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

Marcello M. Bersani

LFC – Politecnico di Milano

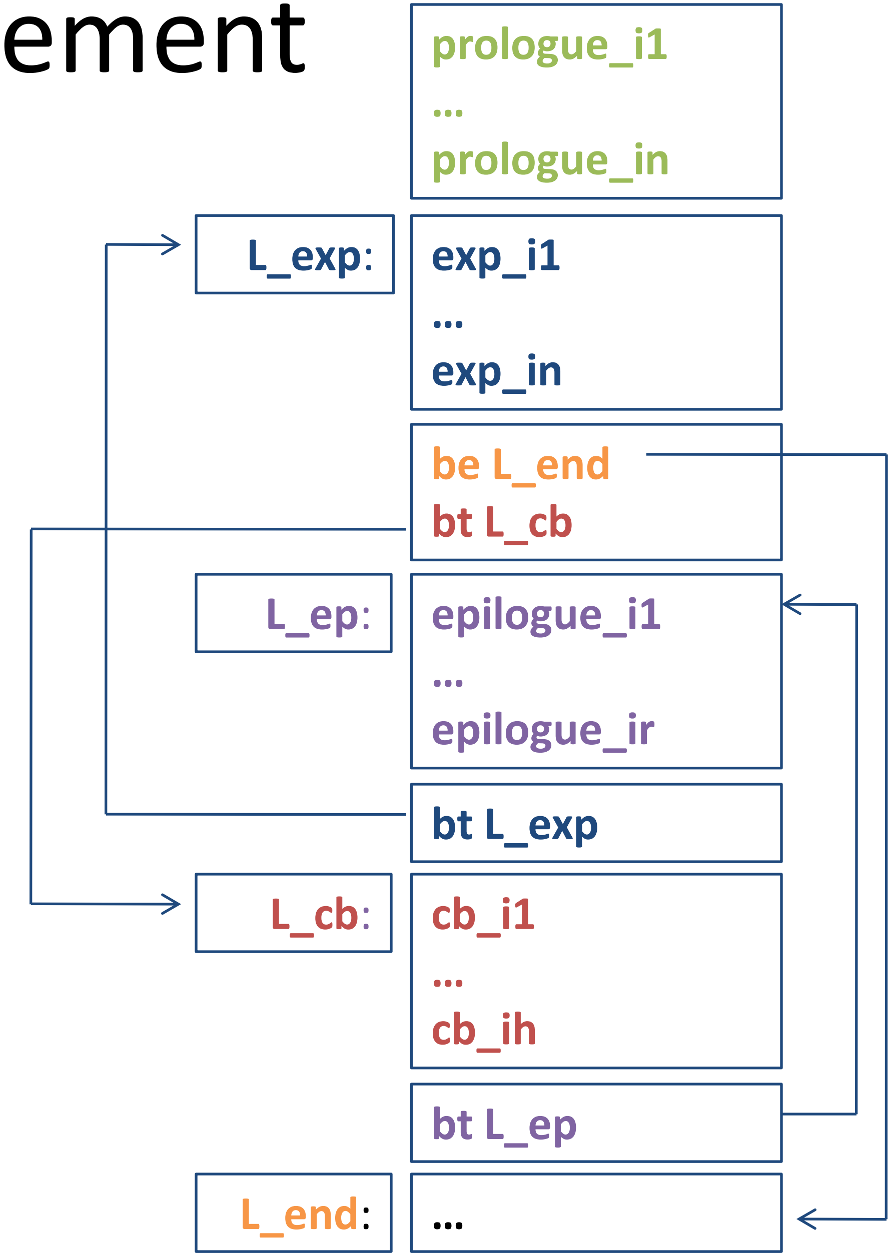
FOR statement

- Define tokens, syntax/semantic rules translating a FOR statement
  - prologue
  - epilogue

```
for (i=0, j=2; i<10; i=i+1){
   code_block
}
```

FOR statement

```
for (prologue; exp; epilogue){
        code_block
}
```



FOR statement

```
for (prologue; exp; epilogue){
    code_block
}
```

- Prologue, epilogue
  - list of assignments (already defined)

```
for (i=0, j=2; exp; i=i+1){
    code_block
}
```

FOR data structure

- Four labels rule for control flow
  - L_exp
  - L_code
  - L_end
  - L_epilogue

	prologue_i1 … prologue_in
L_exp:	exp_i1 … exp_in
	be L_end bt L_cb
L_ep:	epilogue_i1 … epilogue_ir
	bt L_exp
L_cb:	cb_i1 … cb_ih
	bt L_ep
L_end:	…

FOR data structure

```
typedef struct t_for_statement{
   t_axe_label *label_exp;
   t_axe_label *label_end;
   t_axe_label *label_code;
   t_axe_label *label_epilogue;
} t_for_statement;
```

