```
/*****************************************************************************

                                Scanner

*****************************************************************************/
%option noyywrap
%{

#include <string.h>
#include "axe_struct.h"
#include "collections.h"
#include "Acse.tab.h"
#include "axe_constants.h"

/* Variables declared in the lexer for error tracking */
extern int line_num;
extern int num_error;

/* extern declaration of function yyerror */
extern void yyerror(const char*);

%}
/*============================================================================
                        TOKEN DEFINITIONS
============================================================================*/
DIGIT    [0-9]
ID       [a-zA-Z_][a-zA-Z0-9_]*


/*============================================================================
      TOKENS
============================================================================*/

%option noyywrap

%x comment

%%

"\r\n"              { ++line_num; }
"\n"                { ++line_num; }

[ \t\f\v]+          { /* Ignore whitespace. */ }

"//"[^\n]*          { /* ignore comment lines */ }
"/*"                BEGIN(comment);

<comment>[^*\n]*
<comment>[^*\n]*\n      { ++line_num; }
<comment>"*"+[^*/\n]*
<comment>"*"+[^*/\n]*\n { ++line_num; }
<comment>"*"+"/"        BEGIN(INITIAL);

"["                 { return LBRACE; }
"]"                 { return RBRACE; }
"["                 { return LSQUARE; }
"]"                 { return RSQUARE; }
"("                 { return LPAR; }
")"                 { return RPAR; }
";"                 { return SEMI; }
"+"                 { return PLUS; }
"-"                 { return MINUS; }
"*"                 { return MUL_OP; }
"/"                 { return DIV_OP; }
"&"                 { return AND_OP; }
"|"                 { return OR_OP; }
"!"                 { return NOT_OP; }
"="                 { return ASSIGN; }
"<"                 { return LT; }
">"                 { return GT; }
"<<"                { return SHL_OP; }
">>"                { return SHR_OP; }
"=="                { return EQ; }
"!="                { return NOTEQ; }
"<="                { return LTEQ; }
">="                { return GTEQ; }
"&&"                { return ANDAND; }
"||"                { return OROR; }
","                 { return COMMA; }

"do"                { return DO; }
"else"              { return ELSE; }
"if"                { return IF; }
"int"               { yylval.intval = INTEGER_TYPE; return TYPE; }
"while"             { return WHILE; }
"return"            { return RETURN; }
"read"              { return READ; }
"write"             { return WRITE; }

{ID}                { yylval.svalue=strdup(yytext); return IDENTIFIER; }
{DIGIT}+            { yylval.intval = atoi( yytext );
                      return(NUMBER); }

.                   { yyerror("Error: unexpected token");
                      num_error++;
                      return (-1); /* invalid token */ }
<INITIAL><<EOF>>    { return EOF_TOK; }
```



```
%{
/*
 * Andrea Di Biagio
 * Politecnico di Milano, 2007
 *
 * Acse.y
 * Formal Languages & Compilers Machine, 2007/2008
 */

/*****************************************************************************

                    Compiler for the language LANCE

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "axe_struct.h"
#include "axe_engine.h"
#include "symbol_table.h"
#include "axe_target_asm_print.h"
#include "axe_target_transform.h"
#include "axe_errors.h"
#include "collections.h"
#include "axe_expressions.h"
#include "axe_gencode.h"
#include "axe_utils.h"
#include "axe_array.h"
#include "axe_cflow_graph.h"
#include "cflow_constants.h"
#include "axe_transform.h"
#include "axe_reg_alloc.h"
#include "reg_alloc_constants.h"
#include "axe_io_manager.h"
#ifndef NDEBUG
#  include "axe_debug.h"
#endif

/* global variables */
int line_num;         /* this variable will keep track of the
                       * source code line number. Every time that a newline
                       * is encountered while parsing the input file, this
                       * value is increased by 1. This value is then used
                       * for error tracking: if the parser returns an error
                       * or a warning, this value is used in order to notify
                       * in which line of code the error has been found */
int num_error;        /* the number of errors found in the code. This value
                       * is increased by 1 every time a new error is found
                       * in the code. */
int num_warning;      /* As for the 'num_error' global variable, this one
                       * keeps track of all the warning messages displayed */

/* errorcode is defined inside "axe_engine.c" */
extern int errorcode;   /* this variable is used to test if an error is found
                         * while parsing the input file. It also is set
                         * to notify if the compiler internal state is invalid.
                         * When the parsing process is started, the value
                         * of 'errorcode' is set to the value of the macro
                         * 'AXE_OK' defined in "axe_constants.h".
                         * As long as everything (the parsed source code and
                         * the internal state of the compiler) is correct,
                         * the value of 'errorcode' is set to 'AXE_OK'.
                         * When an error occurs (because the input file contains
                         * one or more syntax errors or because something went
                         * wrong in the machine internal state), the errorcode
                         * is set to a value that is different from 'AXE_OK'. */
extern const char *errormsg; /* When errorcode is not equal to AXE_OK,
                              * this variable may be set to an error message to print
                              * if desired. */

extern int cflow_errorcode;   /* As for 'errorcode' this value is used to
                               * test if an error occurs during the creation process of
                               * a control flow graph. More informations can be found
                               * analyzing the file 'axe_cflow_graph.h'. */

/* program informations */
t_program_infos *program;  /* The singleton instance of 'program'.
                            * An instance of 't_program_infos' holds in its
                            * internal structure, all the useful informations
                            * about a program. For example: the assembly
                            * (code and directives); the symbol table;
                            * the label manager (see axe_labels.h) etc. */
t_cflow_Graph *graph;      /* An instance of a control flow graph. This instance
                            * will be generated starting from 'program' and will
                            * be used during the register allocation process */

t_reg_allocator *RA;       /* Register allocator. It implements the "Linear
                            * scan" algorithm */

t_io_infos *file_infos;    /* input and output files used by the compiler */


extern int yylex(void);
extern void yyerror(const char*);

%}
%expect 1
```



```
/*============================================================================
                        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 EOF_TOK /* end of file */
%token LBRACE RBRACE LPAR RPAR LSQUARE RSQUARE
%token SEMI PLUS MINUS MUL_OP DIV_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 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_OP

/*============================================================================
                        BISON GRAMMAR
============================================================================*/
%%

/* 'program' is the starting non-terminal of the grammar.
