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

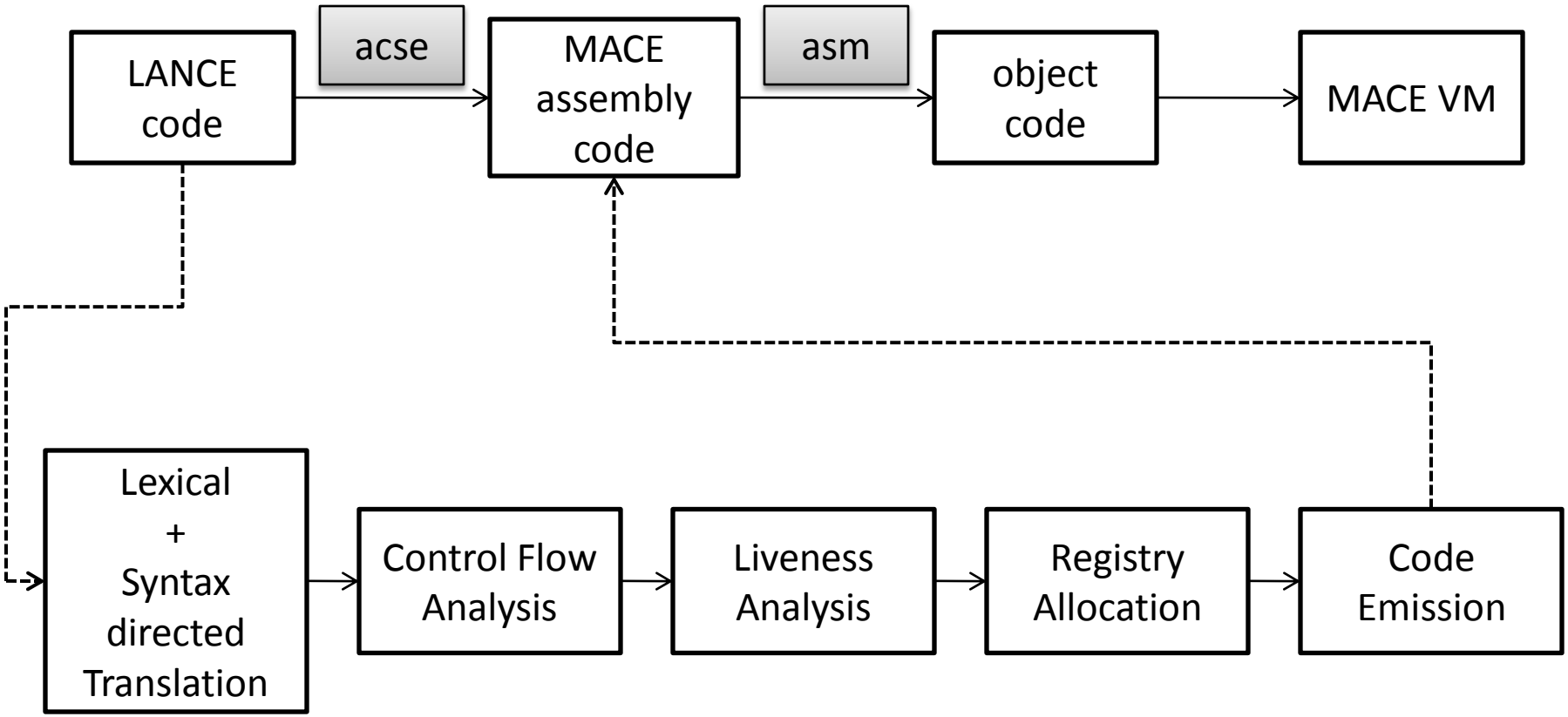
Marcello M. Bersani

LFC – Politecnico di Milano

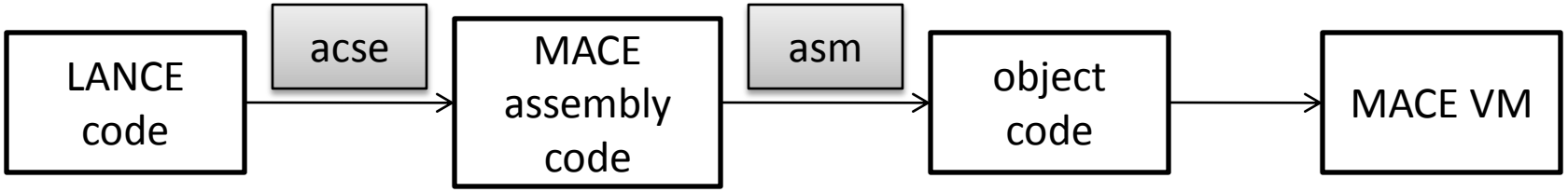
Actors

- ACSE
  - Advanced Compiler Suite for Education
  - Compiler for LANCE language
  - Produces MACE assembly code
- MACE
  - Machine for Advance Compiler Education
  - VM executing object code

Compiling flow



Compiling flow

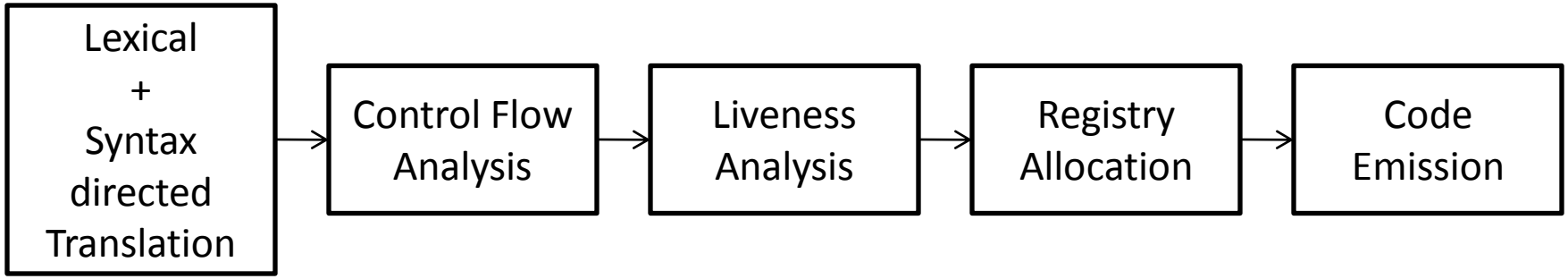



```
int i, j;


for (i=0,j=9 ; i < 10 ; i=i+1, j=j-2) {
   write(i);
   write(j);
 }
```

acse

```
                .data
L0 :            .WORD 0
L1 :            .WORD 0
                .text
                ADDI R1 R0 #0
                STORE R1 L0
                ADDI R2 R0 #9
                STORE R2 L1
L2 :            LOAD R1 L0
                SUBI R0 R1 #10
                STORE R1 L0
                SLT R0 0
                BEQ L3
                BT L4
...
```

