Formal Languages and Compilers Laboratory

ACSE: Building compilers with Bison and Flex

Condensed version

Daniele Cattaneo

Material based on slides by Alessandro Barenghi and Michele Scandale

Contents

ACSE: Advanced Compiler System for Education

ACSE is a simple compiler:

- accepts a C-like source language (called LANCE)
- emits a RISC-like assembly language (called MACE)

It comes with two other helper tools forming an entire toolchain:

asm Assembler (from assembly to machine code)

mace Simulator of the fictional MACE processor

In this course we will see only the ACSE compiler:

- the source is located in `acse` directory
- the code is simple and well documented
- there are a **lot** of helper functions to perform common operations

`LANCE`: LANguage for Compiler Education

`LANCE` is the source language recognized by `ACSE`:

- very small subset of C99
- standard set of arithmetic/logic/comparison operators
- reduced set of control flow statements (`while`, `do-while`, `if`)
- only one scalar type (`int`)
- only one aggregate type (array of `int`s)
- no functions

Very limited support for I/O operations:

- **`read`**`(var)` reads an int from standard input and stores it into `var`
- **`write`**`(expr)` writes `expr` to standard output

LANCE: Syntax

A LANCE source file is composed by two sections:

- variable declarations
- program body as a list of statements

```
int x, y, z = 42;
int arr[10];
int i;

read(x);
read(y);

i=0;
while (i < 10) {
  arr[i] = (y - x) * z;
  i = i + 1;
}
z = arr[9];
write(z);
```

Compilation Process

How does ACSE compile a LANCE file to MACE assembly?

Front-end:

1. The source code is **tokenized** by a **flex-generated scanner**
2. The stream of tokens is **parsed** by a **bison-generated parser**
3. The code is translated to a **temporary intermediate representation** by the **semantic actions in the parser**

Back-end:

4. The intermediate representation is **normalized** to **account for physical limitations of the MACE processor**
5. Each instruction is **printed out** producing the **assembly file**

Same **overall structure** as **more complex** compilers, other details are simplified.

Contents

Root rules

A LANCE source file is split into two sections:

1. Variable declarations; root non-terminal: *var_declarations*
   - We will skip describing these
2. List of statements; root non-terminal: *statements*

The basic grammar rules (expressed in BNF):

$$
\begin{aligned}
\textbf{\textit{program}} &\rightarrow \textit{var\_declarations statements} \dashv \\
\textit{var\_declarations} &\rightarrow \textit{var\_declarations var\_declaration} \\
&\quad \mid \varepsilon \\
\textit{var\_declaration} &\rightarrow \ldots \\
\textit{statements} &\rightarrow \textit{statements statement} \\
&\quad \mid \textit{statement} \\
\textit{statement} &\rightarrow \ldots
\end{aligned}
$$

What is a statement?

> *A **statement** is a syntactic unit of an imperative programming language that expresses some action to be carried out.*
>
> — *Wikipedia*

Statements can be classified as:

Simple Indivisible element of computation

- Assignments, *read*, *write*, …

Compound Statements which contain multiple simple statements

- if, while, do-while

Simple statements

$$
\begin{aligned}
\textit{statement} &\rightarrow \textit{assign\_statement}\ \text{SEMI}\\
&\mid \textit{control\_statement}\\
&\mid \textit{read\_write\_statement}\ \text{SEMI}\\
&\mid \boxed{\text{SEMI}}\\
\textit{control\_statement} &\rightarrow \textit{if\_statement}\\
&\mid \textit{while\_statement}\\
&\mid \textit{do\_while\_statement}\ \text{SEMI}\\
&\mid \textit{return\_statement}\ \text{SEMI}\\
\textit{return\_statement} &\rightarrow \text{RETURN}\\
\textit{assign\_statement} &\rightarrow \text{IDENTIFIER LSQUARE}\ \textit{exp}\ \text{RSQUARE ASSIGN}\ \textit{exp}\\
&\mid \text{IDENTIFIER ASSIGN}\ \textit{exp}\\
\textit{read\_write\_statement} &\rightarrow \textit{read\_statement}\\
&\mid \textit{write\_statement}\\
\textit{read\_statement} &\rightarrow \text{READ LPAR IDENTIFIER RPAR}\\
\textit{write\_statement} &\rightarrow \text{WRITE LPAR}\ \textit{exp}\ \text{RPAR}
\end{aligned}
$$

A semicolon by itself is the NOP statement

Notice:

- ***exp*** is a **generic expression**

Grammar of expressions

Expressions use **bisonprecedence/associativity**

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*

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

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, **weirdnesses** included

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

Bug in the expression grammar

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

$$
\begin{aligned}
\mathit{exp} \rightarrow\ & \ldots \\
\mid\ & \text{MINUS}\ \mathit{exp}
\end{aligned}
$$

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.

