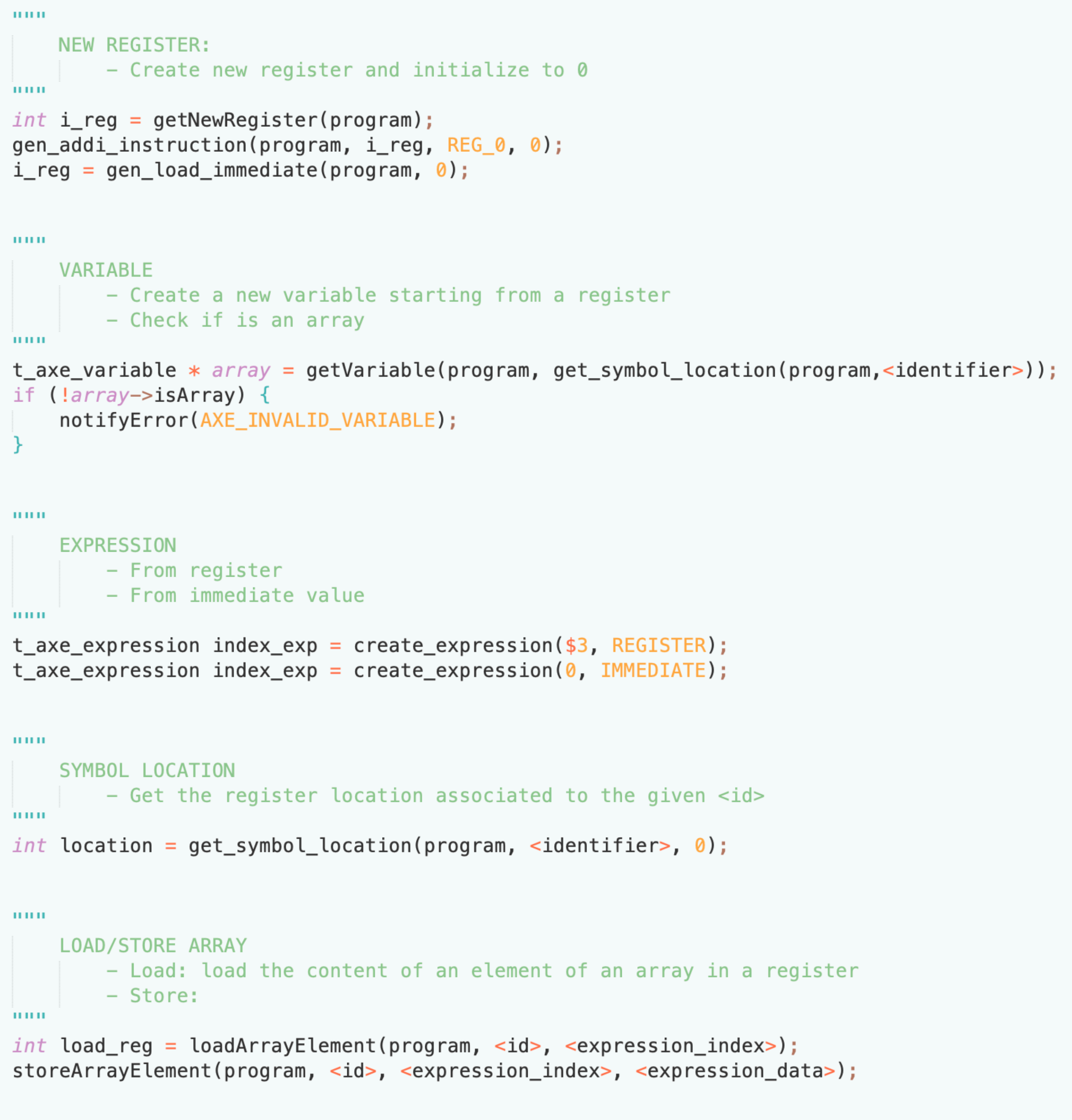
Politecnico di Milano

Formal Languages and Compilers

ACSE Cheatsheet

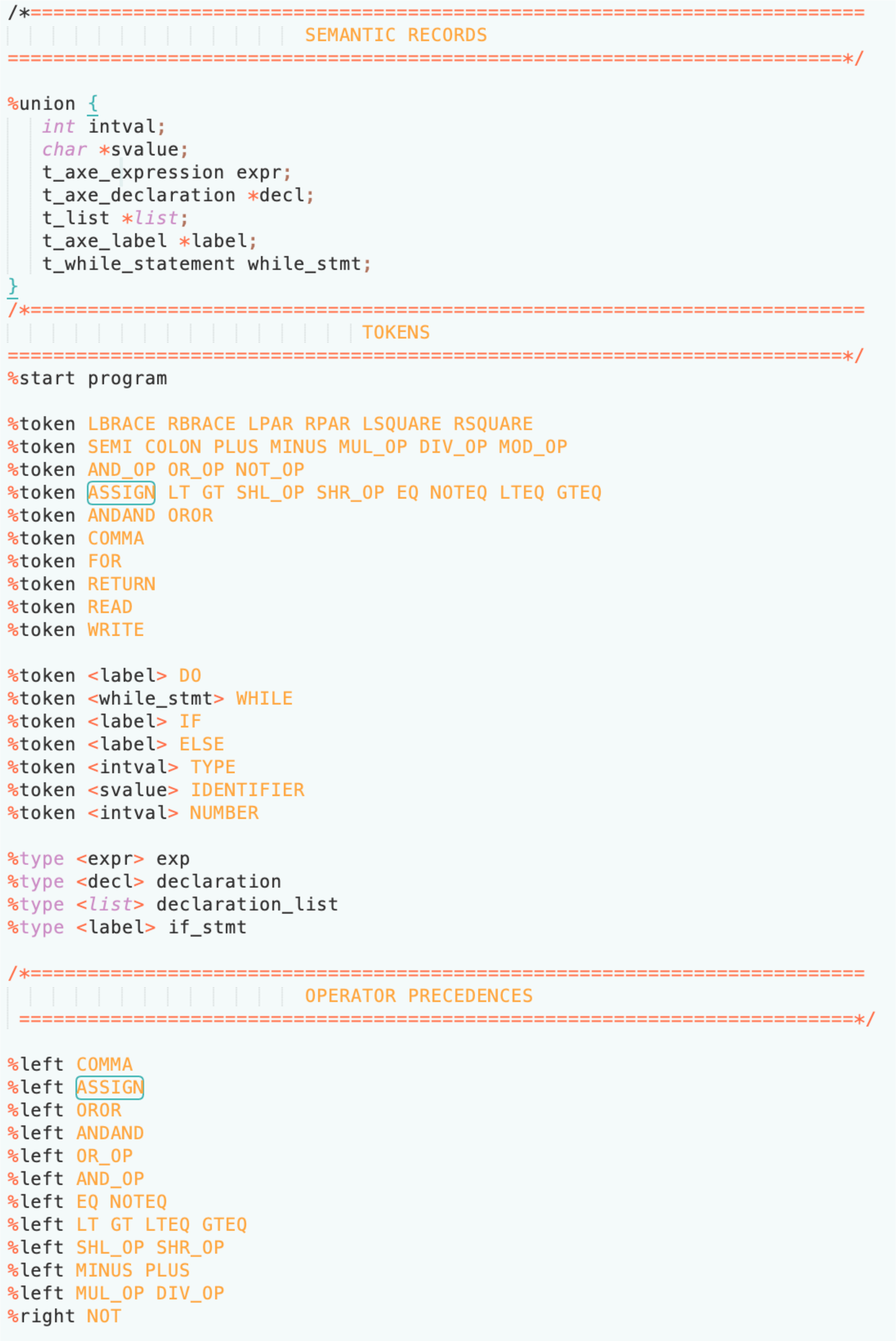
Author
Tommaso Scarlatti



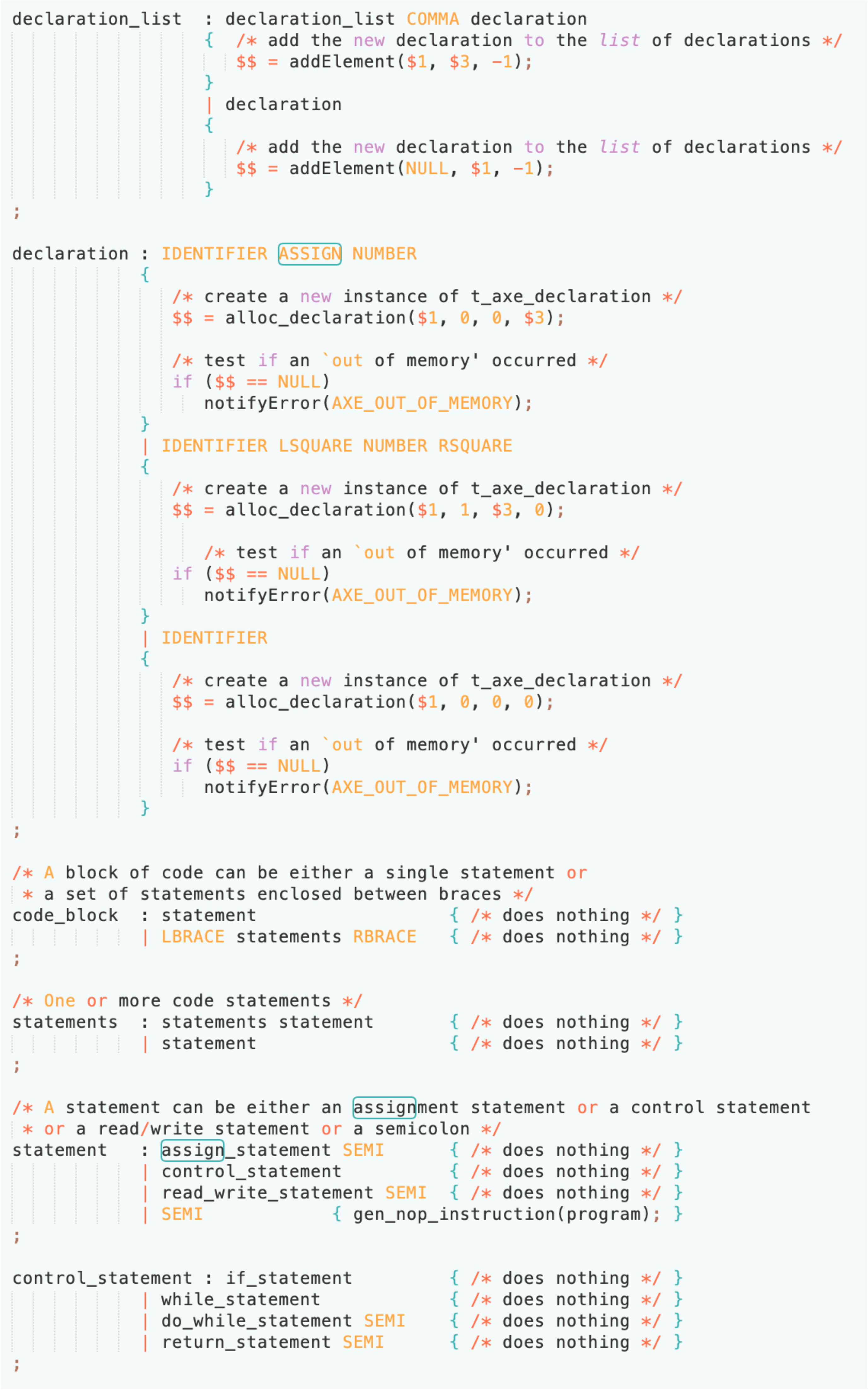
```
"""
    NEW REGISTER:
        - Create new register and initialize to 0
"""
int i_reg = getNewRegister(program);
gen_addi_instruction(program, i_reg, REG_0, 0);
i_reg = gen_load_immediate(program, 0);


"""
    VARIABLE
        - Create a new variable starting from a register
        - Check if is an array
"""
t_axe_variable * array = getVariable(program, get_symbol_location(program,<identifier>));
if (!array->isArray) {
    notifyError(AXE_INVALID_VARIABLE);
}


"""
    EXPRESSION
        - From register
        - From immediate value
"""
t_axe_expression index_exp = create_expression($3, REGISTER);
t_axe_expression index_exp = create_expression(0, IMMEDIATE);


"""
    SYMBOL LOCATION
        - Get the register location associated to the given <id>
"""
int location = get_symbol_location(program, <identifier>, 0);


"""
    LOAD/STORE ARRAY
        - Load: load the content of an element of an array in a register
        - Store:
"""
int load_reg = loadArrayElement(program, <id>, <expression_index>);
storeArrayElement(program, <id>, <expression_index>, <expression_data>);
```

```
"""
    LABELS
        - assignLabel(program, label): set the label where it is called
        - newLabel: create label but not attach
        - assignNewLabel: create + assign
"""
t_axe_label* test = assignNewLabel(program);
t_axe_label* end = newLabel(program);
assignLabel(program, <label_end>);

"""
    LOOP
        - An example of a loop to iterate over an array starting from the end
        - test/end: begin/end of the loop
"""
int index_reg = gen_load_immediate(program, array->arraySize-1);
t_axe_label* test = assignNewLabel(program);
t_axe_label* end = newLabel(program);

handle_binary_comparison(program, create_expression(index_reg,REGISTER),
                                create_expression(0, IMMEDIATE), _LT_);
gen_bne_instruction(program, end, 0);

/// Codeblock ///

gen_subi_instruction(program, index_reg, index_reg, 1);
gen_bt_instruction(program, test, 0);
assignLabel(program, end);


"""
    ARITHMETIC OPERATIONS:
        - Operands are expressions
        - Result expression is returned
        - ADD/MUL/DIV/SUB
"""
t_axe_expression expr = handle_binary_comparison(program, <expr>, <expr>, _NOTEQ_)

"""
    CONDITIONAL OPERATIONS:
