV_R16_IMM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_IMM16 + RW_AX, TRUE );
      cgAddNewByteToCurInstruction( 0, TRUE );
      cgAddNewByteToCurInstruction( 0, TRUE );

      // MOV SS,AX
      cgAddNewInstructionToFunc( MOV_SREG_RM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)MOV_SREG_RM16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_EAX_AX_AL_MM0_XMM0 | SREG_DS_3), TRUE);

      // MOV SS,AX
      cgAddNewInstructionToFunc( MOV_SREG_RM16 );
      cgAddNewByteToCurInstruction( (unsigned
char)MOV_SREG_RM16, TRUE );
      cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_EAX_AX_AL_MM0_XMM0 | SREG_SS_2), TRUE);

      // MOV SP,0xFFFF
      cgAddNewInstructionToFunc( MOV_R16_IMM16 );
      cgAddNewByteToCurInstruction( (unsigned 
char)MOV_R16_IMM16 + RW_SP, TRUE );
      cgAddNewByteToCurInstruction( 0xFF, TRUE );
      cgAddNewByteToCurInstruction( 0xFF, TRUE );

      // MOV SI, absolute main function address
      cgAddNewInstructionToFunc( MOV_R16_RM16 );
      pCurInstruction->Instruction.uCodeSize = 3;
      mainCallLink.pInstruction = pCurInstruction;
      mainCallLink.pFuncId = NULL;

      // CALL SI
      cgAddNewInstructionToFunc( CALL_NEAR_RM16 );
      cgAddNewByteToCurInstruction( (unsigned 
char)CALL_NEAR_RM16, TRUE);
      cgAddNewByteToCurInstruction( (unsigned 
char)(EA16REG_ESI_SI_DH_MM6_XMM6 | 
RO16REG_DL_DX_EDX_MM2_XMM2_2_010), TRUE);

      /* INITIALIZATION BLOCK END */

      while (pCurParseListNode)
      {
            usTypeSum = 0;

            if (pCurParseListNode->FuncNode.uType != 
LEXCLASS_TYPE_STRUCT)
            {
                  pCurBlock = pCurParseListNode-
>FuncNode.pFuncBlock;
                  pFuncLocVar = pCurBlock->pBegLocVarList;
                  cgAddNewFunctionToList(pCurParseListNode-
>FuncNode.uType, pCurParseListNode->FuncNode.pFuncId);
                  if (!strcmp(pCurParseListNode-
>FuncNode.cpFuncName,"main"))
                  {
                        mainCallLink.pFuncId = 
pCurParseListNode->FuncNode.pFuncId;
                  }

                  // Saving previous BP
                  // PUSH BP
                  cgAddNewInstructionToFunc( PUSH_R16 + RW_BP 
);
                  cgAddNewByteToCurInstruction( (unsigned 
char)PUSH_R16 + RW_BP, TRUE );

                  if (pFuncLocVar)
                  {

                        while (pFuncLocVar-
>pNextOperationElem)
                        {
                              pFuncLocVar = pFuncLocVar-
>pNextOperationElem;
                        }
                        // Subtracting the size of all the 
local variables from SP
                        // (allocating the memory)
                        cgAddNewInstructionToFunc( 
SUB_RM16_IMM16 );
                        cgAddNewByteToCurInstruction( 
(unsigned char)SUB_RM16_IMM16, TRUE );
```

```
                        cgAddNewByteToCurInstruction(
(unsigned char)(EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_CH_BP_EBP_MM5_XMM5_5_101), TRUE );
                        while (pFuncLocVar)
                        {
                              pTempIdTableElem =
(TIdTablePtr)pFuncLocVar->pArgument.pvValue;

                              if (pTempIdTableElem-
>pBegIdDecl)
                                    usTypeSum += 2; // for 16-
bit address mode
                              else
                                    usTypeSum +=
semRetTypeSizeByType(pFuncLocVar->pArgument.uLexClass);

