Formal Languages and Compilers
Prof. Breveglieri, Morzenti, Agosta
Written exam: laboratory question 18/01/2022[1]

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] The text and solution to this exam have been adapted to ACSE 2.0 from its initial formulation.

1. Define the tokens (and the related declarations in **scanner.l** and **parser.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/parser.h b/acse/parser.h
--- a/acse/parser.h
+++ b/acse/parser.h
@@ -26,6 +26,11 @@ typedef struct {
    t_label *lExit; ///< Label to the first instruction after the loop.
 } t_whileStmt;

+typedef struct {
+  t_regID rLastValue; ///< Register containing the previous value of the var.
+  t_label *lLoop;     ///< Label to the beginning of the loop.
+} t_convergeStmt;
+
 /**
  * @}
  */
diff --git a/acse/parser.y b/acse/parser.y
--- a/acse/parser.y
+++ b/acse/parser.y
@@ -50,6 +50,7 @@ void yyerror(const char *msg)
    t_label *label;
    t_ifStmt ifStmt;
    t_whileStmt whileStmt;
+  t_convergeStmt convergeStmt;
 }

 /*
@@ -77,6 +78,7 @@ void yyerror(const char *msg)
 %token <label> DO
 %token <string> IDENTIFIER
 %token <integer> NUMBER
+%token <convergeStmt> CONVERGE

 /*
  * Non-terminal symbol semantic value type declarations
@@ -182,6 +184,7 @@ statement
    | return_statement SEMI
    | read_statement SEMI
    | write_statement SEMI
+  | converge_statement
    | SEMI
 ;

@@ -306,6 +309,29 @@ write_statement
    }
 ;

+converge_statement
+  : CONVERGE var_id
+  {
+    // Reserve a register that will buffer the old value of the variable
+    $1.rLastValue = getNewRegister(program);
+    // Generate a label that points to the body of the loop
+    $1.lLoop = createLabel(program);
+    assignLabel(program, $1.lLoop);
+
```

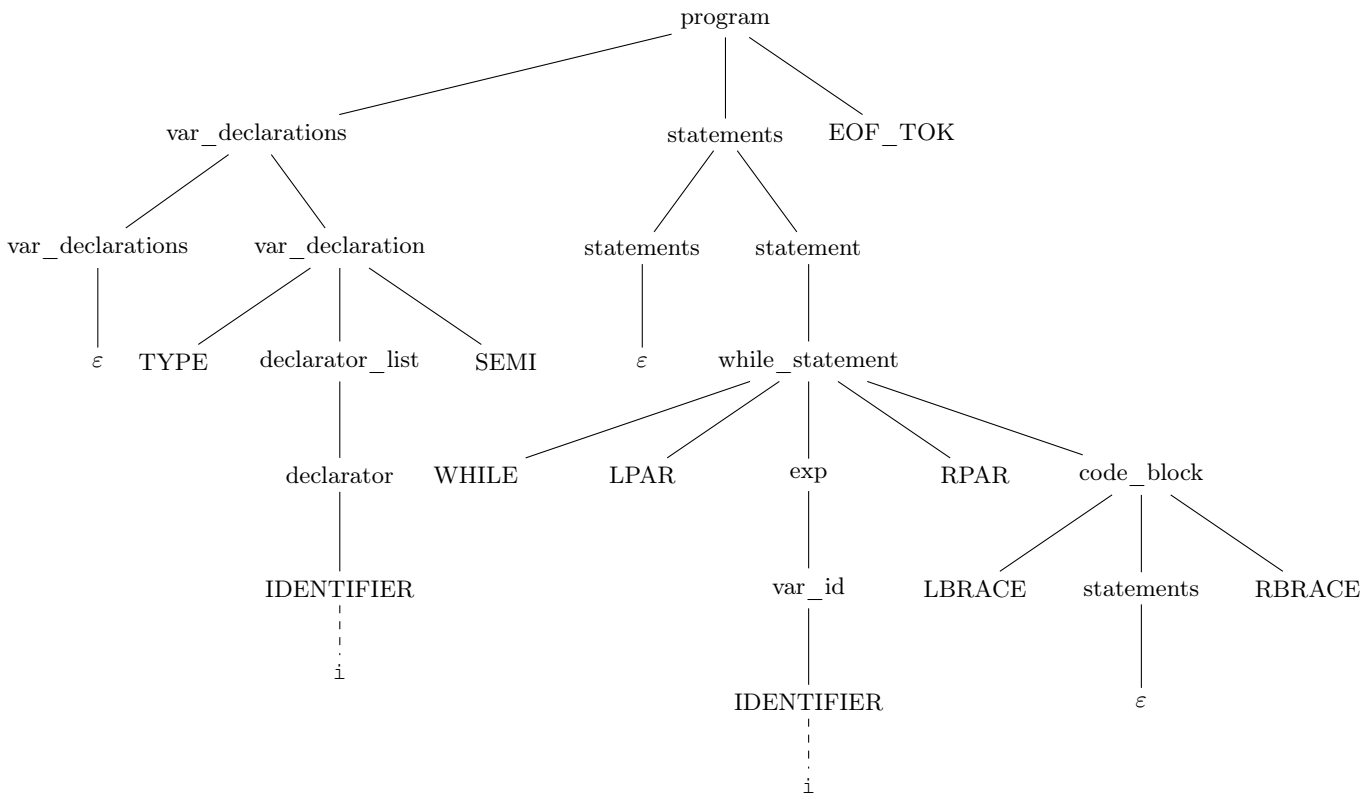
```
+    // Generate code that, just before each execution of the loop body,
+    // saves the value of the variable in the register we reserved earlier
+    t_regID rVar = genLoadVariable(program, $2);
+    genADD(program, $1.rLastValue, REG_0, rVar);
+  }
+  code_block
+  {
+    // Generate a branch that continues the loop if the variable's current
+    // value is different than its previous one.
+    t_regID rVar = genLoadVariable(program, $2);
+    genBNE(program, rVar, $1.rLastValue, $1.lLoop);
+  }
+;
+
 /* The exp rule represents the syntax of expressions. The semantic value of
  * the rule is the register ID that will contain the value of the expression
  * at runtime.
diff --git a/acse/scanner.l b/acse/scanner.l
--- a/acse/scanner.l
+++ b/acse/scanner.l
@@ -81,6 +81,7 @@ ID                          [a-zA-Z_][a-zA-Z0-9_]*
 "return"                  { return RETURN; }
 "read"                    { return READ; }
 "write"                   { return WRITE; }
+"converge"                { return CONVERGE; }

 {ID}                      {
                             yylval.string = strdup(yytext);
diff --git a/tests/converge/converge.src b/tests/converge/converge.src
--- /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;
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**) Briefly explain in plain English words how to modify the solution given to the questions about the `repeat_exp` statement in order to also support the case in which the argument of the statement is an arbitrary expression, as in the following example.

```
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
```