        - Operands are expressions
        - Result expression is returned
        - _LT_/_EQ_/_GT_
"""
t_axe_expression expr = handle_bin_numeric_op(program, <expr>, <expr>, MUL)
```

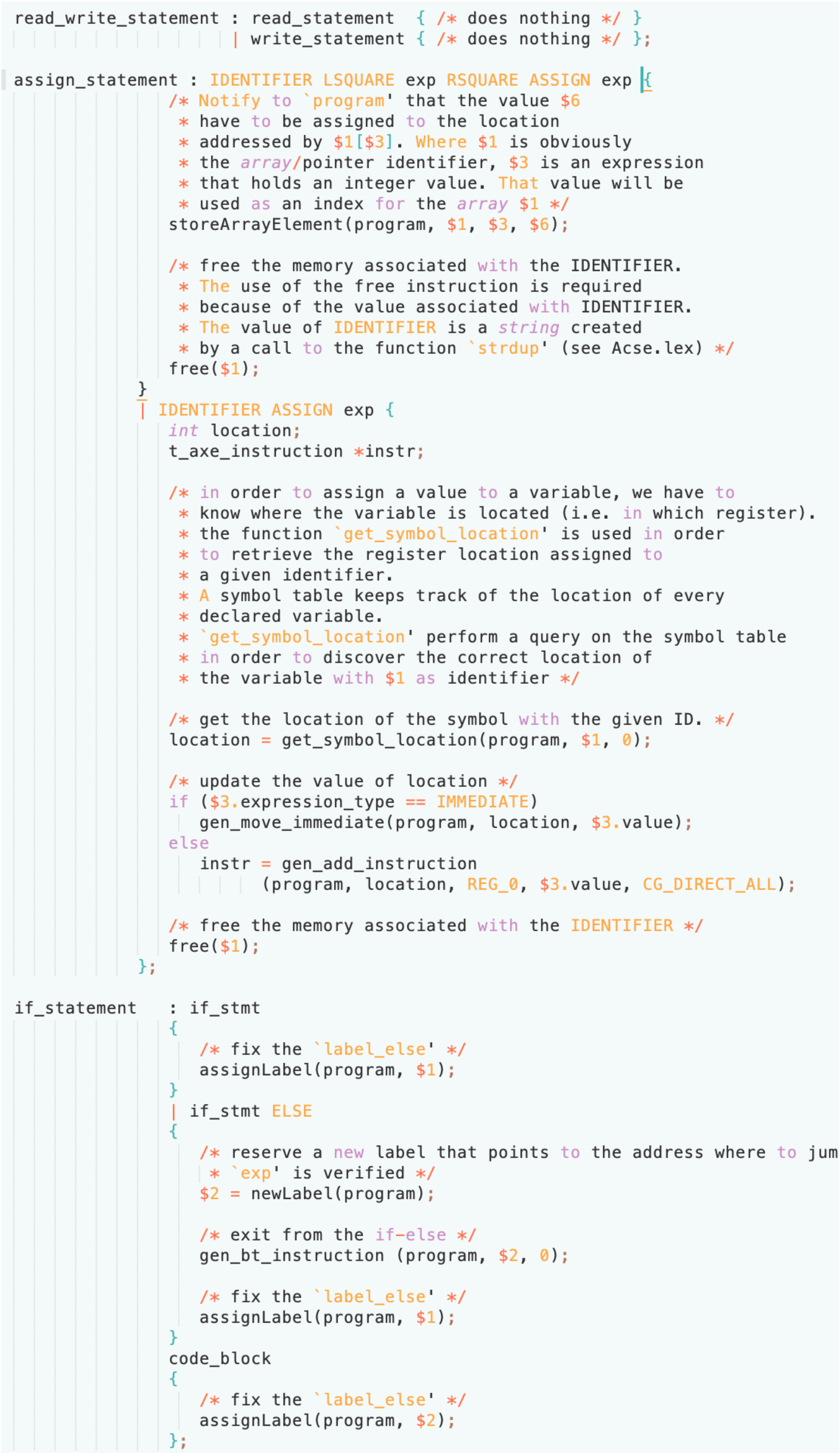
```
"""
    ARITHMETIC ISTRUCTIONS
        - Operands are registers: target, source_1, source_2
"""
gen_andb_instruction(program, i_reg, i_reg, i_reg, CG_DIRECT_ALL);

"""
    CONDITIONAL ISTRUCTIONS
        - The check is performed on the result of the previous expression
"""
gen_beq_instruction(program, <label>, 0);
gen_bne_instruction(program, <label>, 0);
gen_bgt_instruction(program,<label>,0);
gen_bt_instruction(program, <label>, 0);
gen_bmi_instruction(program, <label>,0);
gen_bpl_instruction(program, <label>,0);

"""
    PROGRAM TERMINATION INSTRUCTION
"""
gen_halt_instruction(program);

"""
    GLOBAL ARRAY STRUCT
        - Define a global custom struct for an array with augmented capabilities
"""
struct t_exists{
    char* id;
    int index_reg;
    int array_size;
} exists = {NULL, 0, 0};

"""
    ACSE LIST IMPLEMENTATION
        - Add an element at the beginning of the list, if the list
          is a NULL pointer the list is initialized (i.e. allocated)
          and a valid pointer is returned.
        - Get the element at index from the list, explicit cast is required.
        - Pops the first element from the list, if the list is empty
          after the operation a NULL pointer is returned
"""
t_list *list = NULL;
list = addFirst(list, element);
element_type * l = (element_type *)LDATA(getElementAt(list,index));
list = removeFirst(list);

"""
    NEW VARIABLE DECLARATION
"""
t_axe_declaration * alloc_declaration
      (char *ID, int isArray, int arraySize, int init_val)
```



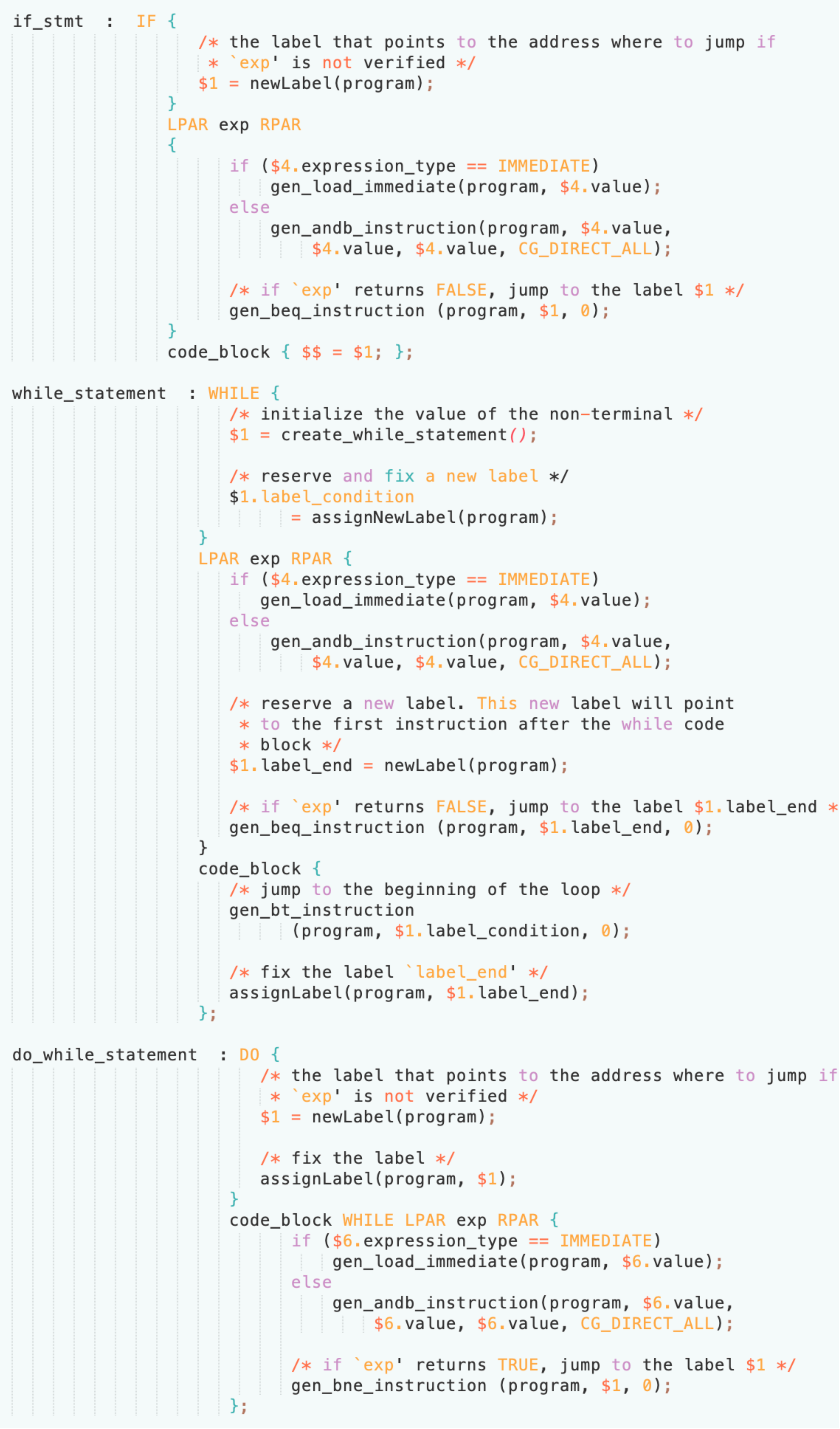
```
/*==========================================================================
                          SEMANTIC RECORDS
==========================================================================*/

%union {
    int intval;
    char *svalue;
    t_axe_expression expr;
    t_axe_declaration *decl;
    t_list *list;
    t_axe_label *label;
    t_while_statement while_stmt;
}
/*==========================================================================
                               TOKENS
==========================================================================*/
%start program

%token LBRACE RBRACE LPAR RPAR LSQUARE RSQUARE
%token SEMI COLON PLUS MINUS MUL_OP DIV_OP MOD_OP
%token AND_OP OR_OP NOT_OP