Grammar of compound statements

$$\boxed{\begin{array}{rl} \mathit{code\_block} \rightarrow & \mathit{statement} \\ & \mid \text{LBRACE } \mathit{statements} \text{ RBRACE} \end{array}}$$

$$\begin{array}{rl}
\mathit{control_statement} \rightarrow & \mathit{if_statement} \\
& \mid \mathit{while_statement} \\
& \mid \mathit{do_while_statement} \text{ SEMI} \\
& \mid \mathit{return_statement} \text{ SEMI} \\
\mathit{if_statement} \rightarrow & \mathit{if_stmt} \\
& \mid \mathit{if_stmt} \text{ ELSE } \mathit{code_block} \\
\mathit{if_stmt} \rightarrow & \text{IF LPAR } \mathit{exp} \text{ RPAR } \mathit{code_block} \\
\mathit{while_statement} \rightarrow & \text{WHILE LPAR } \mathit{exp} \text{ RPAR } \mathit{code_block} \\
\mathit{do_while_statement} \rightarrow & \text{DO } \mathit{code_block} \text{ WHILE LPAR } \mathit{exp} \text{ RPAR}
\end{array}$$

- The ***code_block*** non-terminal is used in all situations where we can have **a list of statements enclosed by braces** or in alternative **a single statement**.

Contents

Orienting inside ACSE

The core elements of ACSE compiler are:

- **scanner** flex source in `Acse.lex`
- **parser** bison source in `Acse.y`
- **codegen** instruction generation functions: `axe_gencode.h`

ACSE is a **syntax directed translator**:

- Produces the compiled output directly **while parsing**
- The order of the compiled instructions depends on the syntax!

`Acse.y` is the most important file in ACSE:

- Contains the (Bison-syntax) grammar of LANCE
- The semantic actions are responsible for the actual translation from LANCE to assembly

ACSE Intermediate Representation

The IR is the data representation used in a compiler to represent the program.

In ACSE it is composed of two main parts:

- The **instruction list**
- The **variable table**

The **semantic actions** modify these two data structures to build the compiled program.

The instructions in the instruction list are **assembly-like**

- Abstracts away all the syntactic details
- Makes later analysis of the program simpler

The program instance

The IR is stored in a **global structure** called **`program`**

- Declaration in `Acse.y`, at the very top
- It also contains other contextual information

```
typedef struct t_program_infos {
  t_list *variables;
  t_list *instructions;

  /* ...Other members not of
   *      interest here...    */

  int current_register;
} t_program_infos;

t_program_infos program;
```

Almost every function in ACSE takes `program` as an argument.

ACSE and MACE instructions

ACSE represents instructions in the program in a RISC-like fashion which is basically identical to the final output (the MACE assembly language)

- Modern compilers use IRs completely different than the target assembly language

Kinds of instructions:

- arithmetic and logic (e.g. `ADD`, `SUB`)
- memory access instructions (e.g. `LOAD`, `STORE`)
- conditional and unconditional branches (e.g. `BEQ`, `BT`)
- special I/O instructions (e.g. `READ`, `WRITE`)

Data storage:

- infinite registers
- infinite memory locations

Register identifiers

The **register identifier** is an integer value that represents **a given register in the bank of infinite registers**

The value of the register identifier is the **number of the register**

- Register `R0` has the register identifier `0`
- Register `R10` has the register identifier `10`
- Register R$\langle n \rangle$ has the register identifier n

ACSE uses these identifiers to represent the operands of each instruction.

Intermediate Representation

Register notes

There are two special registers:

zero `R0` contains the constant value `0`: writes are ignored

status word or PSW: implicitly read/written by arithmetic instructions

The *zero* register is useful to perform **constant value materialization** and copying values across registers:

```
ADDI R1 R0 #10    // put in R1 the constant 10
ADD  R2 R0 R3     // put in R2 the value in R3
```

The PSW register contains four single-bit **flags**, that are exploited mainly by conditional jump instructions:

- **N** negative
- **Z** zero
- **V** overflow
- **C** carry

Instruction Formats

There are 4 kinds of instructions:

Type	Operands	Example
Ternary	1 destination and 2 source registers	`ADD R3 R1 R2`
Binary	1 destination and 1 source register, and 1 immediate operand	`ADDI R3 R1 #4`
Unary	1 destination and 1 address operand	`LOAD R1 L0`
Jump	1 address operand	`BEQ L0`

Adding instructions into a program

ACSE provides a set of **functions** that **add** one instruction **to the end** of the current program:

```
// XXX Rdest Rsrc1 Rsrc2
// XXX = ADD SUB MUL DIV...
// flags = always CG_DIRECT_ALL
extern t_axe_instruction *gen_xxx_instruction(t_program_infos *program,
      int r_dest, int r_source1, int r_source2, int flags)

// XXX Rdest Rsrc1 #immediate
// XXX = ADDI SUBI MULI DIVI...
extern t_axe_instruction *gen_xxx_instruction(
      t_program_infos *program, int r_dest, int r_source1, int immediate)

// Bcc Label
extern t_axe_instruction *gen_bcc_instruction(
      t_program_infos *program, t_axe_label *label, int addr);
```

- r_dest, r_source1, r_source2 are register identifiers
- immediate is the constant that appears directly in the encoding of the instruction

Conditional jump instructions

There are 15 conditional jump instructions, some of them:

- **BT** Unconditional branch
- **BEQ** Branch if last result was zero
- **BNE** Branch if last result was not zero

How do they work:

1. Every arithmetic instruction modifies the PSW **depending on the result of the computation**:
   - **N** set to 1 if the result was negative (otherwise set to 0)
   - **Z** set to 1 if the result was zero (otherwise set to 0)
   - **V** set to 1 if an overflow occurred (otherwise set to 0)
   - **C** set to 1 if there was a carry* (otherwise set to 0)
2. When we arrive at a branch instruction the PSW is checked to decide whether to branch or not

*For italian speakers: “carry” è il riporto dell’addizione in colonna

Conditional jump instructions

Some branch instructions jump depending on a quite obvious condition:

Instruction	Branch condition	Logical test
BT	Always branch	1
BF	Never branch*	0
BPL	Branch if positive	$\neg$N
BMI	Branch if negative	N
BNE	Branch if not zero	$\neg$Z
BEQ	Branch if zero	Z
BVC	Branch if overflow clear	$\neg$V
BVS	Branch if overflow set	V
BCC	Branch if carry clear	$\neg$C
BCS	Branch if carry set	C

*Yes, nobody needs this

Conditional jump instructions

Some conditions are designed to allow numerical comparisons:

Inst.	Branch condition	Logical test
`BNE`	Br. if not equal ($\neq$)	$\neg$Z
`BEQ`	Br. if equal ($=$)	Z
`BGE`	Br. if greater or eq. ($\geq$)	(N $\wedge$ V) $\vee$ ($\neg$N $\wedge$ $\neg$V)
`BLT`	Br. if less than ($<$)	($\neg$N $\wedge$ V) $\vee$ (N $\wedge$ $\neg$V)
`BGT`	Br. if greater than ($>$)	(N $\wedge$ V $\wedge$ $\neg$Z) $\vee$ ($\neg$N $\wedge$ $\neg$V $\wedge$ $\neg$Z)
`BLE`	Br. if less or equal ($\leq$)	Z $\vee$ (N $\wedge$ $\neg$V) $\vee$ ($\neg$N $\wedge$ V)

For these instructions to work as advertised,
the last instruction before the branch **must** be a **subtraction**:

- Example:

```
SUBI R0 R2 #3
BLT L0
```

  branches to `L0` if R2 < 3

Another use for `R0`: discarding the result of a computation (the PSW is updated anyway)

Let’s look at an example of a compiled program, and try to read the intermediate language translation...

