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

Marcello M. Bersani

LFC – Politecnico di Milano

Factorial operator

- Define tokens, syntax/semantic rules translating the factorial operator
  - $x = x \cdot (x-1) \cdot \ldots \cdot 2 \cdot 1$
  - $0! = 1$

Factorial operator

- Standard definition of a loop

```
r = 1;
n = x;

while (n>0){
  r = r * n;
  n = n - 1;
}

return r;
```

```
r ← 1
n ← x + 0
```

```
Lbody:
```

```
be Lend
r ← r * n
n ← n – 1
bgt Lbody
```

```
Lend:
```

Factorial syntactic

exp : …

exp NOT_OP { … }

- Factorial is an unary operator
  - Unique interpretation [no need for assoc]
  - Only precedence matters
    - NOT_OP is the strongest op, we are done!

Factorial operator

- Standard definition of a loop

```
int r_reg = gen_load_immediate(program, 1);
int i_reg = getNewRegister(program);

if ($1.expression_type == IMMEDIATE)
   gen_addi_instruction(…,i_reg, REG_0, $1.value);
else
   gen_add_instruction(…, i_reg, REG_0, $1.value, CG_DIRECT_ALL);

t_axe_label* l_end = newLabel(program);
gen_beq_instruction(program, l_end, 0);
t_axe_label* l_cond = assignNewLabel(program);
gen_mul_instruction(program, r_reg, r_reg, index_reg, CG_DIRECT_ALL);
gen_subi_instruction(program, index_reg, index_reg, 1);
gen_bgt_instruction(program, l_cond,0);
assignLabel(program, l_end);
$$ = create_expression(r_reg, REGISTER);
```

```
r ← 1

n ← x + 0

Lbody:

be Lend

r ← r * n
n ← n – 1
bgt Lbody

Lend:
```

Absolute value operator

- Define tokens, syntax/semantic rules translating the absolute value operator
  - |x| == x if x is *positive*
  - |x| == -x if x is *negative*

Absolute value operator

- Only multiply x by -1 is x is *negative* (non pos)

```
If (x > 0)
   return r;
Else
  return –r;
```

```
r ← x + 0
```

```
bpt Lend
r ← r * (-1)
```

```
Lend:
```

Absolute value syntactic

exp : …

OR_OP exp OR_OP { … }

- Reuse OR_OP
  - Left associativity
  - |v|+1 generates syntax error (PLUS > OR_OP)

Absolute value syntactic

exp : …

OR_OP exp OR_OP **%prec NOT_OP** { … }

- The rule is associated with the highest precedence

Absolute value operator

- Standard definition of a loop

```
exp : …
      |OR_OP exp OR_OP{

if ($2.expression_type == IMMEDIATE)
  $$ = create_expression($2.value > 0 ? $2.value : - $2.value, IMMEDIATE);
else {
  int r_reg = getNewRegister(program);
  gen_add_instruction(…r_reg, REG_0, $2.value, CG_DIRECT_ALL);

  t_axe_label* l_end = newLabel(program);
  gen_bpl_instruction(program, l, 0);
  gen_muli_instruction(program, r_reg, r_reg, -1);
  assignLabel(program, l_end);
  $$ = create_expression(r_reg, REGISTER);
}
```