%token ASSIGN LT GT SHL_OP SHR_OP EQ NOTEQ LTEQ GTEQ
%token ANDAND OROR
%token COMMA
%token FOR
%token RETURN
%token READ
%token WRITE

%token <label> DO
%token <while_stmt> WHILE
%token <label> IF
%token <label> ELSE
%token <intval> TYPE
%token <svalue> IDENTIFIER
%token <intval> NUMBER

%type <expr> exp
%type <decl> declaration
%type <list> declaration_list
%type <label> if_stmt

/*==========================================================================
                          OPERATOR PRECEDENCES
 ==========================================================================*/

%left COMMA
%left ASSIGN
%left OROR
%left ANDAND
%left OR_OP
%left AND_OP
%left EQ NOTEQ
%left LT GT LTEQ GTEQ
%left SHL_OP SHR_OP
%left MINUS PLUS
%left MUL_OP DIV_OP
%right NOT
```



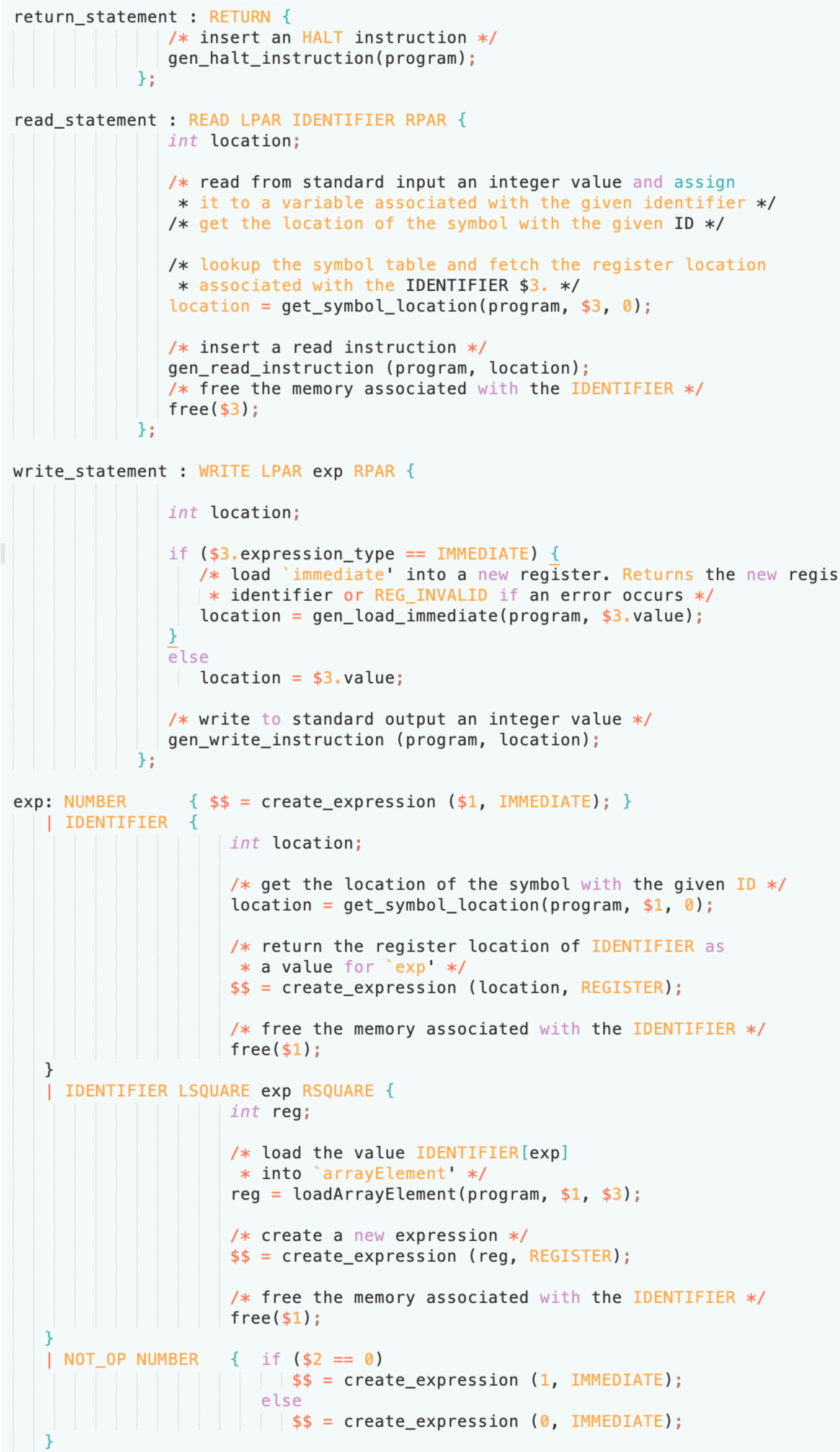
```
declaration_list  : declaration_list COMMA declaration
                  {  /* add the new declaration to the list of declarations */
                     $$ = addElement($1, $3, -1);
                  }
                  | declaration
                  {
                     /* add the new declaration to the list of declarations */
                     $$ = addElement(NULL, $1, -1);
                  }
;

declaration : IDENTIFIER ASSIGN NUMBER
            {
               /* create a new instance of t_axe_declaration */
               $$ = alloc_declaration($1, 0, 0, $3);

               /* test if an `out of memory' occurred */
               if ($$ == NULL)
                  notifyError(AXE_OUT_OF_MEMORY);
            }
            | IDENTIFIER LSQUARE NUMBER RSQUARE
            {
               /* create a new instance of t_axe_declaration */
               $$ = alloc_declaration($1, 1, $3, 0);

                  /* test if an `out of memory' occurred */
               if ($$ == NULL)
                  notifyError(AXE_OUT_OF_MEMORY);
            }
            | IDENTIFIER
            {
               /* create a new instance of t_axe_declaration */
               $$ = alloc_declaration($1, 0, 0, 0);

               /* test if an `out of memory' occurred */
               if ($$ == NULL)
                  notifyError(AXE_OUT_OF_MEMORY);
            }
;

/* A block of code can be either a single statement or
 * a set of statements enclosed between braces */
code_block  : statement                  { /* does nothing */ }
            | LBRACE statements RBRACE   { /* does nothing */ }
;

/* One or more code statements */