                              pFuncLocVar = pFuncLocVar-
>pPrevOperationElem;
                        }
                        ucCode = (unsigned char)usTypeSum;
                        cgAddNewByteToCurInstruction( ucCode,
TRUE );
                        ucCode = usTypeSum >> 8;
                        cgAddNewByteToCurInstruction( ucCode,
TRUE );
                  }
                  // BP with SP equating
                  cgAddNewInstructionToFunc( MOV_R16_RM16 );
                  cgAddNewByteToCurInstruction( (unsigned
char)MOV_R16_RM16, TRUE );
                  cgAddNewByteToCurInstruction( (unsigned
char)(EA16REG_ESP_SP_AH_MM4_XMM4 |
RO16REG_CH_BP_EBP_MM5_XMM5_5_101), TRUE );

                  pContrElem = pCurParseListNode-
>FuncNode.pFuncBlock->pBegContrList;

                  while (pContrElem)
                  {
                        pContrElem-
>pContrElemHandler(pContrElem->pBegArgList);
                        pContrElem = pContrElem-
>pNextContrElem;
                  }
            }
            pCurParseListNode = pCurParseListNode-
>pNextFuncNode;
      }
      if (!mainCallLink.pFuncId)
      {
            semError("linking",0,0,0,"there is no 'main'
function");
      }
      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
```

```
int cgAddEABytesBySemClass(UINT uSemClass, TOperationElemPtr
pTempOE, char cTempRObyte)
{
	TBlockPtr pTempBlock = NULL;
	TBlockPtr pTempCurBlock = NULL;
	unsigned char	ucByte = 0;
	unsigned short	usWord = 0;
	TIdTablePtr	pTempId;
	char		cTempEAbyte;
	TIdTablePtr	pTempIdTableElem = NULL;

	switch(uSemClass)
	{
	case SEMCLASS_VAR8:
	case SEMCLASS_VAR16:
		pTempCurBlock = pCurBlock;
		pTempBlock = ((TIdTablePtr)pTempOE-
>pArgument.pvValue)->pBlock;
		while(pTempCurBlock != pTempBlock)
		{
			usWord = usWord + pTempCurBlock-
>uLocVarSize;
			pTempCurBlock = pTempCurBlock-
>pExternalBlock;
		}

		cgAddNewByteToCurInstruction( (unsigned
char)(EA16MEMLOC_BP_DISP16 | cTempRObyte), TRUE );
		pTempId = (TIdTablePtr)pTempOE-
>pArgument.pvValue;
		usWord = usWord + pTempId->uStackOffset;
		ucByte = (unsigned char)usWord;
		cgAddNewByteToCurInstruction( ucByte, TRUE );
		ucByte = (unsigned char)(usWord>>8);

		cgAddNewByteToCurInstruction( ucByte, TRUE );
		break;
	case SEMCLASS_REG16:
		cTempEAbyte = *((unsigned char*)pTempOE-
>pArgument.pvValue);
		cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );
		break;
	case SEMCLASS_IMM8:
		ucByte = *((unsigned char*)pTempOE-
>pArgument.pvValue);
		cgAddNewByteToCurInstruction( ucByte, TRUE );
		break;
	case SEMCLASS_IMM16:
		usWord = *((unsigned short*)pTempOE-
>pArgument.pvValue);
		ucByte = (unsigned char)usWord;
		cgAddNewByteToCurInstruction( ucByte, TRUE );
		ucByte = (unsigned char)(usWord>>8);

		cgAddNewByteToCurInstruction( ucByte, TRUE );
		break;
	case SEMCLASS_MEM8:
		cTempEAbyte = EA16MEMLOC_DISP16;
		cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );
		usWord = *((unsigned short*)pTempOE-
>pArgument.pvValue);
		ucByte = (unsigned char)usWord;
```

```
            cgAddNewByteToCurInstruction( ucByte, TRUE );
            ucByte = (unsigned char)(usWord>>8);

            cgAddNewByteToCurInstruction( ucByte, TRUE );
            break;
      }

      return TRUE;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgMemoryLocationIndexCase(char cTempRObyte, unsigned char
      cTempEAbyte)
{
      unsigned char   ucByte;
      unsigned short    usWord;

      if (pCurCGOperElem)
      switch (pCurCGOperElem->pArgument.uSemClass)
      {
      case EA16MEMLOC_SI:
            pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
            if (pCurCGOperElem && pCurCGOperElem-
>pArgument.uSemClass == EA16MEMLOC_DISP16)
            {
                  if (!cTempEAbyte)
                        cTempEAbyte =
(char)EA16MEMLOC_SI_DISP16;
                  else
                        switch (cTempEAbyte)
                        {
                        case (char)EA16MEMLOC_BX:
                              cTempEAbyte =
(char)EA16MEMLOC_BX_SI_DISP16;
                              break;
                        case (char)EA16MEMLOCPART_BP:
                              cTempEAbyte =
(char)EA16MEMLOC_BP_SI_DISP16;
                              break;
                        }


                  cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );
                  usWord = *((unsigned short*)pCurCGOperElem-
>pArgument.pvValue);
                  ucByte = (unsigned char)usWord;
                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
                  ucByte = (unsigned char)(usWord>>8);