- Axe_struct.h

FOR syntax/semantics

```
for_statement: FOR     { $1 = create_for_statement();}
      LPAR assign_list SEMI { $1.label_exp = assignNewLabel(program);}
      exp SEMI {
              $1.label_end = newLabel(program);
              gen_beq_instruction(program, $1.label_end, 0);
              $1.label_code = newLabel(program);
              gen_bt_instruction(program, $1.label_code, 0);
              $1.label_epilogue = assignNewLabel(program);
      }
      assign_list RPAR {
              gen_bt_instruction(program, $1.label_exp, 0);
              assignLabel(program, $1.label_code);
      }
      code_block  {
              gen_bt_instruction(program, $1.label_epilogue, 0);
              assignLabel(program, $1.label_end);
      }
;
```

```
          prologue_i1
          ...
          prologue_in
L_exp:    exp_i1
          ...
          exp_in
          be L_end
          bt L_cb
L_ep:     epilogue_i1
          ...
          epilogue_ir
          bt L_exp
L_cb:     cb_i1
          ...
          cb_ih
          bt L_ep
L_end:    ...
```

FOR syntax/semantics

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


Assign_list : {}
        | assign_list COMMA assign_statement    {/* does nothing */}
        | assign_statement                      {/* does nothing */}
;
```

FOR data structure utils

```
extern t_for_statement create_for_statement(){

    t_for_statement result;

    result.label_condition = NULL;
    result.label_end = NULL;
    result.label_code = NULL;
    result.label_epilogue = NULL;

   return result;
}
```

FOR tokens

%union {

…

t_for_statement **for_stmt**;

}

%token <**for_stmt**> FOR

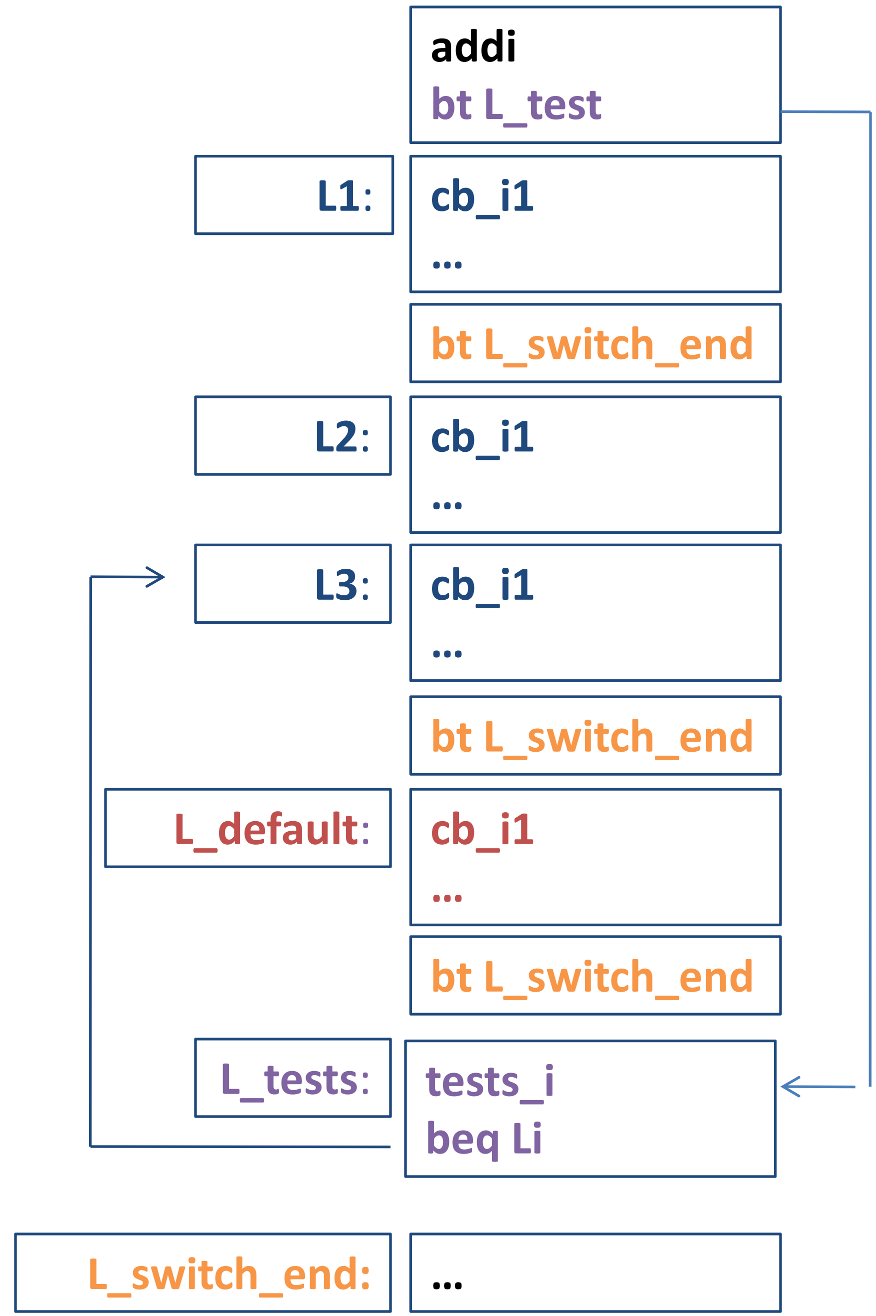
SWITCH statement

- Define tokens, syntax/semantic rules translating a SWITCH statement
  - default