statements  : statements statement       { /* does nothing */ }
            | statement                  { /* does nothing */ }
;

/* A statement can be either an assignment statement or a control statement
 * or a read/write statement or a semicolon */
statement   : assign_statement SEMI      { /* does nothing */ }
            | control_statement          { /* does nothing */ }
            | read_write_statement SEMI  { /* does nothing */ }
            | SEMI            { gen_nop_instruction(program); }
;

control_statement : if_statement         { /* does nothing */ }
            | while_statement            { /* does nothing */ }
            | do_while_statement SEMI    { /* does nothing */ }
            | return_statement SEMI      { /* does nothing */ }
;
```



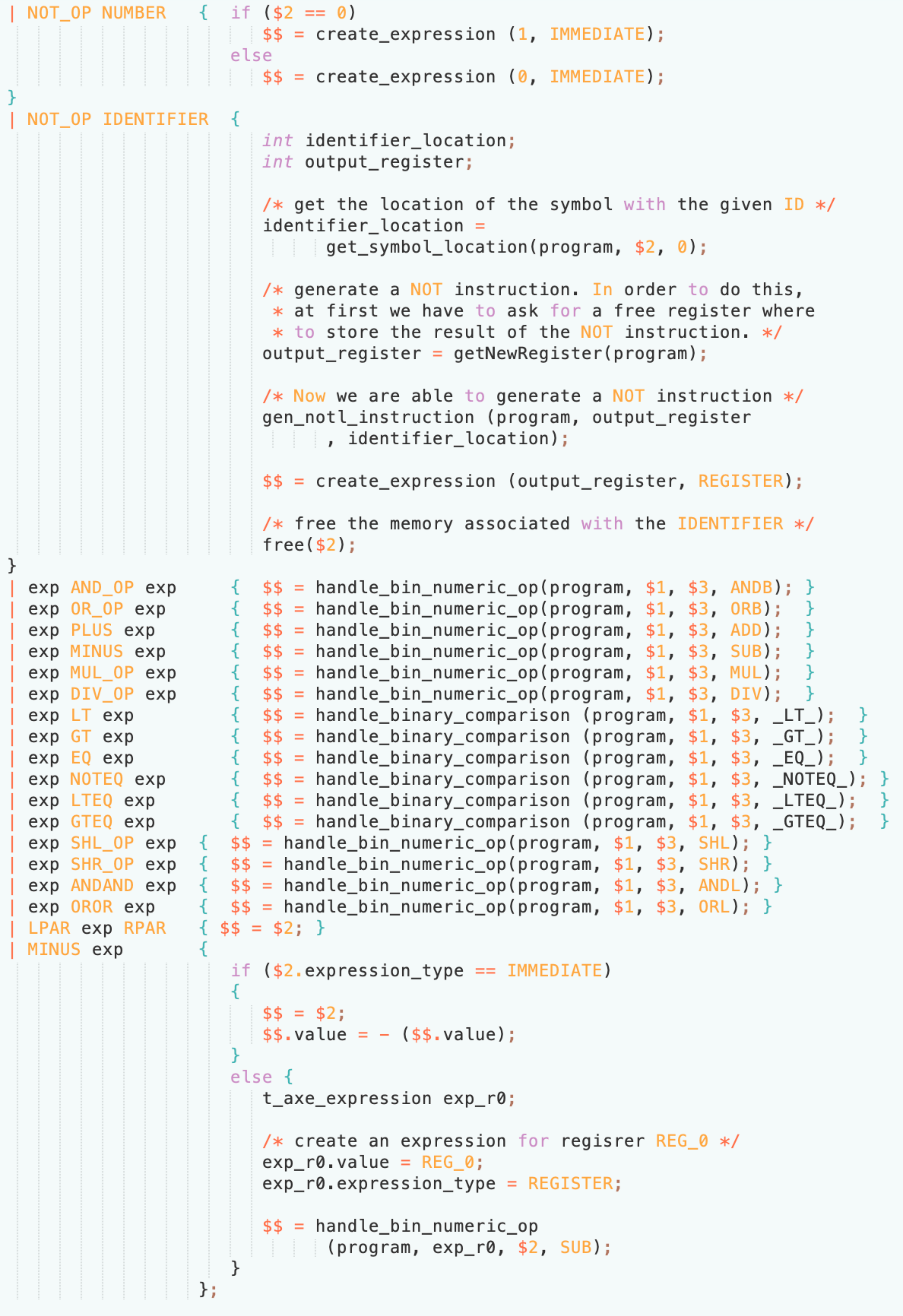
```
read_write_statement : read_statement  { /* does nothing */ }
                     | write_statement { /* does nothing */ };

assign_statement : IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp {
               /* Notify to `program' that the value $6
                * have to be assigned to the location
                * addressed by $1[$3]. Where $1 is obviously
                * the array/pointer identifier, $3 is an expression
                * that holds an integer value. That value will be
                * used as an index for the array $1 */
               storeArrayElement(program, $1, $3, $6);

               /* free the memory associated with the IDENTIFIER.
                * The use of the free instruction is required
                * because of the value associated with IDENTIFIER.
                * The value of IDENTIFIER is a string created
                * by a call to the function `strdup' (see Acse.lex) */
               free($1);
            }
            | IDENTIFIER ASSIGN exp {
               int location;
               t_axe_instruction *instr;

               /* in order to assign a value to a variable, we have to
                * know where the variable is located (i.e. in which register).
                * the function `get_symbol_location' is used in order
                * to retrieve the register location assigned to
                * a given identifier.
                * A symbol table keeps track of the location of every
                * declared variable.
                * `get_symbol_location' perform a query on the symbol table
                * in order to discover the correct location of