                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
            }
            else
            {
                  if (!cTempEAbyte)
                        cTempEAbyte = EA16MEMLOC_SI;
                  else
                        switch (cTempEAbyte)
                  {
                        case (char)EA16MEMLOC_BX:
                              cTempEAbyte = EA16MEMLOC_BX_SI;
                              break;
                        case (char)EA16MEMLOCPART_BP:
                              cTempEAbyte = EA16MEMLOC_BP_SI;
                              break;
                        }

                  cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );
            }
            return cTempEAbyte;
            break;
      case EA16MEMLOC_DI:
            if ((pCurCGOperElem->pNextOperationElem) &&
(pCurCGOperElem->pNextOperationElem->pArgument.uSemClass ==
EA16MEMLOC_DISP16))
            {
                  if (!cTempEAbyte)
                        cTempEAbyte =
(char)EA16MEMLOC_DI_DISP16;
                  else
                        switch (cTempEAbyte)
                        {
                        case (char)EA16MEMLOC_BX:
                              cTempEAbyte =
(char)EA16MEMLOC_BX_DI_DISP16;
                              break;
                        case (char)EA16MEMLOCPART_BP:
                              cTempEAbyte =
(char)EA16MEMLOC_BP_DI_DISP16;
                              break;
                        }
                  cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );

                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;

                  usWord = *((unsigned short*)pCurCGOperElem-
>pArgument.pvValue);
                  ucByte = (unsigned char)usWord;
                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
                  ucByte = (unsigned char)(usWord>>8);

                  cgAddNewByteToCurInstruction( ucByte, TRUE
);
            }
            else
            {
                  if (!cTempEAbyte)
                        cTempEAbyte = (char)EA16MEMLOC_DI;
                  else
                        switch (cTempEAbyte)
                        {
                        case (char)EA16MEMLOC_BX:
                              cTempEAbyte = EA16MEMLOC_BX_DI;
                              break;
                        case (char)EA16MEMLOCPART_BP:
                              cTempEAbyte = EA16MEMLOC_BP_DI;
                              break;
                        }
                  cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );
            }
            return cTempEAbyte;
            break;
      case EA16MEMLOC_DISP16:
            if (!cTempEAbyte)
                  cTempEAbyte = (char)EA16MEMLOC_DISP16;
            else
                  switch (cTempEAbyte)
                  {
                  case (char)EA16MEMLOC_BX:
                        cTempEAbyte =
(char)EA16MEMLOC_BX_DISP16;
                        break;
                  case (char)EA16MEMLOCPART_BP:
                        cTempEAbyte =
(char)EA16MEMLOC_BP_DISP16;
                        break;
                  }
            cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );

            usWord = *((unsigned short*)pCurCGOperElem-
>pArgument.pvValue);
            ucByte = (unsigned char)usWord;
            cgAddNewByteToCurInstruction( ucByte, TRUE );
            ucByte = (unsigned char)(usWord>>8);

            cgAddNewByteToCurInstruction( ucByte, TRUE );
            pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
            return cTempEAbyte;
            break;
      }

      cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );
      return cTempEAbyte;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgMemoryLocationCase(char cTempRObyte)
{
      char  cTempEAbyte;
      switch (pCurCGOperElem->pArgument.uSemClass)
      {
      case EA16MEMLOC_BX:
            pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
            cTempEAbyte =
cgMemoryLocationIndexCase(cTempRObyte, EA16MEMLOC_BX);
            break;
      case EA16MEMLOCPART_BP:
            pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
            cTempEAbyte =
cgMemoryLocationIndexCase(cTempRObyte, EA16MEMLOCPART_BP);
            break;
      case EA16MEMLOC_SI:
      case EA16MEMLOC_DI:
      case EA16MEMLOC_DISP16:
            cTempEAbyte =
cgMemoryLocationIndexCase(cTempRObyte, 0);
            break;
      }

      return cTempEAbyte;
}

/*
*
*
*     @param
*     @param
*     @param
*   @author Nail Sharipov
*/
int cgContrElemInstruction(TOperationElemPtr pBegArgList)
{

