Formal Languages and Compilers
Prof. Breveglieri, Morzenti, Agosta
Written exam: laboratory question 13/06/2022[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 **`zip`** statement. The `zip` statement takes two source arrays, and copies their elements to a destination array in pairs – that is, by alternating elements from the first array and elements from the second.

The source arrays may be of different lengths, or shorter than required to fill the entire destination array. In these cases, the number of pairs of values copied is determined by the length of the shortest input array. The rest of the destination array is left unchanged. In alternative, the destination array may be shorter than required to contain all the elements from the source arrays. In that case the `zip` statement stops as soon as the last element of the destination array is written. If any of the identifiers does not refer to an array, a syntax error is generated and compilation is stopped.

The syntax and operation of the `zip` statement is shown in the example below. In the first example, the c array is filled with the contents of a and b alternatively, in the order in which they appear in the syntax (first a, then b). The a array is shorter than b, (5 elements vs. 6), so the length of a determines the number of pairs to be copied (5 pairs). The number of elements copied is 10 (5 from a, 5 from b) and the rest of the destination array c is left unchanged. In the second example, a is filled with elements from b and c alternatively. The number of pairs to be copied is determined by the shortest array (b) but the a array is shorter than required. As a result only 2 pairs are copied fully, and the last pair is copied only half-way.

```
int a[5], b[6], c[12];
/* a == [1, 2, 3, 4, 5] */
/* b == [10, 20, 30, 40, 50, 60] */
/* c == [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] */

c = zip(a, b);
/* c == [1, 10, 2, 20, 3, 30, 4, 40, 5, 50, 0, 0] */

a = zip(b, c);
/* a == [10, 1, 20, 10, 30] */
```

[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**). (1 point)
2. Define the syntactic rules or the modifications required to the existing ones. (2 points)
3. Define the semantic actions needed to implement the required functionality. (22 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.y b/acse/parser.y
index 6e9c139..4c0e94c 100644
--- a/acse/parser.y
+++ b/acse/parser.y
@@ -70,6 +70,7 @@ void yyerror(const char *msg)
 %token TYPE
 %token RETURN
 %token READ WRITE ELSE
+%token ZIP

 // These are the tokens with a semantic value.
 %token <ifStmt> IF
@@ -182,6 +183,7 @@ statement
   | return_statement SEMI
   | read_statement SEMI
   | write_statement SEMI
+  | zip_statement SEMI
   | SEMI
 ;

@@ -306,6 +308,65 @@ write_statement
   }
 ;

+zip_statement
+  : var_id ASSIGN ZIP LPAR var_id COMMA var_id RPAR
+  {
+    // Check if all three input/output arguments are in fact arrays
+    if (!isArray($1) || !isArray($5) || !isArray($7)) {
+      yyerror("at least one source/destination to zip is not an array!");
+      YYERROR;
+    }
+
+    // Compute the number of pairs of values to copy to the destination array
+    // and load it into a register for later
+    int nPairs = $5->arraySize;
+    if (nPairs > $7->arraySize)
+      nPairs = $7->arraySize;
+    t_regID rNPairs = getNewRegister(program);
+    genLI(program, rNPairs, nPairs);
+    // Load the destination array size in a register for later
+    t_regID rDstSz = getNewRegister(program);
+    genLI(program, rDstSz, $1->arraySize);
+
+    // Reserve two registers used for keeping track of the current index
+    // in the source and destination arrays. Generate code for initializing
+    // both registers to zero.
+    t_regID rSrcI = getNewRegister(program);
+    t_regID rDstI = getNewRegister(program);
+    genLI(program, rSrcI, 0);
+    genLI(program, rDstI, 0);
+
+    // Generate a label at the beginning of the loop which copies the values
+    // at runtime.
+    t_label *lLoop = createLabel(program);
```