```
switch (x){
   case 1: {…; break;}
   case 2: {…}
   case 3: {…; break;}
   default: {…}
}
```

SWITCH statement

```
switch (x){
        case 1: {…; break;}
        case 2: {…}
        case 3: {…; break;}
        default: {…}
}
```



SWITCH data structure

- Register where IDENTIFIER is stored
- Three fixed labels
  - switch_end
  - default_label (possibly NULL)
  - begin_tests
- List of labels
  - Handle cases

```
              addi
              bt L_test
L1:           cb_i1
              …
              bt L_switch_end
L2:           cb_i1
              …
L3:           cb_i1
              …
              bt L_switch_end
L_default:    cb_i1
              …
              bt L_switch_end
L_tests:      tests_i
              beq Li
L_switch_end: …
```

SWITCH data structure

```
typedef struct
{
    int cmp_register;
    t_list *cases;

    t_axe_label *default_label;
    t_axe_label *switch_end;
    t_axe_label *begin_tests;
} t_switch_statement;
```

- Axe_struct.h

	addi bt L_test
L1:	cb_i1 …
	bt L_switch_end
L2:	cb_i1 …
L3:	cb_i1 …
	bt L_switch_end
L_default:	cb_i1 …
	bt L_switch_end
L_tests:	tests_i beq Li
L_switch_end:	…

SWITCH data structure

```
typedef struct
{
   int number;
   t_axe_label *begin_case_label;
} t_case_statement;
```

```
switch (x){
        case 1: {…; break;}
        case 2: {…}
        case 3: {…; break;}
        default: {…}
}
```

	addi bt L_test
L1:	cb_i1 …
	bt L_switch_end
L2:	cb_i1 …
L3:	cb_i1 …
	bt L_switch_end
L_default:	cb_i1 …
	bt L_switch_end
L_tests:	tests_i beq Li
L_switch_end:	…

SWITCH syntactic

```
switch_statement:  SWITCH LPAR IDENTIFIER RPAR LBRACE      {...}
                    switch_block RBRACE                    {...}
                    ;

switch_block:  case_statements                             {...}
        |  case_statements default_statement               {...}
        ;


case_statements: case_statements case_statement
          | case_statement
          ;

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

default_statement: DEFAULT COLON                           {...}
              statements
              ;
```

SWITCH issue

- Switch structures may occur in nested form
- Each rule needs to know which is the current switch

```
break_statement: BREAK
    {  if (!active_switch) {
                        abort();
      }
      else
            gen_bt_instruction(…, current_switch->switch_end);
    }
;
```

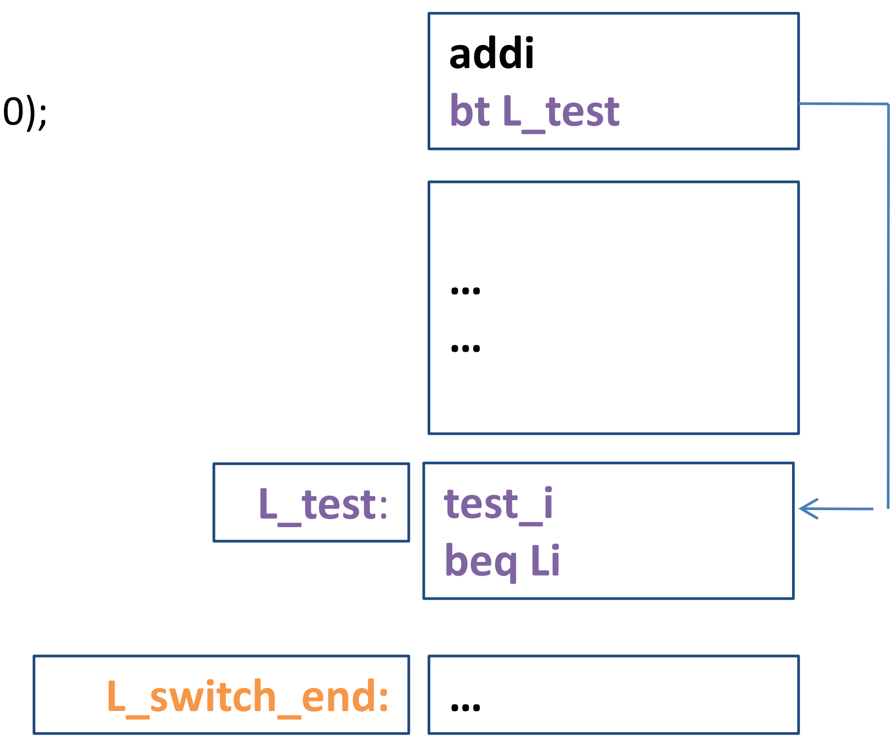
SWITCH issue

- Current switch is kept in a STACK data structure
  - Global access
  - All switch rules can use it

SWITCH