 * A program is composed by:
      1. declarations (zero or more);
      2. A list of instructions. (at least one instruction!).
 * When the rule associated with the non-terminal 'program' is executed,
 * the parser notifies it to the 'program' singleton instance. */
program  : var_declarations statements EOF_TOK
         {
            /* Notify the end of the program. Once called
             * the function 'set_end_Program' - if necessary -
             * introduces a 'HALT' instruction into the
             * list of instructions. */
            set_end_Program(program);

            /* return from yyparse() */
            YYACCEPT;
         }
;

var_declarations : var_declarations var_declaration      { /* does nothing */ }
                 | /* empty */                           { /* does nothing */ }
;

var_declaration      : TYPE declaration_list SEMI
                     {
                        /* update the program infos by adding new variables */
                        set_new_variables(program, $1, $2);
                     }
;

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;

               /* 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
                  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
                 * jump if '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
                      * register 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 exp {
                if ($2.expression_type == IMMEDIATE)
                {
```

```
                  /* IMMEDIATE (constant) expression: compute the value at
                   * compile-time and place the result in a new IMMEDIATE
                   * expression */
                  $$ = create_expression(!($2.value), IMMEDIATE);
               }
               else
               {
                  /* REGISTER expression: generate the code that will compute
                   * the result at compile time */

                  /* Reserve a new register for the result */
                  int output_register = getNewRegister(program);

                  /* Generate a NOTL instruction which will store the negated
                   * logic value into the register we reserved */
                  gen_notl_instruction(program, output_register, $2.value);

                  /* Return a REGISTER expression with the result register */
                  $$ = create_expression(output_register, REGISTER);
               }
   }
   | exp AND_OP exp { $$ = 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 register REG_0 */
                     exp_r0.value = REG_0;
                     exp_r0.expression_type = REGISTER;

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

%%
/*=========================================================================
                                 MAIN
=========================================================================*/
int main (int argc, char **argv)
{
   /* initialize all the compiler data structures and global variables */
   init_compiler(argc, argv);

   /* start the parsing procedure */
   yyparse();

#ifndef NDEBUG
   fprintf(stdout, "Parsing process completed. \n");
   printProgramInfos(program, file_infos->frontend_output);
#endif

   /* test if the parsing process completed succesfully */
   checkConsistency();

   /* do not attach a line number to the instructions generated by the
    * transformations that follow. */
   line_num = -1;

   doTargetSpecificTransformations(program);

#ifndef NDEBUG
   fprintf(stdout, "Creating a control flow graph. \n");
#endif

   /* create the control flow graph */
   graph = createFlowGraph(program->instructions);
   checkConsistency();

#ifndef NDEBUG
   assert(program != NULL);
   assert(program->sy_table != NULL);
   assert(file_infos != NULL);
   assert(file_infos->syTable_output != NULL);
   printSymbolTable(program->sy_table, file_infos->syTable_output);
```



```
   printGraphInfos(graph, file_infos->cfg_1, 0);

   fprintf(stdout, "Updating the basic blocks. \n");
#endif

   /* update the control flow graph by inserting load and stores inside
   * every basic block */
   graph = insertLoadAndStoreInstr(program, graph);

#ifndef NDEBUG
   fprintf(stdout, "Executing a liveness analysis on the intermediate code \n");
#endif
   performLivenessAnalysis(graph);
   checkConsistency();

#ifndef NDEBUG
   printGraphInfos(graph, file_infos->cfg_2, 1);
#endif

#ifndef NDEBUG
   fprintf(stdout, "Starting the register allocation process. \n");
#endif
   /* initialize the register allocator by using the control flow
    * informations stored into the control flow graph */
   RA = initializeRegAlloc(graph);

   /* execute the linear scan algorithm */
   execute_linear_scan(RA);

#ifndef NDEBUG
   printRegAllocInfos(RA, file_infos->reg_alloc_output);
#endif

#ifndef NDEBUG
   fprintf(stdout, "Updating the control flow informations. \n");
#endif
   /* apply changes to the program informations by using the informations
   * of the register allocation process */
   materializeRegisterAllocation(program, graph, RA);
   updateProgramInfos(program, graph);

#ifndef NDEBUG
   fprintf(stdout, "Writing the assembly file... \n");
#endif
   writeAssembly(program, file_infos->output_file_name);

#ifndef NDEBUG
   fprintf(stdout, "Assembly written on file \"%s\".\n", file_infos->output_file_name);
#endif

   /* shutdown the compiler */
   shutdownCompiler(0);

   return 0;
}

/*=========================================================================
                                YYERROR
=========================================================================*/
void yyerror(const char* msg)
{
   errorcode = AXE_SYNTAX_ERROR;
   free((void *)errormsg);
   errormsg = strdup(msg);
}
```



```
/*
 * Andrea Di Biagio
 * Politecnico di Milano, 2007
 *
 * axe_array.h
 * Formal Languages & Compilers Machine, 2007/2008
 *
 * Code generation for array management (load/store)
 */

#ifndef _AXE_ARRAY_H
#define _AXE_ARRAY_H

#include "axe_engine.h"
#include "axe_struct.h"

/* This function generates instructions that load the content of
 * an element of an array in a register.  This function takes as
 * input: a variable identifier (ID) that refers to an array
 * value; an index value that refers to a specific element of
