Formal Languages and Compilers
Prof. Breveglieri, Morzenti, Agosta
Written exam: laboratory question 18/01/2022

Time: 60 minutes. Textbooks and notes can be used. Pencil writing is allowed.
Important: Write your name on any additional sheet.

SURNAME (Cognome): .....

NAME (Nome): .....

Matricola: ..... or Person Code: .....

Instructor: ☐ Prof. Breveglieri ☐ Prof. Morzenti ☐ Prof. Agosta

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 analyser (`flex` input) and the syntactic analyser (`bison` input) and any other source file required to extend the `Lance` language with the `converge` control statement.

The `converge` statement is a loop that, at each iteration, checks if the value of a specified variable did not change since the last execution of its body. When the variable is unchanged, the loop stops. The implementation must raise a syntax error if the specified variable identifier is an array rather than a scalar variable. Additionally, `converge` statements **must** be nestable.

The following code snippet exemplifies how the statement operates. In the first example, the initial value of the variable to be converged (variable a) is 31. The values of a are 10, 3, 1 and 0 after the first, second, third and fourth iteration respectively. After the fifth iteration, the variable a is still equal to 0 (the same as at the end of fourth iteration). Hence, the loop stops (the sixth iteration is not executed). In the second example, the initial value of a is 0. Since the loop body does not change the value of a (hence, after the first iteration it is still 0), the loop stops immediately.

```
int a, b;
a = 31;

converge a {
  a = a / 3;
  write(a);
}
// Iterates 5 times, and writes:
//  10, 3, 1, 0, 0

converge a {
  write(a);
}
// Iterates 1 time, and writes:
//  0
```

1. Define the tokens (and the related declarations in **Acse.lex** and **Acse.y**). (3 points)
2. Define the syntactic rules or the modifications required to the existing ones. (4 points)
3. Define the semantic actions needed to implement the required functionality. (18 points)

Solution

The solution is shown below in *diff* format. All lines that begin with '+' were added, while the lines that begin with '−' were removed. **It is not required and it is not encouraged to provide a solution in *diff* format to get the maximum grade.**

```
diff --git a/acse/Acse.lex b/acse/Acse.lex
index 35f9b73..2921952 100644
--- a/acse/Acse.lex
+++ b/acse/Acse.lex
@@ -93,6 +93,7 @@ ID       [a-zA-Z_][a-zA-Z0-9_]*
 "return"          { return RETURN; }
 "read"            { return READ; }
 "write"           { return WRITE; }
+"converge"        { return CONVERGE; }

 {ID}              { yylval.svalue=strdup(yytext); return IDENTIFIER; }
 {DIGIT}+          { yylval.intval = atoi( yytext );
diff --git a/acse/Acse.y b/acse/Acse.y
index 0636dc6..691d1df 100644
--- a/acse/Acse.y
+++ b/acse/Acse.y
@@ -109,6 +109,7 @@ extern void yyerror(const char*);
    t_list *list;
    t_axe_label *label;
    t_while_statement while_stmt;
+   t_converge_statement converge_stmt;
 }
 /*============================================================================
                                  TOKENS
@@ -133,6 +134,7 @@ extern void yyerror(const char*);
 %token <intval> TYPE
 %token <svalue> IDENTIFIER
 %token <intval> NUMBER
+%token <converge_stmt> CONVERGE

 %type <expr> exp
 %type <decl> declaration
@@ -254,12 +256,43 @@ control_statement : if_statement         { /* does nothing */ }
             | while_statement            { /* does nothing */ }
             | do_while_statement SEMI    { /* does nothing */ }
             | return_statement SEMI      { /* does nothing */ }
+            | converge_statement
 ;

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

+converge_statement: CONVERGE IDENTIFIER
+   {
+      /* Check that the variable is indeed a scalar */
+      t_axe_variable *v_var = getVariable(program, $2);
+      if (!v_var || v_var->isArray) {
+         yyerror("error: converge only works on scalars");
+      }
+
+      /* Reserve a register that will buffer the old value of the variable */
+      $1.r_oldvalue = getNewRegister(program);
+      /* Generate a label that points to the body of the loop */
+      $1.l_loop = assignNewLabel(program);
```

