Formal Languages and Compilers Laboratory

ACSE: Building simple compilers in Bison and Flex

Daniele Cattaneo

Material based on slides by Alessandro Barenghi and Michele Scandale

Contents

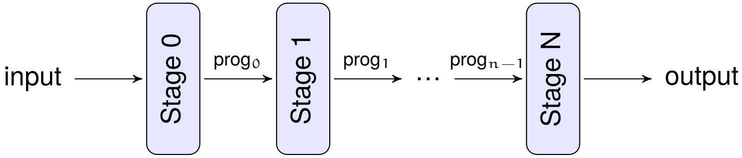
What does a compiler do?

The purpose of a compiler is:

- it translates a program written in a language L_0 into a **semantically equivalent** program expressed in language L_1.

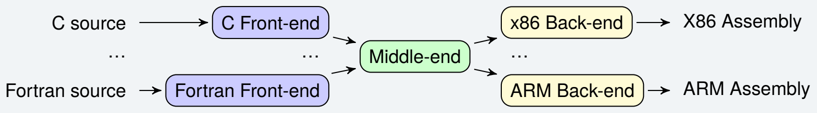
A compiler is organized as a pipeline:

- each stage applies a transformation to the input program producing an output program



Compiler pipeline

Generic compiler structure

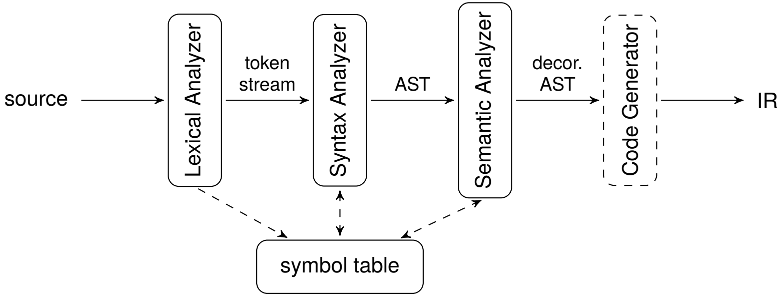


Each stage has its own purpose:

front-end	converts from source language into an intermediate form
middle-end	applies transformations and optimizations on the intermediate form
back-end	convert from the intermediate form into target machine language

Front-end structure

The front-end translates a program from a source language to an intermediate form.



Main tasks:

- recognize language constructs
- verify syntactical correctness
- verify semantical correctness

A real world example: LLVM

Many frontends:

- Various languages (C, C++, Objective C, Swift, Rust, Go)
- **clang** is the frontend for C-like languages
- Each frontend outputs a unified language called LLVM-IR

The intermediate representation is optimized in the **middle-end**

- Removes redundant or unused computations
- Rearranges loops to speed them up
- Tries to vectorize your code
- And more!

At last:

- Each intermediate instruction is mapped to real target CPU using a graph covering algorithm
- “Printers” emit assembly code or directly the binary

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)` stores an integer read from standard input into `var`
- **`write`**`(var)` writes `var` 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**

There is no **middle-end** because no **optimizations** are made

Contents

What’s an Intermediate Representation (IR)?

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

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

Let’s focus on this special **assembly language**

Intermediate Representation

ACSE uses a RISC-like intermediate assembly language which is almost 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:

- unbounded registers
- unbounded memory locations

Registers, immediates, labels

Registers are the **variables** of intermediate language

- All registers have the same type: 32-bit signed integer
- It’s the instruction that decides if the contents of a register must be interpreted as a pointer

Immediates are **constant values** encoded directly in the instruction

Labels are **constant pointers** to a given location in memory

- Used both for conditional jumps and for pointing to the location of a statically-allocated array
- The actual address is decided by the **assembler**

Instruction Formats

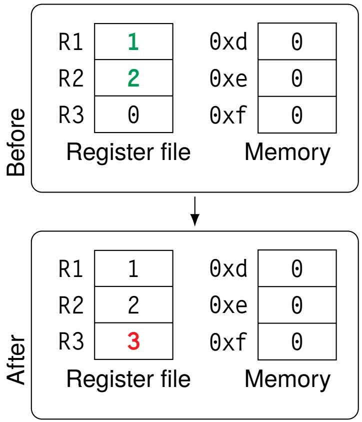
Type	Operands	Example
Ternary	1 destination and 2 source registers	`ADD R3 R1 R2`
Binary	1 destination and 1 source register, and 1 immediate operand	`ADDI R3 R1 #4`
Unary	1 destination and 1 address operand	`LOAD R1 L0`
Jump	1 address operand	`BEQ L0`

