Flex , Bison and the **ACSE** compiler suite

Marcello M. Bersani

LFC – Politecnico di Milano

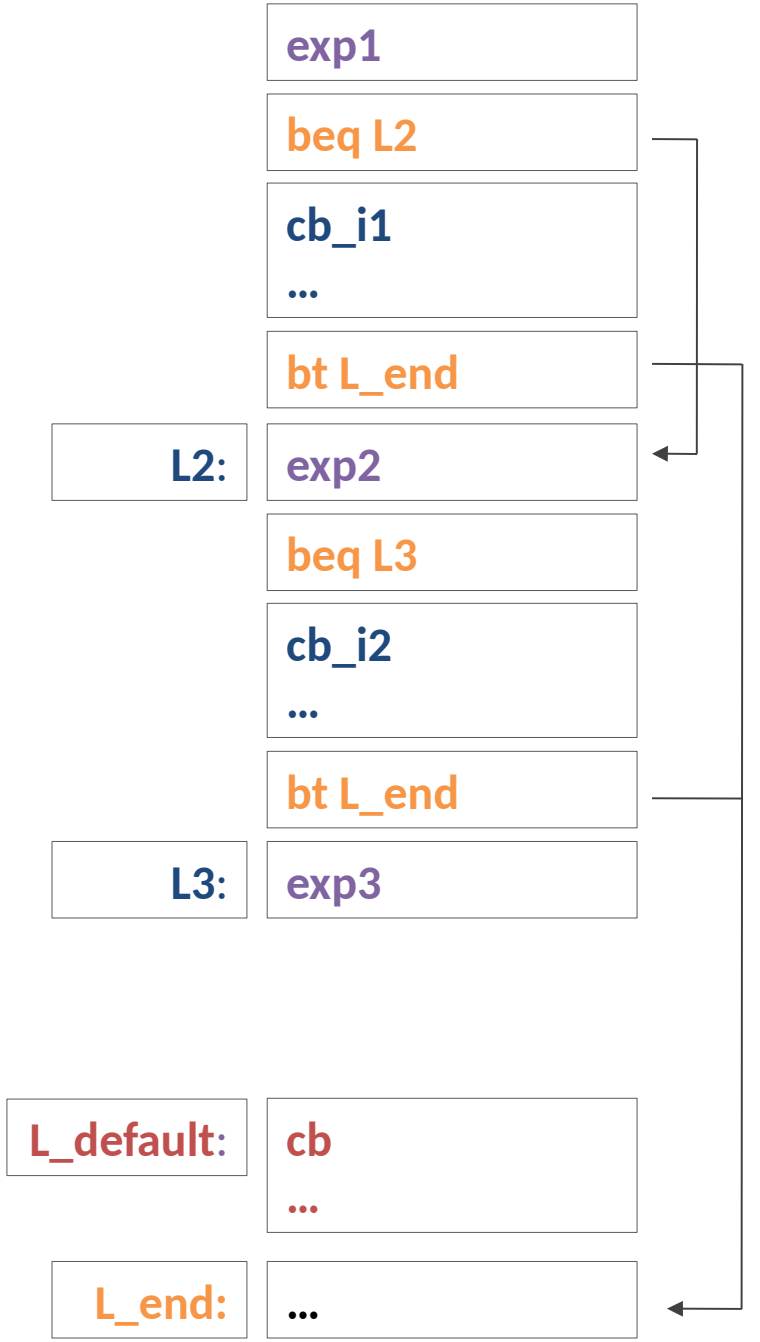
COND statement

- Define tokens, syntax/semantic rules translating a COND statement
  - Default
  - meet-first: first case having positive condition is executed

```
cond {
   case x: x=0;
   case y>0: y=y+x; x=1;
   case x+y: x=0; y=0;
   default: y=0;
}
```

COND statement

```
cond {
    case x: x=0;
    case y>0: y=y+x; x=1;
    case x+y: x=0; y=0;
    default: y=0;
}
```



COND data structure

- Two labels
  - L_end: defined immediately when COND is recognized
  - L_next: labels the next case block where to jump if the current exp is false

COND data structure

```
typedef struct
{
    t_axe_label *L_end;
    t_axe_label *L_next;
} t_cond_statement;
```