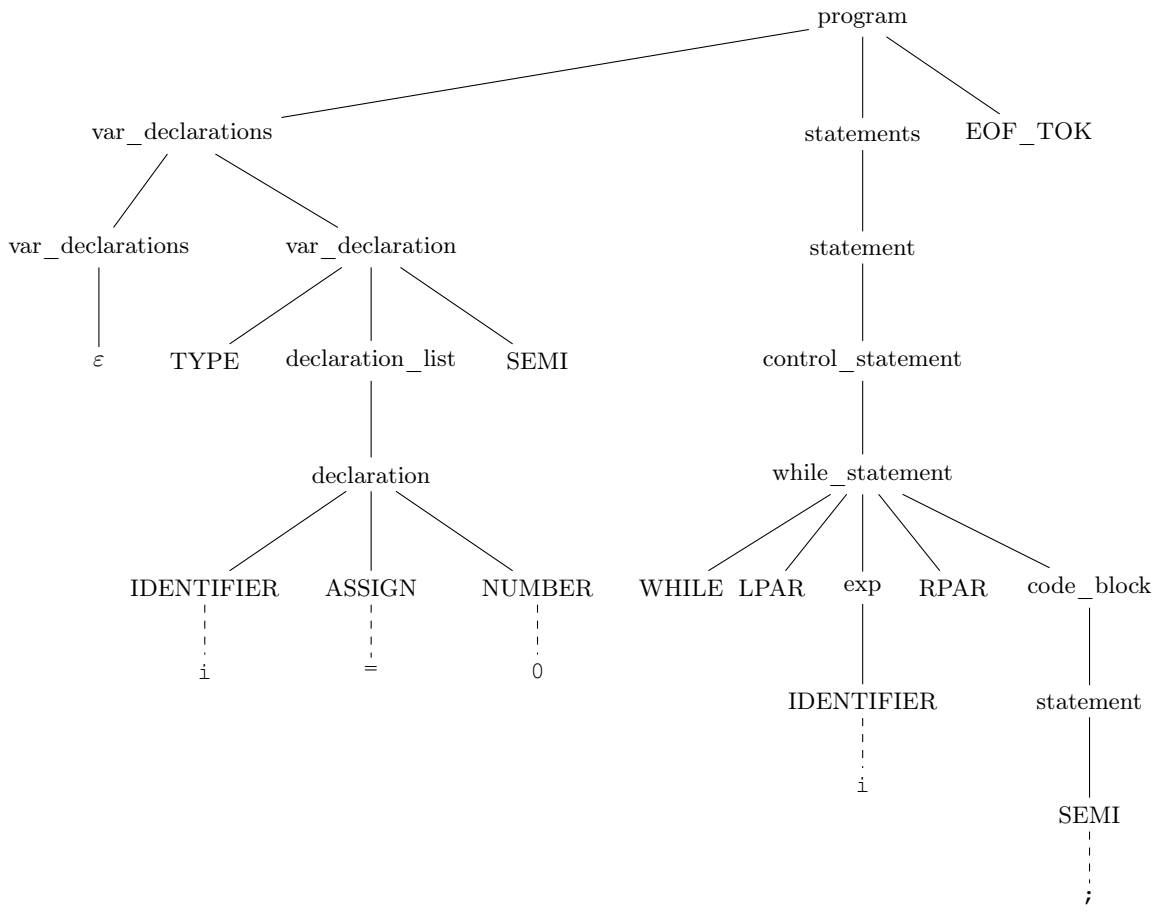
```
+
+      /* Generate code that, just before each execution of the loop body,
+       * saves the value of the variable in the register we reserved earlier */
+      int r_var = get_symbol_location(program, $2, 0);
+      gen_add_instruction(program, $1.r_oldvalue, REG_0, r_var, CG_DIRECT_ALL);
+   }
+   code_block
+   {
+      /* Generate a subtraction that updates the processor flags for checking
+       * if the variable's current value is equal to its old value */
+      int r_var = get_symbol_location(program, $2, 0);
+      gen_sub_instruction(program, REG_0, $1.r_oldvalue, r_var, CG_DIRECT_ALL);
+      /* Generate a branch that continues the loop if the variable's current
+       * value is different than its previous one. */
+      gen_bne_instruction(program, $1.l_loop, 0);
+   }
+;
+
 assign_statement : IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp
             {
                /* Notify to 'program' that the value $6
diff --git a/acse/axe_struct.h b/acse/axe_struct.h
index 00cb86f..99d48d0 100644
--- a/acse/axe_struct.h
+++ b/acse/axe_struct.h
@@ -110,6 +110,11 @@ typedef struct t_while_statement
                                      * that follows the while construct */
 } t_while_statement;

+typedef struct t_converge_statement {
+   int r_oldvalue;
+   t_axe_label *l_loop;
+} t_converge_statement;
+
 /* create a label */
 extern t_axe_label *alloc_label(int value);

diff --git a/tests/converge/converge.src b/tests/converge/converge.src
new file mode 100644
index 0000000..8906873
--- /dev/null
+++ b/tests/converge/converge.src
@@ -0,0 +1,11 @@
+int a, b;
+a = 31;
+
+converge a {
+  a = a / 3;
+  write(a);
+}
+
+converge a {
+  write(a);
+}
```

4. Given the following `Lance` code snippet:

```
int i=0;
while(i);
```

write down the syntactic tree generated during the parsing with the Bison grammar described in Acse.y *starting from the axiom.* (5 points)

Solution



5. (**Bonus**) Consider the case in which the `converge` statement is able to handle arbitrary expressions instead of variables, as in the following example. Describe at least one case in which it is possible to check at compile time if the statement will loop endlessly, and how to detect it.

```
int a, b, c;

a = 10;
b = -5;
converge (a - b) {
  c = (a + b) / 2;
  a = (a + c) / 2;
  b = (b + c) / 2;
  write(a - b);
}
// Iterates 5 times, and writes:
//  7, 3, 1, 0, 0
```

Solution

Due to a typographical error, the question asked whether it was possible to check at compile time if the statement will loop **endlessly** rather than **once**.

For the question as it appeared in the text as printed, an acceptable answer would be as follows. When the expression is of type REGISTER and grows monotonically at each iteration, the statement loops endlessly. In some cases, this situation is detectable by specific code analysis techniques (such as *scalar evolution*). As code analysis is not a topic of the course, answers that do not describe how to detect the condition were nonetheless accepted.

For the question as it was intended, the answer is as follows. When the expression is of type IMMEDIATE, then it is certain that its value will not change at each iteration. This causes the loop to only execute once.