```
switch_statement: SWITCH LPAR IDENTIFIER RPAR LBRACE {
      $1 = (t_switch_statement *)malloc(sizeof(t_switch_statement));
      $1->cmp_register = getNewRegister(program);
      gen_addi_instruction(program, $1->cmp_register, get_symbol_location(program,$3,0), 0);
      $1->begin_test = newLabel(program);
      $1->switch_end = newLabel(program);
      switchStack = addFirst(switchStack, $1);
      gen_bt_instruction(program, $1->begin_test, 0);
       }
       switch_block RBRACE  {…}


}
;
```



SWITCH

```
switch_statement: SWITCH LPAR IDENTIFIER RPAR LBRACE {…}
      switch_block RBRACE{
        t_list *p;
        int cmpReg;
        assignLabel(program, $1->begin_test);
        cmpReg = getNewRegister(program);
        p = $1->cases;
        while (p!=NULL){
          gen_subi_instruction(…,cmpReg,$1->cmp_register,((t_case_statement *)p->data)->number);
          gen_beq_instruction(…,((t_case_statement *)p->data)->begin_case_label, 0);
          p = p->next;
        }
        if ($1->default_label != NULL)
          gen_bt_instruction(program,$1->default_label,0);
          assignLabel(program,$1->switch_end);
          switchStack = removeFirst(switchStack);
      }
 ;
```

addi
bt L_test

…

L_test:

subi R n
beq L1

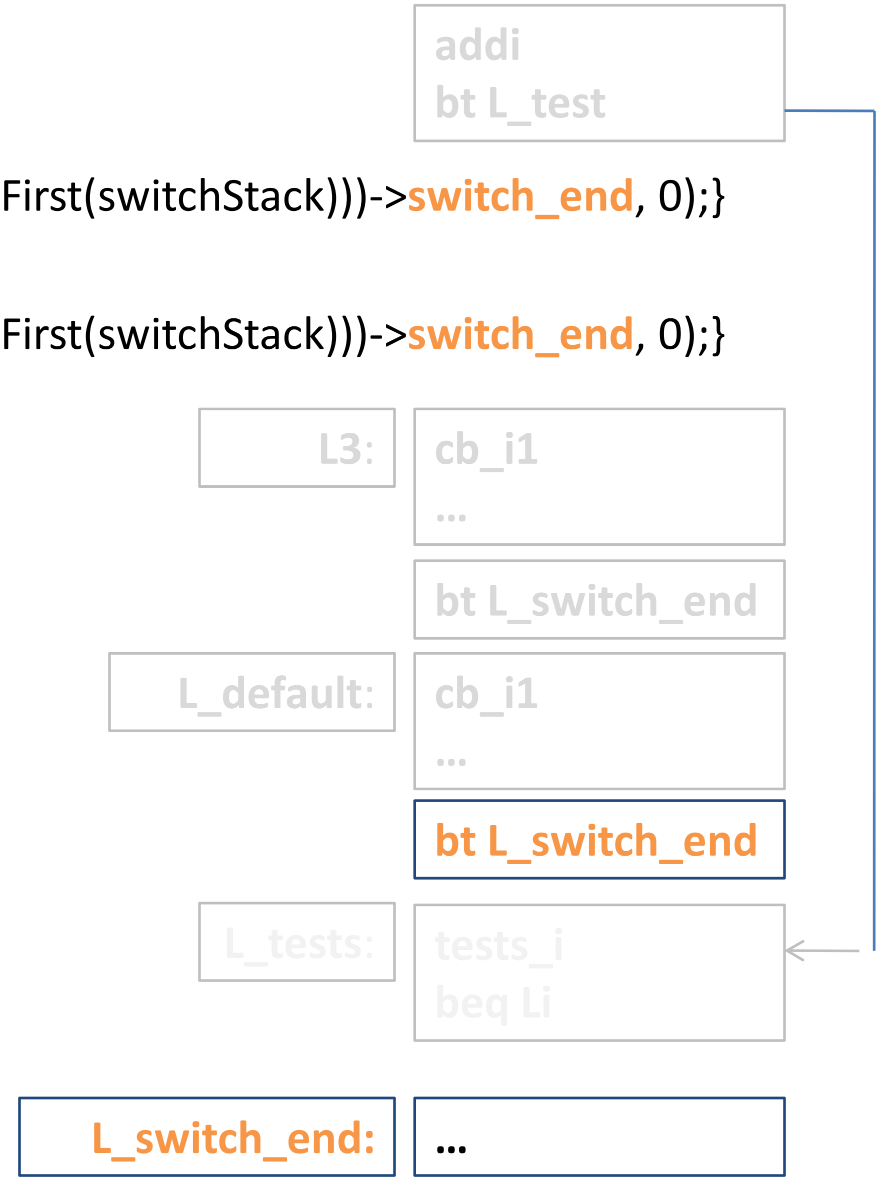
bt L_default

L_switch_end:

…

SWITCH