- Axe_struct.h

COND syntactic

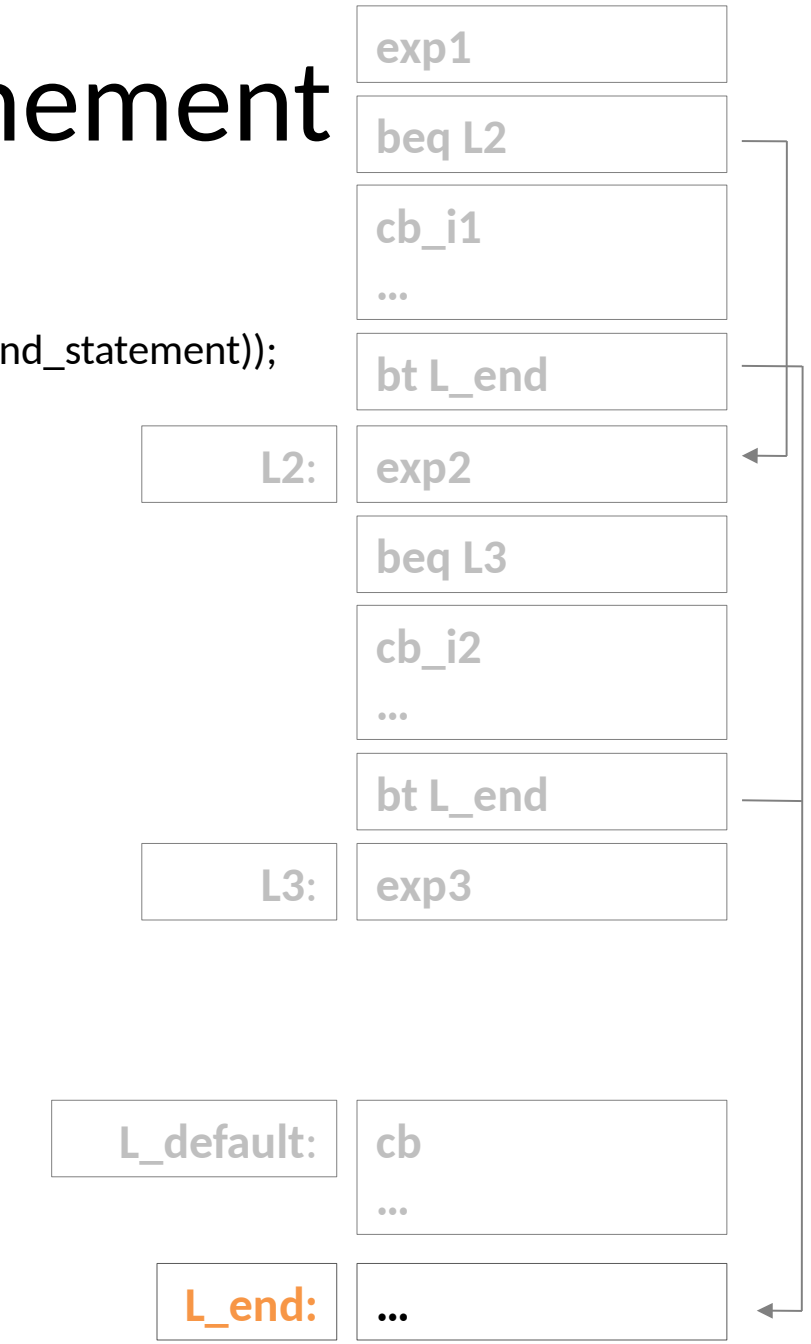
```
cond_statement:  COND LBRACE                    {...}
        cond_block RBRACE          {...}
     ;

cond_block:  case_statements
        |  case_statements default_statement
        ;


case_statements: case_statements case_statement
           | case_statement
           ;

case_statement: CASE exp COLON       {...}
            statements       {...}
          ;

default_statement: DEFAULT COLON
     statements
     ;
```

COND stack

- COND structures may occur in nested form
- Each rule must know which is the current one
- As for SWITCH, we use a stack
  - condStack

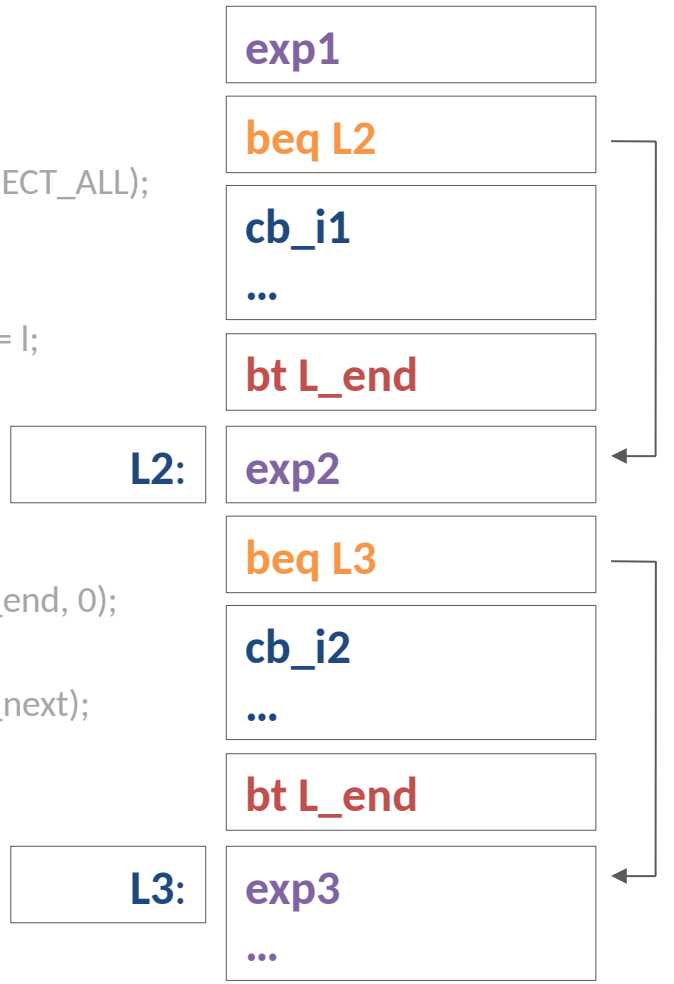
COND statemement

```
cond_statement: COND LBRACE {
                $1 = (t_cond_statement *)malloc(sizeof(t_cond_statement));


                $1->label_end = newLabel(program);
                condStack = addFirst(condStack, $1);
               }
    cond_block RBRACE
               {
                assignLabel(program,$1->label_end);
                condStack = removeFirst(condStack);
               }
               ;
```