 * the array.  It returns the location identifier for the
 * register that will contain the value of the array element at
 * position 'index'.  'index' is an expression: its value can be
 * either a register location (i.e., the value of 'index' is
 * stored inside a register) or an immediate value. */
extern int loadArrayElement(
      t_program_infos *program, char *ID, t_axe_expression index);

/* This function generates instructions that load the address of
 * an element of an array in a regester.  This function takes as
 * input: a variable identifier (ID) that refers to an array
 * value; an index value that refers to a specific element of
 * the array.  It returns the location identifier for the
 * register that will contain the address of the array element
 * at position 'index'.  'index' is an expression: its value can
 * be either a register location (i.e., the value of 'index' is
 * stored inside a register) or an immediate value. */
extern int loadArrayAddress(
      t_program_infos *program, char *ID, t_axe_expression index);

/* This function generates instructions that store a value
 * specified by 'data' into the element at position 'index' of
 * the array 'ID'.  This function takes as input: a variable
 * identifier (ID) that refers to an array value; an index value
 * that refers to a specific element of the array; a value to be
 * stored (data).  'data' and 'index' are expressions: their
 * value can be either register locations (i.e. their values are
 * stored inside a register) or immediate values. */
extern void storeArrayElement(t_program_infos *program, char *ID,
      t_axe_expression index, t_axe_expression data);

#endif
```

```c
/*
 * Andrea Di Biagio
 * Politecnico di Milano, 2007
 *
 * axe_constants.h
 * Formal Languages & Compilers Machine, 2007/2008
 *
 */

#ifndef _AXE_CONSTANTS_H
#define _AXE_CONSTANTS_H

/* registers */
#define REG_INVALID -1
#define REG_0 0

/* MACE opcodes */
#define ADD 0
#define SUB 1
#define ANDL 2
#define ORL 3
#define EORL 4
#define ANDB 5
#define ORB 6
#define EORB 7
#define MUL 8
#define DIV 9
#define SHL 10
#define SHR 11
#define ROTL 12
#define ROTR 13
#define NEG 14
#define SPCL 15
#define ADDI 16
#define SUBI 17
#define ANDLI 18
#define ORLI 19
#define EORLI 20
#define ANDBI 21
#define ORBI 22
#define EORBI 23
#define MULI 24
#define DIVI 25
#define SHLI 26
#define SHRI 27
#define ROTLI 28
#define ROTRI 29
#define NOTL 30
#define NOTB 31
#define NOP 32
#define MOVA 33
#define JSR 34
#define RET 35
#define HALT 36
#define SEQ 37
#define SGE 38
#define SGT 39
#define SLE 40
#define SLT 41
#define SNE 42
#define BT 43
#define BF 44
#define BHI 45
#define BLS 46
#define BCC 47
#define BCS 48
#define BNE 49
#define BEQ 50
#define BVC 51
#define BVS 52
#define BPL 53
#define BMI 54
#define BGE 55
#define BLT 56
#define BGT 57
#define BLE 58
#define LOAD 59
#define STORE 60
#define AXE_READ 61
#define AXE_WRITE 62
#define INVALID_OPCODE -1

/* data types */
#define INTEGER_TYPE 0
#define UNKNOWN_TYPE -1

/* WARNINGS */
#define WARN_DIVISION_BY_ZERO 1
#define WARN_INVALID_SHIFT_AMOUNT 2

/* errorcodes */
#define AXE_OK 0
#define AXE_OUT_OF_MEMORY 1
#define AXE_PROGRAM_NOT_INITIALIZED 2
#define AXE_INVALID_INSTRUCTION 3
#define AXE_VARIABLE_ID_UNSPECIFIED 4
#define AXE_VARIABLE_ALREADY_DECLARED 5
#define AXE_INVALID_TYPE 6
```



```c
#define AXE_FOPEN_ERROR 7
#define AXE_FCLOSE_ERROR 8
#define AXE_INVALID_INPUT_FILE 9
#define AXE_FWRITE_ERROR 10
#define AXE_INVALID_DATA_FORMAT 11
#define AXE_INVALID_OPCODE 12
#define AXE_INVALID_REGISTER_INFO 13
#define AXE_INVALID_LABEL 14
#define AXE_INVALID_ARRAY_SIZE 15
#define AXE_INVALID_VARIABLE 16
#define AXE_INVALID_ADDRESS 17
#define AXE_INVALID_EXPRESSION 18
#define AXE_UNKNOWN_VARIABLE 19
#define AXE_LABEL_ALREADY_ASSIGNED 20
#define AXE_INVALID_LABEL_MANAGER 21
#define AXE_SY_TABLE_ERROR 22
#define AXE_NULL_DECLARATION 23
#define AXE_INVALID_CFLOW_GRAPH 24
#define AXE_INVALID_REG_ALLOC 25
#define AXE_REG_ALLOC_ERROR 26
#define AXE_TRANSFORM_ERROR 27
#define AXE_SYNTAX_ERROR 28
#define AXE_UNKNOWN_ERROR 29

/* DIRECTIVE TYPES */
#define DIR_WORD 0
#define DIR_SPACE 1
#define DIR_INVALID -1

/* ADDRESS TYPES */
#define ADDRESS_TYPE 0
#define LABEL_TYPE 1

/* CODEGEN FLAGS */
#define CG_DIRECT_ALL 0       /*  DEST =  SRC1 <OP>  SRC2  */
#define CG_INDIRECT_ALL 3     /* [DEST] = SRC1 <OP> [SRC2] */
#define CG_INDIRECT_DEST 1    /* [DEST] = SRC1 <OP>  SRC2  */
#define CG_INDIRECT_SOURCE 2  /*  DEST =  SRC1 <OP> [SRC2] */

/* EXPRESSION TYPES */
#define IMMEDIATE 0
#define REGISTER 1
#define INVALID_EXPRESSION -1

/* binary comparison constants */
#define _LT_ 0
#define _GT_ 1
#define _EQ_ 2
#define _NOTEQ_ 3
#define _LTEQ_ 4
#define _GTEQ_ 5

#endif
```



```c
/*
 * Andrea Di Biagio
 * Politecnico di Milano, 2007
 *
 * axe_engine.h
 * Formal Languages & Compilers Machine, 2007/2008
 *
 * Contains t_program_infos and some functions for label management
 * (reserve, fix, assign)
 */

#ifndef _AXE_ENGINE_H
#define _AXE_ENGINE_H

#include "axe_struct.h"
#include "axe_labels.h"
#include "collections.h"
#include "symbol_table.h"

typedef struct t_program_infos
{
  t_list *variables;
  t_list *instructions;
  t_list *instrInsPtrStack;
  t_list *data;
  t_axe_label_manager *lmanager;
  t_symbol_table *sy_table;
  int current_register;
} t_program_infos;


/* initialize the informations associated with the program. This function is
 * called at the beginning of the translation process. This function
 * is called once: its only purpouse is to initialize an instance of the struct
 * 't_program_infos' that will contain all the informations about the program
 * that will be compiled */
extern t_program_infos *allocProgramInfos();

/* add a new instruction to the current program. This function is directly
 * called by all the functions defined in 'axe_gencode.h' */
extern void addInstruction(t_program_infos *program, t_axe_instruction *instr);

/* remove an instruction from the program, given its link in the instruction
 * list. */
extern void removeInstructionLink(t_program_infos *program, t_list *instrLi);

/* Save the current insertion point in the instruction list, and replace it
 * with `ip`. New instructions will be inserted after the `ip` instruction.