```
switch_block:  case_statements
          { gen_bt_instruction(…, ((…)LDATA(getFirst(switchStack)))->switch_end, 0);}
        |  case_statements default_statement
          { gen_bt_instruction(…, ((…)LDATA(getFirst(switchStack)))->switch_end, 0);}
        ;
```



SWITCH

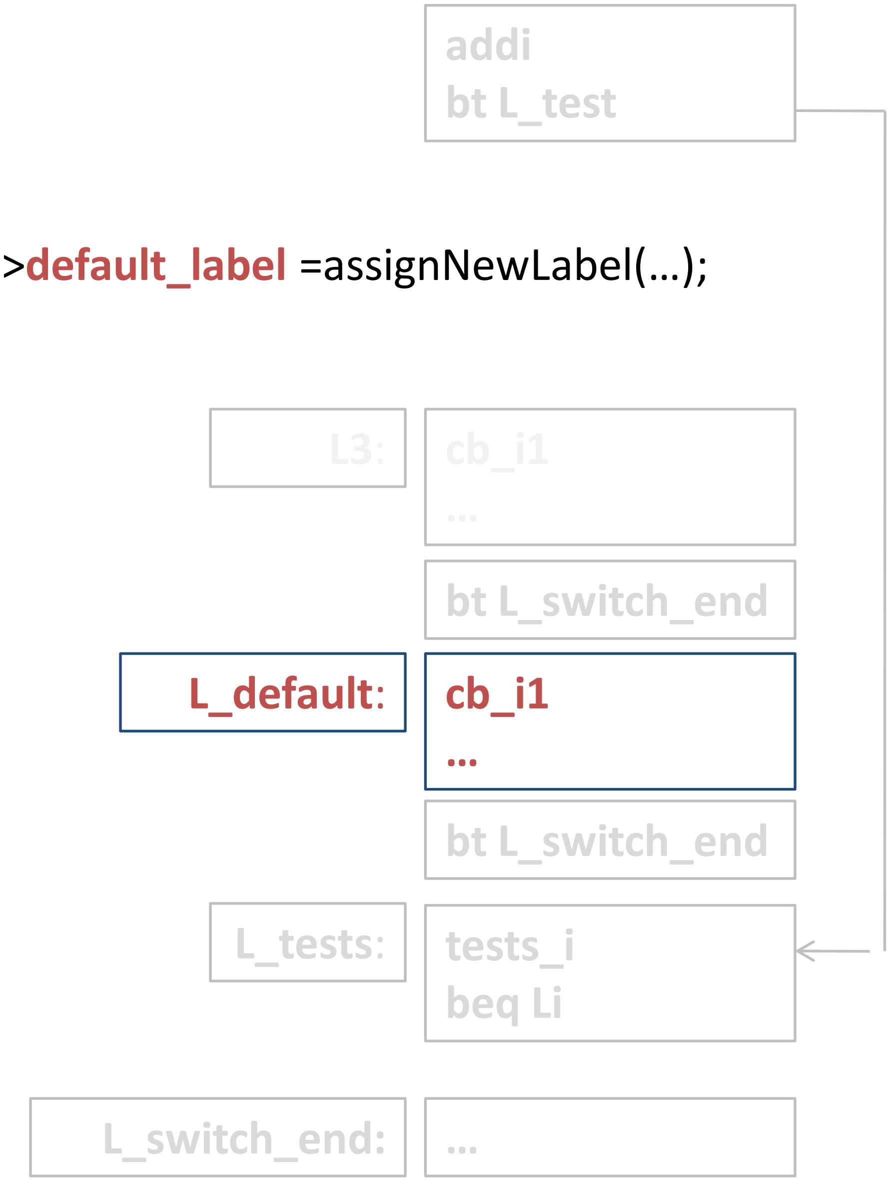
```
case_statements: case_statements case_statement
          | case_statement
          ;

case_statement: CASE NUMBER COLON {
  t_case_statement *c = (...malloc(sizeof(t_case_statement));
  c->number = $2;
  c->begin_case_label = assignNewLabel(program);
  ((...LDATA(getFirst(switchStack)))->cases = addLast(((t_switch_statement
     *)LDATA(getFirst(switchStack)))->cases,c);
            }
            statements
    ;
```

	addi bt L_test
L3:	**cb_i1** **...**
	bt L_switch_end
L_default:	cb_i1 ...
	bt L_switch_end
L_tests:	tests_i beq Li
L_switch_end:	...

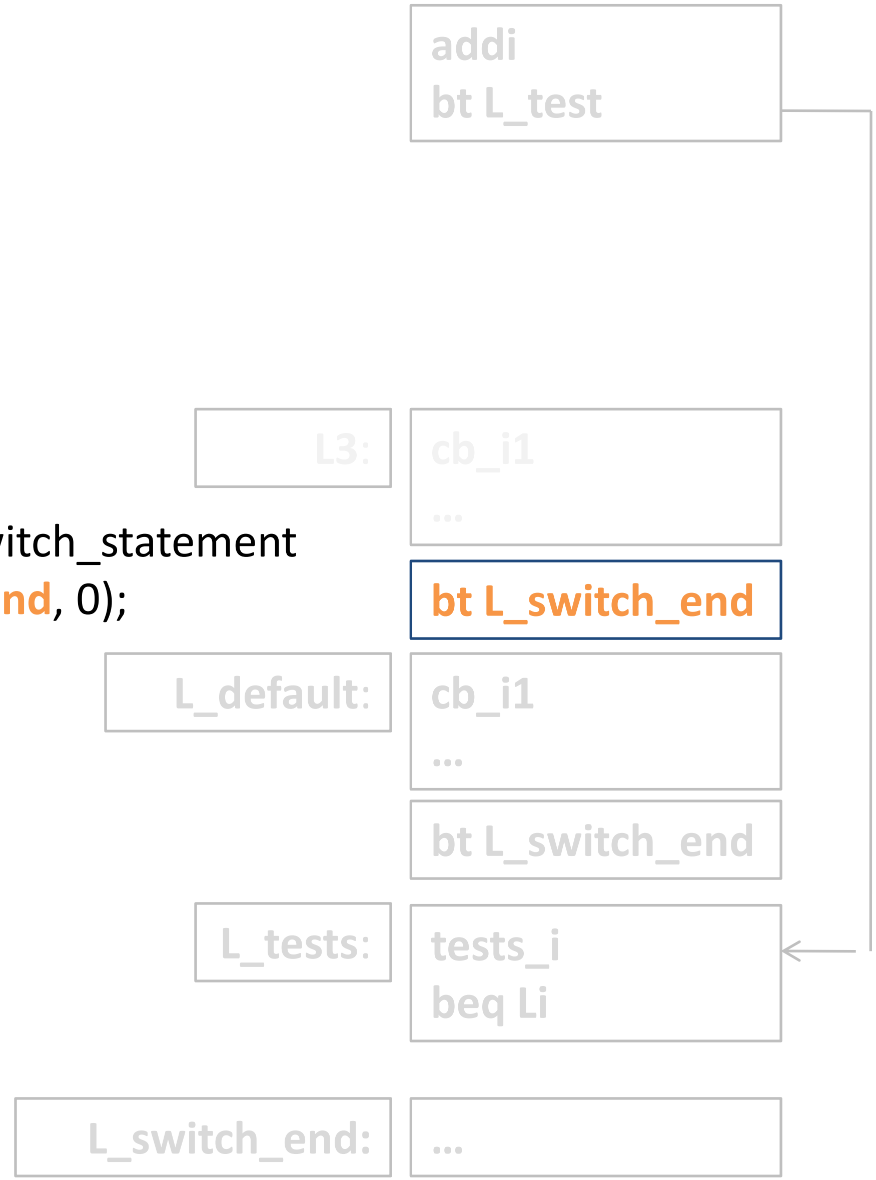
SWITCH

```
default_statement: DEFAULT COLON
        { ((…}LDATA(getFirst(switchStack)))->default_label =assignNewLabel(…);
        }
        statements
        ;
```



SWITCH

```
break_statement: BREAK
      {  if (switchStack == NULL) {
                abort();
         }
         else
                gen_bt_instruction(program, ((t_switch_statement
      *)LDATA(getFirst(switchStack)))->switch_end, 0);
      }
;
```



SWITCH

```
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 */ }
        | switch_statement          { /* does nothing */ }
;
```

SWITCH

```
%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_switch_statement *switch_stmt;
}

%token <switch_stmt> SWITCH
```

SHIFT statement

- Define tokens, syntax/semantic rules translating a SHIFT statement over arrays
  - Left/right
  - 0 fills free places

```
a << 2;
a >> x;
```

SHIFT statement

- a = [0,1,2,3,4,5]

a << 2;

    [2,3,4,5,0,0]

a >> x;

    [0,0,1,2,3,4]   if x==1

Left SHIFT statement

- ACSE code mimics iterations over the array
  - After defining index where shift starts (sha)
  - move each element to the final position
  - Then fill with 0 from (size-sha)

a = [0,1,2,3,4,5]

a << 2

[0,1,**2**,3,4,5]

[**2**,3,4,5,**0**,0]

Left SHIFT statement

- Definition of index where shift starts (sha)

```
array_shift_statement : IDENTIFIER SHL_OP exp {
    t_axe_variable* id = getVariable(…, $1);
    if( ! id->isArray)  exit(-1);
    int array_size = id->arraySize;
    int sha_reg = getNewRegister(…);
    t_axe_expression size_exp = create_expression(array_size, IMMEDIATE);
    t_axe_expression zero_exp = create_expression(0, IMMEDIATE);
    int size_reg = getNewRegister(…);
    gen_addi_instruction(…, size_reg, REG_0, size_exp.value);
    t_axe_expression shift_amount = handle_bin_numeric_op(…, $3, zero_exp, ADD);
    if(shift_amount.expression_type == IMMEDIATE)
       gen_addi_instruction(…, sha_reg, REG_0, shift_amount.value);
    else
       gen_add_instruction(…, sha_reg, REG_0, shift_amount.value, CG_DIRECT_ALL);
```

Left SHIFT statement

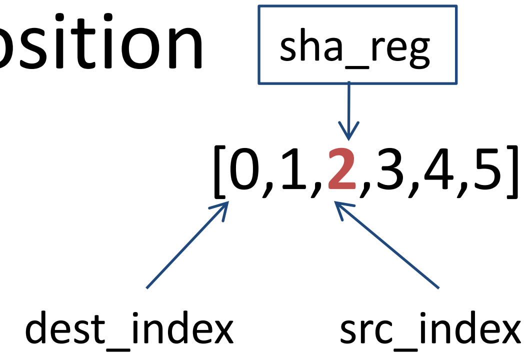
- Definition of index where shift starts (sha)
  - Possible solution: reduce shift until its value is less than the array size

```
s_start  tr=sha_reg-size_reg
         ble s_end
         sha_reg=sha_reg-size_reg
         bt s_start
s_end    ...
```

```
int tr = getNewRegister(...);
t_axe_label * settings_end = newLabel(...);
t_axe_label * settings_start = assignNewLabel(...);
gen_sub_instruction(..., tr, sha_reg, size_reg, CG_DIRECT_ALL);
gen_ble_instruction(..., settings_end, 0);
gen_sub_instruction(..., sha_reg, sha_reg, size_reg, CG_DIRECT_ALL);
gen_bt_instruction(..., settings_start, 0);
assignLabel(..., settings_end);
```

Left SHIFT statement

- move each element to the final position




```
int temp_reg;
int src_index = getNewRegister(…);
int dest_index = getNewRegister(…);
gen_add_instruction(…, src_index, REG_0, sha_reg, CG_DIRECT_ALL);
gen_add_instruction(…, dest_index, REG_0, REG_0, CG_DIRECT_ALL);
t_axe_label * stop_shifting = newLabel(…);
t_axe_label * continue_shifting = assignNewLabel(…);
t_axe_expression src_exp, dest_exp;
src_exp = create_expression(src_index, REGISTER);
dest_exp = create_expression(dest_index, REGISTER);

//check if the right bound of the array has been reached...
gen_sub_instruction(…, tr, size_reg, src_index, CG_DIRECT_ALL);
gen_beq_instruction(…., stop_shifting, 0);
temp_reg = loadArrayElement(…, $1, src_exp);
t_axe_expression temp_exp = create_expression(temp_reg, REGISTER);
storeArrayElement(…, $1, dest_exp, temp_exp);
gen_addi_instruction(…, src_index, src_index, 1);
gen_addi_instruction(…, dest_index, dest_index, 1);
gen_bt_instruction(…, continue_shifting, 0);
assignLabel(…, stop_shifting);
```

```
        src_index = 0 + sha_reg
        dest_index = 0
c_shift tr=size_reg-src_index
        beq stop_shifting
        temp_reg = loadArrayEl(a, src_exp)
        storeArrayEl(dest_exp,temp_exp)
        src_index=src_index+1
        dest_index=dest_index+1
        bt c_shift
stop_shift …
```

Left SHIFT statement

- fill with 0 from (size-sha)

size_reg-sha_reg

[2,3,4,5,0,0]

gen_sub_instruction(…, sha_reg, size_reg, sha_reg, CG_DIRECT_ALL);
t_axe_label * **stop_filling** = newLabel(…);
t_axe_label * **continue_filling** = assignNewLabel(…);
gen_sub_instruction(…, tr, size_reg, sha_reg, CG_DIRECT_ALL);
gen_beq_instruction(…, **stop_filling**, 0);
shift_amount = create_expression(sha_reg, REGISTER);
storeArrayElement(…., $1, shift_amount, zero_exp);
gen_addi_instruction(…, sha_reg, sha_reg, 1);
gen_bt_instruction(…, **continue_filling**, 0);
assignLabel(…, stop_filling);

	sha_reg= size_reg - sha_reg
c_fill	tr=size_reg-sha_reg
	beq stop_fill
	storeArrayEl(dest_exp,zero_exp)
	sha_reg=sha_reg+1
	bt c_fill
stop_fill	…

Conditional exp

- It is an inline if-then-else

b = a * 2 **if** a > 2 * b **else** 0;

- Take care to specify the precedence and associativity of the operators

Conditional exp

- $r = exp$ **if** $\mathbf{exp_c}$ **else** exp;

$R \leftarrow \mathbf{exp_c} == 0$

$Mask \leftarrow R - 1$ [$Mask=-1 = (1_{31}1_{30} \ldots 1_0)$ if $\mathbf{exp_c} \neq 0$]

$Nmask \leftarrow NOTB(Mask)$

$Rr \leftarrow (exp \text{ ANDB } Mask) \text{ ORB } (exp \text{ ANDB } Nmask)$

Conditional exp