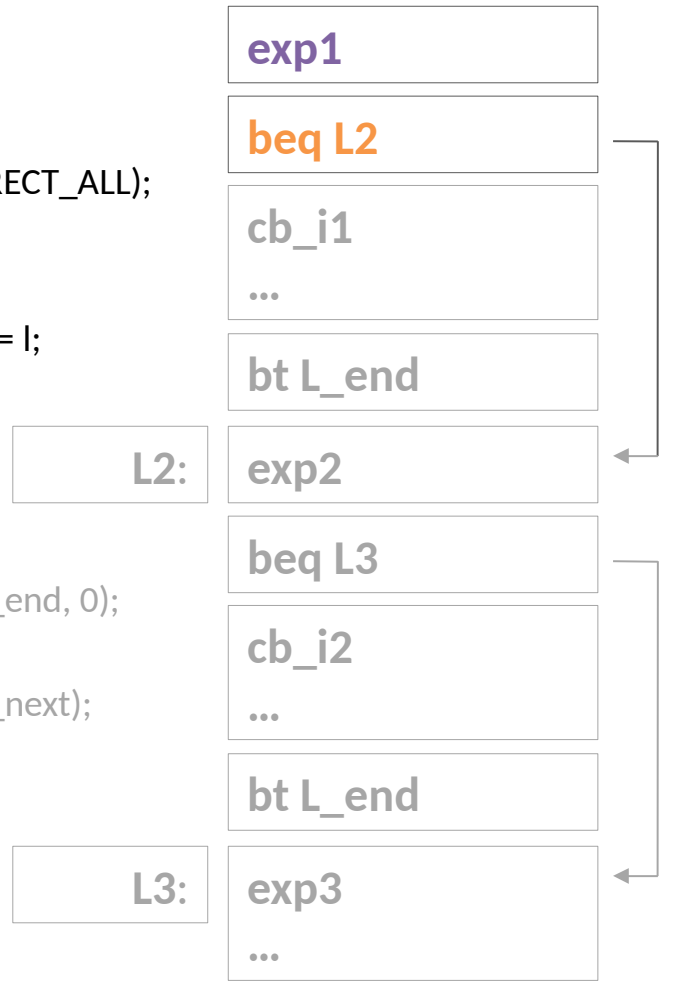
COND case

```
case_statement: CASE exp COLON  {
    if ($2.expression_type == IMMEDIATE)
       gen_load_immediate(program, $2.value);
    else
       gen_andb_instruction(program, $2.value, $2.value, $2.value, CG_DIRECT_ALL);

    t_axe_label* l = newLabel(program);
    ((t_cond_statement *)LDATA(getElementAt(condStack,0)))->label_next = l;
    gen_beq_instruction (program, l, 0);
}
statements {
    gen_bt_instruction(program,
            ((t_cond_statement *)LDATA(getElementAt(condStack,0)))->label_end, 0);
    assignLabel(program,
            ((t_cond_statement *)LDATA(getElementAt(condStack,0)))->label_next);
  }
;
```



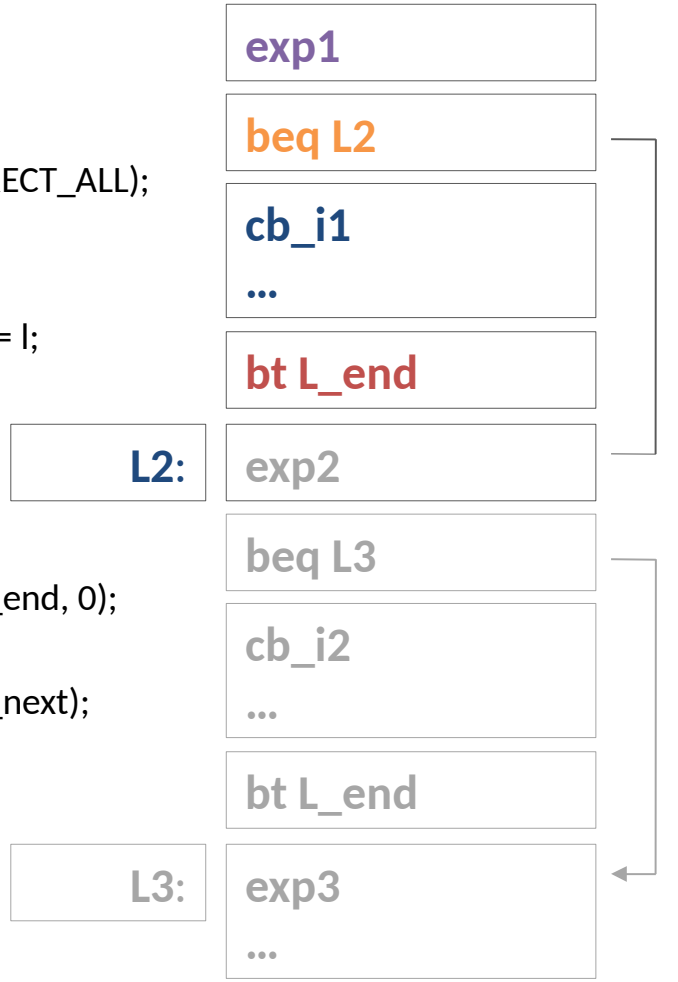
COND case

```
case_statement: CASE exp COLON  {
    if ($2.expression_type == IMMEDIATE)
       gen_load_immediate(program, $2.value);
    else
       gen_andb_instruction(program, $2.value, $2.value, $2.value, CG_DIRECT_ALL);

    t_axe_label* l = newLabel(program);
    ((t_cond_statement *)LDATA(getElementAt(condStack,0)))->label_next = l;
    gen_beq_instruction (program, l, 0);
}
statements {
    gen_bt_instruction(program,
            ((t_cond_statement *)LDATA(getElementAt(condStack,0)))->label_end, 0);
    assignLabel(program,
            ((t_cond_statement *)LDATA(getElementAt(condStack,0)))->label_next);
  }
;
```



COND case

```
case_statement: CASE exp COLON  {
    if ($2.expression_type == IMMEDIATE)
       gen_load_immediate(program, $2.value);
    else
       gen_andb_instruction(program, $2.value, $2.value, $2.value, CG_DIRECT_ALL);

    t_axe_label* l = newLabel(program);
    ((t_cond_statement *)LDATA(getElementAt(condStack,0)))->label_next = l;
    gen_beq_instruction (program, l, 0);
}
statements {
    gen_bt_instruction(program,
            ((t_cond_statement *)LDATA(getElementAt(condStack,0)))->label_end, 0);
    assignLabel(program,
            ((t_cond_statement *)LDATA(getElementAt(condStack,0)))->label_next);
  }
;
```



COND

```
control_statement : if_statement     { /* does nothing */ }
        | do_while_statement SEMI        { /* does nothing */ }
        | while_statement            { /* does nothing */ }
        | return_statement SEMI      { /* does nothing */ }
        | break_statement SEMI       { /* does nothing */ }
        | cond_statement             { /* does nothing */ }
;
```

COND type

```
%union {
   int intval;
   char *svalue;
   t_axe_expression expr;
   t_axe_declaration *decl;
   t_list *list;
   t_axe_label *label;
   t_while_statement while_stmt;
   t_cond_statement *cond_stmt;
}

%token <cond_stmt> COND
```

COND lexer

“cond” {return COND;}

Array equals operator

- Define tokens, syntax/semantic rules translating array equal operator over arrays

int v1[3];

int v2[3];

Int v3[5];

…

if (v1 **=a=** v2) …

If (v1 **=a=** v3) … // false!

Array equals operator

- =a= is an operator then it must be defined as an expression **exp**

exp: …

| IDENTIFIER EQARRAY IDENTIFIER { … }

Array equals operator

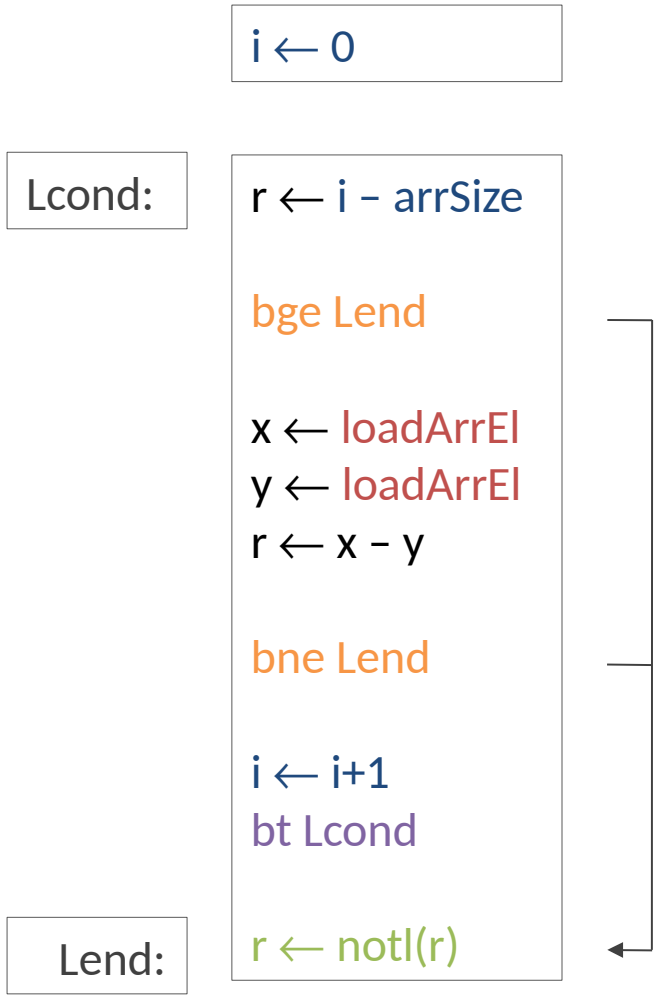
- First verify that both the IDs are array, otherwise error

exp:

```
| IDENTIFIER EQARRAY IDENTIFIER {
    t_axe_variable* id1 = getVariable(program, $1);
    if( ! id1->isArray) exit(-1);

    t_axe_variable* id2 = getVariable(program, $3);
    if( ! id2->isArray) exit(-1);

    if (id1->arraySize != id2->arraySize)
    $$ = create_expression(0, IMMEDIATE);
    else { ... }
```

Array equals operator

```
int array_size = id1->arraySize;
int i = getNewRegister(program);
int r = getNewRegister(program);

gen_addi_instruction(program, i, REG_0, 0);

t_axe_label* lcond = assignNewLabel(program);
gen_subi_instruction(program, r, i, id1->arraySize);

t_axe_label* lend = newLabel(program);
gen_bge_instruction(program, lend, 0);
t_axe_expression i_exp = create_expression(i, REGISTER);
int x = loadArrayElement(program, $1, i_exp);
int y = loadArrayElement(program, $3, i_exp);
gen_sub_instruction(program, r, x, y, CG_DIRECT_ALL);

gen_bne_instruction(program, lend, 0);

gen_addi_instruction(program, i, i, 1);
gen_bt_instruction(program, lcond, 0);

assignLabel(program, lend);
gen_notl_instruction(program, r, r);

$$ = create_expression(r, REGISTER);
```