 * To insert instructions at the beginning of the program, ip shall be NULL. */
extern void pushInstrInsertionPoint(t_program_infos *p, t_list *ip);

/* Restore the last insertion point in the instruction list. Returns the
 * previous position of the instruction insertion point. */
extern t_list *popInstrInsertionPoint(t_program_infos *p);

/* reserve a new label identifier and return the identifier to the caller */
extern t_axe_label *newLabel(t_program_infos *program);

/* assign the given label identifier to the next instruction. Returns
 * the label assigned; otherwise (an error occurred) NULL */
extern t_axe_label *assignLabel(t_program_infos *program, t_axe_label *label);

/* reserve and fix a new label. It returns either the label assigned or
 * NULL if an error occurred */
extern t_axe_label *assignNewLabel(t_program_infos *program);

/* Like the above functions, but with the ability to give a name to the label.
 * If another label with the same name already exists, the name assigned to
 * the new label will be modified to remove any ambiguity. */
extern t_axe_label *newNamedLabel(t_program_infos *program, const char *name);
extern t_axe_label *assignNewNamedLabel(
      t_program_infos *program, const char *name);

/* add a variable to the program */
extern void createVariable(t_program_infos *program, char *ID, int type,
      int isArray, int arraySize, int init_val);

/* get a previously allocated variable */
extern t_axe_variable *getVariable(t_program_infos *program, char *ID);

/* get the label that marks the starting address of the variable
 * with name "ID" */
extern t_axe_label *getLabelFromVariableID(t_program_infos *program, char *ID);

/* get a register still not used. This function returns
 * the ID of the register found*/
extern int getNewRegister(t_program_infos *program);

/* finalize all the data structures associated with 'program' */
extern void finalizeProgramInfos(t_program_infos *program);

#endif
```

```
/*
 * Andrea Di Biagio
 * Politecnico di Milano, 2007
 *
 * axe_expressions.h
 * Formal Languages & Compilers Machine, 2007/2008
 *
 * Support functions for t_axe_expressions.
 */

#ifndef _AXE_EXPRESSIONS_H
#define _AXE_EXPRESSIONS_H

#include "axe_engine.h"

/* This function generates instructions for binary numeric
 * operations.  It takes as input two expressions and a binary
 * operation identifier, and it returns a new expression that
 * represents the result of the specified binary operation
 * applied to 'exp1' and 'exp2'.
 * If the two expressions are both IMMEDIATE, no instructions are generated
 * and an IMMEDIATE expression is returned.
 *
 * Valid values for 'binop' are:
 * ADD
 * ANDB
 * ANDL
 * ORB
 * ORL
 * EORB
 * EORL
 * SUB
 * MUL
 * SHL
 * SHR
 * DIV  */
extern t_axe_expression handle_bin_numeric_op(t_program_infos *program,
     t_axe_expression exp1, t_axe_expression exp2, int binop);

/* This function generates instructions that perform a
 * comparison between two values.  It takes as input two
 * expressions and a binary comparison identifier, and it
 * returns a new expression that represents the result of the
 * specified binary comparison between 'exp1' and 'exp2'.
 * If the two expressions are both IMMEDIATE, no instructions are generated
 * and an IMMEDIATE expression is returned.
 *
 * Valid values for 'condition' are:
 * _LT_     (used to test if the value of 'exp1' is less than
 *           the value of 'exp2')
 * _GT_     (used to test if the value of 'exp1' is greater than
 *           the value of 'exp2')
 * _EQ_     (used to test if the value of 'exp1' is equal to
 *           the value of 'exp2')
 * _NOTEQ_  (used to test if the value of 'exp1' is not equal to
 *           the value of 'exp2')
 * _LTEQ_   (used to test if the value of 'exp1' is less than
 *           or equal to the value of 'exp2')
 * _GTEQ_   (used to test if the value of 'exp1' is greater than
 *           the value of 'exp2')  */
extern t_axe_expression handle_binary_comparison(t_program_infos *program,
     t_axe_expression exp1, t_axe_expression exp2, int condition);

#endif
```



```
/*
 * Andrea Di Biagio
 * Politecnico di Milano, 2007
 *
 * axe_gencode.h
 * Formal Languages & Compilers Machine, 2007/2008
 *
 * Code generation functions. See also axe_utils.h for gen_load_immediate()
 * and gen_move_immediate().
 */

#ifndef _AXE_GENCODE_H
#define _AXE_GENCODE_H

#include "axe_engine.h"
#include "axe_struct.h"

/*-----------------------------------------------------
 *                    NOP & HALT
 *-----------------------------------------------------*/

/* By calling this function, a new NOP instruction will be added
 * to 'program'. A NOP instruction doesn't make use of
 * any kind of parameter */
extern t_axe_instruction *gen_nop_instruction(t_program_infos *program);

/* By calling this function, a new HALT instruction will be added
 * to 'program'. An HALT instruction doesn't require
 * any kind of parameter */
extern t_axe_instruction *gen_halt_instruction(t_program_infos *program);

/*-----------------------------------------------------
 *                    UNARY OPERATIONS
 *-----------------------------------------------------*/

/* A LOAD instruction requires the following parameters:
 * 1.  A destination register where the requested value will be loaded
 * 2.  A label information (can be a NULL pointer. If so, the addess
 *     value will be taken into consideration)
 * 3.  A direct address (if label is different from NULL) */
extern t_axe_instruction *gen_load_instruction(
     t_program_infos *program, int r_dest, t_axe_label *label, int address);

/* A READ instruction requires only one parameter:
 * A destination register (where the value
 * read from standard input will be loaded). */
extern t_axe_instruction *gen_read_instruction(
     t_program_infos *program, int r_dest);

/* A WRITE instruction requires only one parameter:
 * A destination register (where the value
 * that will be written to the standard output is located). */
extern t_axe_instruction *gen_write_instruction(
     t_program_infos *program, int r_dest);

/* A STORE instruction copies a value from a register to a
 * specific memory location. The memory location can be
 * either a label identifier or a address reference.
 * In order to create a STORE instruction the caller must
 * provide a valid register location ('r_dest') and an
 * instance of 't_axe_label' or a numeric address */
extern t_axe_instruction *gen_store_instruction(
     t_program_infos *program, int r_dest, t_axe_label *label, int address);

/* A MOVA instruction copies an address value into a register.