                * the variable with $1 as identifier */

               /* get the location of the symbol with the given ID. */
               location = get_symbol_location(program, $1, 0);

               /* update the value of location */
               if ($3.expression_type == IMMEDIATE)
                  gen_move_immediate(program, location, $3.value);
               else
                  instr = gen_add_instruction
                        (program, location, REG_0, $3.value, CG_DIRECT_ALL);

               /* free the memory associated with the IDENTIFIER */
               free($1);
            };

if_statement   : if_stmt
               {
                  /* fix the `label_else' */
                  assignLabel(program, $1);
               }
               | if_stmt ELSE
               {
                  /* reserve a new label that points to the address where to jum
                   * `exp' is verified */
                  $2 = newLabel(program);

                  /* exit from the if-else */
                  gen_bt_instruction (program, $2, 0);

                  /* fix the `label_else' */
                  assignLabel(program, $1);
               }
               code_block
               {
                  /* fix the `label_else' */
                  assignLabel(program, $2);
               };
```



```
if_stmt  :  IF {
                  /* the label that points to the address where to jump if
                   * `exp' is not verified */
                  $1 = newLabel(program);
               }
               LPAR exp RPAR
               {
                     if ($4.expression_type == IMMEDIATE)
                         gen_load_immediate(program, $4.value);
                     else
                         gen_andb_instruction(program, $4.value,
                             $4.value, $4.value, CG_DIRECT_ALL);

                     /* if `exp' returns FALSE, jump to the label $1 */
                     gen_beq_instruction (program, $1, 0);
               }
               code_block { $$ = $1; };

while_statement  : WHILE {
                     /* initialize the value of the non-terminal */
                     $1 = create_while_statement();

                     /* reserve and fix a new label */
                     $1.label_condition
                           = assignNewLabel(program);
                  }
                  LPAR exp RPAR {
                     if ($4.expression_type == IMMEDIATE)
                        gen_load_immediate(program, $4.value);