r ← x + 0

bpl Lend

r ← r * -1

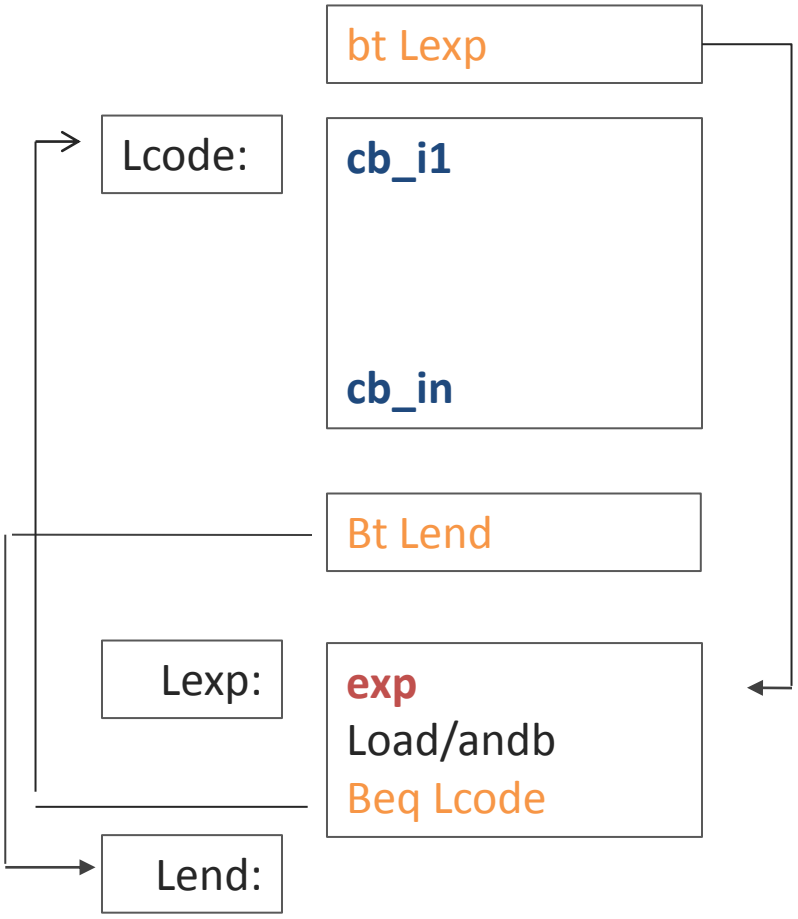
Lend:

Eval-unless statement

- Eval-unless construct allows conditional execution of the code block. If the expression after **unless** evaluates to false, the code block is executed otherwise the code block is skipped.

```
eval {
   y = 1;
} unless x==5;
```