Array equals operator

- Lexer

  "=a=" {return EQARRAY; }

- Token:

  %token … GTEQ EQARRAY

- If we use == instead of =a= a shift/reduce conflict is produced
  - %expect … (sets the number of S/R to tolerate)

Array comprehension

- Define construct allowing to assign each element of the destination array the result of an expression evaluated on the value of the corresponding element of the source array

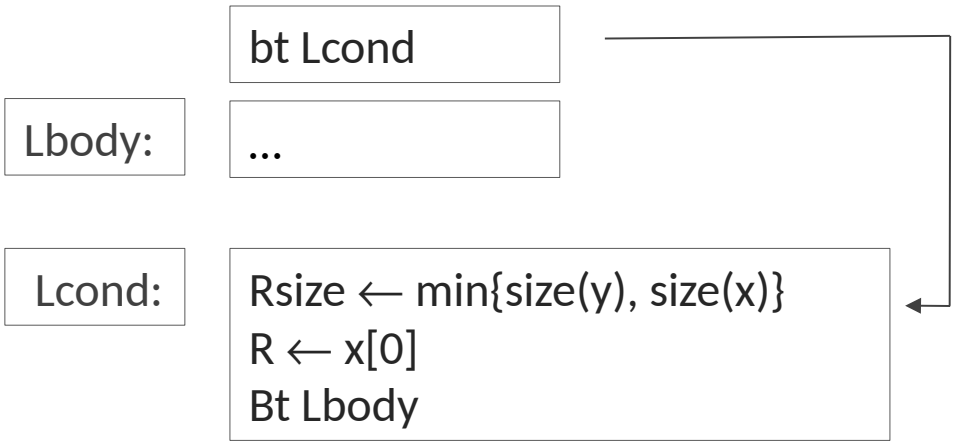
```
int i, x[5], y[7];

x[0] = 1; x[1] = 2; x[2] = 3;
x[3] = 4; x[4] = 5;

// y = {-2, 1, 6, 13, 22, undef, undef}
y = [i * i - 3 for i in x];
```

Array comprehension

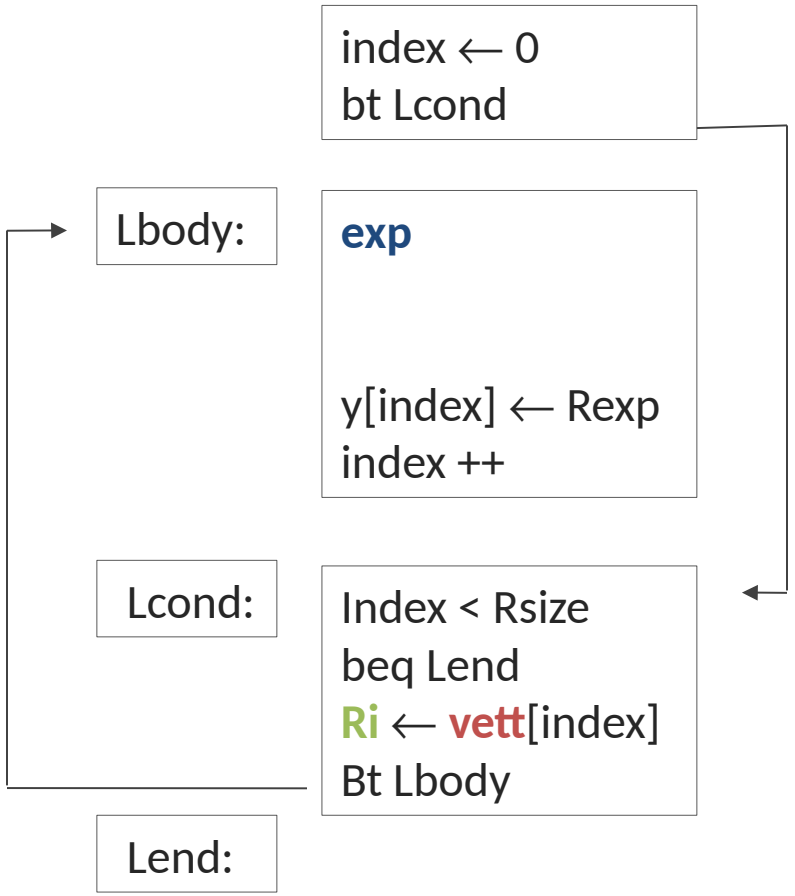
- The vector ID is in the last position

y = [i * i - 3 for i in **vett**];

- To retrieve its size, jump to the end to initialize the loop



Array comprehension



y = [**exp**

for i in **vett**];

Array comprehension

- We use a global variable to store
  - Lcond, Lbody
  - The index register

```
struct {
  t_axe_label *Lbody;
  t_axe_label *Lcond;
  int index_reg;
} ac;
```