      TByteListPtr      pTempByte;
      char  cTempRObyte;
      char  cTempEAbyte;

      cgAddNewInstructionToFunc( MOV_RM16_R16 );
      pCurCGOperElem = pBegArgList;
      while (pCurCGOperElem)
      {

            switch(pCurCGOperElem->pArgument.uSemClass)
            {
            case SEMCLASS_PREFIX:

                  cgAddNewByteToCurInstruction( *((unsigned
char*)pCurCGOperElem->pArgument.pvValue), TRUE );
                  break;

            case SEMCLASS_INSTRUCTION_PTR16_16:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_JMP:
                        cgAddNewByteToCurInstruction(
(unsigned char)JMP_PTR_16_16, TRUE );
                        break;
                  }

                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem->pNextOperationElem;

                  switch(pCurCGOperElem->pArgument.uSemClass)
                  {
                  case SEMCLASS_IMM8:
                        cgAddEABytesBySemClass(SEMCLASS_IMM8,
pCurCGOperElem, 0);
```

```
                        cgAddNewByteToCurInstruction( 0, TRUE
);
                        break;
                  case SEMCLASS_IMM16:

      cgAddEABytesBySemClass(SEMCLASS_IMM16, pCurCGOperElem,
0);
                        break;
                  }

                  pCurCGOperElem = pCurCGOperElem-
>pPrevOperationElem;

                  switch(pCurCGOperElem->pArgument.uSemClass)
                  {
                  case SEMCLASS_IMM8:
                        cgAddEABytesBySemClass(SEMCLASS_IMM8,
pCurCGOperElem, 0);
                        cgAddNewByteToCurInstruction( 0, TRUE
);
                        break;
                  case SEMCLASS_IMM16:

      cgAddEABytesBySemClass(SEMCLASS_IMM16, pCurCGOperElem,
0);
                        break;
                  }

                  break;


            case SEMCLASS_INSTRUCTION_IMM8:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_INT:
                        cgAddNewByteToCurInstruction(
(unsigned char)INT_IMM8, TRUE );
                        break;
                  }

                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;

                  if (pCurCGOperElem->pArgument.uSemClass ==
SEMCLASS_IMM8)
                        cgAddEABytesBySemClass(SEMCLASS_IMM8,
pCurCGOperElem, 0);

                  break;


            /*
             *    MOV Sreg, r/m16
             */
            case SEMCLASS_INSTRUCTION_Sreg_MEM16:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_SREG_RM16, TRUE );
                        break;
                  }

                  pCurCGOperElem = pCurCGOperElem-
```

```
>pNextOperationElem;
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;

                  if (pCurCGOperElem->pArgument.uSemClass ==
SEMCLASS_REG16)

      cgAddEABytesBySemClass(SEMCLASS_REG16, pCurCGOperElem,
cTempRObyte);

                  if (pCurCGOperElem->pArgument.uSemClass ==
SEMCLASS_VAR16)

      cgAddEABytesBySemClass(SEMCLASS_VAR16, pCurCGOperElem,
cTempRObyte);

                  break;
            case SEMCLASS_INSTRUCTION_Sreg_GPReg16:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_SREG_RM16, TRUE );
                        break;
                  }

                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  if (pCurCGOperElem->pArgument.uSemClass ==
SEMCLASS_REG16)

      cgAddEABytesBySemClass(SEMCLASS_REG16, pCurCGOperElem,
cTempRObyte);
                  break;

            /*
             *    MOV r/m8, r8
             */
            case SEMCLASS_INSTRUCTION_MEM8_GPReg8:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_RM8_R8, TRUE );
                        break;
                  }



                  pTempByte = pCurByte;