 * An address can be either an instance of 't_axe_label'
 * or a number (numeric address) */
extern t_axe_instruction *gen_mova_instruction(
     t_program_infos *program, int r_dest, t_axe_label *label, int address);

/*
 * STATUS REGISTER TEST INSTRUCTIONS
 */

/* A SGE instruction tests the content of the STATUS REGISTER. To be more
 * specific, a SGE instruction sets to #1 the content of the register
 * 'r_dest' if the condition (N.V + ~N.~V) is TRUE; otherwise the content
 * of 'r_dest' is set to 0.
 * (I.e.: r_dest will be set to #1 only if the value computed by
 * the last numeric operation returned a value
 * greater or equal to zero). */
extern t_axe_instruction *gen_sge_instruction(
     t_program_infos *program, int r_dest);

/* A SEQ instruction tests the content of the STATUS REGISTER. In particular,
 * a SEQ instruction sets to #1 the content of the register
 * 'r_dest' if the condition Z is TRUE; otherwise the content of 'r_dest' is set
 * to 0. (I.e.: r_dest will be set to #1 only if the value computed by
 * the last numeric operation returned a value equal to zero). */
extern t_axe_instruction *gen_seq_instruction(
     t_program_infos *program, int r_dest);

/* A SGT instruction tests the content of the STATUS REGISTER. In particular,
 * a SGT instruction sets to #1 the content of the register
 * 'r_dest' if the condition (N.V.~Z + ~N.~V.~Z) is TRUE;
 * otherwise the content of 'r_dest' is set to 0. (I.e.: r_dest will be
 * set to #1 only if the value computed by the last numeric operation
 * returned a value greater than zero). */
```



```
extern t_axe_instruction *gen_sgt_instruction(
     t_program_infos *program, int r_dest);

/* A SLE instruction tests the content of the STATUS REGISTER. In particular,
 * a SLE instruction sets to #1 the content of the register
 * 'r_dest' if the condition (Z + N.~V + ~N.V) is TRUE;
 * otherwise the content of 'r_dest' is set to 0. (I.e.: r_dest will be
 * set to #1 only if the value computed by the last numeric operation
 * returned a value less than zero). */
extern t_axe_instruction *gen_sle_instruction(
     t_program_infos *program, int r_dest);

/* A SLT instruction tests the content of the STATUS REGISTER. In particular,
 * a SLT instruction sets to #1 the content of the register
 * 'r_dest' if the condition (N.~V + ~N.V) is TRUE;
 * otherwise the content of 'r_dest' is set to 0. (I.e.: r_dest will be
 * set to #1 only if the value computed by the last numeric operation
 * returned a value less than or equal to zero). */
extern t_axe_instruction *gen_slt_instruction(
     t_program_infos *program, int r_dest);

/* A SNE instruction tests the content of the STATUS REGISTER. In particular,
 * a SNE instruction sets to #1 the content of the register
 * 'r_dest' if the condition ~N is TRUE;
 * otherwise the content of 'r_dest' is set to 0. (I.e.: r_dest will be
 * set to #1 only if the value computed by the last numeric operation
 * returned a value different from zero). */
extern t_axe_instruction *gen_sne_instruction(
     t_program_infos *program, int r_dest);

/*-----------------------------------------------------
 *                    BINARY OPERATIONS
 *-----------------------------------------------------*/

/* Used in order to create and assign to the current 'program'
 * an ADDI instruction. The semantic of an ADDI instruction
 * is the following: ADDI r_dest, r_source1, immediate. 'RDest' is a register
 * location identifier: the result of the ADDI instruction will be
 * stored in that register. Using an RTL (Register Transfer Language)
 * representation we can say that an ADDI instruction of the form:
 * ADDI R1 R2 #IMM can be represented in the following manner: R1 <-- R2 + IMM.
 * 'Rsource1' and '#IMM' are the two operands of the binary numeric
 * operation. 'r_dest' is a register location, 'immediate' is an immediate
 * value. The content of 'r_source1' is added to the value of 'immediate'
 * and the result is then stored into the register 'RDest'. */
extern t_axe_instruction *gen_addi_instruction(
     t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * a SUBI instruction. The semantic of an SUBI instruction
 * is the following: SUBI r_dest, r_source1, immediate. 'RDest' is a register
 * location identifier: the result of the SUBI instruction will be
 * stored in that register. Using an RTL representation we can say
 * that a SUBI instruction of the form: SUBI R1 R2 #IMM can be represented
 * in the following manner: R1 <-- R2 - IMM.
 * 'Rsource1' and '#IMM' are the two operands of the binary numeric
 * operation. 'r_dest' is a register location, 'immediate' is an immediate
 * value. The content of 'r_source1' is subtracted to the value of 'immediate'
 * and the result is then stored into the register 'RDest'. */
extern t_axe_instruction *gen_subi_instruction(
     t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * an ANDLI instruction. An example RTL representation of ANDLI R1 R2 #IMM is:
 * R1 <-- R2 && IMM.
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_andli_instruction(
     t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * a ORLI instruction. An example RTL representation of ORLI R1 R2 #IMM is:
 * R1 <-- R2 || IMM.
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_orli_instruction(
     t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * a EORLI instruction. An example RTL representation of EORLI R1 R2 #IMM is:
 * R1 <-- R2 XOR IMM (Where XOR is the operator: logical exclusive OR).
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_eorli_instruction(
     t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * an ANDBI instruction. An example RTL representation of ANDBI R1 R2 #IMM is:
 * R1 <-- R2 & IMM (bitwise AND).
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_andbi_instruction(
     t_program_infos *program, int r_dest, int r_source1, int immediate);
```

```
/* Used in order to create and assign to the current 'program'
 * a MULI instruction. An example RTL representation of MULI is:
 * R1 <-- R2 * IMM.
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_muli_instruction(
      t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * an ORBI instruction. An example RTL representation of ORBI R1 R2 #IMM is:
 * R1 <-- R2 | IMM.
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_orbi_instruction(
      t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * a EORBI instruction. An example RTL representation of EORBI R1 R2 #IMM is:
 * R1 <-- R2 ^ IMM.
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_eorbi_instruction(
      t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * a DIVI instruction. An example RTL representation of DIVI R1 R2 #IMM is:
 * R1 <-- R2 / IMM.
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_divi_instruction(
      t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * a SHLI instruction. An example RTL representation of SHLI R1 R2 #IMM is:
 * R1 <-- R2 / IMM.
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_shli_instruction(
      t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * a SHRI instruction. An example RTL representation of SHRI R1 R2 #IMM is:
 * R1 <-- R2 / IMM.