                     else
                         gen_andb_instruction(program, $4.value,
                             $4.value, $4.value, CG_DIRECT_ALL);

                     /* reserve a new label. This new label will point
                      * to the first instruction after the while code
                      * block */
                     $1.label_end = newLabel(program);

                     /* if `exp' returns FALSE, jump to the label $1.label_end *
                     gen_beq_instruction (program, $1.label_end, 0);
                  }
                  code_block {
                     /* jump to the beginning of the loop */
                     gen_bt_instruction
                           (program, $1.label_condition, 0);

                     /* fix the label `label_end' */
                     assignLabel(program, $1.label_end);
                  };

do_while_statement  : DO {
                        /* the label that points to the address where to jump if
                         * `exp' is not verified */
                        $1 = newLabel(program);

                        /* fix the label */
                        assignLabel(program, $1);
                     }
                     code_block WHILE LPAR exp RPAR {
                           if ($6.expression_type == IMMEDIATE)
                               gen_load_immediate(program, $6.value);
                           else
                               gen_andb_instruction(program, $6.value,
                                   $6.value, $6.value, CG_DIRECT_ALL);

                           /* if `exp' returns TRUE, jump to the label $1 */
                           gen_bne_instruction (program, $1, 0);
                     };
```



```
return_statement : RETURN {
               /* insert an HALT instruction */
               gen_halt_instruction(program);
            };

read_statement : READ LPAR IDENTIFIER RPAR {
               int location;

               /* read from standard input an integer value and assign
                * it to a variable associated with the given identifier */
               /* get the location of the symbol with the given ID */

               /* lookup the symbol table and fetch the register location
                * associated with the IDENTIFIER $3. */
               location = get_symbol_location(program, $3, 0);

               /* insert a read instruction */
               gen_read_instruction (program, location);