                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;

                  cTempEAbyte = cgMemoryLocationCase(0);
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pTempByte->pNextByte->cByte = (unsigned
char)(cTempEAbyte | cTempRObyte);
                  break;
            case SEMCLASS_INSTRUCTION_MEM16_GPReg16:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_RM16_R16, TRUE );
                        break;
                  }

                  pTempByte = pCurByte;
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempEAbyte = cgMemoryLocationCase(0);
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pTempByte->pNextByte->cByte = (unsigned
char)(cTempEAbyte | cTempRObyte);
                  break;

            case SEMCLASS_INSTRUCTION_MEM32_GPReg32:
                  break;

            case SEMCLASS_INSTRUCTION_REG8_IMM8:
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cgAddNewByteToCurInstruction( (unsigned
char)(MOV_R8_IMM8 + cTempRObyte), TRUE );

                  if (pCurCGOperElem->pArgument.uSemClass ==
SEMCLASS_IMM8)
                        cgAddEABytesBySemClass(SEMCLASS_IMM8,
pCurCGOperElem, cTempRObyte);

                  break;

            case SEMCLASS_INSTRUCTION_REG16_IMM16:
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cgAddNewByteToCurInstruction( (unsigned
char)(MOV_R16_IMM16 + cTempRObyte), TRUE );

                  if (pCurCGOperElem->pArgument.uSemClass ==
SEMCLASS_IMM16)

      cgAddEABytesBySemClass(SEMCLASS_IMM16, pCurCGOperElem,
cTempRObyte);
                  break;
            case SEMCLASS_INSTRUCTION_REG16_MEM16:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_R16_RM16, TRUE );
                        break;
                  }

                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempEAbyte =
cgMemoryLocationCase(cTempRObyte);

                  if (pCurCGOperElem)
                  switch (pCurCGOperElem-
>pArgument.uSemClass)
                  {
                  case SEMCLASS_VAR16:

      cgAddEABytesBySemClass(SEMCLASS_VAR16, pCurCGOperElem,
cTempRObyte);
                        break;
                  }
                  break;
            case SEMCLASS_INSTRUCTION_REG8_MEM8:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_R8_RM8, TRUE );
                        break;
                  }
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempEAbyte =
cgMemoryLocationCase(cTempRObyte);

                  if (pCurCGOperElem)
                  switch (pCurCGOperElem-
>pArgument.uSemClass)
                  {
                  case SEMCLASS_VAR8:
                        cgAddEABytesBySemClass(SEMCLASS_VAR8,
pCurCGOperElem, cTempRObyte);
                        break;
                  }
                  break;

            case SEMCLASS_INSTRUCTION_MEM8_IMM8:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_RM8_IMM8, TRUE );
                        break;
                  }

                  cTempRObyte =
RO16REG_AL_AX_EAX_MM0_XMM0_0_000;
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempEAbyte =
cgMemoryLocationCase(cTempRObyte);
                  if (pCurCGOperElem->pArgument.uSemClass ==
SEMCLASS_IMM8)
                        cgAddEABytesBySemClass(SEMCLASS_IMM8,
pCurCGOperElem, cTempRObyte);
                  break;

            case SEMCLASS_INSTRUCTION_MEM16_IMM16:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_RM16_IMM16, TRUE );
                        break;
                  }

                  cTempRObyte =
RO16REG_AL_AX_EAX_MM0_XMM0_0_000;
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;

                  cTempEAbyte = cgMemoryLocationCase(0);

                  if (pCurCGOperElem->pArgument.uSemClass ==
SEMCLASS_IMM16)

      cgAddEABytesBySemClass(SEMCLASS_IMM16, pCurCGOperElem,
cTempRObyte);
                  break;

            case SEMCLASS_INSTRUCTION_REG8_REG8:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_R8_RM8, TRUE );
                        break;
                  }

                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempEAbyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );
                  break;
            case SEMCLASS_INSTRUCTION_REG16_REG16:
                  switch(pCurCGOperElem->pArgument.uLexClass)
                  {
                  case LEXCLASS_IA32INSTRUCTIONS_MOV:
                        cgAddNewByteToCurInstruction(
(unsigned char)MOV_R16_RM16, TRUE );
                        break;
                  }

                  pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
                  cTempRObyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  pCurCGOperElem = pCurCGOperElem-
```

```
>pNextOperationElem;
                  cTempEAbyte = *((unsigned
char*)pCurCGOperElem->pArgument.pvValue);
                  cgAddNewByteToCurInstruction( (unsigned
char)(cTempEAbyte | cTempRObyte), TRUE );
                  break;
            case SEMCLASS_INSTRUCTION_REG32_REG32:
                  break;
            }
            if (pCurCGOperElem)
            pCurCGOperElem = pCurCGOperElem-
>pNextOperationElem;
      }
      return TRUE;
}
```

Figure 10. analysis_cg.c