Formal Languages and Compilers
Prof. Breveglieri, Morzenti, Agosta
Written exam: laboratory question 04/07/2024[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: ....

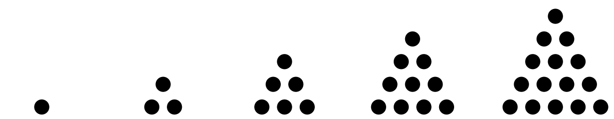
Instructor: ☐ Prof. Breveglieri ☐ Prof. Morzenti ☐ Prof. Agosta

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

Triangular numbers are integers that count objects arranged in a triangle. The value of the nth triangular number is defined by the following equation:

$$T_n = \frac{n(n+1)}{2}$$

The zero-th triangular number is defined equal to zero ($T_0 = 0$) and there are no preceding triangular numbers (in other words, n cannot be less than zero). Examples of triangular numbers are shown in the following picture.



$T_1 = 1$ $T_2 = 3$ $T_3 = 6$ $T_4 = 10$ $T_5 = 15$

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 ability to compute the nth triangular number using a new **expression operator**, called **`tri`**, with the following syntax:

`tri` `(`⟨*exp.*⟩`)`

The only **argument** to the `tri` operator is an **arbitrary expression** which specifies the index of the triangular number to compute. For example, `tri(5)` will compute the value 15. The operator is also **applicable to negative numbers**, but it always returns **zero**. For example, `tri(-10)` has the value 0.

Example

The following program shows an example of use of the `tri` operator.

```
int i = 0;
// Print T_1, T_2, ... T_10
while (i < 10) {
  write(tri(i + 1));
  i = i + 1;
}
```

[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.y b/acse/parser.y
index 6e9c139..334ad47 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 TRI

 // These are the tokens with a semantic value.
 %token <ifStmt> IF
@@ -438,6 +439,23 @@ exp
     $$ = getNewRegister(program);
     genOR(program, $$, rNormalizedOp1, rNormalizedOp2);
   }
+  | TRI LPAR exp RPAR
+  {
+    // Reserve a register for the result, and generate an instruction to
+    // initialize it to zero.
+    $$ = getNewRegister(program);
+    genLI(program, $$, 0);
+    // Generate a branch that skips the computation if the expression is
+    // negative or zero
+    t_label *lNegative = createLabel(program);
+    genBLE(program, $3, REG_0, lNegative);
+    // Generate the computation of the triangular number
+    genADDI(program, $$, $3, 1);  // $$ = n + 1
+    genMUL(program, $$, $$, $3);  // $$ = n * (n + 1)
+    genDIVI(program, $$, $$, 2);  // $$ = n * (n + 1) / 2
+    // Assign the label to the end of the code
+    assignLabel(program, lNegative);
+  }
 ;

 var_id
diff --git a/acse/scanner.l b/acse/scanner.l
index 3d35cd5..2d1b11d 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; }
+"tri"                     { return TRI; }

 {ID}                      {
                             yylval.string = strdup(yytext);
diff --git a/tests/tri/tri.src b/tests/tri/tri.src
new file mode 100644
index 0000000..1d0af52
--- /dev/null
+++ b/tests/tri/tri.src
@@ -0,0 +1,7 @@
+int i;
```

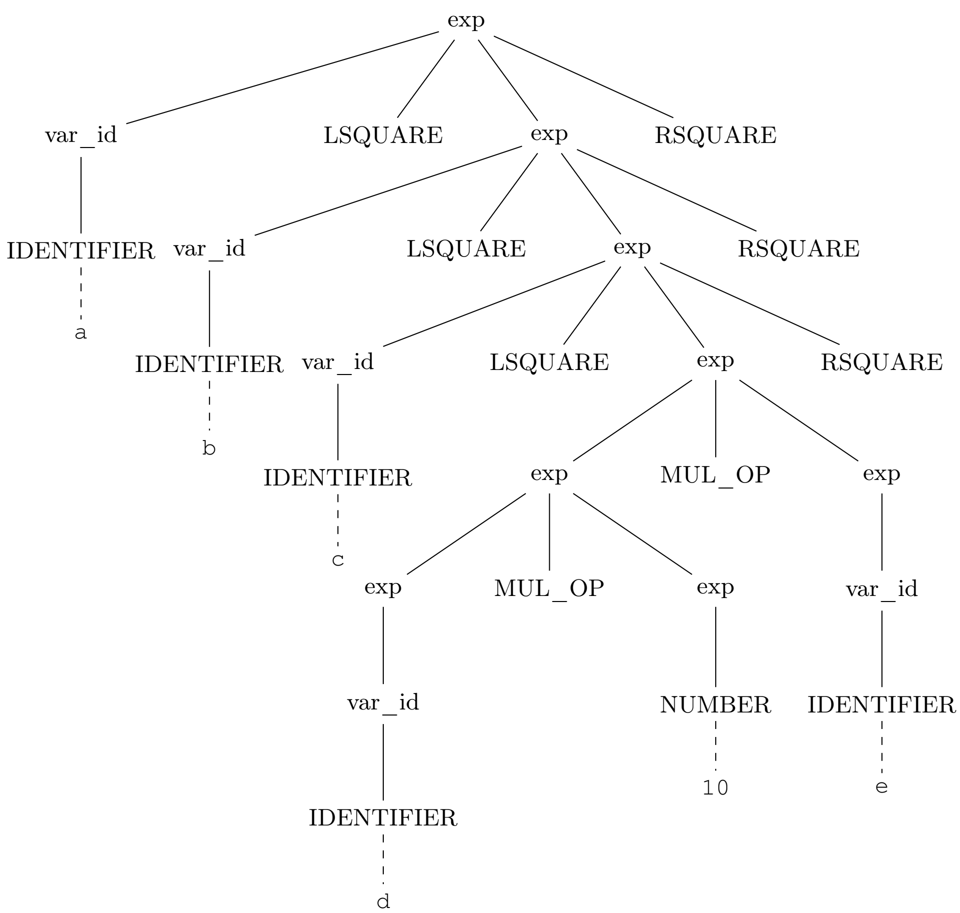
```
+
+i = 0;
+while (i < 15) {
+  write(tri(i - 5));
+  i = i + 1;
+}
```

4. Given the following `Lance` code snippet:

```
a[b[c[d*10*e]]]
```

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**) Briefly discuss in plain English words why it is **not necessary** to provide precedence and associativity declarations for the `tri` operator.

   Additionally, show an alternative syntax for `tri` that **does require** the declaration of its precedence and associativity with respect to other expression operators.

Solution

The syntax of the operator `tri` as required by the text allows for an unambiguous grammar without defining precedence and associativity rules; hence they are not necessary. Specifically, the mandatory pair of parentheses around the expression argument forces it to have a single derivation tree.

On the other hand, if the operator did *not* require the parentheses around the expression, then it would have been necessary to define the precedence rules for `tri` to enforce the non-ambiguity of the grammar. For example, without precedence rules, the string

`tri` `1 + 2`

allows both of the following parse trees: