Formal Languages and Compilers

ACSE: Assignments and Expressions

Alessandro Barenghi & Michele Scandale

January 8, 2020

ACSE: Statements

A LANCE source file is split into two sections:

- variable declarations already discussed
- a list of statements

The program logic is described as a sequence of actions (formally statements) that must be executed in order to compute the program results.

Statements can be classified as:

sequential the control flow is linear, statements are executed in order

conditional the control flow is split according to a condition value into different paths which will join later on

iterative the control flow is cyclic

LANCE: Statements

Looking at the ACSE grammar:

```
statements   : statements statement
             | statement
             ;

statement    : assign_statement SEMI
             | control_statement
             | read_write_statement SEMI
             | SEMI { gen_nop_instruction(program); }
             ;
```

- `control_statement` represents the set of statements of LANCE where the control flow is modified (i.e. conditional and iterative)
- `assign_statement`, `read_write_statement` are sequential statements

ACSE: Assign statement

An **assign statement** represents the action of changing the value of a variable or an array element.

```
int a, v[10];

a = 4;
v[5] = a + v[1] - 1;
```

An assignment is composed of:

left-hand side: the modified element (e.g., `a`, `v[5]`)

right-hand side: expression computing the value to be assigned (e.g., `4`, `a + v[1] - 1`)

Generally type compatibility between *RHS* and *LHS* is required through:

- RHS having the same type of LHS
- implicit casts or promotion rules

Important: in ACSE there is only **one** type, so type checking is not required!

ACSE: Assign statement

The grammar rules for the assign statement are:

```
assign_statement : IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp
                   {
                     storeArrayElement(program, $1, $3, $6);
                     free($1);
                   }
                 | IDENTIFIER ASSIGN exp
                   {
                     // ...
                   }
                 ;
```

Important: in LANCE only assignments to *scalars* are allowed.

- The RHS is always an expression (nonterminal exp)
- A specific syntactic rule for each LHS (scalar or vector cell) is needed

ACSE: Expressions

A good part of the LANCE is dealt with employing expressions:

- RHSs of assignments
- arrays indexing
- control statements conditions

An expression is internally represented by the `t_axe_expression` structure (see `axe_struct.h`):

```
typedef struct t_axe_expression {
  int value;
  int expression_type;
} t_axe_expression;
```

ACSE: Expressions

Framework

The expression framework:

- allows to combine expressions together
- emits the code computing expressions
- is described in `axe_expression.h`

The structure is by nature recursive:

- two base cases: `IMMEDIATE` (numeric constant) and `REGISTER` (variables)
- temporary values are kept into `REGISTER` expressions
- base constructor `create_expression(int value, int expr_type)`

Helper functions for expression manipulation are:

- `handle_bin_numeric_op` – arithmetic, shifts, logic and bitwise operations (+, -, *, /, <<, >>, &&, ||, &, |)
- `handle_binary_comparison` – relational operations (==, !=, >, <, >=, <=)

ACSE: Expressions

Framework

The two functions **handle_bin_numeric_op** and **handle_binary_comparison** are operators for expressions:

```
t_axe_expression handle_bin_numeric_op(t_program_infos *program,
                                       t_axe_expression exp1,
                                       t_axe_expression exp2,
                                       int binop);
t_axe_expression handle_binary_comparison(t_program_infos *program,
                                          t_axe_expression exp1,
                                          t_axe_expression exp2,
                                          int condition);
```

Important: in these function is implemented a simple optimization named *constant folding*, thus the result of an operation over two immediate expression is still an immediate expression computed at **compile time**, no instructions are emitted!

ACSE: Expressions

Framework

```
exp : NUMBER      { $$ = create_expression ($1, IMMEDIATE); }
    | IDENTIFIER
      { $$ = create_expression(get_symbol_location(program, $1, 0).
                               REGISTER);
        free($1); }
    | IDENTIFIER LSQUARE exp RSQUARE
      { $$ = create_expression (loadArrayElement(program, $1, $3), REGISTER);
        free($1); }
    | exp PLUS exp    { $$ = handle_bin_numeric_op(program, $1, $3, ADD); }
    | exp MINUS exp   { $$ = handle_bin_numeric_op(program, $1, $3, SUB); }
    | exp MUL_OP exp  { $$ = handle_bin_numeric_op(program, $1, $3, MUL); }
    | exp DIV_OP exp  { $$ = handle_bin_numeric_op(program, $1, $3, DIV); }
    | exp SHL_OP exp  { $$ = handle_bin_numeric_op(program, $1, $3, SHL); }
    | exp SHR_OP exp  { $$ = handle_bin_numeric_op(program, $1, $3, SHR); }
    | exp ANDAND exp  { $$ = handle_bin_numeric_op(program, $1, $3, ANDL); }
    | exp OROR exp    { $$ = handle_bin_numeric_op(program, $1, $3, ORL); }
    | exp AND_OP exp  { $$ = handle_bin_numeric_op(program, $1, $3, ANDB); }
    | exp OR_OP exp   { $$ = handle_bin_numeric_op(program, $1, $3, ORB); }
    | exp LT exp      { $$ = handle_binary_comparison(program, $1, $3, _LT_); }
    | exp GT exp      { $$ = handle_binary_comparison(program, $1, $3, _GT_); }
    | exp EQ exp      { $$ = handle_binary_comparison(program, $1, $3, _EQ_); }
    | exp NOTEQ exp   { $$ = handle_binary_comparison(program, $1, $3, _NOTEQ_); }
    | exp LTEQ exp    { $$ = handle_binary_comparison(program, $1, $3, _LTEQ_); }
    | exp GTEQ exp    { $$ = handle_binary_comparison(program, $1, $3, _GTEQ_); }
    | /* ... */
    ;
```

ACSE: Expressions

Unboxing

Depending on the **expression_type** field, the **value** field can represent:

- immediate: the value of the immediate
- register: the identifier of the register that **at runtime** will contain value of the expression

The manipulation of a **t_axe_expression** always requires to handle the two cases:

```
assign_statement : IDENTIFIER LSQUARE exp RSQUARE ASSIGN exp
                   {
                     // ...
                   }
                 | IDENTIFIER ASSIGN exp
                   {
                     int dest = get_symbol_location(program, $1, 0);
                     if ($3.expression_type == IMMEDIATE)
                       gen_addi_instruction(program, dest,
                                            REG_0, $3.value);
                     else
                       gen_add_instruction(program, dest, REG_0,
                                           $3.value, CG_DIRECT_ALL);
                     free($1);
                   }
                 ;
```

Class task I

Let's extend the `ACSE` compiler in order to introduce the **modulo** operator.

```
int a;
read(a);
write(a % 5); // it prints the remainder of the division by 5
```

Important: the target ISA does not have a machine instruction that implements the modulo operation.

Class task II

Let's extend the `ACSE` compiler in order to introduce the **circular shift** operators.

```
int a = 43981; // 0xABCD
int b = 10;

write(a >>> 4); // it prints 0xD0000ABC
b = - b; // -10 == 0xFFFFFFF6
write(b <<< 2); // it prints -37 == 0xFFFFFFDB
```

The rotate operators (<<<, >>>) are similar to shifts: bits are moved in a circular fashion instead of linearly.

Hints: think how to split the bit-sequence so that the operation is equivalent to a swap of the two subsequences.

Class task III

Let's extend the ACSE compiler in order to introduce the **implicit variable**.

```
int a;
read(a);
// the result is assigned to the '$implicit' variable
a * a + 2 * a - 5;
write($implicit);
```

An expression can be a statement whose semantic is the assignment of the expression to the *implicit* variable.