Formal Languages and Compilers
Proff. Breveglieri, Crespi Reghizzi, Morzenti
Written exam[1]: laboratory question
10/07/2012

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 `continue` statement:

```
int sum=0;
int a=0;

while(a<20) {
  a = a+1;

  if ((a/2)*2 != a) {
    /* If a is odd */
    continue;
  }

  sum = sum + a;
}

write(sum);
```

```
int sum=0;
int a=0;

do {
  a = a+1;

  if ((a/2)*2 != a) {
    /* If a is odd */
    continue;
  }

  sum = sum + a;
} while (a<20);

write(sum);
```

The semantics of `continue` is the same as in the C programming language: when `continue` is executed, all the remaining statements of the current iteration of the innermost loop are ignored, the condition is checked again and the next iteration of the same loop starts.

The `continue` statement has to work both in `while` loops and `do-while` loops.

If `continue` is used outside any loop, the compiler should print an error message and exit.

Explicit any other assumption you made to implement the support for the `continue` construct.

[1] Time 60'. 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)

   The solution is in the attached patch.

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

   The solution is in the attached patch.

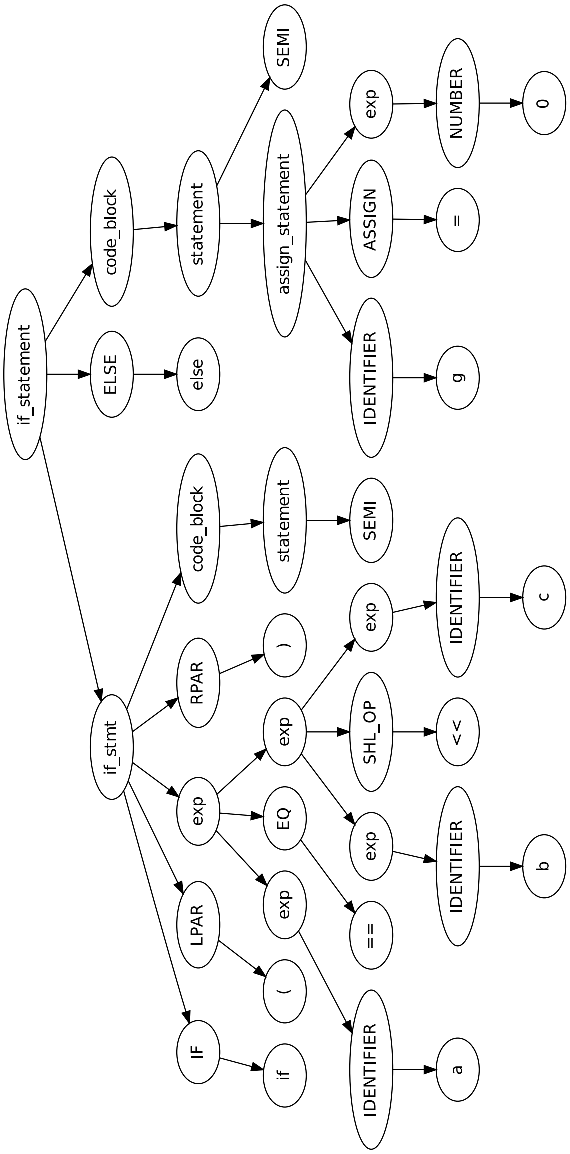
3. Define the semantic actions needed to implement the required functionality. (17 points)

   The solution is in the attached patch.

4. Given the following code snipped:

   ```
   if(a == b << c) ; else g = 0;
   ```

   Write down the syntactic tree generated during the parsing with the Bison grammar described in Acse.y *starting from the* `if_statement` *nonterminal.* (10 points)



5. (Bonus) Implement the support for the variable-nesting `continue` statement defined as follows.

Example 1

```
int a,i;

a = 0;
i = 0;
while (a < 20) /*Cycle 1*/
{
  a = a + 1;
  i = -1;
  while(i < a) {/*Cycle 2*/
    i = i + 1;
    if ((i/2)*2 != i) {
      continue 0;
    }
    write(i);
  }
  write(a);
}
```

Example 2

```
int a,i;

a = 0;
i = 0;
while (a < 20) /*Cycle 1*/
{
  a = a + 1;
  i = -1;
  while(i < a) {/*Cycle 2*/
    i = i + 1;
    if ((i/2)*2 != i) {
      continue 1;
    }
    write(i);
  }
  write(a);
}
```

When `continue` is invoked with parameter 0 (as in Example 1), it is equivalent to the usual `continue`, so the execution skips to the next iteration of Cycle 2.

When `continue` is invoked with parameter 1 (as in Example 2), it jumps to the cycle 1 step above in the loop nest (therefore, to the next iteration of Cycle 1).

In general,

`continue K`

jumps to the next iteration of the loop `K` levels above in the loop nest.

NB: Consider `K` as an immediate value.

(3 points)

Change the `continue_statement` rule so that it becomes:

```
continue_statement : CONTINUE NUMBER {

  if ($2 >= getLength(loop_nest)) {
    printMessage("This continue statement is not inside enough cycles!\n");
    exit(-1);
  }

  t_list *go_to = getElementAt(loop_nest, $2);
  t_axe_label *next_iter = go_to->data;
  gen_bt_instruction(program, next_iter, 0);
}
```