# Formal Languages and Compilers
# Proff. Breveglieri, Crespi Reghizzi, Morzenti
# Written exam[1]: laboratory question
# 27/06/2009

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 *use assignments as a part of an expression* with the following syntax (the same as C programming language):

```
int x, y, z, a[3];

x = y = z = a[0] = 1;
write( x = y + 3 );
z = (y = 2) * x;
x = z + y;
write( x );
```

Compiling this code snippet and executing it would result in a printout of the values “4” and “10”.

The solution must comply with the following specifications:

- The assignment operator has the lowest priority and is *right* associative.
- An assignment expression may appear in the grammar everywhere a nonterminal expression does (i.e., everywhere a nonterminal exp is placed).
- The left hand side of an assignment may be either a variable or an element of an array.
- The right hand side of an assignment may be any valid expression of the language, including an assignment.

Your modifications have to allow the Acse compiler to both correctly analyze the syntactical correctness of the aforementioned constructs and to generate a correct translation in the Mace assembly language.

Take care not to build an ambiguous grammar.

[1]Time 45’. Textbooks and notes can be used.
Pencil writing is allowed. Write your name on any additional sheet.

The answers given here are generally more verbose than those regarded as fully satisfactory during the evaluation of a student's work.

1. Define the syntactic rules needed (or the modifications to the existing ones) in order to obtain the required functionalities, including possible directives for precedence and associativity. (7 points)

   As any expression can be an assignment, add two possible expansions to the *exp* non-terminal:

   ```
   exp : ...
       |    IDENTIFIER ASSIGN exp
       |    IDENTIFIER LSQUARE exp RSQUARE ASSIGN
   exp
   ;
   ```

   There is already a statement setting the correct precedence for the assignment:

   `%left ASSIGN`

   The grammar specified above is not ambiguous, as any assignment expression has an identifier as the first operand, so there is no need to modify the associativeness[2]. Modifying the above statement in

   `%right ASSIGN`

   is okay, though.

   An alternative solution is to make use of the non-terminal *assign_statement*:

   ```
   exp : ...
       |    assign_statement
   ;
   ```

   Again, the grammar is not ambiguous, for the same reason as above.

[2]If the rule were "*exp* : *exp* ASSIGN *exp*" the resulting grammar would be ambiguous, and the associativeness must be specified correctly.

2. Define the semantic actions needed (or the modifications to the existing ones) in order to obtain the required functionalities. (8 points)

   The semantic actions for the new rules are basically a copy of the actions for *assign_statement*, with the addition of the code to propagate the value of the expression:

```
exp : IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp
    {
        storeArrayElement(program, $1, $3, $6);
        free($1);
        $$ = $6;
    }
 | IDENTIFIER ASSIGN exp
    {
        int location;
        t_axe_instruction *instr;
        location = get_symbol_location(program, $1, 0);
        if ($3.expression_type == IMMEDIATE)
           instr = gen_addi_instruction
              (program, location, REG_0, $3.value);
        else
           instr = gen_add_instruction
                 (program, location, REG_0, $3.value, CG_DIRECT_ALL);
        free($1);
        $$ = $3;
    }
;
```

The alternative solution requires to add a semantic attribute to the non-terminal *assign_statement*

```
%type <expr> assign_statement
```

and to perform the propagation of the value of the expression both in the new expansion for *exp* and in the actions for *assign_statement*:

```
assign_statement : IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp
    {
        ...
        $$ = $6;
    }
 | IDENTIFIER ASSIGN exp
    {
        ...
        $$ = $3;
    }
;

exp : assign_statement
    {
        $$ = $1;
    }
;
```

3. Given the following code snippet:

```
int a = 2;
int b;

b = a*5 + (a = a - 1) + a*2;
write( b )
```

(a) Write down the syntactic tree of the assignment statement on the third line, starting from the non-terminal `statements` according to the grammar specified before. (4 points)

(b) According to the grammar specified before, write down the evaluation order of the operations performed in the aforementioned statement and compute the final value of the variable `b`. (5 points)

(a) See figure 1

(b) In an LR parser the tree is built bottom-up starting from the left, in the order that is the reverse of the rightmost derivation. Therefore, the operations are evaluated in the following order:

| **Expression** | **Expr. value** | **Value of** a |
|---|---|---|
| a $\boxed{*}$ 5 | 10 | 2 |
| a $\boxed{-}$ 1 | 1 | 2 |
| a $\boxed{=}$ a - 1 | 1 | 1 |
| a $*$ 5 $\boxed{+}$ (a $=$ a - 1) | 6 | 1 |
| a $\boxed{*}$ 2 | 2 | 1 |
| a $*$ 5 $+$ (a $=$ a - 1) $\boxed{+}$ a $*$ 2 | 13 | 1 |