Operands & Addressing modes

Operand type	Syntax	Notes
Register direct	`R`$\langle n \rangle$	The *n*-th register
Register indirect	`(R`$\langle n \rangle$`)`	Data at the address contained in the *n*-th register
Symbolic address	`L`$\langle n \rangle$ $\langle \textit{label id} \rangle$	The address identified by the label
Immediate	$\#\langle n \rangle$	The scalar integer constant *n*

Addressing modes example

`ADD R3 R1 R2`



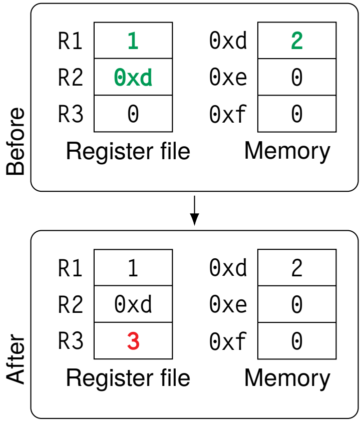
Before

	Register file		Memory
R1	1	0xd	0
R2	2	0xe	0
R3	0	0xf	0

After

	Register file		Memory
R1	1	0xd	0
R2	2	0xe	0
R3	3	0xf	0

`ADD R3 R1 (R2)`



Before

	Register file		Memory
R1	1	0xd	2
R2	0xd	0xe	0
R3	0	0xf	0

After

	Register file		Memory
R1	1	0xd	2
R2	0xd	0xe	0
R3	3	0xf	0

Green: Read **Red**: Write

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

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

Carry vs. Overflow

Carry and overflow are **almost** the same thing:

- Carry is overflow for **unsigned** numbers
- Overflow refers to **signed** numbers

Example with 8-bit numbers (ACSE has 32-bit registers but the concept is the same):

Carry

```
 11111111       255
        1 +       1 +
------------  -------
 00000000         0
```

The result (256) does not fit in 8 bits no matter what!

Overflow

```
 01111111       127
        1 +       1 +
------------  --------
 10000000      -128
```

The result would fit in 8 bits if it was unsigned. But since it is signed, it interferes with the sign bit and it "wraps around".

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)

The variable table

The variable table contains the list of variables declared in the program

A variable can be either a **scalar** or an **array**:

- **Scalars** have a **register** where they are stored and an **initial value**
- **Arrays** have a **size** and a **label** to a **memory location** where they will be stored

Additionally each variable has a unique **identifier** (the name of the variable).

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

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 **list of instructions while parsing**
- The order of the compiled instructions depends on the syntax!
- Real compilers first build a **syntactic tree** (AST), then they transform it to a **list of instructions**

`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

Basics of the LANCE grammar

In a LANCE source file is split into two sections:

1. Variable declarations; root non-terminal: *var_declarations*
2. List of statements; root non-terminal: *statements*

The basic grammar rules (expressed in BNF):

program	:	*var_declarations statements* ⊣
var_declarations	:	*var_declarations var_declaration*
	|	ε
statements	:	*statements statement*
	|	*statement*
code_block	:	*statement*
	|	LBRACE *statements* RBRACE
var_declaration	:	…
statement	:	…

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

The most basic statement

Let's start with an example, by looking at the most obvious statement of them all:

- **`return`** statement: exits the program
- Translation to ASM: a single instruction, **`HALT`**

The Bison code responsible:

```
statement         : /* ... */
                  | control_statement
                  | /* ... */
;

control_statement : /* ...*/
                  | return_statement SEMI
;

return_statement  : RETURN
                  {
                     /* insert an HALT instruction */
                     gen_halt_instruction(program);
                  }
;
```

The code generation functions

The translation is done by the semantic action of *return_statement*

- The bison-generated parser executes it every time it encounters a `return` statement in a program
- The only thing it does: **call the `gen_halt_instruction()` function**

`gen_halt_instruction()` is a **normal C function**

- Declaration in `axe_gencode.h`
- Implementation in `axe_gencode.c`
- It **generates a `HALT` instruction**

What does it mean to **generate an instruction**?

Code generation

Remember the simple **infix expression compiler**?

- It produced a C program that computed a given expression
- The generation of the program was performed by calling the `printf` function

Conceptually, generating an instruction is just like a call to `printf`!

```
return_statement  : RETURN
                    {
                       printf("HALT\n");
                    }
;
```

Not so simple

However, ACSE cannot make do simply with printfs alone...

- The machinery inside ACSE needs to adjust the instructions *after-the-fact* for various reasons
- The most important reason is that the “physical” MACE machine does **not** have **infinite registers**

To allow these adjustements, instead of performing the `printf` immediately, ACSE **allocates an instruction structure and adds it to a linked list**

- The linked list is inside the **global `program` structure**

```
return_statement: RETURN
  {
    t_axe_instruction *inst = malloc(sizeof(t_axe_instruction));
    inst->opcode = HALT;
    program->instructions = addLast(program->instructions, inst);
  }
;
```

Still fairly simple

Once the entire program is parsed, and the registers have been fixed up, a separate function **prints all the instructions***:

```
void writeAssembly(t_program_infos *program, char *file)
{
  FILE *fp = fopen(file, "w");

  t_list *cur = program->instructions;
  while (cur) {
    t_axe_instruction *cur_inst = cur->data;

    switch (cur_inst->opcode) {
      /* ... */
      case HALT:
        printf("HALT\n");
        break;
      /* ... */
    }

    cur = cur->next;
  }

  fclose(fp);
}
```

*The code has been intentionally simplified, the real function is a bit more complex

Back to the codegen functions

So, in summary, `gen_halt_instruction()` allocates the instruction structure and adds it to the list of instructions:

```
void gen_halt_instruction(t_program_infos *program)
{
  t_axe_instruction *inst = malloc(sizeof(t_axe_instruction));
  inst->opcode = HALT;
  program->instructions = addLast(program->instructions, inst);
}
```

Similar functions exist for **all the instructions in the intermediate representation**

The program instance

We have seen that the instruction list is contained in a **global structure** called **`program`**

- Declaration in `Acse.y`, at the very top
- It contains **the intermediate representation** of the program being compiled
- 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;
```

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

Recap

1. ACSE is a compiler for a language called LANCE
2. Reading LANCE files is taken care of through a bison/flex generated parser
3. During the parsing process, the **semantic actions** rewrite each statement encountered to *semantically equivalent* instructions in an assembly-like **intermediate representation or language**
    - This process is called **syntactic-directed translation**
4. The intermediate representation consists of linked lists (variable and instructions) contained in the `program` structure
5. At the end of the compilation, the lists in the `program` structure are traversed and each instruction is printed to produce the final assembly program

Where to go from here

From now on we will examine **how each language feature in LANCE is implemented in the ACSE codebase**

- Variable declaration, access and assignment
- Arithmetic expressions and constant folding
- Control statements (if, while, do-while)

In the process we will see further “building blocks” in the IR:

- The variable list
- Register identifiers
- Labels

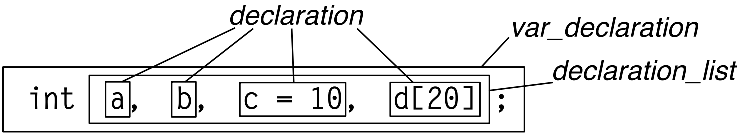
Final goal: **know ACSE well enough to be able to add features to it**

Contents

Syntax of variable declarations

The syntax of variable declarations is similar to C:

var_declaration	:	TYPE *declaration_list* SEMI
declaration_list	:	*declaration_list* COMMA *declaration*
	|	*declaration*
declaration	:	IDENTIFIER ASSIGN NUMBER
	|	IDENTIFIER LSQUARE NUMBER RSQUARE
	|	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 $\longrightarrow$

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

Tokens for constant integers

Another token appears, `NUMBER`, which 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

Creating a variable

Each variable declared must be **added to the variable list**

- All scalar variables are associated with a register
- All arrays are associated with a memory location
- The variable list lets us associate the register/memory location of the variable/array from the identifier

The function for adding a variable to the list is this:

```
void createVariable(
   t_program_infos *program,
              char *id,        ← The variable identifier
               int  type,      ← The type (always INTEGER_TYPE)
               int  isArray,   ← 1 if the var. will be an array
               int  arraySize, ← Arrays only: number of elements
               int  init_val   ← Scalars only: initial value
   );
```

Important! This function **already frees the identifier for you**

Creating a scalar variable

The semantic action for variable declarations without initializer is super easy:

```
declaration : IDENTIFIER
            {
              createVariable(
                program,
                $1,              /* char *id         */
                INTEGER_TYPE,    /*  int  type       */
                0,               /*  int  isArray    */
                0,               /*  int  arraySize  */
                0,               /*  int  init_val   */
              );
            }
            | /* ... other expansions ... */
;
```

Creating a scalar variable

Now with an initializer:

```
declaration : IDENTIFIER ASSIGN NUMBER
            {
              createVariable(
                program,
                $1,            /* char *id        */
                INTEGER_TYPE,  /*  int  type      */
                0,             /*  int  isArray   */
                0,             /*  int  arraySize */
                $3,            /*  int  init_val  */
              );
            }
            | /* ... other expansions ... */
;
```

Creating an array

The NUMBER in arrays is the size of the array:

```
declaration : IDENTIFIER LSQUARE NUMBER RSQUARE
            {
              createVariable(
                program,
                $1,               /* char *id        */
                INTEGER_TYPE,     /*  int  type      */
                1,                /*  int  isArray   */
                $3,               /*  int  arraySize */
                0,                /*  int  init_val  */
              );
            }
            | /* ... other expansions ... */
;
```

One more complication

In the semantic actions we have seen so far the **type** was assumed to always be **int**.

This is true in ACSE **today**, but **tomorrow** it might not be.

This is the reason why, instead of the **simplified** semantic actions we have just seen, ACSE implements declarations in a slightly different way...

- A list is built with all the declarations in a single line
- In the semantic action of *var_declaration* the list is enumerated and all the variables are created (by calling `createVariable()`) with the specified type

We won't discuss this into detail now. If you are curious, look at the end of these slides (or at the source code).

Accessing variables

A clarification

Creating a variable only tells ACSE that it exists:

- No space for **its value** is reserved in the memory of the compiler
- However, the compilers knows in which register or memory location its value will be **at runtime**
- Compilation is **a form of planning**

Accessing variables

A clarification

Analogy: International Space Station Mission Control

Mission control	**ACSE Compiler**
Plans in advance what the astronauts will do when they go into orbit	Plans in advance what the CPU will do when the program is executed
Is constrained by human and operational limitations (sleep time, equipment repairs, ...)	Is constrained by the limits of the CPU architecture (instruction set, ...)
The plan must achieve a certain set of objectives	The compiled program must perform the same computation specified in the source code
Does not go into orbit to do the job of the astronauts!	**Does not execute any of the statements in the program!**

Accessing variables

Knowing where each variable is stored **allows us to generate the code for accessing it**

Primitive kinds of accesses:

- Read a value from a scalar variable
- Write (assign) a new value to a scalar variable
- Read a value from a given array element
- Write (assign) a new value to a given array element

For now we will focus on scalar variables, we will see arrays later

Retrieving the register of a variable

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

- Read a variable = Use the register
- Assign a variable = Define the register

Some technical compiler-world terms...

Use: accessing a register
Define: assign (store) a value in a register

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 doesn't free the ID string for you!

*This is not true for arrays, but we will see this later

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

Be careful not to **mix up normal integer variables in the compiler with register identifiers!**

- 50% of students make this mistake at the exam!

Infallible secret trick that evil professors don’t tell:

- Doing **any kind of arithmetic or comparison operation on register identifiers is wrong**

Generating an assignment to a variable

1. Fetch the register ID associated to the variable
2. Generate an instruction to assign that register

Example: assign the constant value *42* to the variable “a”

```
int r_var = get_symbol_location(program, "a", 0);
gen_addi_instruction(program, r_var, REG_0, 42);
```

Notice:

- The `gen_`*xxx*`i_instruction()` family of functions allows to generate instructions with an immediate parameter
- Arguments: 2 register identifiers, 1 integer (the immediate)

Another example

Let’s write the code to generate the assembly implementing this statement:

```
a = b + c;
```

Compiler code that generates the corresponding IR

```
int r1 = get_symbol_location(program, "a", 0);
int r2 = get_symbol_location(program, "b", 0);
int r3 = get_symbol_location(program, "c", 0);
gen_add_instruction(program, r1, r2, r3, CG_DIRECT_ALL);
```

Notice:

- The `gen_`*xxx*`_instruction()` family of functions allows to generate ternary instructions
- Arguments: 3 register identifiers, 1 flags parameter
- The last argument specifies whose registers are indirectly addressed

The *read* statement

The **read** statement assigns a value to a variable

- At runtime, the value is read from the terminal by the special `READ` instruction
- Notice that the variable identifier is **freed**

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

read_write_statement : read_statement
                      | /* ...*/
;

read_statement        : READ LPAR IDENTIFIER RPAR
                      {
                         int location;
                         location = get_symbol_location(program, $3, 0);
                         gen_read_instruction(program, location);
                         free($3);
                      }
;
```

Contents

Homework

Modify ACSE to add a **new statement** called **`fma`**:

- The statement must compute the expression $a \times b + c$ (where a, b, c are the three arguments)
- The result of the computation goes back into a

Example of the syntax

```
int res, v1, v2;
/* ... */
fma(res, v1, v2);
```

Steps to follow to get you started*:

1. Add new tokens to the Acse.lex file
2. Add token declarations and grammar rules to Acse.y
3. Write the semantic actions
4. Compile the modified ACSE by running **make**
5. Test the modified ACSE with a small program (look at the Readme file for detailed instructions...)

*The solution is in the zip file associated to these slides

Contents

Acse.lex

```
/* ... */
"int"               { yylval.intval = INTEGER_TYPE; return TYPE; }
/* ... */
```

Acse.y

```
%union {
   /* ... */
   t_axe_declaration *decl;
   t_list *list;
   /* ... */
}

%type <decl> declaration
%type <list> declaration_list
%token <intval> TYPE
```

```
var_declaration  : TYPE declaration_list SEMI
                 { set_new_variables(program, $1, $2); }
;

declaration_list : declaration_list COMMA declaration
                 { $$ = addElement($1, $3, -1); }
                 | declaration
                 { $$ = addElement(NULL, $1, -1); }
;

declaration      : IDENTIFIER ASSIGN NUMBER
                 {
                   $$ = alloc_declaration($1, 0, 0, $3);
                   if ($$ == NULL) notifyError(AXE_OUT_OF_MEMORY);
                 }
                 | IDENTIFIER LSQUARE NUMBER RSQUARE
                 {
                   $$ = alloc_declaration($1, 1, $3, 0);
                   if ($$ == NULL) notifyError(AXE_OUT_OF_MEMORY);
                 }
                 | IDENTIFIER
                 {
                   $$ = alloc_declaration($1, 0, 0, 0);
                   if ($$ == NULL) notifyError(AXE_OUT_OF_MEMORY);
                 }
;
```

axe_struct.h

```
typedef struct t_axe_declaration
{
   int isArray;   /* must be TRUE if the current variable is an array */
   int arraySize; /* the size of the array. This information is useful
                   * only if the field `isArray' is TRUE */
   int init_val;  /* initial value of the current variable. */
   char *ID;      /* variable identifier (should never be a NULL pointer
                   * or an empty string "") */
} t_axe_declaration;
```

axe_struct.c

```
t_axe_declaration * alloc_declaration(
      char *ID, int isArray, int arraySize, int init_val)
{
   t_axe_declaration *result =
         (t_axe_declaration *) malloc(sizeof(t_axe_declaration));
   if (result == NULL)
      return NULL;

   result->isArray = isArray;
   result->arraySize = arraySize;
   result->ID = ID;
   result->init_val = init_val;
   return result;
}
```

axe_utils.c

```
void set_new_variables(
      t_program_infos *program, int varType, t_list *variables)
{
   t_list *current_element = variables;
   t_axe_declaration *current_decl;

   while (current_element != NULL) {
      current_decl = (t_axe_declaration *)LDATA(current_element);
      createVariable(program,
            current_decl->ID,
            varType,
            current_decl->isArray,
            current_decl->arraySize,
            current_decl->init_val);
      current_element = LNEXT(current_element);
   }

   /* free the linked list */
   current_element = variables;
   while (current_element != NULL) {
      current_decl = (t_axe_declaration *)LDATA(current_element);
      /* ... */
      free(current_decl);
      current_element = LNEXT(current_element);
   }
   freeList(variables);
}
```