# Formal Languages and Compilers
# Proff. Breveglieri, Crespi Reghizzi, Morzenti
# Written exam[1]: laboratory question
# 06/07/2010

SURNAME: .....
NAME: ..... Student ID: .....
Course: ○ Laurea Specialistica ○ V. O. ○ Laurea Triennale ○ Other:.....
Instructor: ○ Prof. Breveglieri ○ Prof. Crespi ○ Prof Morzenti

The laboratory question must be answered taking into account the implementation of the `Acse` compiler given with the exam text.

Modify the specification of the lexical analyzer (`flex` input) and the syntactic analyzer (`bison` input) and any other source file required to extend the `Lance` language with the ability to handle the `eval-unless` construct:

```
int x, y;

read(x);

y = 0;

eval {
  y = 1;
} unless x==5;

write(y);
```

Between `eval` and `unless` there is a code block. The `unless` keyword is followed by an expression. If the expression evaluates to false, the code block is executed. If the condition is true, the code block is not executed.

In the code sample above, if the `x` variable is initialized with the value 5, the execution will end printing 0 on the screen. For every other value in `x`, the code block containing `y = 1` will be executed and the execution will end printing 1 on the screen.

Explicit any other assumption you made to implement the support for the `eval-unless` construct.

[1] Time 45'. Textbooks and notes can be used.
Pencil writing is allowed. Write your name on any additional sheet.

1. Define the tokens (and the related declarations in **Acse.lex** e **Acse.y**). (1 points)

   Two tokens are needed, and they have to be declared in the Acse.lex file, adding the following lines.

   ```
   "eval"   { return EVAL; }
   "unless"   { return UNLESS; }
   ```

2. Define the syntactic rules or the modifications required to the existing ones. (4 points)

   Declare a new structure in the axe_struct.h file. It will contain the labels needed to generate the code for the construct:

   ```
   typedef struct t_unless_stmt {
     t_axe_label *label_condition;
     t_axe_label *label_code;
     t_axe_label *label_end;
   } t_unless_stmt;
   ```

   Then, modify the Acse.y as described hereafter. Expand the semantic record with a reference to the `t_unless_stmt` structure:

   ```
   %union {
     ...
     t_unless_stmt unless_stmt;
   }
   ```

   Declare the tokens. The `eval` token will contain an `unless_stmt` record.

   ```
   %token UNLESS
   %token <unless_stmt> EVAL
   ```

   The unless statement is a new kind of control statement:

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

   Finally, here is the rule describing the unless statement:

   ```
   unless_statement : EVAL code_block UNLESS exp
   ;
   ```

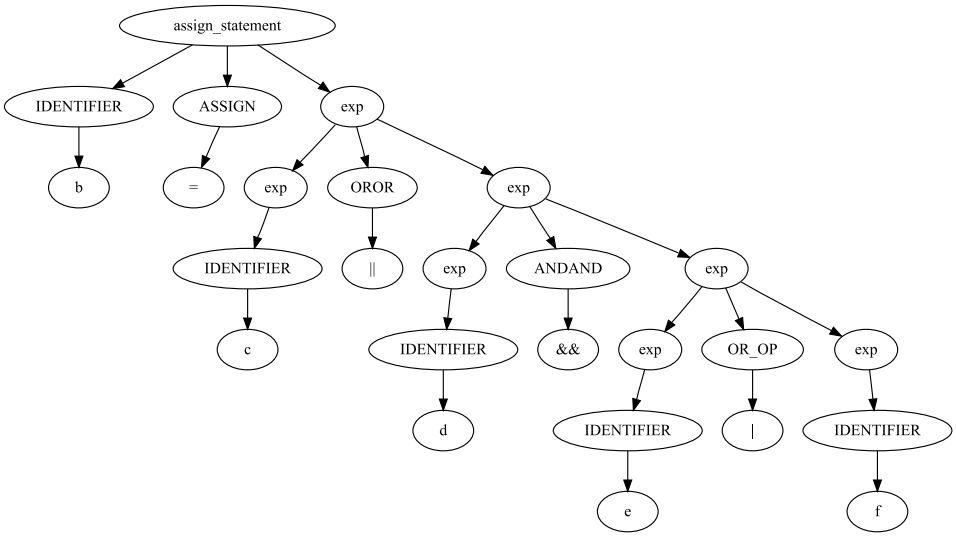
3. Define the semantic actions needed to implement the required functionality. (20 points)

```
unless_statement : EVAL
{
    $1.label_condition = newLabel(program);

    /* Jump to the evaluation of the expression */
    gen_bt_instruction(program, $1.label_condition, 0);

    /* Set the label that identifies the code block */
    $1.label_code = newLabel(program);
    assignLabel(program, $1.label_code);

} code_block
{
$1.label_end = newLabel(program);
gen_bt_instruction (program, $1.label_end, 0);
} UNLESS
{
             /* Set the label that identifies the condition evaluation */
             assignLabel(program, $1.label_condition);
}
exp
{
  if ($7.expression_type == IMMEDIATE)
                gen_load_immediate(program, $7.value);
  else
               gen_andb_instruction(program, $7.value
, $7.value, $7.value, CG_DIRECT_ALL);

  /* If the expression is FALSE, jump back to
   * the execution of the code block */
  gen_beq_instruction (program, $1.label_code, 0);

             /* Label identifying the end of the construct */
  assignLabel(program, $1.label_end);
}
;
```

4. Given the following code snipped:

```
b = c || d && e | f
```

Write down the syntactic tree generated during the parsing with the Bison grammar described in Acse.y *starting from the* `assign_statement` *nonterminal*. (5 points)



5. (Bonus) Is it possible to produce optimized code for the `eval-unless` construct in case the value of the condition is already known at compile time (i.e. it is constant)? Explain your answer. (3 points)

   In order to produce optimized code it should be possible to disable the generation of the code block (in case at compile time the condition is known to be true) or to generate the code block but not the condition computation (in case the condition is known to be true). Unfortunately, in this construct, the condition appears after the code block, therefore it is evaluated after the code of the code block has already been generated, and there is no way to optimize it with a single parsing pass.