Input Program / **IR Code**

```
int x, y, z = 42;                    ADDI R1 R0 #0
                                     ADDI R2 R0 #0
                                     ADDI R3 R0 #42
int arr[10];
int i;                               ADDI R4 R0 #0

read(x);                             READ R1 0
read(y);                             READ R2 0
i=0;                                 ADDI R4 R0 #0
while (i < 10) {           L5        SUBI R5 R4 #10
                                     BGE L6
  arr[i] = (y - x) * z;              SUB R6 R2 R1
                                     MUL R7 R6 R3
                                     MOVA R8 _arr
                                     ADD R8 R8 R4
                                     ADD (R8) R0 R7
  i = i + 1;                         ADDI R4 R4 #1
}                                    BT L5
z = arr[9];                L6        MOVA R10 _arr
                                     ADDI R10 R10 #9
                                     ADD R3 R0 (R10)
write(z);                            WRITE R3 0
                                     HALT
```

Var. Table

Name	Reg.	Size	Lab.
x	R1	–	–
y	R2	–	–
z	R3	–	–
arr	–	10	_arr
i	R4	–	–

Contents

Assignment and Expressions

Now let's consider the **assignment statement**:

```
assign_statement: IDENTIFIER ASSIGN exp
  {
    /* ? */
  }
;
```

In assembly language, an assignment translates to two steps:

1. Compute the value to be assigned (described by the expression)
2. Store the value to the variable

The computation of the value is delegated to the **expression semantic actions**

1. Semantic actions of *exp* will write some code that puts the result of the expression in a **register**
2. Our semantic action will write the store to the variable

We need a way to identify which register contains the value!

Semantic Values in ACSE

The *exp* rule will tell us where the result of the expression is by modifying its **semantic value**

In `Acse.y` the **union declaration** defines the following types for semantic values:

```
%union {
   int intval;
   char *svalue;
   t_list *list;
   t_axe_expression expr;
   t_axe_label *label;
   ...
}
```

intval	Generic integer
svalue	Generic string
list	Generic linked list
expr	Intermediate expression value
label	Label in the generated assembly

Semantic Values in ACSE

The *exp* rule is of type **expr**, which corresponds to this struct declaration:

```
#define IMMEDIATE 0
#define REGISTER 1

typedef struct t_axe_expression {
   int value;           // a constant or a register identifier
   int expression_type; // IMMEDIATE or REGISTER
} t_axe_expression;
```

Two cases:

- Type == **REGISTER**: Value **identifies the register** which will contain the value at runtime
- Type == **IMMEDIATE**: Value **is the value** of the expression itself, therefore the expression is **constant** and could be computed at **compile time**

Assignment and Expressions

We can start sketching the outline of the code in our semantic action:

```
assign_statement: IDENTIFIER ASSIGN exp
  {
    if ($3.expression_type == REGISTER) {
      /* the value to store is in register R($3.value) */
    } else /* if ($3.expression_type == IMMEDIATE) */ {
      /* the value to store is itself $3.value */
    }
  }
;
```

When we have a *t_axe_expression* we **always** need to check its type first!

Now we need to check how to store a value in a variable.

Handling variables

In ACSE every **scalar** variable is stored in a register:

- Read a variable = Use the register
- Assign a variable = Set a value to the register
- This is **different** from what is shown in other courses (ACSO)

Function for retrieving this register:

```
int get_symbol_location(
   t_program_infos *program,
   char *ID,                  /* the variable name */
   int genLoad                /* always zero */
);
```

It returns an **integer**: the **register identifier**
It needs as input the **name (or identifier) of the variable**

We can get the identifier of the variable from the semantic value of IDENTIFIER

Tokens for variable identifiers

The token for variable identifiers is (fittingly) called `IDENTIFIER`:

- Semantic value: a C string with the name of the variable
- The string is **dynamically allocated** by the lexer
  - That means we need to **free** it in the semantic actions!

Acse.y

```
%union {
   ...
   char *svalue;
   ...
}

%token <svalue> IDENTIFIER
```

Acse.lex

```
ID    [a-zA-Z_][a-zA-Z0-9_]*
%%
{ID} {
        yylval.svalue =
          strdup(yytext);
        return IDENTIFIER;
     }
```

For those who don't know, `strdup()` is implemented like this →

```
char *strdup(char *s)
{
  char *new = malloc(strlen(s)+1);
  strcpy(new, s);
  return new;
}
```

Assignment and Expressions

Now we can complete the semantic action

To cause a change of the value of the register allocated to the variable we can generate an **ADD** or **ADDI** instruction:

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

    if ($3.expression_type == REGISTER) {
      gen_add_instruction(
          program, r_var, REG_0, $3.value, CG_DIRECT_ALL);
    } else {
      gen_addi_instruction(program, r_var, REG_0, $3.value);
    }

    free($1);
  }
;
```

Assignment to arrays

There is a second expansion for assignments that stores to arrays:

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

This one is shorter because the **storeArrayElement()** helper function does the hard work of writing the correct code for us.

Helper functions for 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.

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

How expressions work

In assembly (or the IR) we can only represent operations between 2 registers at most!

- For computing arbitrary expressions we need some **temporary registers** where to place **intermediate results**

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!

gen_load_immediate()

If we want our new register to be initialized with a constant we can use the *gen_load_immediate()* function.

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

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

   imm_register = getNewRegister(program);
   gen_addi_instruction(program, imm_register, REG_0, imm);

   return imm_register;
}
```

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!

Simple code generation of operators

An **initial** implementation of expressions might look like this:

```
exp : /* ... */
    | exp PLUS exp
    {
       if ($1.expression_type == REGISTER &&
             $3.expression_type == REGISTER) {

          $$.expression_type = REGISTER;
          $$.value = getNewRegister(program);
          gen_add_instruction(program,
             $$.value, $1.value, $3.value, CG_DIRECT_ALL);

       } else {
          ...
       }
    }
    | /* ... */
```

Handling constant folding

Now let's consider what we have to do to support **compile-time computation of constant expressions**.

All **variables** are considered **unknowns** at compile time. Therefore:

- $1 + 2$: known at compile time
- $1 + c$: unknown at compile time
- $b + 1$: unknown at compile time
- $b + c$: unknown at compile time

Two cases:

- Both operands are **IMMEDIATE**: result is **IMMEDIATE**
- Otherwise result is **REGISTER**

Code generation of operators

The simple code above becomes:

```
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 = getNewRegister(program);
         gen_addi_instruction(program, r1, REG_0, $1.value);
      } else {
         r1 = $1.value;
      }
      if ($3.expression_type == IMMEDIATE) {
         r2 = getNewRegister(program);
         gen_addi_instruction(program, r2, REG_0, $3.value);
      } else {
         r2 = $3.value;
      }
      rres = getNewRegister(program);
      gen_add_instruction(program, rres, r1, r2, CG_DIRECT_ALL);
      $$ = create_expression(rres, REGISTER);
    }
  }
```

Utility function for initializing a t_axe_exp.

We reduce the mixed reg/imm cases to just reg/reg

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);
```

Constants, variable accesses

Let's conclude with the definition of the root recursion rules of *exp*

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;
    }
    | /* ... */
```

Tokens for constant integers

The `NUMBER` token corresponds to **constant integers**:

- Semantic value: the integer value that appeared in the LANCE source code

Acse.y

```
%union {
   int intval
   ...
}

%token <intval> NUMBER
```

Acse.lex

```
DIGIT    [0-9]
%%
{DIGIT}+ {
           yylval.intval =
              atoi(yytext);
           return NUMBER;
         }
```

Of course, `atoi()` (short for Ascii TO Integer) “converts” strings to integers

Contents

Branches in assembly

Let us look at this example program:

```
int a;
read(a);
if (a == 10) {
  write(10);
}
```

The equivalent assembly code is:

```
  READ R1 0          // read(a);

  SUBI R0 R1 #10
  SEQ R2 0           // R2 = (a == 10);

  BEQ L0             // if (R2==0) goto L0;
  ADDI R3 R0 #10
  WRITE R3 0         // write(10);
L0:
```

We need a way to write in our compiled program:

- The conditional branch instruction BEQ
- The label L0 we branch to

Branches

In general there are two kinds of branches:

Forward The label is **after** the branch

Backward The label is **before** the branch

Forward branch

```
    /* ... */
    BT L0
    /* ... */
L0: /* ... */
```

Backward branch

```
    /* ... */
L0: /* ... */
    /* ... */
    BT L0
```

Creating labels

Problem: ACSE is a **syntactic directed translator**

- Normally, labels that appear after a branch must be also **generated** after the branch
- We need a way to **allocate** a label without **generating** it – in other words, without **inserting it** into the instruction list

This is why ACSE provides **3** primary functions for creating labels:

- *newLabel()*
- *assignLabel()*
- *assignNewLabel()*

Creating labels

newLabel() creates a label structure **without inserting it into the instruction list**.

```
t_axe_label *newLabel(t_program_infos *program);
```

assignLabel() inserts the label in the instruction list.

```
void assignLabel(t_program_infos *program,
                     t_axe_label *label);
```