Compiling flow



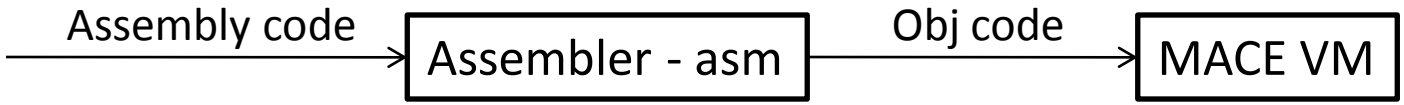
- **CF analysis**: analizes the control flow of the code by individuating jumps
- **Liveness analysis**: determines where a variable is "live"; i.e., the interval between a write and a read instruction (live vars are kept into registry)
- **Registry alloc**: maps used vars into a finite set of registries for reason of efficiency. Effect: insert suitable LOAD/STORE instruction to move vars into registry

ACSE main() code

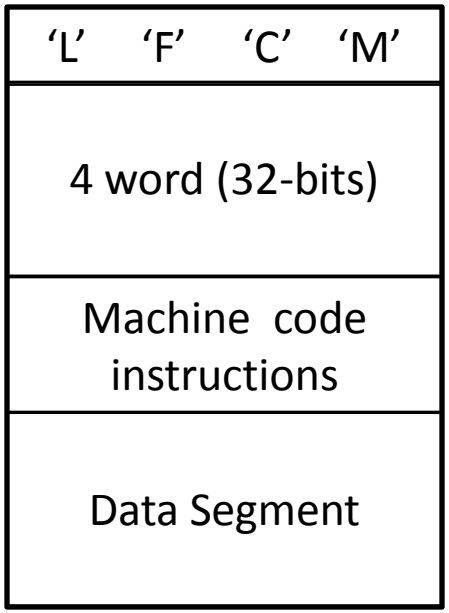
```
/* initialize all the compiler data structures and global variables */
  init_compiler(argc, argv);
  /* start the parsing procedure */
  yyparse();

…
  /* create the control flow graph */
  graph = createFlowGraph(program->instructions);

/* update the control flow graph by inserting load and stores inside every basic block */
  graph = insertLoadAndStoreInstr(program, graph);

  performLivenessAnalysis(graph);

/* initialize the register allocator by using the control flow informations stored into the control flow graph */
  RA = initializeRegAlloc(graph);

  /* execute the linear scan algorythm */
  execute_linear_scan(RA);

/* apply changes to the program informations by using the informations of the register allocation process */
  updateProgramInfos(program, graph, RA);

  writeAssembly(program, file_infos->output_file_name);
…
  return 0;
```

Assembly for MACE

- Assembly: human-like language




```
            .data
L0 :        .WORD 0
L1 :        .WORD 0
            .text
            ADDI R1 R0 #0
            STORE R1 L0
            ADDI R2 R0 #9
            STORE R2 L1
L2 :        LOAD R1 L0
            SUBI R0 R1 #10
            STORE R1 L0
            SLT R0 0
            BEQ L3
            BT L4
...
```



Assembly for MACE

```
Label: Instruction/Directive [/*comment*/]
```

- Instruction: **opcode + operands**
  - register identifiers, directly or indirectly addressed;
  - immediate values
  - address values (typically labels).
  - Type
    - Unary/binary: direct addressing always
    - ternary: direct/indirect with limitations

L1: ADD R3 R2 R1
L1: ADD (R3) R2 (R1)
L1: BEQ L5
L1: ADD R2 R1 #1

Assembly for MACE

```
Label: Instruction/Directive [/*comment*/]
```

- Directive
  - .**data**: beginning of DS
  - .**text**: beginning of CS
  - .**word** VAL: reserve a 32-bit memory location in DS; its initial value is equal to VAL
  - .**space** VAL: reserve VAL contiguous byte in DS

MACE in a nutshell

- Emulated machine

- Architecture
  - 32 general-purpose 32-bit registers
  - 32-bit PC
  - 32-bit status register (PSW)
  - 32-bit memory words

MACE in a nutshell

- Bootstrap
  - Reserve 2KB; DS and CS are loaded
  - Registers set to 0; set PC
- Execution
  - Fetch PC instruction
  - Decode/execute
  - Update registers, PC, PSW
  - Until HALT