```
+    assignLabel(program, lLoop);
+    // Reserve a label for exiting the loop
+    t_label *lExit = createLabel(program);
+
+    // Now we need to generate the code of the body of the loop.
+    // First we generate a copy from the first source array to the destination
+    t_regID rTemp = genLoadArrayElement(program, $5, rSrcI);
+    genStoreRegisterToArrayElement(program, $1, rDstI, rTemp);
+    genADDI(program, rDstI, rDstI, 1);
+    // Generate a branch out of the loop if we are at the end of the dest. array
+    genBGE(program, rDstI, rDstSz, lExit);
+    // Now generate a copy from the second source array to the destination
+    rTemp = genLoadArrayElement(program, $7, rSrcI);
+    genStoreRegisterToArrayElement(program, $1, rDstI, rTemp);
+    genADDI(program, rDstI, rDstI, 1);
+    // Generate a branch out of the loop if we are at the end of the dest. array
+    genBGE(program, rDstI, rDstSz, lExit);
+
+    // Generate code to increment the index in the source arrays and continue
+    // the loop if we are not at the end
+    genADDI(program, rSrcI, rSrcI, 1);
+    genBLT(program, rSrcI, rNPairs, lLoop);
+
+    // Assign the label for exiting the loop
+    assignLabel(program, lExit);
+  }
+;
+
 /* 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
index 3d35cd5..b082310 100644
--- 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; }
+"zip"                   { return ZIP; }

 {ID}                    {
                           yylval.string = strdup(yytext);
diff --git a/tests/zip/zip.src b/tests/zip/zip.src
new file mode 100644
index 0000000..174cf57
--- /dev/null
+++ b/tests/zip/zip.src
@@ -0,0 +1,32 @@
+int a[5], b[6], c[12];
+int i;
+
+i = 0;
+while (i < 5) {
+  a[i] = i + 1;
+  i = i + 1;
+}
+i = 0;
+while (i < 6) {
+  b[i] = (i + 1) * 10;
+  i = i + 1;
+}
+i = 0;
+while (i < 12) {
+  c[i] = 0;
+  i = i + 1;
+}
+
+c = zip(a, b);
```

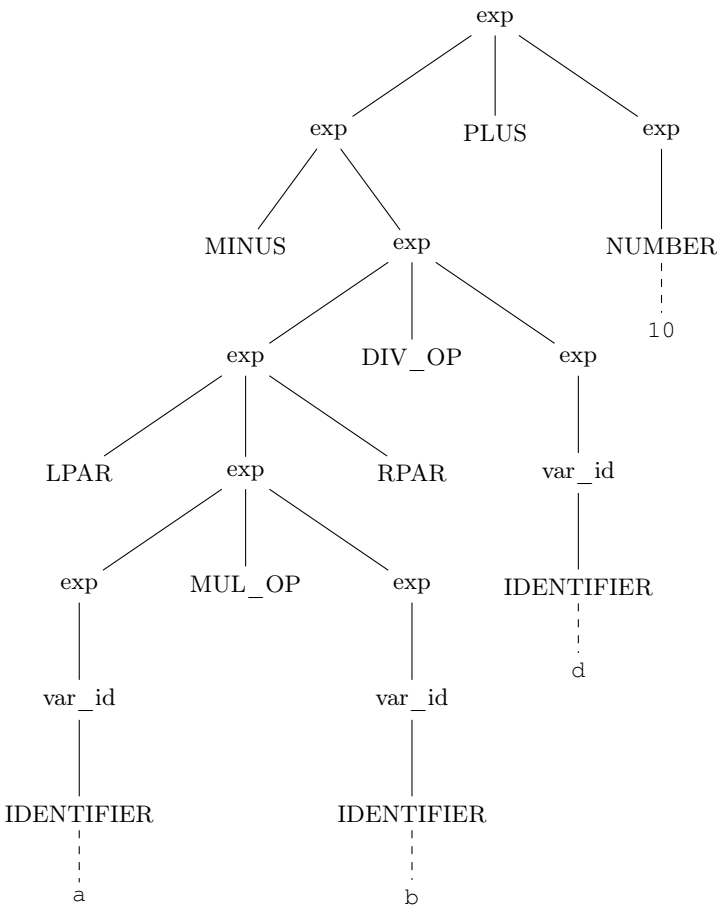
```
+i = 0;
+while (i < 12) {
+  write(c[i]);
+  i = i + 1;
+}
+
+a = zip(b, c);
+i = 0;
+while (i < 5) {
+  write(a[i]);
+  i = i + 1;
+}
```

4. Given the following `Lance` code snippet:

```
-(a * b) / d + 10
```

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

Solution



5. (**Bonus**) Describe in plain English words how the given implementation of the `zip` statement can be extended such that, when the destination array has not been declared yet, it is instantiated automatically.

   In particular, describe which ACSE utility function needs to be called in order to instantiate the array, and how the size of the array is derived.