               /* free the memory associated with the IDENTIFIER */
               free($3);
            };

write_statement : WRITE LPAR exp RPAR {

               int location;

               if ($3.expression_type == IMMEDIATE) {
                  /* load `immediate' into a new register. Returns the new regis
                   * identifier or REG_INVALID if an error occurs */
                  location = gen_load_immediate(program, $3.value);
               }
               else
                  location = $3.value;

               /* write to standard output an integer value */
               gen_write_instruction (program, location);
            };

exp: NUMBER      { $$ = create_expression ($1, IMMEDIATE); }
   | IDENTIFIER  {
                     int location;

                     /* get the location of the symbol with the given ID */
                     location = get_symbol_location(program, $1, 0);

                     /* return the register location of IDENTIFIER as
                      * a value for `exp' */
                     $$ = create_expression (location, REGISTER);

                     /* free the memory associated with the IDENTIFIER */
                     free($1);
   }
   | IDENTIFIER LSQUARE exp RSQUARE {
                     int reg;

                     /* load the value IDENTIFIER[exp]
                      * into `arrayElement' */
                     reg = loadArrayElement(program, $1, $3);

                     /* create a new expression */
                     $$ = create_expression (reg, REGISTER);

                     /* free the memory associated with the IDENTIFIER */
                     free($1);
   }
   | NOT_OP NUMBER   {  if ($2 == 0)
                           $$ = create_expression (1, IMMEDIATE);
                        else
                           $$ = create_expression (0, IMMEDIATE);
   }
```



```
| NOT_OP NUMBER   {  if ($2 == 0)
                        $$ = create_expression (1, IMMEDIATE);
                     else
                        $$ = create_expression (0, IMMEDIATE);
}
| NOT_OP IDENTIFIER  {
                        int identifier_location;
                        int output_register;

                        /* get the location of the symbol with the given ID */
                        identifier_location =
                              get_symbol_location(program, $2, 0);

                        /* generate a NOT instruction. In order to do this,