MACE in a nutshell

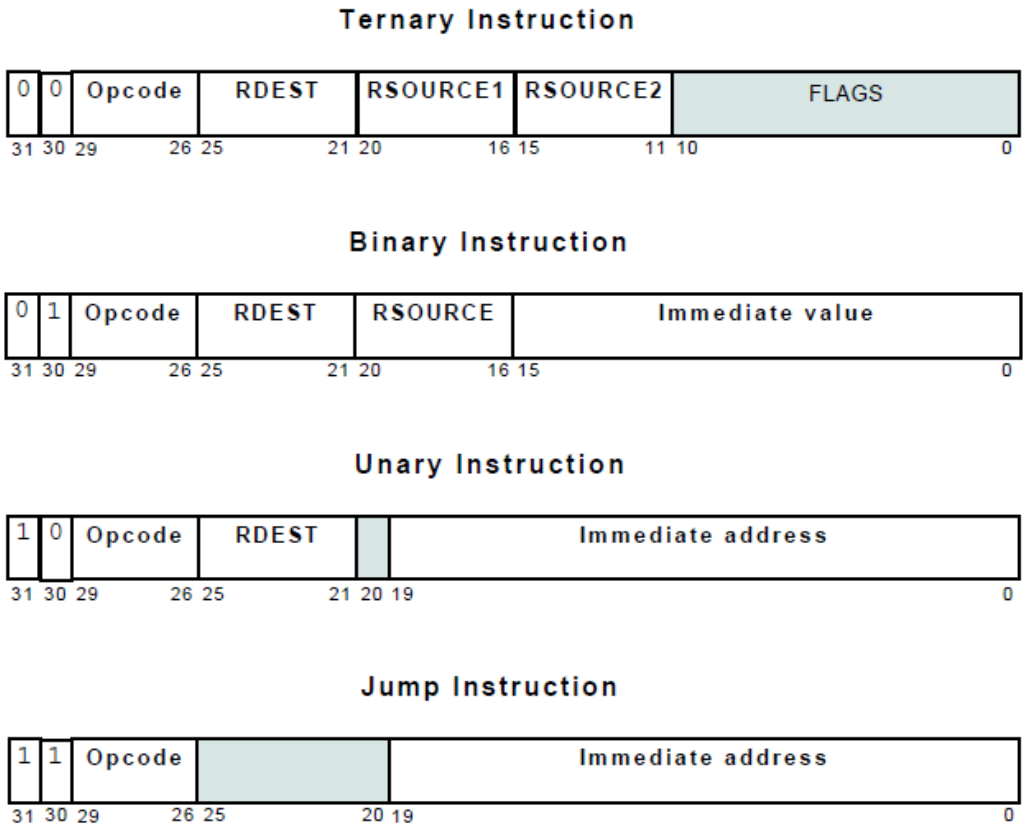


Figure 3.3: Instruction Formats

MACE in a nutshell

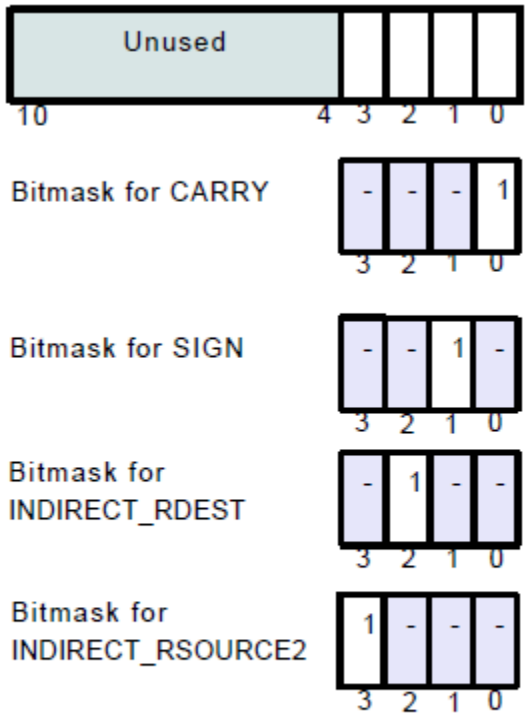


Figure 3.4: Flag bits for ternary instructions

LANCE in a nutshell

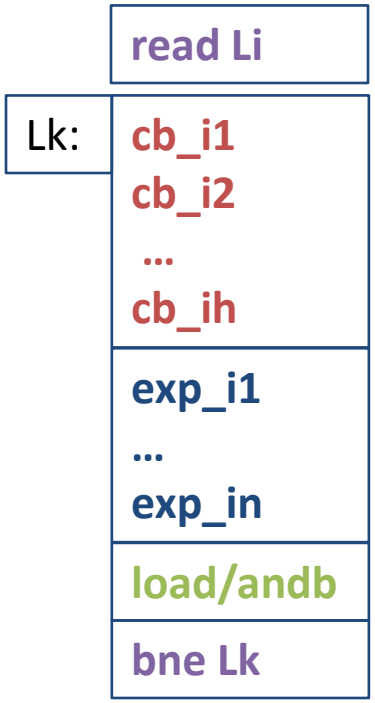
- Data type: int, array
- Basic I/O: read(), write()
- Arithmetic expressions
- If-then-else
- While
- Do-while

LANCE example: DO-WHILE

```
int i;

read(i);
do
   code_block
while (i>0);
```

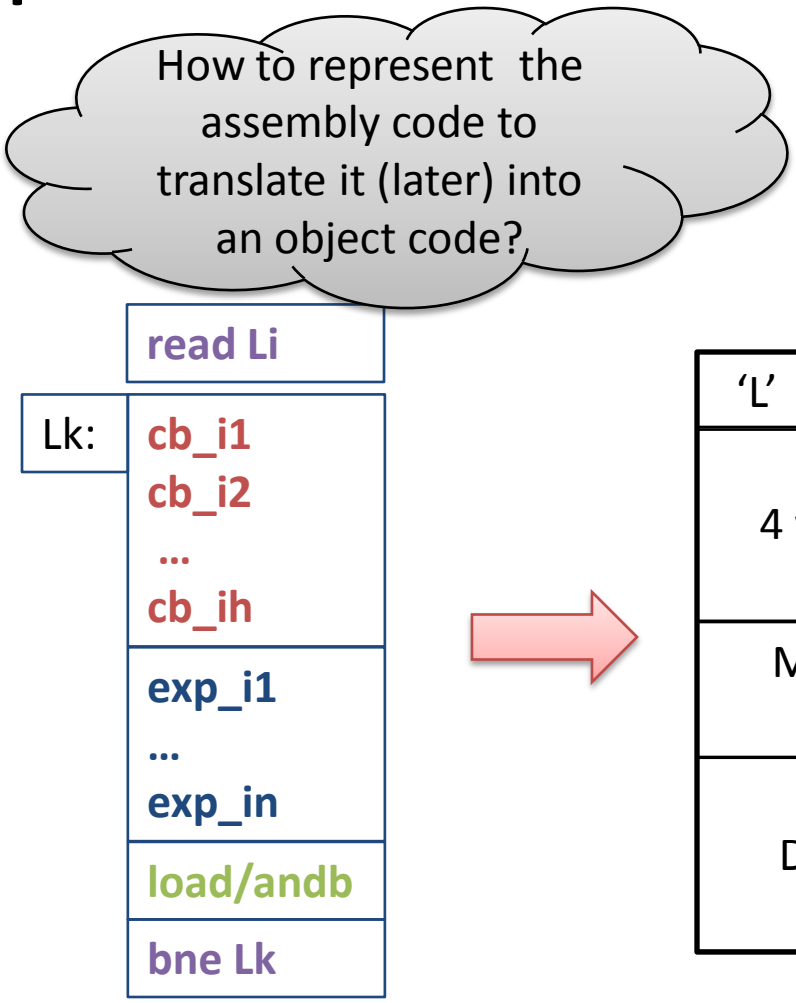
How to implement a DO-WHILE

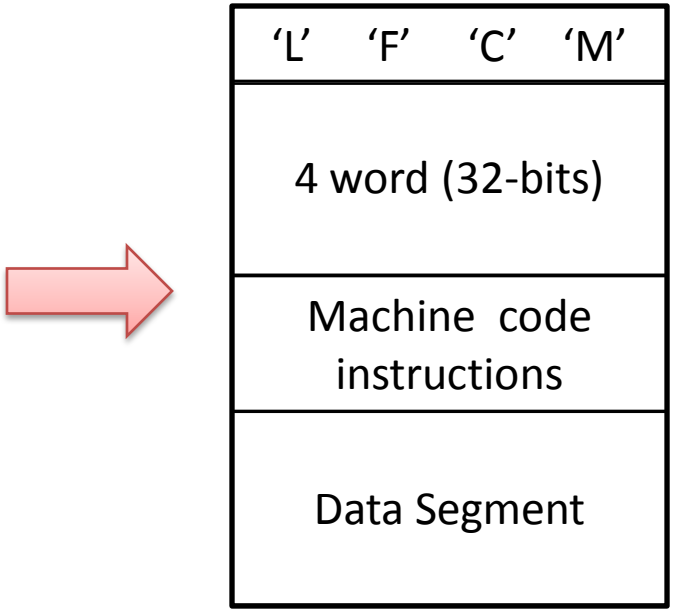
```
int i;

read(i);
do
   code_block
while (i>0);
```



How to implement a DO-WHILE

```
int i;

read(i);
do
   code_block
while (i>0);
```



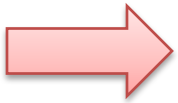


How to implement a DO-WHILE

```
int i;

read(i);
do
   code_block
while (i>0);
```

How to represent the assembly code to translate it (later) into an object code?

```
        read Li
Lk:     cb_i1
        cb_i2
        ...
        cb_ih
        exp_i1
        ...
        exp_in
        load/andb
        bne Lk
```



‘L’ ‘F’ ‘C’ ‘M’
4 word (32-bits)
Machine code instructions
Data Segment

We need a data structure to store:
-instructions
-labels
-variables
- …

LANCE-to-Assembly

- Essential infos collected while parsing are stored into data structure **program_info**

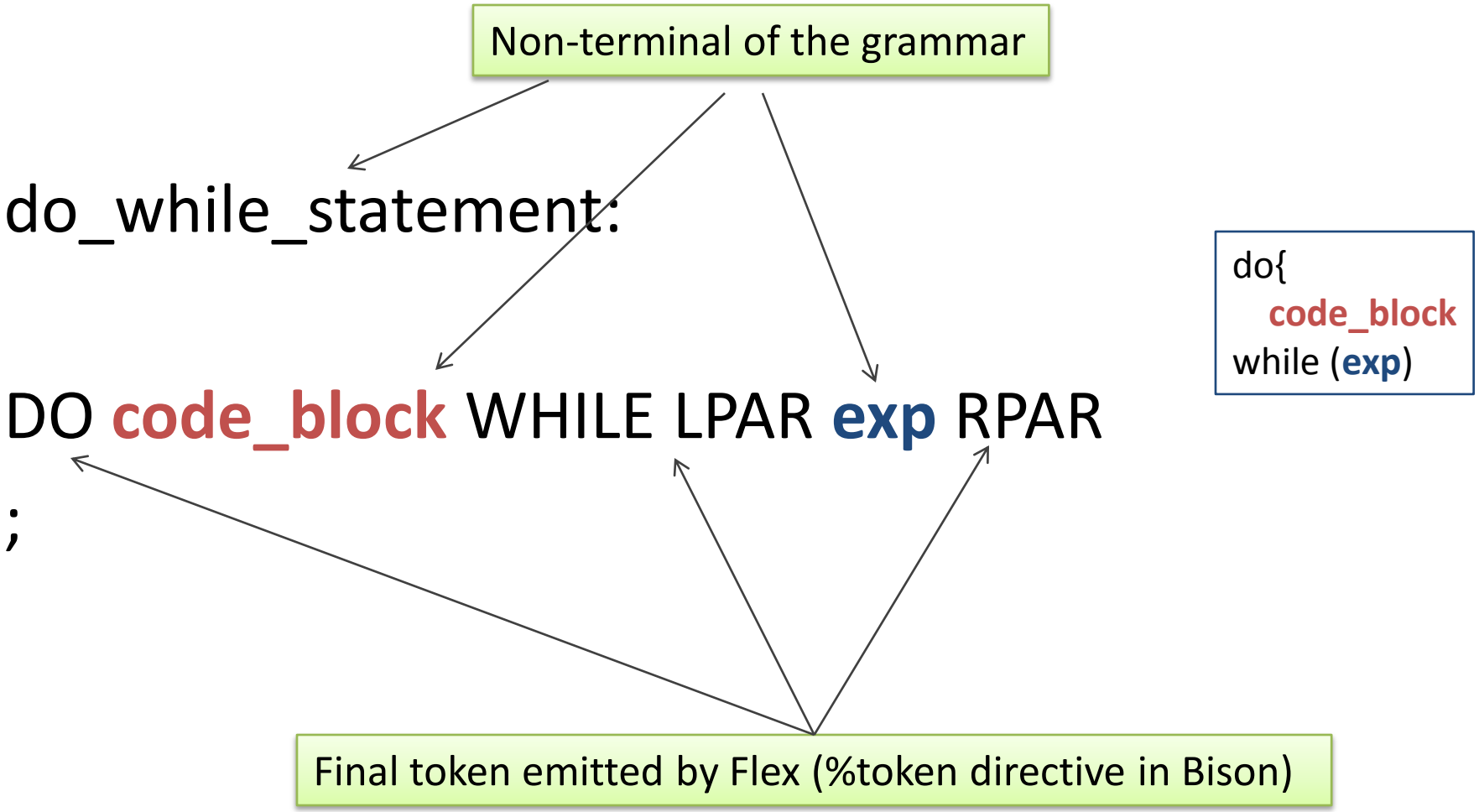
- While parsing the syntax tree is decorated by semantic info
  - Used to produce final assembly code
  - “external”: related to LANCE code
    - Vars, type declarations, expressions, …
  - “internal”: related to assembly code
    - Labels, instructions, registers, …