Eval-unless statement



eval{

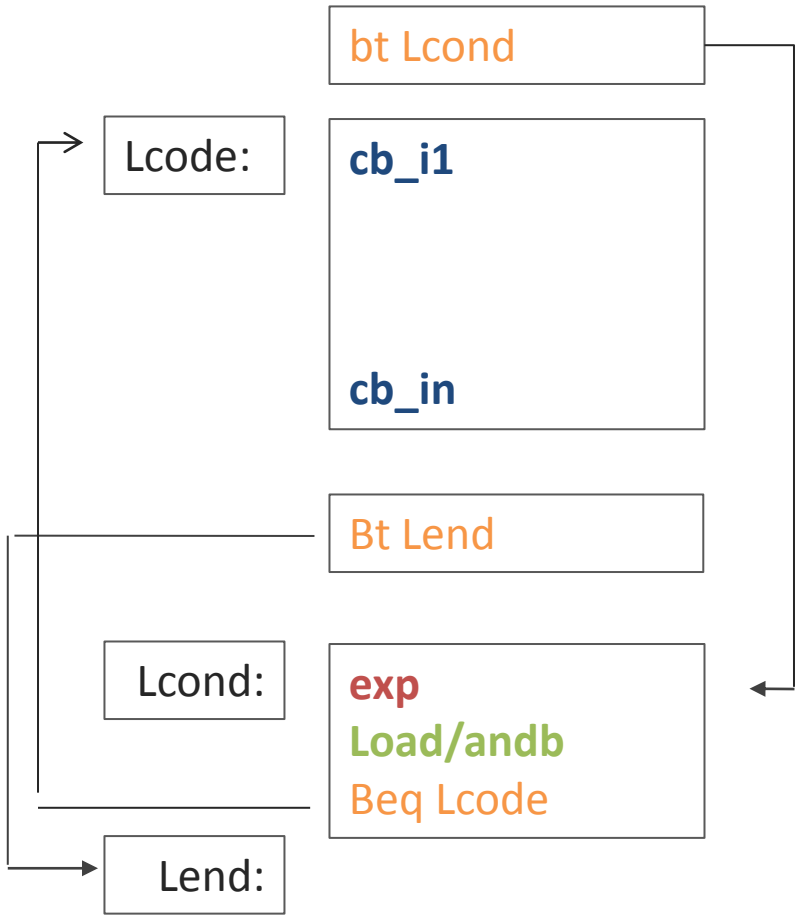
code_block

}unless **exp**

Eval-unless statement

```
unless_statement : EVAL
{
  $1.label_condition = newLabel(…);
  gen_bt_instruction(…, $1.label_condition, 0);

  $1.label_code = newLabel(…);
  assignLabel(…, $1.label_code);
}
code_block
{
  $1.label_end = newLabel(…);
  gen_bt_instruction (…, $1.label_end, 0);
} UNLESS
{
  assignLabel(…, $1.label_condition);
}
exp
{
  if ($7.expression_type == IMMEDIATE)
    gen_load_immediate(…, $7.value);
  else
    gen_andb_instruction(…, $7.value, $7.value, $7.value, CG_DIRECT_ALL);
  gen_beq_instruction (…, $1.label_code, 0);
  assignLabel(…, $1.label_end);
}
```



Eval-unless statement

- Descriptor

```
typedef struct t_unless_stmt {
   t_axe_label *label_condition;
   t_axe_label *label_code;
   t_axe_label *label_end;
} t_unless_stmt;
```

Eval-unless statement

```
%union {
  ...
  t_unless_stmt unless_stmt;
}

%token UNLESS
%token <unless_stmt> EVAL
```

Define directive

- Define the **define** instruction associating an identifier with an immediate value.

```
define ANSWER 42;
define QUESTION 9;
int x;
read( x );
x = ANSWER * x;
write( x );
```

Define syntactic

program : **macro_defs** var_declarations statements;

macro_defs : macro_defs macro_def {}
| {};

macro_def : DEFINE IDENTIFIER NUMBER SEMICOLON;

declaration : . . .
| IDENTIFIER ASSIGN IDENTIFIER
| IDENTIFIER LSQUARE IDENTIFIER RSQUARE
;

Define data structure

- Descriptor of a definition

```
typedef struct {
   char *id;
   int val;
} DATA;
```

```
define ANSWER 42;
```

- List to store all the descriptor of a definition

```
%{
  ...
  t_list macros;
%}
```

Define directive

```
macro_def : DEFINE IDENTIFIER NUMBER SEMICOLON
{
   DATA *p;
   p = getMacro( $2 );
   if (p) {
      /* boom! */
   }
   p = malloc( sizeof(DATA) );
   addList( &macros, p );
   p->id = $2;
   p->val = $3;
}
;
```

Define directive

```
declaration : IDENTIFIER ASSIGN IDENTIFIER
{
   DATA *p = getMacro( $3 );
   if (! p) {
     /* boom! */
   }
   $$ = alloc_declaration( $1, 0, 0, p->val );
}
| IDENTIFIER LSQUARE IDENTIFIER RSQUARE
{
   DATA *p = getMacro( $3 );
   if (! p) {
     /* boom*/
   }
   $$ = alloc_declaration( $1, 1, p->val, 0 );
}
;
```

Define directive

```
exp : IDENTIFIER
{
   DATA *p = getMacro( $1 );
   if (! p) {
     /* Code of the old action */
   } else {
     $$ = create_expression( p->val, IMMEDIATE );
   }
}
| NOT_OP IDENTIFIER
{
   DATA *p = getMacro( $2 );
   if (! p) {
     /* Code of the old action */
    }
    $$ = create_expression( p->val == 0, IMMEDIATE );
}
;
```