Think of this function as if it **prints the label to the output** (like *gen_xxx_instruction()* functions print *instructions* to the output)

assignNewLabel() creates and inserts the label simultaneously.

```
t_axe_label *assignNewLabel(t_program_infos *program)
{
  t_axe_label *label = newLabel(program);
  return assignLabel(program, label);
}
```

Creating branches

For writing branches in the compiled program we have the usual set of `gen_xxx_instruction` functions:

```
extern t_axe_instruction *gen_bt_instruction(
      t_program_infos *program, t_axe_label *label, int addr);
extern t_axe_instruction *gen_bne_instruction(
      t_program_infos *program, t_axe_label *label, int addr);
extern t_axe_instruction *gen_beq_instruction(
      t_program_infos *program, t_axe_label *label, int addr);
extern t_axe_instruction *gen_bge_instruction(
      t_program_infos *program, t_axe_label *label, int addr);
extern t_axe_instruction *gen_blt_instruction(
      t_program_infos *program, t_axe_label *label, int addr);
extern t_axe_instruction *gen_bgt_instruction(
      t_program_infos *program, t_axe_label *label, int addr);
extern t_axe_instruction *gen_ble_instruction(
      t_program_infos *program, t_axe_label *label, int addr);
```

- The **label** argument specifies the target
- The **addr** argument is unused (leave as zero)

Semantics of the statement

Let's apply this to the implementation of the `if` statement.

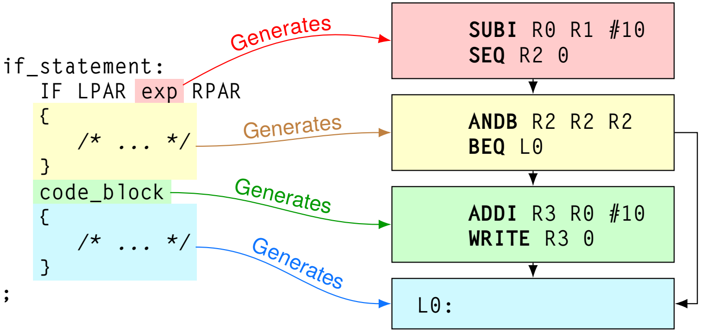
The grammar for this statement, **without the else part**, is like this:

```
if_statement: IF LPAR exp RPAR code_block;
```

The expression inside the parenthesis is called the **condition**
The code block is executed only if the **condition** is **not equal to zero**

Semantic actions

- We need two semantic actions:
  - One for the code **before** the body
  - One for assigning the label **after** the body
- The additional ANDB forces the update of the flags in case the *exp*'s code didn't do that.



Note: parsing the *exp* non-terminal not always results in generation of code because of constant folding.

Semantic actions

The label must be shared between the first and second action:

- Trick: we store it in the semantic value of the IF token

```
%union {
   /* ... */
   t_axe_label *label;
   /* ... */
}
%token <label> IF

/* ... */

if_statement:
   IF LPAR exp RPAR    {  /* ... */  }
   code_block          {  /* ... */  }
;
```

```
/* ... */
if_statement:
  IF LPAR exp RPAR
  {
    /* Ensure that the value of the condition is materialized */
    int r_cond;
    if ($3.expression_type == REGISTER) {
      r_cond = $3.value;
    } else {
      r_cond = getNewRegister(program);
      gen_addi_instruction(program, r_cond, REG_0, $3.value);
    }

    gen_andb_instruction(program,
          r_cond, r_cond, r_cond, CG_DIRECT_ALL);
    $1 = newLabel(program);
    /* Generate the branch */
    gen_beq_instruction(program, $1, 0);
  }
  code_block
  {
    /* Generate the label */
    assignLabel(program, $1);
  }
;
```

Contents

Conclusion

We saw the basic APIs provided by ACSE:

- **getNewRegister()** to get temporary registers for intermediate computations
- **get_symbol_location()** to access variables
- **gen_xxx_instruction()** to generate instructions (mostly by example)
- **new/assign/assignNewLabel()** to generate labels
- **load/storeArrayElement()** to generate accesses to arrays

The two following ACSE lectures will focus on practical exercises that will use these APIs (plus some extra helpers) to implement new statements.

In past years students clamored for more exercises and less talking... so that's what you get. I hope you understood something anyway!