ACSE data structure

```
typedef struct t_program_infos
{
  t_list *variables;
  t_list *instructions;
  t_list *data;
  t_axe_label_manager *lmanager;
  t_symbol_table *sy_table;
  int current_register;
} t_program_infos;
```

- Axe_engine.h

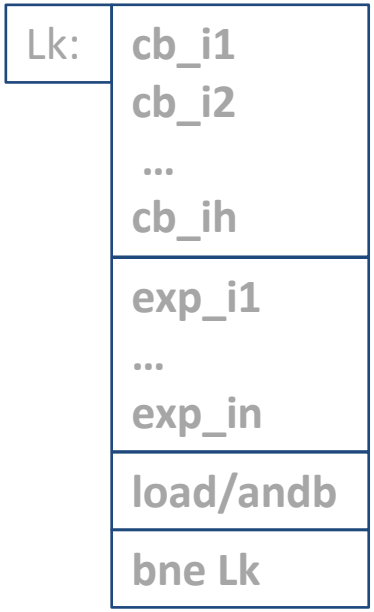
DO-WHILE


Non-terminal of the grammar
do_while_statement:
DO code_block WHILE LPAR exp RPAR
;
do{
code_block
while (exp)
Final token emitted by Flex (%token directive in Bison)

DO-WHILE

```
do_while_statement  : DO
   {
       $1 = newLabel(program);
       assignLabel(program, $1);
   }
   code_block WHILE LPAR exp RPAR
   {
        if ($6.expression_type == IMMEDIATE)
              gen_load_immediate(program, $6.value);
        else
              gen_andb_instruction(program,$6.value,$6.value, $6.value, CG_DIRECT_ALL);

        gen_bne_instruction (program, $1, 0);
   }
;
```

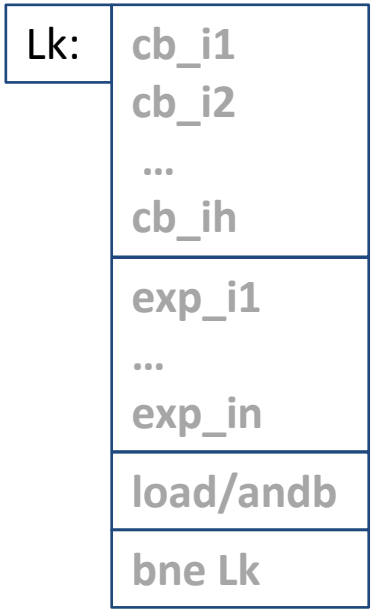
```
do{
   code_block
while (exp)
```



DO-WHILE

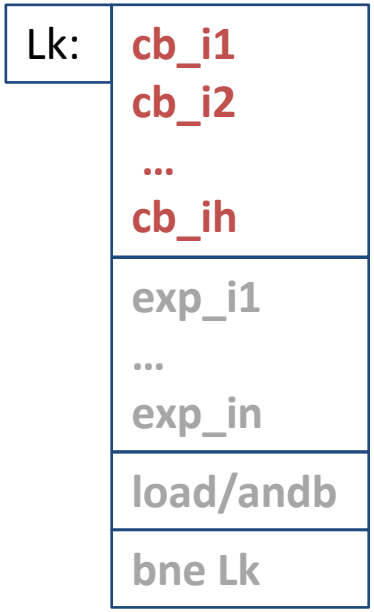
```
do_while_statement  : DO
   {
      $1 = newLabel(program);
      assignLabel(program, $1);
   }
   code_block WHILE LPAR exp RPAR
   {
       if ($6.expression_type == IMMEDIATE)
            gen_load_immediate(program, $6.value);
       else
            gen_andb_instruction(program,$6.value,$6.value, $6.value, CG_DIRECT_ALL);

       gen_bne_instruction (program, $1, 0);
   }
;
```

```
do{
   code_block
while (exp)
```



DO-WHILE

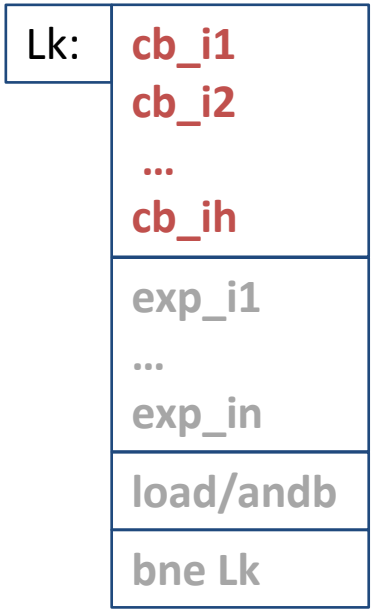
```
do_while_statement  : DO
   {
      $1 = newLabel(program);
      assignLabel(program, $1);
   }
   code_block WHILE LPAR exp RPAR
   {
      if ($6.expression_type == IMMEDIATE)
            gen_load_immediate(program, $6.value);
      else
            gen_andb_instruction(program,$6.value,$6.value, $6.value, CG_DIRECT_ALL);

      gen_bne_instruction (program, $1, 0);
   }
;
```



DO-WHILE

```
do_while_statement  : DO
   {
       $1 = newLabel(program);
       assignLabel(program, $1);
   }
   code_block WHILE LPAR exp RPAR
   {
       if ($6.expression_type == IMMEDIATE)
             gen_load_immediate(program, $6.value);
       else
             gen_andb_instruction(program,$6.value,$6.value, $6.value, CG_DIRECT_ALL);

       gen_bne_instruction (program, $1, 0);
   }
;
```