```
exp: …
| exp IF exp ELSE exp
  {
   if ($3.expression_type == IMMEDIATE) {
    $$ = $3.value ? $1 : $5;
   } else {
    t_axe_expression zero = create_expression(0, IMMEDIATE);
    t_axe_expression cmp = handle_binary_comparison(program, $3, zero, _EQ_);
    t_axe_expression one = create_expression(1, IMMEDIATE);
    t_axe_expression mask = handle_bin_numeric_op(program, cmp, one, SUB);
    int r = getNewRegister(program);
    gen_notb_instruction(program, r, mask.value);
    t_axe_expression nmask = create_expression(r, REGISTER);
    $$ = handle_bin_numeric_op(program,
                        handle_bin_numeric_op(program, $1, mask, ANDB),
                        handle_bin_numeric_op(program, $5, nmask, ANDB),
                        ORB);
   }
  }
;
```

Conditional exp

- **%left** IF ELSE

Write(a if b else c if b if c else b < 2 else -100);

[a if b else c] if b if c else b < 2 else -100

[exp] if [b if c else b < 2] else -100

[exp] if [exp] else -100

Splice expression

- r = **a** $ **b** @ k;

- r is defined as
  - k m.s. bits of **a** followed by
  - 32 - k l.s. bits of **b**

$$r = [\mathbf{a}_{31}\ldots\mathbf{a}_{(31-(k-1))}\mathbf{b}_{(31-k)}\ldots\mathbf{b}_{0}]_{0<k<32}$$

$$r = [\mathbf{a}_{31}\ldots\mathbf{a}_{0}]_{k>=32}$$

$$r = [\mathbf{b}_{31}\ldots\mathbf{b}_{0}]_{k=0}$$

Splice expression

- r = **a** $ **b** @ 5;

- $r = [\mathbf{a}_{31}\mathbf{a}_{30}\mathbf{a}_{29}\mathbf{a}_{28}\mathbf{a}_{27}\mathbf{b}_{26}\ldots\mathbf{b}_{0}]$

- How to do it?

$$r = \underbrace{(\mathbf{a}\ \text{ANDB Mask})}_{\mathbf{r\_e1}}\ \text{ORB}\ \underbrace{(\mathbf{b}\ \text{ANDB NMask})}_{\mathbf{r\_e2}}$$

$[\mathbf{1}_{31}\mathbf{1}_{30}\mathbf{1}_{29}\mathbf{1}_{28}\mathbf{1}_{27}\mathbf{0}_{26}\ldots\mathbf{0}_{0}]$

$[\mathbf{0}_{31}\mathbf{0}_{30}\mathbf{0}_{29}\mathbf{0}_{28}\mathbf{0}_{27}\mathbf{1}_{26}\ldots\mathbf{1}_{0}]$

Splice expression

```
exp:
| exp DOLLAR exp AT exp {
  int e_c;

  if ($5.value>32)
    e_c = 0;
  else
    e_c = 32-$5.value;

  int r_e2 = gen_load_immediate(program, 0);
  int r_index = gen_load_immediate(program, e_c);

  $4 = newLabel(program); /*label end*/
  $2 = assignNewLabel(program); /*label condition*/

  gen_beq_instruction(program, $4, 0);
  gen_shli_instruction(program, r_e2, r_e2, 1);
  gen_addi_instruction(program, r_e2, r_e2, 1);
  gen_subi_instruction(program, r_index, r_index, 1);
  gen_bt_instruction(program, $2, 0);
  assignLabel(program, $4);

  int r_e1 = getNewRegister(program);
  gen_notb_instruction(program, r_e1, r_e2); /*define mask for e1 through*/

  …
```

```
        r_e2 ← 0
        r_index ← 32 - e_c

L_cond: beq L_end
        r_e2 ← shiftl(r_e2)
        r_e2 ← r_e2+1
        r_index ← r_index-1
        bt L_cond
L_end:  …

        r_e1 ← notb(r_e2)
```

Splice expression

```
exp:
| exp DOLLAR exp AT exp {
  …

   /*get r_e1 bits of exp1*/
   if ($1.expression_type == IMMEDIATE)
     gen_andb_instruction(program, r_e1, r_e1, gen_load_immediate(program, $1.value), CG_DIRECT_ALL);
   else
     gen_andb_instruction(program, r_e1, r_e1, $1.value, CG_DIRECT_ALL);

   /*get 32-e1 bits of exp2*/
   if ($3.expression_type == IMMEDIATE)
     gen_andb_instruction(program, r_e2, r_e2, gen_load_immediate(program, $3.value), CG_DIRECT_ALL);
    else
     gen_andb_instruction(program, r_e2, r_e2, $3.value, CG_DIRECT_ALL);

   int r = getNewRegister(program);
   gen_orb_instruction(program, r, r_e1, r_e2, CG_DIRECT_ALL);
   $$ = create_expression(r, REGISTER);
}
 ;
```