- %token IN

Array comprehension

y = [**exp** for **i** in **vett**];

```
assign_statement: …
| IDENTIFIER ASSIGN
  {
    ac.index_reg = gen_load_immediate(program, 0);
    ac.Lcond = newLabel(program);
    gen_bt_instruction(program, ac.Lcond, 0);
    ac.Lbody = assignNewLabel(program);
  }
  LSQUARE exp
  {
    int index_reg = array_compr.index_reg;
    t_axe_expression index_expr = create_expression(index_reg, REGISTER);
    storeArrayElement(program, $1, index_expr, $5);
    gen_addi_instruction(program, index_reg, index_reg, 1);
  }
```

index ← 0
bt Lcond

Lbody:

exp

y[index] ← Rexp
index ++

Array comprehension

```
assign_statement: ...
| IDENTIFIER ASSIGN
  {
    ac.index_reg = gen_load_immediate(program, 0);
    ac.Lcond = newLabel(program);
    gen_bt_instruction(program, ac.Lcond, 0);
    ac.Lbody = assignNewLabel(program);
  }
  LSQUARE exp
  {
    int index_reg = array_compr.index_reg;
    t_axe_expression index_expr = create_expression(index_reg, REGISTER);
    storeArrayElement(program, $1, index_expr, $5);
    gen_addi_instruction(program, index_reg, index_reg, 1);
  }
```

y = [exp for i in vett];

index ← 0
bt Lcond

Lbody:

exp

y[index] ← Rexp
index ++

Array comprehension

y = [**exp** for **i** in **vett**];

```
assign_statement: ...
| IDENTIFIER ASSIGN
  {
    ac.index_reg = gen_load_immediate(program, 0);
    ac.Lcond = newLabel(program);
    gen_bt_instruction(program, ac.Lcond, 0);
    ac.Lbody = assignNewLabel(program);
  }
  LSQUARE exp
  {
    t_axe_expression index_expr = create_expression(ac.index_reg, REGISTER);
    storeArrayElement(program, $1, index_expr, $5);
    gen_addi_instruction(program, index_reg, index_reg, 1);
  }
```

index ← 0
bt Lcond

Lbody:

exp

y[index] ← Rexp
index ++

Array comprehension

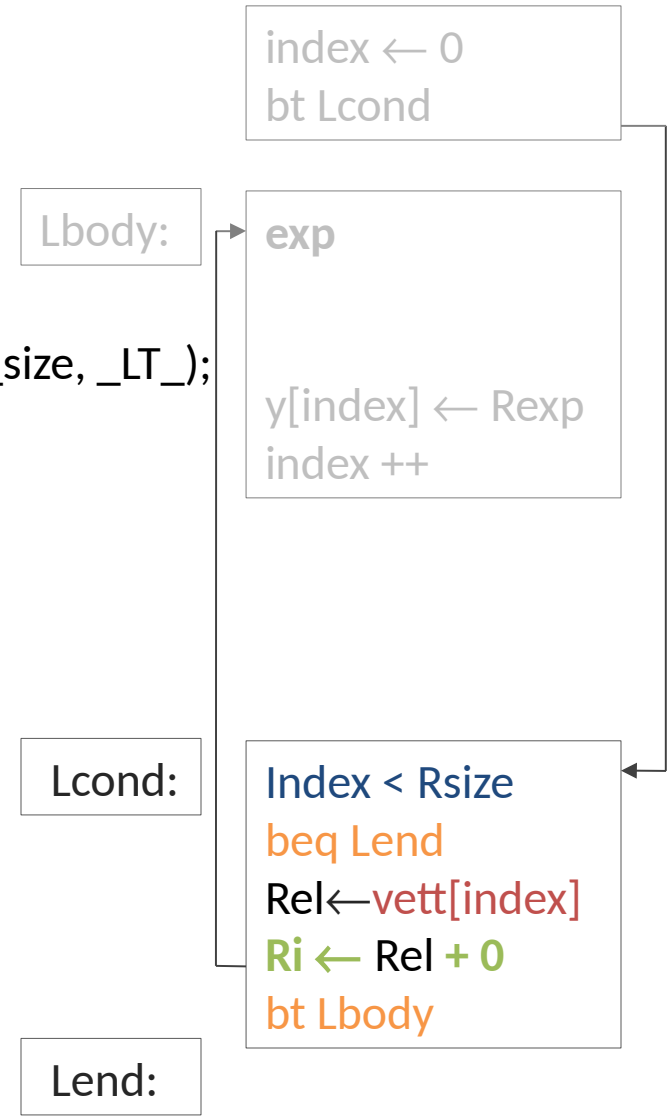
FOR IDENTIFIER IN IDENTIFIER RSQUARE

y = [exp for i in vett];

```
{
   t_axe_variable *dest = getVariable(program, $1);
   t_axe_variable *iv = getVariable(program, $8);
   t_axe_variable *src = getVariable(program, $10);

   if (!dest->isArray || iv->isArray || !src->isArray)
     exit(-1);

   t_axe_expression min_size =  create_expression(dest->arraySize < src->arraySize ?
                                                  dest->arraySize : src->arraySize,
                                                  IMMEDIATE);


  int iv_reg = get_symbol_location(program, $8, 0);
  t_axe_expression index_expr = create_expression(ac.index_reg, REGISTER);
```

Array comprehension

```
...
  t_axe_label *Lend = newLabel(program);

   assignLabel(program, ac.Lcond);

   t_axe_expression cmp =
      handle_binary_comparison(program, index_expr, min_size, _LT_);

   gen_beq_instruction(program, Lend, 0);

   int elem = loadArrayElement(program, $10, index_expr);
   gen_addi_instruction(program, iv_reg, elem, 0);

   gen_bt_instruction(program, ac.Lbody, 0);
   assignLabel(program, Lend);
   free($1);
   free($8);
   free($10);
};
```