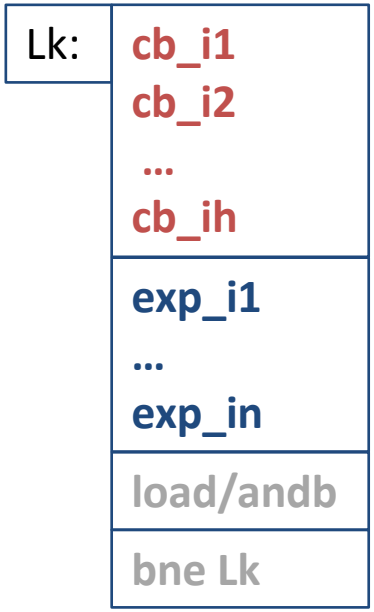
```
do
   code_block
while (exp)
```



DO-WHILE

```
do_while_statement  : DO
   {
      $1 = newLabel(program);
      assignLabel(program, $1);
   }
   code_block WHILE LPAR exp RPAR
   {
      if ($6.expression_type == IMMEDIATE)
            gen_load_immediate(program, $6.value);
      else
            gen_andb_instruction(program,$6.value,$6.value, $6.value, CG_DIRECT_ALL);

      gen_bne_instruction (program, $1, 0);
   }
;
```

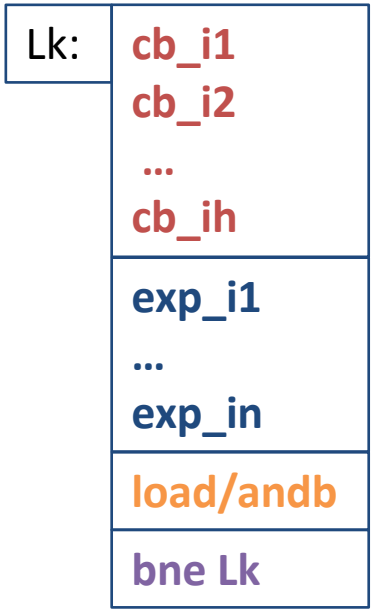
```
do
   code_block
while (exp)
```



DO-WHILE

```
do_while_statement  : DO
   {
       $1 = newLabel(program);
       assignLabel(program, $1);
   }
   code_block WHILE LPAR exp RPAR
   {
        if ($6.expression_type == IMMEDIATE)
              gen_load_immediate(program, $6.value);
        else
              gen_andb_instruction(program,$6.value,$6.value, $6.value, CG_DIRECT_ALL);

        gen_bne_instruction (program, $1, 0);
   }
;
```

```
do
   code_block
while (exp)
```



ACSE data structure

```
typedef struct t_program_infos
{
  t_list *variables;
  t_list *instructions;
  t_list *data;
  t_axe_label_manager *lmanager;
  t_symbol_table *sy_table;
  int current_register;
} t_program_infos;
```

- Axe_engine.h

Instructions

- “Internal” description of assembly instructions
- List of t_axe_instruction (axe_struct.h)
  - Instructions are temporary; refined by Registry Allocation analysis through Liveness infos

L1: ADD (R3) R2 (R1)

L1: BEQ L5

L1: ADD (R3) R2 1

Instructions

```
typedef struct t_axe_instruction
{
  int opcode;                  /* instruction opcode (for example: AXE_ADD ) */
  t_axe_register *reg_1;              /* destination register */
  t_axe_register *reg_2;              /* first source register */
  t_axe_register *reg_3;              /* second source register */
  int immediate;               /* immediate value */
  t_axe_address *address;      /* an address operand */
  char *user_comment;          /* if defined it is set to the source code
    instruction that generated the current assembly. This string will be written
    into the output code as a comment */
  t_axe_label *labelID;        /* a label associated with the current instruction*/
}t_axe_instruction;
```

Registry

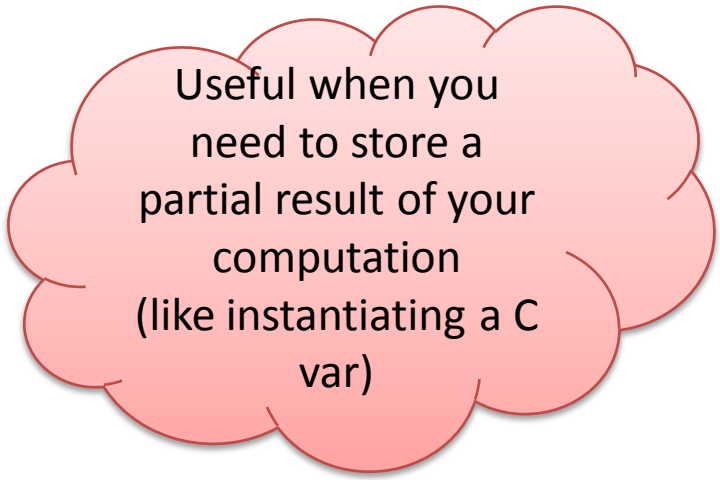
- “Internal” description of a register (axe_struct.h)
- Each register is uniquely identified by a number

```
typedef struct t_axe_register
{
  int ID;          /* an identifier of the register */
  int indirect;   /* a boolean value: 1 if the register value is a pointer
    */
}t_axe_register;
```

Registry

- Function handling register (axe_engine.c)

```
/* get a register still not used. This function returns
 * the ID of the register found*/
[extern] int getNewRegister(t_program_infos *program)
{
  int result;

  /* test the preconditions */
  ...

  result = program->current_register;
  program->current_register++;

  /* return the current label identifier */
  return result;
}
```



Address

- Internal description of an phisical address (refined by the assembler when building the final object code)
- An address is
  - label or
  - Absolute value of PC

```
typedef struct t_axe_address
{
  int addr;                    /* a Program Counter */
  t_axe_label *labelID;        /* a label identifier */
  int type;            /* one of ADDRESS_TYPE or LABEL_TYPE */
}t_axe_address;
```

Generating assembly instructions

- axe_gencode.h/axe_gencode.c
  - Module for generating (all) assembly

```
t_axe_instruction * gen_bt_instruction(t_program_infos *program, t_axe_label
    *label, int addr){
  return gen_jump_instruction (program, BT, label, addr);
}

t_axe_instruction * gen_add_instruction (t_program_infos *program, int r_dest, int
    r_source1, int r_source2, int flags){
  return gen_ternary_instruction(program, ADD, r_dest, r_source1, r_source2, flags);
}

t_axe_instruction * gen_addi_instruction (t_program_infos *program, int r_dest, int
    r_source1, int immediate){
  return gen_binary_instruction(program, ADDI, r_dest, r_source1, immediate);
}

…
```