 * 'r_source1' and 'immediate' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location, 'immediate' is an immediate
 * value. */
extern t_axe_instruction *gen_shri_instruction(
      t_program_infos *program, int r_dest, int r_source1, int immediate);

/* Used in order to create and assign to the current 'program'
 * a NOTL instruction. An example RTL representation of NOTL R1 R2 is:
 * R1 <-- !R2. */
extern t_axe_instruction *gen_notl_instruction(
      t_program_infos *program, int r_dest, int r_source1);

/* Used in order to create and assign to the current 'program'
 * a NOTB instruction. An example RTL representation of NOTB R1 R2 is:
 * R1 <-- ~R2. */
extern t_axe_instruction *gen_notb_instruction(
      t_program_infos *program, int r_dest, int r_source1);

/*-------------------------------------------------------
 *                    TERNARY OPERATIONS
 *-------------------------------------------------------*/

/* Used in order to create and assign to the current 'program'
 * a ADD instruction. An example RTL representation of ADD R1 R2 R3 is:
 * R1 <-- R2 + R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_add_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a SUB instruction. An example RTL representation of SUB R1 R2 R3 is:
 * R1 <-- R2 - R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_sub_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a ANDL instruction. An example RTL representation of ANDL R1 R2 R3 is:
 * R1 <-- R2 && R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_andl_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
```



```
 * a ORL instruction. An example RTL representation of ORL R1 R2 R3 is:
 * R1 <-- R2 || R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_orl_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a EORL instruction. An example RTL representation of EORL R1 R2 R3 is:
 * R1 <-- R2 XORL R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_eorl_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a ANDB instruction. An example RTL representation of ANDB R1 R2 R3 is:
 * R1 <-- R2 & R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_andb_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a ORB instruction. An example RTL representation of ORB R1 R2 R3 is:
 * R1 <-- R2 | R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_orb_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a EORB instruction. An example RTL representation of EORB R1 R2 R3 is:
 * R1 <-- R2 XORB R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_eorb_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a MUL instruction. An example RTL representation of MUL R1 R2 R3 is:
 * R1 <-- R2 * R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_mul_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a DIV instruction. An example RTL representation of DIV R1 R2 R3 is:
 * R1 <-- R2 / R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_div_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a SHL instruction. An example RTL representation of SHL R1 R2 R3 is:
 * R1 <-- R2 shifted to left by R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_shl_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a SHR instruction. An example RTL representation of SHR R1 R2 R3 is:
 * R1 <-- R2 shifted to right by R3.
 * 'r_source1' and 'r_source2' are the two operands of the binary numeric
 * comparison. 'r_dest' is a register location. 'r_dest' and 'r_source2'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_shr_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/* Used in order to create and assign to the current 'program'
 * a NEG instruction. An example RTL representation of NEG R1 R2 is:
 * as follows: R1 <-- (-R2).
 * 'r_source' is the only operand for this instruction.
 * 'r_dest' is a register location. 'r_dest' and 'r_source'
 * are register locations that can be directly or indirectly addressed. */
extern t_axe_instruction *gen_neg_instruction(
      t_program_infos *program, int r_dest, int r_source, int flags);

/* This instruction is reserved for future implementation. */
extern t_axe_instruction *gen_spcl_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags);

/*-------------------------------------------------------
 *                    JUMP INSTRUCTIONS
 *-------------------------------------------------------*/

/* create a branch true instruction. By executing this instruction the control
```



```
 * is always passed to either the instruction with the label 'label' associated
 * with, or (if 'label' is a NULL pointer) to the explicit 'address' */
extern t_axe_instruction *gen_bt_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a branch false instruction. By executing this instruction the control
 * is always passed to the next instruction in the program
 * (i.e.: the instruction pointed by PC + 1). */
extern t_axe_instruction *gen_bf_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create an unsigned "branch on higher than" instruction. According to the
 * value of the status register, the branch will be taken if the expression
 * ~(C + Z) is TRUE. */
extern t_axe_instruction *gen_bhi_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create an unsigned "branch on less than (or equal)" instruction. According
 * to the value of the status register, the branch will be taken if the
 * expression (C + Z) is TRUE. */
extern t_axe_instruction *gen_bls_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on carry clear" instruction. If the bit 'C' of the
 * status register is not set, then the branch is taken. */
extern t_axe_instruction *gen_bcc_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on carry set" instruction. If the bit 'C' of the
 * status register is set, then the branch is taken. */
extern t_axe_instruction *gen_bcs_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on not equal" instruction. If the bit 'Z' of the
 * status register is not set, then the branch is taken. */
extern t_axe_instruction *gen_bne_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on equal" instruction. If the bit 'Z' of the
 * status register is set, then the branch is taken. */
extern t_axe_instruction *gen_beq_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on overflow clear" instruction. If the bit 'V' of the
 * status register is not set, then the branch is taken. */
extern t_axe_instruction *gen_bvc_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on overflow set" instruction. If the bit 'V' of the
 * status register is set, then the branch is taken. */
extern t_axe_instruction *gen_bvs_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on plus (i.e. positive)" instruction. If the bit 'N' of the
 * status register is not set, then the branch is taken. */
extern t_axe_instruction *gen_bpl_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on minus (i.e. negative)" instruction. If the bit 'N' of the
 * status register is set, then the branch is taken. */
extern t_axe_instruction *gen_bmi_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on greater or equal" instruction. According to the value
 * of the status register, the branch will be taken if the expression
 * (N.V + ~N.~V) is TRUE. */
extern t_axe_instruction *gen_bge_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on less than" instruction. According to the value
 * of the status register, the branch will be taken if the expression
 * (N.~V + ~N.V) is TRUE. */
extern t_axe_instruction *gen_blt_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on greater than" instruction. According to the value
 * of the status register, the branch will be taken if the expression
 * (N.V.~Z + ~N.~V.~Z) is TRUE. */
extern t_axe_instruction *gen_bgt_instruction(
      t_program_infos *program, t_axe_label *label, int addr);

/* create a "branch on less than or equal" instruction. According to the value