MAP statement

Map applies in-place transformation to the elements of an array. For each element, the code block representing the transformation, which is suffixed to **as** keyword is executed and finally the processed array element is written back to its location.

```
int vett[100];
int elem;

// vett = {2, -10, 9}
map elem on vett as {
    elem = elem * 2;
}
// now vett = {4, -20, 18}
```

MAP statement

map elem **on** vett **as**

code_block

Ri ← size(vet)

Lbody: **Relem** ← **vett**[Ri]

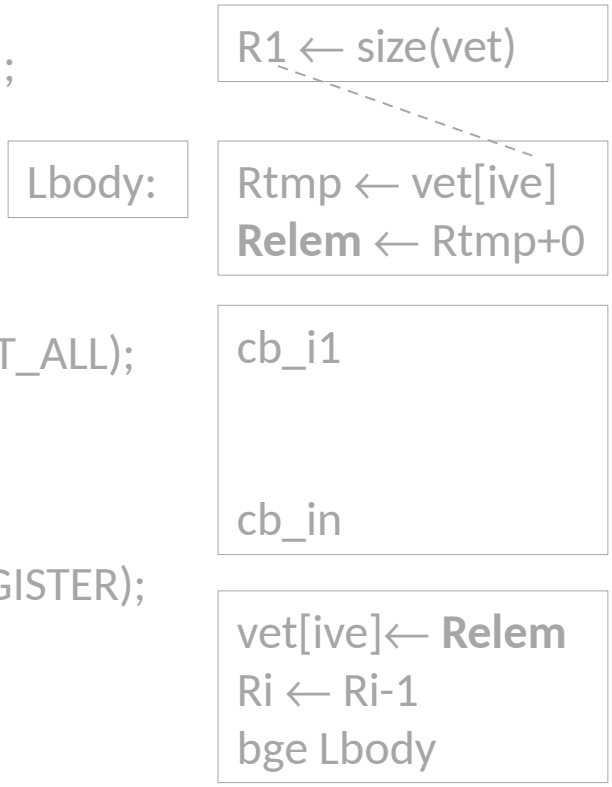
cb_i1

cb_in

vett[Ri]← **Relem**
Ri ← Ri-1
bge Lbody

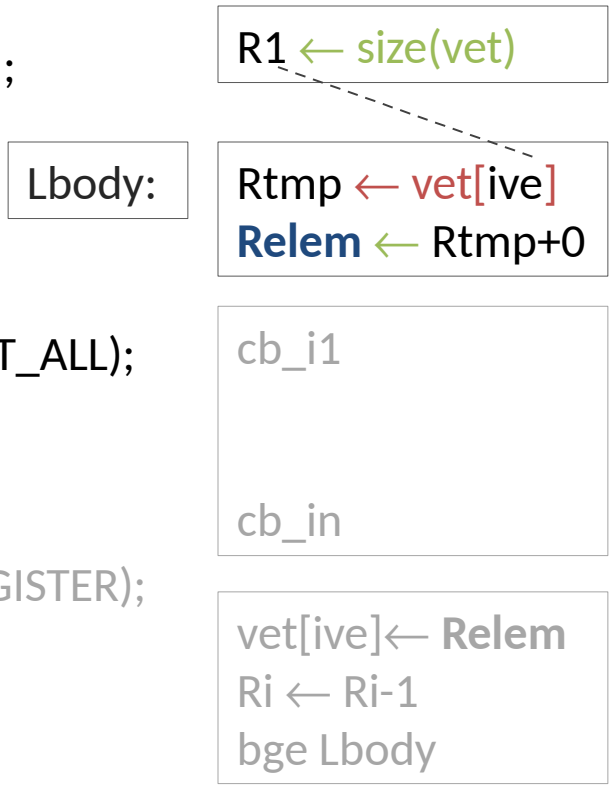
MAP statement

```
map_statement : MAP IDENTIFIER ON IDENTIFIER AS {
  t_axe_variable *var_elem = getVariable(program, $2);
  t_axe_variable *var_arr = getVariable(program, $4);

  int elem_reg = get_symbol_location(program, $2, 0);
  $1 = gen_load_immediate(program, var_arr->arraySize - 1);
  $3 = assignNewLabel(program);

  t_axe_expression ive = create_expression($1, REGISTER);
  int tmp = loadArrayElement(program, $4, ive);
  gen_add_instruction(..., elem_reg, REG_0, tmp, CG_DIRECT_ALL);
}  code_block {
   int elem_reg = get_symbol_location(program, $2, 0);
   t_axe_expression ive = create_expression($1, REGISTER);
   t_axe_expression elem = create_expression(elem_reg, REGISTER);
   storeArrayElement(program, $4, ive, elem);
   gen_subi_instruction(program, $1, $1, 1);
   gen_bge_instruction(program, $3, 0);

  free($2);  free($4); }
```



MAP statement

```
map_statement : MAP IDENTIFIER ON IDENTIFIER AS {
  t_axe_variable *var_elem = getVariable(program, $2);
  t_axe_variable *var_arr = getVariable(program, $4);

  int elem_reg = get_symbol_location(program, $2, 0);
  $1 = gen_load_immediate(program, var_arr->arraySize - 1);
  $3 = assignNewLabel(program);

  t_axe_expression ive = create_expression($1, REGISTER);
  int tmp = loadArrayElement(program, $4, ive);
  gen_add_instruction(..., elem_reg, REG_0, tmp, CG_DIRECT_ALL);
} code_block {
  int elem_reg = get_symbol_location(program, $2, 0);
  t_axe_expression ive = create_expression($1, REGISTER);
  t_axe_expression elem = create_expression(elem_reg, REGISTER);
  storeArrayElement(program, $4, ive, elem);
  gen_subi_instruction(program, $1, $1, 1);
  gen_bge_instruction(program, $3, 0);

  free($2);  free($4); }
```