Generating assembly instructions

- axe_gencode.h/axe_gencode.c
  - Module for generating (all) assembly
  - Wrappers for

```
static t_axe_instruction * gen_unary_instruction (t_program_infos *program,
    int opcode, int r_dest, t_axe_label *label, int addr);
static t_axe_instruction * gen_binary_instruction (t_program_infos *program,
    int opcode, int r_dest, int r_source1, int immediate);
static t_axe_instruction * gen_ternary_instruction (t_program_infos *program,
    int opcode, int r_dest, int r_source1, int r_source2, int flags);
static t_axe_instruction * gen_jump_instruction (t_program_infos *program,
    int opcode, t_axe_label *label, int addr);
```

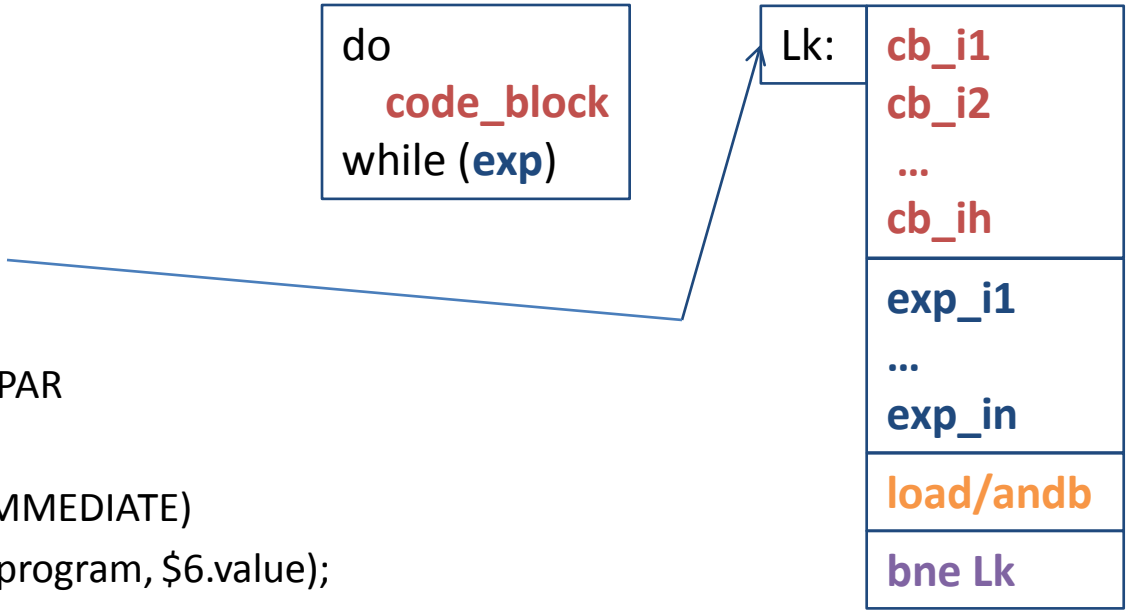
Generating assembly instructions

```
static t_axe_instruction * gen_unary_instruction (t_program_infos *program,
    int opcode, int r_dest, t_axe_label *label, int addr);
static t_axe_instruction * gen_binary_instruction (t_program_infos *program,
    int opcode, int r_dest, int r_source1, int immediate);
static t_axe_instruction * gen_ternary_instruction (t_program_infos *program,
    int opcode, int r_dest, int r_source1, int r_source2, int flags);
static t_axe_instruction * gen_jump_instruction (t_program_infos *program,
    int opcode, t_axe_label *label, int addr);
```

- Allocate and init a **t_axe_instruction**
- Update the list of instrs by **addInstructions**(…)
- Return the instruction

DO-WHILE

```
do_while_statement  : DO
   {
       $1 = newLabel(program);
       assignLabel(program, $1);
   }
   code_block WHILE LPAR exp RPAR
   {
        if ($6.expression_type == IMMEDIATE)
              gen_load_immediate(program, $6.value);
        else
              gen_andb_instruction(program,$6.value,$6.value, $6.value, CG_DIRECT_ALL);

        gen_bne_instruction (program, $1, 0);
   }
;
```



Labels

- “internal” description of labels to be used in the assembly code

L1: .word 0

L2 : LOAD R1 L0

Labels

```
typedef struct t_axe_label
{
   int labelID;      /* label identifier */
} t_axe_label;
```

- Defined in axe_struct.h
- See later the Label Manager

ACSE data structure

```
typedef struct t_program_infos
{
  t_list *variables;
  t_list *instructions;
  t_list *data;
  t_axe_label_manager *lmanager;
  t_symbol_table *sy_table;
  int current_register;
} t_program_infos;
```

- Axe_engine.h

Label Manager

- List of **t_label** used in the assembly so far
- Current label, next label
- axe_label.h/axe_labels.c

```
struct t_axe_label_manager
{
   t_list *labels;
   int current_label_ID;
   t_axe_label *label_to_assign;
};
```

Label Manager

- Labels are added to the assembly instruction when **addInstruction**() is called
  - Label is put when needed (label_to_assign!=NULL)

- Labels can be managed by user
  - newLabel: return a new label L
  - assignLabel: label L is assigned to the next instruction
    - label_to_assign is set (!= NULL)

Label Manager

```
/* retrieve the label that will be assigned to the next instruction */
[extern] t_axe_label * assign_label(t_axe_label_manager *lmanager);
{
  t_axe_label *result;

  /* precondition: lmanager must be different from NULL */
  ...

  /* the label that must be returned (can be a NULL pointer) */
  result = lmanager->label_to_assign;

  /* update the value of `lmanager->label_to_assign' */
  lmanager->label_to_assign = NULL;

  /* return the label */
  return result;
}
```

Label Manager

- Wrappers [axe_engine.h/axe_engine.c]
  - newLabelID → newLabel
  - assignLabelID → assignLabel
  - newLabel+assignLabel → assignNewLabel

