Formal Languages and Compilers Laboratory

ACSE: Expressions and Arrays

Daniele Cattaneo

Material based on slides by Alessandro Barenghi and Michele Scandale

Contents

Expressions

In the LANCE language, *expressions* appear in many places:

- Right-hand-side (RHS) of assignments
- Array indices
- Conditions in control statements

Almost all operators that are present in the C language are supported:

- Basic arithmetic (+, -, *, /)
- Bitwise operators (&, |, <<, >>)
- Logical operators (&&, ||, !)
- Comparison operators (!=, ==, >, <, >=, <=)

Grammar of expressions

```
exp :   NUMBER
     |  IDENTIFIER
     |  NOT_OP exp
     |  exp AND_OP exp
     |  exp OR_OP exp
     |  exp PLUS exp
     |  exp MINUS exp
     |  exp MUL_OP exp
     |  exp DIV_OP exp
     |  exp LT exp
     |  exp GT exp
     |  exp EQ exp
     |  exp NOTEQ exp
     |  exp LTEQ exp
     |  exp GTEQ exp
     |  exp SHL_OP exp
     |  exp SHR_OP exp
     |  exp ANDAND exp
     |  exp OROR exp
     |  LPAR exp RPAR
     |  MINUS exp
```

```
NOT_OP   !
AND_OP   &
OR_OP    |
PLUS     +
MINUS    -
MUL_OP   *
DIV_OP   /
LT       <
GT       >
EQ       ==
NOTEQ    !=
LTEQ     <=
GTEQ     >=
SHL_OP   <<
SHR_OP   >>
ANDAND   &&
OROR     ||
LPAR     (
RPAR     )
```

Operator precedences

The expression grammar of LANCE is the same as the example infix expression grammar we have seen when we discussed *bison*

Of course we need to declare operator precedence and associativity:

```
%left OROR
%left ANDAND
%left OR_OP
%left AND_OP
%left EQ NOTEQ
%left LT GT LTEQ GTEQ
%left SHL_OP SHR_OP
%left MINUS PLUS
%left MUL_OP DIV_OP
%right NOT_OP
```

Same as C, **bugs** included

- Operators & and | have **LOWER** priority than comparisons!

One more bug in the grammar

The LANCE grammar supports **unary minus syntax** for negation:

$$\begin{array}{rll} \textit{exp} : & \ldots \\ | & \text{MINUS } \textit{exp} \end{array}$$

But there’s a problem:

- MINUS is **left associative** and has the **same priority** as PLUS
- This is correct for the normal **subtraction** operator
- But it is not correct for **negation**

Expression	Normal interpretation	LANCE interpretation
`- 1 * 2 - 3`	`((- 1) * 2) - 3`	`(- (1 * 2)) - 3`

At the exam, don’t fall into the trap of forgetting how LANCE mis-interprets unary minus!

This bug can actually be fixed, look at the *bison* bonus slides to learn how. However, the reason it’s **not** fixed is lost in the mists of time.

Semantic Actions: Basics

Remember the **expression compiler examples** we have already seen:

- Semantic value of *exp*: **register identifier**
- It's the register where we place **the intermediate result** of that subexpression

Up to now we have only said that **registers are associated to variables**...

- ...but in the intermediate language can only represent operations between 2 registers at most!
- For more complex expressions we need some **temporary registers** where to place **intermediate results**
- These **temporary registers** are **NOT** associated to variables
- In other words, we need more registers than variables

Temporary registers

LANCE Expression

```
a + b * c / 15
```

Intermediate Representation

```
MUL  R4 R2 R3
ADDI R5 R0 #15
DIV  R6 R4 R5
ADD  R7 R1 R6
```

Registers containing variables:

- R1 associated with a
- R2 associated with b
- R3 associated with c

Temporary registers:

- R4 = b * c
- R5 = 15
- R6 = b * c / 15
- R7 = a + b * c / 15

Temporary registers

ACSE provides an easy way to retrieve a **register identifier never before seen in the translated intermediate representation** to use as a temporary register:

```
/* Get a register still not used. */
int getNewRegister(t_program_infos *program)
{
   int result;
   result = program->current_register;
   program->current_register++;
   return result;
}
```

Usage and implementation are super-simple!

Temporary registers

A word of caution!

Retrieving a new temporary register **does not generate any code**

- Book-keeping of register identifiers is done **purely at compile-time**
- In theory, in the intermediate language **all infinite registers already exist *a priori***
- The only thing we are doing is **deciding which register to use in the generated code** out of the infinite ones

In fact, *getNewRegister()* does not add anything to the list of instructions: as a result it certainly does not generate any code!

Basic code generation for expressions

Constants Generate code to put the constant in a new temporary register

Variables Reuse the register associated to the variable

Parenthesis Reuse the register identifier of the subexpression

```
exp : NUMBER
    {
       $$ = getNewRegister(program);
       gen_addi_instruction(program, $$, REG_0, $1);
    }
    | IDENTIFIER
    {
       $$ = get_symbol_location(program, $1, 0);
       free($1);
    }
    | LPAR exp RPAR
    {
       $$ = $2;
    }
    | /* ... */
```

gen_load_immediate()

The pattern of generating code to put a constant in a register is very common:

gen_load_immediate() Put a constant in a **new** register

gen_move_immediate() Put a constant in **any** register

Putting a constant in a register is sometimes called **materialization**

```
void gen_move_immediate(t_program_infos *program, int dest, int imm)
{
   gen_addi_instruction(program, dest, REG_0, imm);
}

int gen_load_immediate(t_program_infos *program, int imm)
{
   int imm_register;

   imm_register = getNewRegister(program);
   gen_move_immediate(program, imm_register, imm);

   return imm_register;
}
```

Code generation of operators

Many operators directly correspond to MACE instructions:

```
exp : /* ... */
    | exp PLUS exp
    {
       $$ = getNewRegister(program);
       gen_add_instruction(program, $$, $1, $3, CG_DIRECT_ALL);
    }
    | exp AND_OP exp
    {
       $$ = getNewRegister(program);
       gen_andb_instruction(program, $$, $1, $3, CG_DIRECT_ALL);
    }
    | exp OROR exp
    {
       $$ = getNewRegister(program);
       gen_orl_instruction(program, $$, $1, $3, CG_DIRECT_ALL);
    }
    | /* ... */
```

Code generation of comparisons

Comparison operators require more complex code sequences using the **set instructions**:

- They come in variants like branch instructions:
  `SLT, SGT, SEQ, SNE, SLE, SGE`
- They implement the same logic conditions as branching
- Instead of branching they **set a register to zero or one**

```
exp : /* ... */
    | exp LT exp
    {
       $$ = getNewRegister(program);
       gen_sub_instruction(program, REG_0, $1, $3, CG_DIRECT_ALL);
       gen_slt_instruction(program, $$);
    }
    | /* ... */
```

More or less, for expressions **that is it!**

An optimization: Constant Folding

The semantic actions we have seen would be enough to completely implement expressions...

But ACSE does not stop there!

- ACSE implements one optimization: **constant folding**
- Instead of computing **constant expressions** at runtime, it computes them at **compile time**!
- This makes the program faster at the expense of some extra work in the compiler

Before constant folding

```
a + 3 * 4 - 2 + c
```

$\Downarrow$

After constant folding

```
a + 10 + c
```

The role of compilers and optimizations

Up until now we have said that **the compiler shall not execute the statements in the program**...

But **constant folding** means we **ARE** in fact executing **SOME PARTS** of statements at **compile time**

When it is allowed to do things at compile time **anyway**?

- When doing something at compile time **does not change the behavior of the program in any observable circumstance**
  - The process of computing an expression value is **invisible** to the LANCE programmer, this is why we can play with it
  - Whether we have constant folding or not, the compiled program **works in the same way**
- For real-world programming languages there are specification documents that detail what is considered “observable”

How to implement constant folding

The idea is to have a **double meaning** for the semantic value of *exp*:

- Constant integer
- Register identifier

Then, in each action, we check the operands:

- Are both constants?
  1. Compute the operation at compile time
  2. Result expression: another constant
- Is at least one of them not a constant?
  1. If there’s a constant, materialize it into a register
  2. **Generate code** which will compute the result at runtime
  3. Result expression: the register identifier which will hold the result

Double meaning for a semantic value

The obvious approach for implementing a double meaning for a semantic value is to define **a new type of semantic value**:

```
/* in axe_struct.h: */
typedef struct t_axe_expression {
   int value;
   int expression_type;  // IMMEDIATE or REGISTER
} t_axe_expression;

/* in Acse.y: */
%union {
   /* ... */
   t_axe_expression expr;
   /* ... */
}

%type <expr> exp
```

The **expression_type** element decides the meaning of **value**:

- Constant integer when *IMMEDIATE*
- Register identifier when *REGISTER*

create_expression()

To set the semantic value of *exp* just assign the members of *t_axe_expression*:

```
exp : /* ... */
    {
       $$.expression_type = /* IMMEDIATE or REGISTER */
       $$.value = /* the constant or the register ident. */
    }
    /* ... */
```

Easier way: use the helper function *create_expression()*

```
t_axe_expression create_expression(int value, int type)
{
   t_axe_expression expression;

   expression.value = value;
   expression.expression_type = type;

   return expression;
}
```

Expressions with constant folding

Let's go back to the semantic actions we have just seen...

Constants Constant expression, don't materialize the value

Variables Register expression with the variable's register

Parenthesis No changes

```
exp : NUMBER
    {
      $$ = create_expression($1, IMMEDIATE);
    }
    | IDENTIFIER
    {
      int location = get_symbol_location(program, $1, 0);
      $$ = create_expression(location, REGISTER);
      free($1);
    }
    | LPAR exp RPAR
    {
      $$ = $2;
    }
    | /* ... */
```

Code generation of operators

For operators we have to check the expression type of the operands:

```
exp: exp PLUS exp
  {
    if ($1.expression_type==IMMEDIATE && $3.expression_type==IMMEDIATE)
    {
      $$ = create_expression($1.value + $3.value, IMMEDIATE);
    }
    else
    {
      int r1, r2, rres;

      if ($1.expression_type == IMMEDIATE)
         r1 = gen_load_immediate(program, $1.value);
      else
         r1 = $1.value;

      if ($3.expression_type == IMMEDIATE)
         r2 = gen_load_immediate(program, $3.value);
      else
         r2 = $3.value;

      rres = getNewRegister(program);
      gen_add_instruction(program, rres, r1, r2, CG_DIRECT_ALL);
      $$ = create_expression(rres, REGISTER);
    }
  }
```

Yet another helper function...

Copy-pasting the semantic action we have just seen for every operator is stupid!

ACSE consolidates the logic for handling constant folding in two helper functions defined in **axe_expressions.h**:

handle_bin_numeric_op()	Arithmetic and logical operations
handle_binary_comparison()	Comparisons

```
/* Valid values for `binop' are:                              *
 * ADD   SUB   MUL   DIV   ANDL  ORL   EORL                   *
 * ANDB  ORB   EORB  SHL   SHR                                */
t_axe_expression handle_bin_numeric_op(t_program_infos *program,
                                       t_axe_expression exp1,
                                       t_axe_expression exp2,
                                       int binop);

/* Valid values for `condition' are:                          *
 * _LT_  _GT_  _EQ_  _NOTEQ_  _LTEQ_  _GTEQ_                  */
t_axe_expression handle_binary_comparison(t_program_infos *program,
                                          t_axe_expression exp1,
                                          t_axe_expression exp2,
                                          int condition);
```

Operators: final version

We can rewrite a **third** time the semantic actions for binary operators:

```
exp: /* ... */
   | 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 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 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_); }
   | 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); }
   | /* ... */
```

Non-binary operators are handled without using helper functions.

Contents

Assignments

Now that we have seen expressions we can take a closer look at **assignments**.

statement :	*assign_statement* SEMI
	…
assign_statement :	IDENTIFIER ASSIGN *exp*

Since the right-hand-side of an assignment is an **expression**, there are two cases:

1. The expression is constant
   - The constant must be materialized to assign it
2. The expression is a register
   - Generate code to copy the expression's value register into the variable's register

Semantic action for assignments

Remember: every variable is associated with a register...

```
assign_statement: IDENTIFIER ASSIGN exp
   {
      int location;
      location = get_symbol_location(program, $1, 0);

      if ($3.expression_type == IMMEDIATE)
         gen_move_immediate(program, location, $3.value);
      else
         gen_add_instruction(program,
            location, REG_0, $3.value, CG_DIRECT_ALL);

      free($1);
   }
;
```

Checking if an expression is constant (IMMEDIATE) and materializing it is **very common**

The write statement

```
statement             : /* ... */
                      | read_write_statement SEMI
                      | /* ... */
;

read_write_statement : /* ... */
                      | write_statement
;

write_statement: WRITE LPAR exp RPAR
   {
      int location;

      if ($3.expression_type == IMMEDIATE)
         location = gen_load_immediate(program, $3.value);
      else
         location = $3.value;

      gen_write_instruction(program, location);
   }
;
```

Contents

Constant folding for free

Sometimes you want to expand ACSE with a new statement which could be rewritten as an expression, and which must perform **constant folding** if possible

- A statement that does something that in the language is already possible is called **syntactic sugar**

There are two approaches:

1. Write the code to generate instructions only as long as the result is **not** constant
   - For some tasks it's the only choice
   - Tends to be laborious!
2. Mis-use the expression helper functions to simulate a *real* expression

Example: FMA statement

Let's look back at the first ACSE-related homework:

Exercise

Implement the FMA statement: `fma(a, b, c)`
Equivalent to `a = a * b + c;`

If all three arguments are variable names, the solution would be this:

Solution

```
fma_statements:
  FMA LPAR IDENTIFIER COMMA IDENTIFIER COMMA IDENTIFIER RPAR
  {
    int r_v1 = get_symbol_location(program, $3, 0);
    int r_v2 = get_symbol_location(program, $5, 0);
    int r_v3 = get_symbol_location(program, $7, 0);
    gen_mul_instruction(program, r_v1, r_v1, r_v2, CG_DIRECT_ALL);
    gen_add_instruction(program, r_v1, r_v1, r_v3, CG_DIRECT_ALL);
    free($3);
    free($5);
    free($7);
  }
;
```

Example: FMA statement

Let's modify the statement to take two expressions as the last parameters:

Solution with expressions

```
fma_statement:
  FMA LPAR IDENTIFIER COMMA exp COMMA exp RPAR
  {
    int r_v1 = get_symbol_location(program, $3, 0);

    /* Materialization of constants */
    int r_v2, r_v3;
    if ($5.expression_type == IMMEDIATE)
      r_v2 = gen_load_immediate(program, $5.value);
    else
      r_v2 = $5.value;
    if ($7.expression_type == IMMEDIATE)
      r_v2 = gen_load_immediate(program, $7.value);
    else
      r_v2 = $7.value;

    gen_mul_instruction(program, r_v1, r_v1, r_v2, CG_DIRECT_ALL);
    gen_add_instruction(program, r_v1, r_v1, r_v3, CG_DIRECT_ALL);
    free($3);
  }
;
```

Example: FMA statement

We can use *handle_bin_numeric_op()* to skip manual materialization:

Solution with expressions

```
fma_statement:
  FMA LPAR IDENTIFIER COMMA exp COMMA exp RPAR
  {
    int r_v1 = get_symbol_location(program, $3, 0);
    t_axe_expression e_v1 = create_expression(r_v1, REGISTER);

    t_axe_expression e_mul =
        handle_bin_numeric_op(program, e_v1, $5, MUL);
    t_axe_expression e_add =
        handle_bin_numeric_op(program, e_mul, $7, ADD);

    /* At this point we don't need to check the expression_type of
     * e_add because it already depends on e_v1 which is a REGISTER
     * expression */
    gen_add_instruction(program, r_v1, REG_0, e_add.value, CG_DIRECT_ALL);
    free($3);
  }
;
```

Example: FMA operator

The case in which *FMA* is an **operator** is even simpler, and will perform constant folding for you:

FMA operator: solution with expressions

```
exp: /* ... */
  | FMA LPAR exp COMMA exp COMMA exp RPAR
  {
    t_axe_expression e_mul =
        handle_bin_numeric_op(program, $3, $5, MUL);
    $$ = handle_bin_numeric_op(program, e_mul, $7, ADD);
  }
;
```

To conclude this topic:

- You can use *handle_bin_numeric_op()* and *handle_binary_comparison()* to easily create statements that are **sintactic sugar** over a given expression
- In some scenarios this requires caution...
    - Sometimes you want code to be generated regardless of constants

Contents

Arrays vs scalars

Remember that when we talked about **definitions** we talked about **arrays**...

- ... and then we promptly forgot about them!
- I consciously did that *for you* to simplify things

Let us put things straight and **add arrays to ACSE**!

Where do we need to make modifications?

- Expressions
- Assignments

That's it! But first we need to know **how to access arrays**...

Accessing arrays

Arrays have a crucial difference with respect to variables:

- They are stored in **memory**
- Their memory location is identified by a **label**

Remember: labels are **identifiers of constant pointers**

- We are in the compiler and as a result we don't know the address
- But we can still use the label to refer to the location of an array
- On the contrary, the **offset** of each array item from the start of the array is known!

In the IR the **MOVA** instruction loads an address to a register...

1. Load the address into a register
2. Add the offset of the desired element to the address
3. Use **indirect addressing** to read/write to the array element

Accessing arrays

Let's look at how a simple program with array accesses is translated into the intermediate representation:

- The label pointing to the array's memory is `_array`

LANCE input

```
int array[10];

array[5] = 10;

write(array[3]);
```

Instruction list

```
ADDI R1 R0 #10
MOVA R2 _array
ADDI R2 R2 #5
ADD (R2) R0 R1

MOVA R3 _array
ADDI R3 R3 #3
ADD R4 R0 (R3)
WRITE R4 0
```

Helper functions to access arrays

Writing the code for generating this sequence of instruction every time you need a statement to access arrays is cumbersome...

- ACSE provides two **primitive helper functions** that allow generating array accesses quickly and easily
- They are defined in **axe_array.h**

```
int loadArrayElement(t_program_infos *program,
                                char *ID,
                     t_axe_expression  index);

void storeArrayElement(t_program_infos *program,
                                char *ID,
                     t_axe_expression  index,
                     t_axe_expression  data);
```

Helper functions to access arrays

```
int loadArrayElement(t_program_infos *program,
                                    char *ID,
                          t_axe_expression  index);

void storeArrayElement(t_program_infos *program,
                                     char *ID,
                           t_axe_expression  index,
                           t_axe_expression  data);
```

The arguments:

ID The variable identifier of the array

index A constant or a reg. ID with the subscript to access

data A constant or a reg. ID with the value to put in the array (*storeArrayElement()* only)

loadArrayElement() returns the **identifier of the register** which will contain the value read from the array element.

Helper functions to access arrays

Of course these functions are **not magic**:

- They simply generate the code we have seen in the example
- Let's look at the implementation of *loadArrayElement()*:*

```
int loadArrayElement(t_program_infos *program, char *ID, t_axe_expression index)
{
   /* Generate a load of the label's address into a register */
   t_axe_label *l_array = getLabelFromVariableID(program, ID);
   int r_addr = getNewRegister(program);
   gen_mova_instruction(program, r_addr, l_array, 0);

   /* Generate computation of the desired element's address */
   if (index.expression_type == IMMEDIATE)
      gen_addi_instruction(program, r_addr, r_addr, index.value);
   else
      gen_add_instruction(program, r_addr, r_addr, index.value, CG_DIRECT_ALL);

   /* Generate a load of the array element into the result register */
   int r_elem = getNewRegister(program);
   gen_add_instruction(program, r_elem, REG_0, r_addr, CG_INDIRECT_SOURCE);
   return r_elem;
}
```

---

* *storeArrayElement()* is similar

Arrays in expressions

Now let's look at the semantic actions in ACSE for handling arrays, starting with expressions:

```
exp: /* ... */
   | IDENTIFIER LSQUARE exp RSQUARE {
       int reg;
       reg = loadArrayElement(program, $1, $3);
       $$ = create_expression(reg, REGISTER);
       free($1);
   }
```

When an array appears in an expression:

1. Generate a load the array element into a new register
2. The expression semantic value is that register identifier

Assignments to array

Assignments to arrays are even simpler:

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

storeArrayElement() is doing all the hard work for us!

Checking a variable's properties

Remember: inside ACSE, arrays and scalars are both kinds of **variables**!

When working with arrays it is sometimes necessary to check the properties of a variable:

- Verify if it's an array or not
- Check the size of the array

The function for retrieving this information: **getVariable()**

```
t_axe_variable *getVariable(
   t_program_infos *program,
              char *ID);
```

```
typedef struct t_axe_variable {
   char *ID;
   int type;
   int isArray;    // <- !
   int arraySize;  // <- !
   int location;
   t_axe_label *labelID;
} t_axe_variable;
```

Checking if a variable is an array

A common pattern: check if a given **identifier** is associated to an **array**:

```
char *the_id;

t_axe_variable *v_ident = getVariable(program, the_id);
if (!v_ident->isArray) {
   yyerror("The␣specified␣variable␣is␣not␣an␣array!");
   YYERROR;
}
```

Remember: *yyerror()* is the standard Bison function for signaling syntax errors. The *YYERROR* macro is what actually stops the syntactic action.*

*Actually, many exam solutions do not use the YYERROR macro, so you are exempted from using it as well

Contents

Homework 1/3

Extend the `ACSE` compiler in order to introduce the **modulo** operator.

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

Important: there is **no instruction** for computing the modulo!

But there's a trick... Remember these identities (memories of high school!) about **integer division**...

$$q = \frac{a}{b}$$
$$r = a - q \times b$$

Where q is called **quotient** and r is the **remainder**...

Homework 2/3

At the moment the **read** operator in LANCE can only be applied to **scalar variables**

Extend the ACSE compiler in order to introduce the **array read statement:**

```
int a[10];
read(a[5]);
```

Additionally, make sure that compilation stops with a **syntax error** when the array's identifier is not actually associated with an array variable.

Homework 3/3

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

```
int a;
read(a);

a * a + 2 * a - 5;
/* the result is assigned       *
 * to the '$implicit' variable */

write($implicit);
```

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

Important: the set of characters allowed in identifiers does not include the dollar sign (`$`)...