Formal Languages and Compilers
Proff. Breveglieri, Crespi Reghizzi, Morzenti
Written exam[1]: laboratory question
28/06/2008

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 a new iterative construct *computed goto* resembling the one in the following sample.

```
int a;
...
label3: ...
...
goto a in label1,label2,label3;
...
label1: ...
...
label2: ...
...
```

The semantic of the construct is the following one: if the runtime value of the control variable (`a` in the example) is a positive integer n , then the goto structure jumps to the n-th label after the keyword `in` (e.g.: if $a = 1$ the control flow jumps to "label1" in the aforementioned code). If the control variable assumes either a value less than 1, or a value greater than the number of labels, no alterations in the control flow are made and the program continues its execution.

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

In order to correctly solve the question we suggest you to use the `getNewRegister` function contained in `axe_engine.[h,c]`, which is able to reserve a free register for your purposes. The prototype of the function is the following one.

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

```
/* get a register still not used. This function returns
 * the ID of the register found */
int getNewRegister(t_program_infos *program);
```

The integer returned by the function represents, in a non ambiguous way, the register which has been allocated.

Please notice that the present solution is more verbose than what is needed to get the maximum grade.

As usual, there are many different possible ways to generate assembly code that complies with the specifications. The most straightforward one is the following:

`Lance`	Assembly
label3:	label3:
...	...
goto a **in**	
label1,	**SUBI** Ry Rx ♯1
	BEQ label1
mylabel,	**SUBI** Ry Rx ♯2
	BEQ mylabel
label3;	**SUBI** Ry Rx ♯3
	BEQ label3
...	...
label1:	label1:
...	...
mylabel:	mylabel:
...	...

where Rx is the register associated with variable `a` and Ry is a temporary register. The assembly code compares the value of the variable `a` with 1, 2, 3... (**SUBI** instructions), and it jumps to the correspondent label when the terms of a comparison are equal (i.e., a subtraction returns 0).

There are two points in the above code that need some attention: the generation of the constants 1, 2, 3... to be associated to the labels in the **goto** statement, and the correspondence between `Lance` and assembly labels. For the first point, the label list in the **goto** statement should be generated by the grammar with left recursion; a synthesized attribute can easily count the positions of the labels in the list. For the second point, we can make use of the symbol table already present in the `Acse` compiler; we add a new type of entries, *label*, which associates names of labels in the `Lance` language with assembly labels.

1. Define the tokens and the Acse.lex and Acse.y declarations needed to achieve the required functionality. (3 points)

   The new construct makes use of two new keywords, **goto** and **in**. The existing token IDENTIFIER can be used anywhere a label appears.

   In **Acse.lex** we add:

   ```
   "goto"        { return GOTO; }
   "in"          { return IN; }
   ```

   and in **Acse.y**:

   ```
   % token GOTO
   % token IN
   ```

2. Define the syntactic rules needed to achieve the required functionality. (8 points)

   A new possibility for the non-terminal *statement* generates labeled statements:

   statement : ...
   | *labeled_statement*
   ;

   labeled_statement : IDENTIFIER COLON statement ;

   In this way, a label has always a statement unambiguously associated to it.

   The *goto* statements are generated by a new possible expansion for the non-terminal *control_statement*:

   control_statement : ...
   | *goto_statement*
   ;

   goto_statement : GOTO IDENTIFIER IN *goto_list* SEMI ;

   goto_list : *goto_list* COMMA IDENTIFIER
   | IDENTIFIER
   ;

   The list of labels uses left recursion, so that the **SUBI** and **BEQ** instructions can be generated in the correct order.

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

   We have to define new functions to handle the insertion and the look-up of labels in the symbol table, similar to `set_new_variables` and `get_symbol_location`:

   ```
   t_axe_label *new_lance_label( t_program_infos *program,
           char *id );
   t_axe_label *get_lance_label( t_program_infos *program,
           char *id );
   ```

   `new_lance_label` creates a new label (by calling `reserveLabel`, add it as an entry of the new type *label* in the symbol table, and returns it. `get_lance_label` search a label entry in the symbol table by name and returns it; if it does not find any label with the given name, it returns `NULL`.

   (As an alternative, a modified version of `get_lance_label` could replace the two functions as they are described above: The modified `get_lance_label` also creates a new entry in the symbol table whenever it does not find the requested label in the table.)

   Some global definitions. We define the synthesized attribute used to count the labels in a **goto** statement:

   ```
   %union {
      ...
      int count;  // Counter for labels in goto statements
   }

   %type <count> goto_list
   ```

   We need also an inherited attribute to pass the register that contains the value of the goto variable (a, in the example) to the actions of the non-terminal *goto_list*, which are responsible for the generation of the **SUBI** instructions. In Bison, where there is no support for inherited attributes, we have to rely on global structures. In this case there is no recursion (a **goto** statement cannot contain other statements inside), so a simple global variable is enough:

   ```
   %{
       ...
       int goto_var_reg;  // Register associated with the
                          // variable in a goto statement
                          // (i.e., the register Rx in the
                          // translation scheme)
   %}
   ```

The semantic action for the *labeled_statement* rule sets the address of a label. We have to handle both the case when the label is already present in the symbol table (because of a previous **goto** statement) and the case when the label is new.

```
labeled_statement : IDENTIFIER COLON
    {
        t_axe_label *lab =
             get_lance_label( program, $1 );
        if (! lab)
            lab = new_lance_label( program, $1 );
        fixLabel( program, lab );
    }
    statement ;
```

The action for the *goto_statement* rule just sets the goto_var_reg variable.

```
goto_statement : GOTO IDENTIFIER
    {
        goto_var_reg = get_symbol_location( program,
            $2, 0 );
    }
    IN goto_list SEMI ;
```

The actions of the non-terminal *goto_list* count the labels present in the list, and use the value of the counter to generate the **SUBI** instructions with the correct arguments. The labels contained in a list may or may not be already present in the symbol; labels not yet present are created by calling new_lance_label (but they are not assigned by calling fixLabel; labels are just referenced when they occurs in a **goto** statement).

```
goto_list : goto_list COMMA IDENTIFIER
    {
        int reg;           // Temporary register
        t_axe_label *lab;  // Assembly label
        reg = getNewRegister( program );
        lab = get_lance_label( program, $3 );
        if (! lab)
            lab = new_lance_label( program, $3 );
        gen_subi_instruction( program, reg,
             goto_var_reg, $1 + 1 );
        gen_beq_instruction( program, lab, 0 );
        $$ = $1 + 1;       // Position in the label list
    }
```

```
| IDENTIFIER
   {
       int reg;           // Temporary register
       t_axe_label *lab;  // Assembly label
       reg = getNewRegister( program );
       lab = get_lance_label( program, $1 );
       if (! lab)
           lab = new_lance_label( program, $1 );
       gen_subi_instruction( program, reg,
            goto_var_reg, 1 );
       gen_beq_instruction( program, lab, 0 );
       $$ = 1;            // Position in the label list
   }
;
```

4. Define the semantic actions needed to raise an error if there are undeclared labels within the goto statement. (5 points)

   The above solution does not check that a label referenced in a **goto** statement is assigned somewhere in the program source, nor does it check that no label is declared twice. So, we have to keep track of which labels are not declared, and raise an error whenever a label is declared for the second time; also, if there is any undeclared label left at the end of the parsing, an error is to be raised.

   We add a new field `declared` to the entries of the symbol table; this field is initialized to 0 in calls to `new_lance_label`. The action of the non-terminal *labeled_statement* first checks the `declared` field, to ensure that the label is not already declared. Then, the action marks the label as declared by setting the `declared` field to 1.

   At the end of the parsing, i.e., at the beginning of action of the non-terminal *program*), we run through all the labels in the symbol table and check if there is any undeclared label.