MAP statement

```
map_statement : MAP IDENTIFIER ON IDENTIFIER AS {
  t_axe_variable *var_elem = getVariable(program, $2);
  t_axe_variable *var_arr = getVariable(program, $4);

  int elem_reg = get_symbol_location(program, $2, 0);
  $1 = gen_load_immediate(program, var_arr->arraySize - 1);
  $3 = assignNewLabel(program);

  t_axe_expression ive = create_expression($1, REGISTER);
  int tmp = loadArrayElement(program, $4, ive);
  gen_add_instruction(..., elem_reg, REG_0, tmp, CG_DIRECT_ALL);
} code_block {
   int elem_reg = get_symbol_location(program, $2, 0);
   t_axe_expression ive = create_expression($1, REGISTER);
   t_axe_expression elem = create_expression(elem_reg, REGISTER);
   storeArrayElement(program, $4, ive, elem);
   gen_subi_instruction(program, $1, $1, 1);
   gen_bge_instruction(program, $3, 0);

  free($2);  free($4); }
```

Rive ← size(vet)

Lbody:

Rtmp ← vet[ive]
Relem ← Rtmp+0

cb_i1

cb_in

vet[ive]← **Relem**
Rive ← Rive-1
bge Lbody

MAP statement

- %token <intval> MAP
- %token AS

- control_statement : …
  | map_statement
  ;

Reduce statement

Reduce applies a function reducing the elements of an array into a single scalar (e.g. computing their average). The result, held in the support variable specified after **into** keyword, is updated computing the reduction expression, enclosed between double square braces, for each element of the array.

```
int vett[100];
int elem, t, sum;

read(t);
sum = 0;

// vett = {2, 6, 13}
reduce elem into sum
  as [[ sum + t * elem ]]
  on vett;
// sum = 21
```

Reduce statement

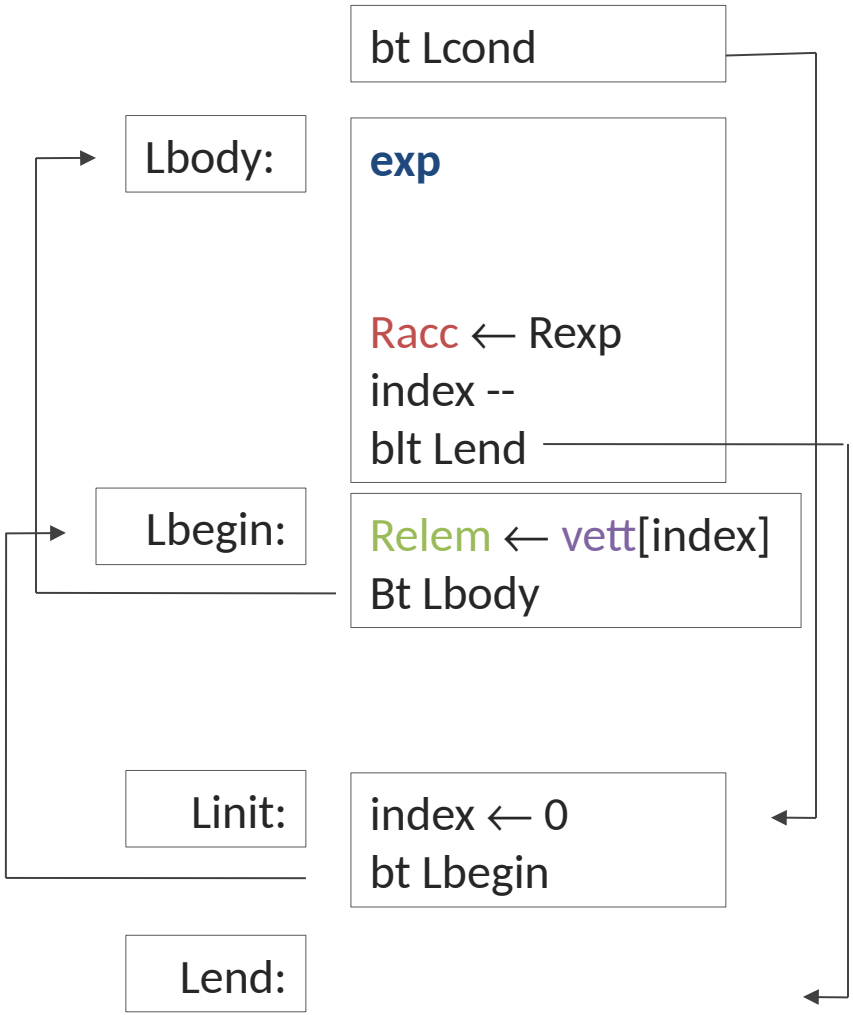
- Similar to array comprehension

- The vector ID is in the last position of the stmt

```
reduce elem into sum
  as [[ sum + t * elem ]]
  on vett;
```

- To retrieve its size jump to the end to initialize the loop

Reduce statement



REDUCE **elem** INTO **acc** AS [[

exp]] ON **vett**

Reduce statement

- Three labels are stored on tokens

%token <**label**> ON REDUCE INTO

- Lbegin is local in the last semantic action
- control_statement : …

| reduce_statement SEMI

;