                         * at first we have to ask for a free register where
                         * to store the result of the NOT instruction. */
                        output_register = getNewRegister(program);

                        /* Now we are able to generate a NOT instruction */
                        gen_notl_instruction (program, output_register
                              , identifier_location);

                        $$ = create_expression (output_register, REGISTER);

                        /* free the memory associated with the IDENTIFIER */
                        free($2);
}
| exp AND_OP exp     {  $$ = handle_bin_numeric_op(program, $1, $3, ANDB); }
| exp OR_OP exp      {  $$ = handle_bin_numeric_op(program, $1, $3, ORB);  }
| exp PLUS exp       {  $$ = handle_bin_numeric_op(program, $1, $3, ADD);  }
| exp MINUS exp      {  $$ = handle_bin_numeric_op(program, $1, $3, SUB);  }
| exp MUL_OP exp     {  $$ = handle_bin_numeric_op(program, $1, $3, MUL);  }
| exp DIV_OP exp     {  $$ = handle_bin_numeric_op(program, $1, $3, DIV);  }
| exp LT exp         {  $$ = handle_binary_comparison (program, $1, $3, _LT_);  }
| exp GT exp         {  $$ = handle_binary_comparison (program, $1, $3, _GT_);  }
| exp EQ exp         {  $$ = handle_binary_comparison (program, $1, $3, _EQ_);  }
| exp NOTEQ exp      {  $$ = handle_binary_comparison (program, $1, $3, _NOTEQ_); }
| exp LTEQ exp       {  $$ = handle_binary_comparison (program, $1, $3, _LTEQ_);  }
| exp GTEQ exp       {  $$ = handle_binary_comparison (program, $1, $3, _GTEQ_);  }
| exp SHL_OP exp  {  $$ = handle_bin_numeric_op(program, $1, $3, SHL); }
| exp SHR_OP exp  {  $$ = handle_bin_numeric_op(program, $1, $3, SHR); }
| exp ANDAND exp  {  $$ = handle_bin_numeric_op(program, $1, $3, ANDL); }
| exp OROR exp    {  $$ = handle_bin_numeric_op(program, $1, $3, ORL); }
| LPAR exp RPAR   { $$ = $2; }
| MINUS exp       {
                     if ($2.expression_type == IMMEDIATE)
                     {
                        $$ = $2;
                        $$.value = - ($$.value);
                     }
                     else {
                        t_axe_expression exp_r0;

                        /* create an expression for regisrer REG_0 */
                        exp_r0.value = REG_0;
                        exp_r0.expression_type = REGISTER;

                        $$ = handle_bin_numeric_op
                              (program, exp_r0, $2, SUB);
                     }
                  };
```