Label Manager

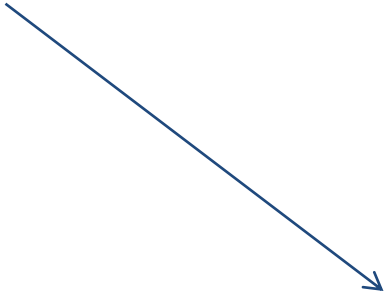
```
/* reserve a new label identifier and return the identifier to the caller */
[extern] t_axe_label * newLabelID(t_axe_label_manager *lmanager);
{
  t_axe_label *result;

  /* preconditions: lmanager must be different from NULL */
  ...

  /* initialize a new label */
  result = alloc_label(lmanager->current_label_ID);

  /* update the value of `current_label_ID' */
  lmanager->current_label_ID++;

  /* tests if an out of memory occurred */
  if (result == NULL)
    return NULL;

  /* add the new label to the list of labels */
  lmanager->labels = addElement(lmanager->labels, result, -1);

  /* return the new label */
  return result;
}
```

Label Manager

```
/*assign the given label identifier to the next instruction. Returns * FALSE if an error occurred; otherwise true*/
[extern] t_axe_label * assignLabelID(t_axe_label_manager *lmanager, t_axe_label *label);
{
  /* precondition: lmanager must be different from NULL */
  ...

  /* precondition: label must be different from NULL and
   * must always carry a valid identifier */
  ...

  /* test if the next instruction has already a label */
  if (  (lmanager->label_to_assign != NULL)
       && ((lmanager->label_to_assign)->labelID != LABEL_UNSPECIFIED) )
  {
    label->labelID = (lmanager->label_to_assign)->labelID;
  }
  else
    lmanager->label_to_assign = label;

  /* all went good */
  return label;
}
```

Example – read instruction

```
read_statement : READ LPAR IDENTIFIER RPAR
        {
          int location;

          location = get_symbol_location(program, $3, 0);

          /* insert a read instruction */
          gen_read_instruction (program, location);

          /* free the memory associated with the IDENTIFIER */
          free($3);
        }
```



ACSE data structure

```
typedef struct t_program_infos
{
  t_list *variables;
  t_list *instructions;
  t_list *data;
  t_axe_label_manager *lmanager;
  t_symbol_table *sy_table;
  int current_register;
} t_program_infos;
```

- Axe_engine.h

Symbol Table

- List of **t_symbols** (symbols_table.h)
- “internal” representation of relation
  - Var name
  - Type
  - Associated register

```
typedef struct
{
   char *ID;            /* symbol identifier */
   int type;           /* type associated with the symbol */
   int reg_location;    /* a register location */
} t_symbol;
```

Symbol Table

- Remark:
  - The underlying machine is supposed to have an unbounded number of regs
  - Each var is associated with a unique register
  - Liveness analysis/Registers Allocation fill the gap
    - Put suitable LOAD/STORE to map unbounded set of registers to a physical machine (like MACE)

Symbol Table

- API (symbols_table.c/symbols_table.h)

```
/* put a symbol into the symbol table */
extern int putSym(t_symbol_table *table, char *ID, int type);

/* set the location of the symbol with ID as identifier */
extern int setLocation(t_symbol_table *table, char *ID, int reg);

/* get the location of the symbol with the given ID */
extern int getLocation(t_symbol_table *table, char *ID, int *errorcode);

/* get the type associated with the symbol with ID as identifier */
extern int getTypeFromID(t_symbol_table *table, char *ID, int type);

/* given a register identifier (location), it returns the ID of the variable
 * stored inside the register `location'. This function returns NULL
 * if the location is an invalid location. */
extern char * getIDfromLocation(t_symbol_table *table, int location, int *errorcode);
…
```

Symbol Table - utils

int **get_symbol_location**(t_program_infos *program, char *ID, int genLoad); [axe_utils.c]

- Given an ID, returns the **register** where ID is stored
  - If ID has never been loaded, searches for a new reg; assign the register to ID; then returns it
  - genLoad forces LOAD from DS

ACSE data structure

```
typedef struct t_program_infos
{
  t_list *variables;
  t_list *instructions;
  t_list *data;
  t_axe_label_manager *lmanager;
  t_symbol_table *sy_table;
  int current_register;
} t_program_infos;
```

- Axe_engine.h

Variables

- “external” description of LANCE vars
- List of t_axe_variable (axe_struct.h)

```
int v1 = 10;
Int vect[3];

read(v1);
read(vect[0]);
```

Variables

```
typedef struct t_axe_variable
{
  int type;            /* a valid data type @see `axe_constants.h' */
  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. Actually it is
                                                  * implemented as a integer value.
                        `int' is
                        * the only supported type at the moment,
                        * future developments could consist of a modification of
                        * the supported type system. Thus, maybe init_val will
                                be modified in future. */
  char *ID;            /* variable identifier (should never be a NULL
                        * pointer or an empty string "") */
  t_axe_label *labelID;   /* a label that refers to the location
                                * of the variable inside the data segment */
} t_axe_variable;
```

Variables

- A variable is a location into DS
  - Identified by a label

```
                .data
L0 :            .WORD 0
L1 :            .WORD 0
                .text
                ADDI R1 R0 #0
                STORE R1 L0
                ADDI R2 R0 #9
                STORE R2 L1
L2 :            LOAD R1 L0
                SUBI R0 R1 #10
                STORE R1 L0
                SLT R0 0
                BEQ L3
                BT L4
...
```

Variables

- Functions managing vars (axe_utils.h/axe_utils.c)

```
/* add a variable to the program */
extern void createVariable(t_program_infos *program, char *ID, int
    type, int isArray, int arraySize, int init_val);

/* get a previously allocated variable */
extern t_axe_variable * getVariable(t_program_infos *program, char
    *ID);

/* get the label that marks the starting address of the variable
 * with name "ID" */
extern t_axe_label * getLabelFromVariableID(t_program_infos
    *program, char *ID);
```

ACSE data structure

```
typedef struct t_program_infos
{
  t_list *variables;
  t_list *instructions;
  t_list *data;
  t_axe_label_manager *lmanager;
  t_symbol_table *sy_table;
  int current_register;
} t_program_infos;
```

- Axe_engine.h

Data infos

- List of **t_axe_data** (axe_struct.h)
- “internal” description of DS
  - The list is created after finishing parsing the block "declarations of vars"
  - associating with each declared variable the appropriate memory segment .data

```
typedef struct t_axe_data
{
  int directiveType;          /* the type of the current directive
                                 * (for example: DIR_WORD) */
  int value;                  /* the value associated with the directive */
  t_axe_label *labelID;   /* label associated with the current data */
}t_axe_data;
```