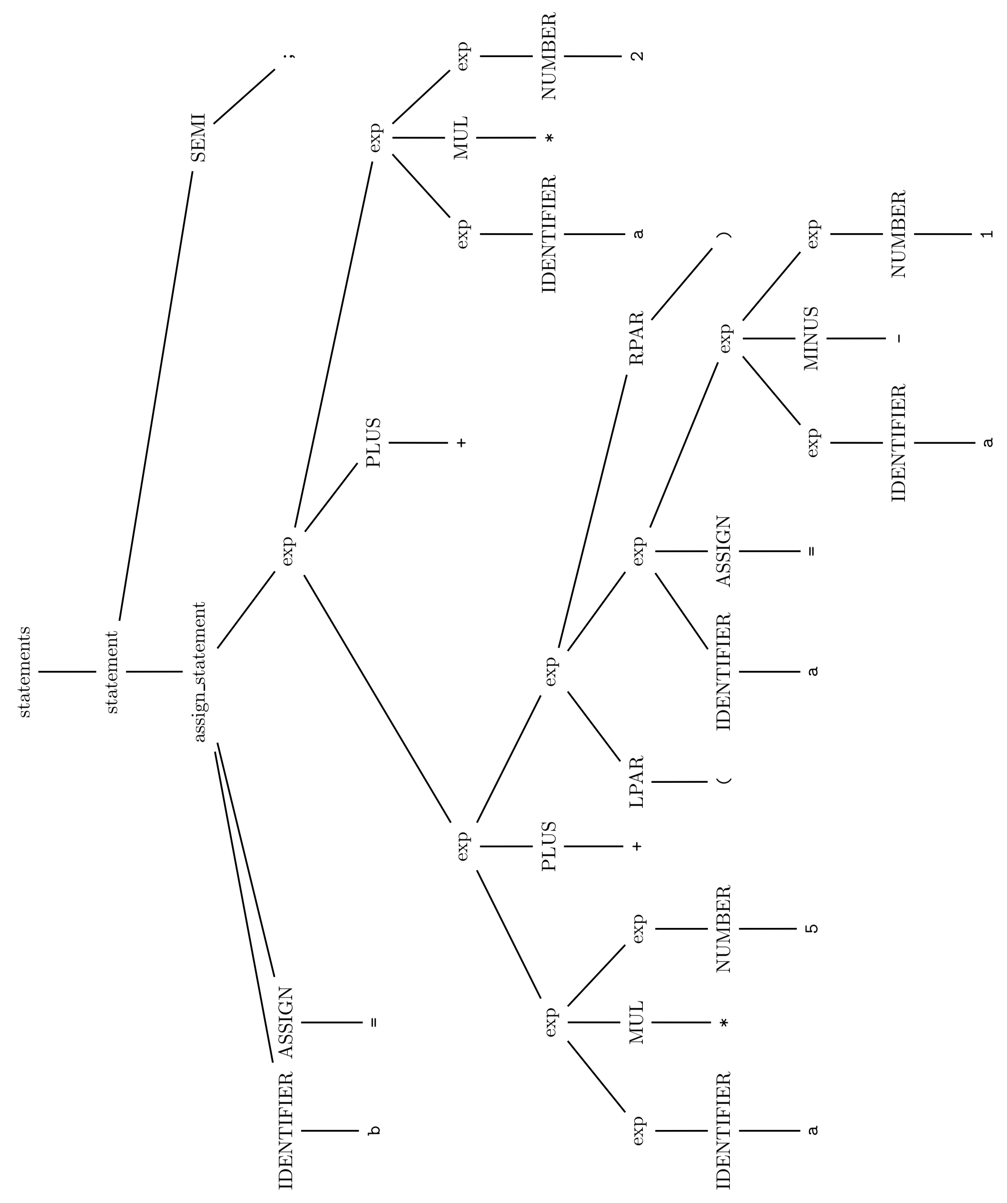

Figure 1: Syntactic tree of the statement `b = a*5 + (a = a - 1) + a*2;`

4. In the code snippet reported in the question 3 the value of `b` depends on the order in which the operands are evaluated. Since a typical programming language doesn't suffer from such issues, delineate which semantic check may be applied by the compiler in order to produce a warning in situations like the ones in the example. It is not required to write the actual code, but the semantic actions and the structures involved must be specified correctly. (9 points)

The problem happens whenever a variable is both used and assigned in the same expression (but it is okay to use a variable to compute the value to assign to the same variable, as in `a = a + 1`). Briefly, we need to keep track of the variables used and assigned in expressions, and raise a warning whenever the intersection between the two sets is not empty.

A way to do this is to add two fields to the `t_axe_expression` structure: `assigned` and `used`. They are both sets of identifiers.

In the rules

```
exp : IDENTIFIER
exp : NOT_OP IDENTIFIER
```

`used` is initialized with the identifier, while `assigned` is empty. In

```
exp : NUMBER
exp : NOT_OP NUMBER
```

both sets are empty.

In the rule

```
exp : IDENTIFIER ASSIGN exp {
    $$.assigned = $3.assigned ∪ { $1 }
    $$.used = $3.assigned \ { $1 }
}
```

the identifier is added to *assigned* and removed from `used`. The removal from `used` is needed to avoid issuing warning in case of expressions like **a = a + 1**, which are safe to use. A check is performed to make sure the assigned variable is not already present in the `assigned` set of the right-hand *exp*. This check discovers double assignments, as in `a = 1 + (a = 2)`.

For the rules of the type

```
exp : exp OPERATOR exp {
    $$.assigned = $1.assigned ∪ $3.assigned
    $$.used = $1.used ∪ $3.used
}
```

The value of `assigned` and `used` of the left-hand side *exp* is the union of values on the right-hand side. Two checks are performed in these rules:

(a) The intersection of `assigned` sets on the right side should be empty. This is to avoid double assignments as in `(a = 3) + (a = 4)`.

(b) The intersection of the `assigned` and `used` should be empty. This is to avoid the assignment and the use of the same variable in one expression, as in `a + (a = 3)`.

In rules where there is one *exp* in the right-hand side, the values of `assigned` and `used` are propagated unchanged.

The above solution does not address the problem related to the use and assignment of array elements. Array identifiers could be added to the `assigned` and `used` sets and handled in the same way as variables. Such a solution generates false positives, because it does not distinguish between individual elements of the array. This is unavoidable, as array could be index with variables, whose values is not known at compile time.