 * of the status register, the branch will be taken if the expression
 * (Z + N.~V + ~N.V) is TRUE. */
extern t_axe_instruction *gen_ble_instruction(
      t_program_infos *program, t_axe_label *label, int addr);


#endif
```

```c
/*
 * Andrea Di Biagio
 * Politecnico di Milano, 2007
 *
 * axe_struct.h
 * Formal Languages & Compilers Machine, 2007/2008
 *
 * Fundamental data structures
 */

#ifndef _AXE_STRUCT_H
#define _AXE_STRUCT_H

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include "axe_constants.h"

typedef struct t_axe_label
{
   unsigned int labelID; /* label identifier */
   char *name;           /* Name of the label. If NULL, the name will be
                          * automatically generated in the form L<ID>. */
} t_axe_label;

typedef struct t_axe_register
{
   int ID;        /* an identifier of the register */
   int indirect;  /* a boolean value: 1 if the register value is a pointer */
}t_axe_register;

typedef struct t_axe_address
{
   int addr;              /* a Program Counter */
   t_axe_label *labelID;  /* a label identifier */
   int type;              /* one of ADDRESS_TYPE or LABEL_TYPE */
}t_axe_address;

/* A structure that defines the internal data of a 'Acse variable' */
typedef struct t_axe_variable
{
   int type;      /* a valid data type @see 'axe_constants.h' */
   int isArray;   /* must be TRUE if the current variable is an array */
   int arraySize; /* the size of the array. This information is useful only
                   * if the field 'isArray' is TRUE */
   int init_val;  /* initial value of the current variable. Actually it is
                   * implemented as a integer value. 'int' is
                   * the only supported type at the moment,
                   * future developments could consist of a modification of
                   * the supported type system. Thus, maybe init_val will be
                   * modified in future. */
   char *ID;              /* variable identifier (should never be a NULL
                           * pointer or an empty string "") */
   t_axe_label *labelID;  /* a label that refers to the location
                           * of the variable inside the data segment */
} t_axe_variable;

/* a symbolic assembly instruction */
typedef struct t_axe_instruction
{
   int opcode;                  /* instruction opcode (for example: ADD) */
   t_axe_register *reg_1;       /* destination register */
   t_axe_register *reg_2;       /* first source register */
   t_axe_register *reg_3;       /* second source register */
   int immediate;               /* immediate value */
   t_axe_address *address;      /* an address operand */
   char *user_comment;          /* if defined it is set to the source code
                                 * instruction that generated the current
                                 * assembly. This string will be written
                                 * into the output code as a comment */
   t_axe_label *labelID;        /* a label associated with the current
                                 * instruction */
}t_axe_instruction;

/* this structure is used in order to define assembler directives.
 * Directives are used in many cases such the definition of variables
 * inside the data segment. Every instance 't_axe_data' contains
 * all the informations about a single directive.
 * An example is the directive .word that is required when the assembler
 * must reserve a word of data inside the data segment. */
typedef struct t_axe_data
{
   int directiveType;      /* the type of the current directive
                            * (for example: DIR_WORD) */
   int value;              /* the value associated with the directive */
   t_axe_label *labelID;   /* label associated with the current data */
}t_axe_data;

typedef struct t_axe_expression
{
   int value;           /* an immediate value or a register identifier */
   int expression_type; /* actually only integer values are supported */
} t_axe_expression;

typedef struct t_axe_declaration
{
   int isArray;          /* must be TRUE if the current variable is an array */
   int arraySize;        /* the size of the array. This information is useful
                          * only if the field 'isArray' is TRUE */
   int init_val;         /* initial value of the current variable. */
   char *ID;             /* variable identifier (should never be a NULL pointer
                          * or an empty string "") */
} t_axe_declaration;

typedef struct t_while_statement
{
   t_axe_label *label_condition;   /* this label points to the expression
                                    * that is used as loop condition */
   t_axe_label *label_end;         /* this label points to the instruction
                                    * that follows the while construct */
} t_while_statement;

/* create a label */
extern t_axe_label *alloc_label(int value);

/* free a label */
extern void free_label(t_axe_label *lab);

/* create an expression */
extern t_axe_expression create_expression(int value, int type);

/* create an instance that will mantain infos about a while statement */
extern t_while_statement create_while_statement();

/* create an instance of 't_axe_register' */
extern t_axe_register *alloc_register(int ID, int indirect);

/* create an instance of 't_axe_instruction' */
extern t_axe_instruction *alloc_instruction(int opcode);

/* create an instance of 't_axe_address' */
extern t_axe_address *alloc_address(int type, int address, t_axe_label *label);

/* create an instance of 't_axe_data' */
extern t_axe_data *alloc_data(int directiveType, int value, t_axe_label *label);

/* create an instance of 't_axe_variable' */
extern t_axe_variable *alloc_variable(
      char *ID, int type, int isArray, int arraySize, int init_val);

/* finalize an instance of 't_axe_variable' */
extern void free_variable(t_axe_variable *variable);

/* create an instance of 't_axe_variable' */
extern t_axe_declaration *alloc_declaration(
      char *ID, int isArray, int arraySize, int init_val);

/* finalize an instruction info. */
extern void free_Instruction(t_axe_instruction *inst);

/* finalize a data info. */
extern void free_Data(t_axe_data *data);

#endif
```



```c
/*
 * Andrea Di Biagio
 * Politecnico di Milano, 2007
 *
 * axe_utils.h
 * Formal Languages & Compilers Machine, 2007/2008
 *
 * Contains important functions to access the list of symbols and other
 * utility functions and macros.
 */

#ifndef _AXE_UTILS_H
#define _AXE_UTILS_H

#include "axe_engine.h"
#include "axe_struct.h"
#include "axe_constants.h"
#include "collections.h"

/* maximum and minimum between two values */
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define MIN(x, y) ((x) > (y) ? (y) : (x))

/* create a variable for each 't_axe_declaration' inside
 * the list 'variables'. Each new variable will be of type
 * 'varType'. */
extern void set_new_variables(
      t_program_infos *program, int varType, t_list *variables);

/* Given a variable/symbol identifier (ID) this function
 * returns a register location where the value is stored
 * (the value of the variable identified by 'ID').
 * If the variable/symbol has never been loaded from memory
 * to a register, first this function searches
 * for a free register, then it assign the variable with the given
 * ID to the register just found.
 * Once computed, the location (a register identifier) is returned
 * as output to the caller.
 * This function generates a LOAD instruction
 * only if the flag 'genLoad' is set to 1; otherwise it simply reserve
 * a register location for a new variable in the symbol table.
 * If an error occurs, get_symbol_location returns a REG_INVALID errorcode */
extern int get_symbol_location(t_program_infos *program, char *ID, int genLoad);

/* Generate the instruction to load an 'immediate' value into a new register.
 * It returns the new register identifier or REG_INVALID if an error occurs */
extern int gen_load_immediate(t_program_infos *program, int immediate);

/* Generate the instruction to move an 'immediate' value into a register. */
extern void gen_move_immediate(t_program_infos *program, int dest, int imm);

/* Returns 1 if 'instr' is a jump (branch) instruction. */
extern int isJumpInstruction(t_axe_instruction *instr);

/* Returns 1 if 'instr' is a unconditional jump instruction (BT, BF) */
extern int isUnconditionalJump(t_axe_instruction *instr);

/* Returns 1 if 'instr' is either the HALT instruction or the RET
 * instruction. */
extern int isHaltOrRetInstruction(t_axe_instruction *instr);

/* Returns 1 if 'instr' is the LOAD instruction. */
extern int isLoadInstruction(t_axe_instruction *instr);

/* Returns 1 if the opcode corresponds to an instruction with an immediate
 * argument (i.e. if the instruction mnemonic ends with 'I'). */
extern int isImmediateArgumentInstrOpcode(int opcode);

/* Switches the immediate form of an opcode. For example, ADDI is transformed
 * to ADD, and ADD is transformed to ADDI. Returns the original opcode in case
 * there is no immediate or non-immediate available. */
extern int switchOpcodeImmediateForm(int orig);

/* Notify the end of the program. This function is directly called
 * from the parser when the parsing process is ended */
extern void set_end_Program(t_program_infos *program);

/* Once called, this function destroys all the data structures
 * associated with the compiler (program, RA, etc.). This function
 * is typically automatically called before exiting from the main
 * or when the compiler encounters some error. */
extern void shutdownCompiler();

/* Once called, this function initialize all the data structures
 * associated with the compiler (program, RA etc..) and all the
 * global variables in the system. This function
 * is typically automatically called at the beginning of the main
 * and should NEVER be called from the user code */
extern void init_compiler(int argc, char **argv);

#endif
```

```
/*
 * Andrea Di Biagio
 * Politecnico di Milano, 2007
 *
 * collections.h
 * Formal Languages & Compilers Machine, 2007/2008
 *
 * A double-linked list. 'prev' pointer of first element and 'next' pointer of
 * last element are NULL.
 */

#ifndef _COLLECTIONS_H
#define _COLLECTIONS_H

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>


/* create a list data item from an integer value */
#define INTDATA(data)               ((void *)((intptr_t)(data)))

/* get the next list item. NULL if item is the last item in the list. */
#define LNEXT(item)                 ((item)->next)
/* get the previous list item. NULL if item is the first item in the list. */
#define LPREV(item)                 ((item)->prev)
/* get the data associated to this list item. */
#define LDATA(item)                 ((item)->data)
/* get the integer value data associated to this list item. */
#define LINTDATA(item)              ((int)((intptr_t)LDATA(item)))

/* set the next list item. */
#define SET_NEXT(item, _next)       ((item)->next = (_next))
/* set the previous list item. */
#define SET_PREV(item, _prev)       ((item)->prev = (_prev))
/* set the data associated to this list item. */
#define SET_DATA(item, _data)       ((item)->data = (_data))
/* set an integer value as the data associated to this list item. */
#define SET_INTDATA(item, _data)    ((item)->data = INTDATA(_data))


/* a list element */
typedef struct t_list
{
   void  *data;
   struct t_list *next;
   struct t_list *prev;
}t_list;


/* add an element 'data' to the list 'list' at position 'pos'. If pos is
 * negative, or is larger than the number of elements in the list, the new
 * element is added on to the end of the list. Function 'addElement' returns a
 * pointer to the new head of the list */
extern t_list *addElement(t_list *list, void *data, int pos);

/* add sorted */
extern t_list *addSorted(
      t_list *list, void *data, int (*compareFunc)(void *a, void *b));

/* add an element to the end of the list */
extern t_list *addLast(t_list *list, void *data);

/* add an element at the beginning of the list */
extern t_list *addFirst(t_list *list, void *data);

/* Add an element before a given element already in the list.
 * Returns the newly added element. */
extern t_list *addBefore(t_list *listPos, void *data);

/* Add an element after a given element already in the list.
 * Returns the newly added element. */
extern t_list *addAfter(t_list *listPos, void *data);

/* remove an element at the beginning of the list */
extern t_list *removeFirst(t_list *list);

/* remove an element from the list */
extern t_list *removeElement(t_list *list, void *data);

/* remove a link from the list 'list' */
extern t_list *removeElementLink(t_list *list, t_list *element);

/* find an element inside the list 'list'. The current implementation calls the
 * CustomfindElement' passing a NULL reference as 'func' */
extern t_list *findElement(t_list *list, void *data);

/* find an element inside the list 'list'. */
extern t_list *CustomfindElement(
      t_list *list, void *data, int (*compareFunc)(void *a, void *b));

/* find the position of an 'element' inside the 'list'. -1 if not found */
extern int getPosition(t_list *list, t_list *element);

/* find the length of 'list' */
extern int getLength(t_list *list);
```



```
/* remove all the elements of a list */
extern void freeList(t_list *list);

/* get the last element of the list. Returns NULL if the list is empty
 * or list is a NULL pointer */
extern t_list *getLastElement(t_list *list);

/* retrieve the list element at position 'position' inside the 'list'.
 * Returns NULL if: the list is empty, the list is a NULL pointer or
 * the list holds less than 'position' elements. */
extern t_list *getElementAt(t_list *list, unsigned int position);

/* create a new list with the same elements */
extern t_list *cloneList(t_list *list);

/* add a list of elements to another list */
extern t_list *addList(t_list *list, t_list *elements);

/* add a list of elements to a set */
extern t_list *addListToSet(t_list *list, t_list *elements,
      int (*compareFunc)(void *a, void *b), int *modified);


#endif
```



```
*****************************************************************************
                              IMPORTANT
                    HOW TO GENERATE CONDITIONAL JUMPS
*****************************************************************************

This is an example:

gen_beq_instruction( ... label ... )
    Generate a jump-if-equal instruction (i.e., jump if flag ZERO is SET) to
    'label'. That means a jump to 'label' if the preceding expression is
    FALSE.
    This is because when comparison result is ZERO, the comparison is FALSE
    and flag zero is SET.
```