# Formal Languages and Compilers Laboratory

# ACSE: control statements

Daniele Cattaneo

Material based on slides by Alessandro Barenghi and Michele Scandale

# Contents

# A (somewhat) simple exercise

Let us try to add the **save** statement to ACSE:

**Example**

```
int a = 10;

save a {
  a = 7;
  write(a);   // prints "7"
}
write(a);     // prints "10"
```

The save statement

- **saves** the content of a given variable at the beginning of the block
- **restores** the variable to the previous value at the end of the block

# Token definitions, grammar

New token: the `save` keyword. The grammar rules are also simple.

**Acse.lex**

```
/* ... */
"save"              { return SAVE; }
/* ... */
```

**Acse.y**

```
/* ... */
%token SAVE
/* ... */
statement    : /* ... */
             | save_statement
;
save_statement : SAVE IDENTIFIER code_block
;
/* ... */
```

# The logic of the statement

What code should we generate to save a variable?

- **Before** the block: copy its value into a **temporary register**
- **After** the block: restore the original value from the register

This statement is **syntactic sugar**: a programmer could obtain the same effect manually in this way:

**Example**

```
int a = 10;

save a {
  a = 7;
  write(a);
}
write(a);
```

**Equivalent code**

```
int a = 10, temp_a;

temp_a = a;
a = 7;
write(a);
a = temp_a;
write(a);
```

**Tip:** It's always useful to "de-sugar" a construct you are implementing to clear up any doubt you might have about its implementation

# Implementing the code generation

ACSE is a **syntactic directed translator**:

- The code **before** the block is generated by the **semantic action before** the *code_block* non-terminal
- The code **after** the block is generated by the **semantic action after** the *code_block* non-terminal

```
save_statement :
    SAVE IDENTIFIER
    {
        // Generate code to save the variable
        // The code will appear before the block
    }
    code_block
    {
        // Generate code to restore the variable
        // The code will appear after the block
    }
;
```

# Implementing the code generation

It's easy enough to implement the save part, but there is a problem:

- Semantic actions are **independent blocks or scopes**
- Variables declared in a semantic action **are local to that action!**

```
save_statement :
   SAVE IDENTIFIER
   {
      int r_save = getNewRegister(program);
      int r_var = get_symbol_location(program, $2, 0);
      gen_add_instruction(program,
            r_save, REG_0, r_var, CG_DIRECT_ALL);
   }
   code_block
   {
      // ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
      // The r_save variable CANNOT BE ACCESSED HERE
      // ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
   }
;
```

# Sharing variables between semantic actions

In general you sometimes want to **declare a variable that is accessible to multiple semantic actions**

There are several ways to do this:

1. Use a **global variable**
2. Use a **global stack**
3. Re-purpose a **symbol's semantic value** as a variable

All these methods have distinct pros and cons...

# Method 1 - Global variables

We simply declare our shared variable as a global instead of a local:

```
int r_save;

/* ... */

save_statement :
   SAVE IDENTIFIER
   {
      r_save = getNewRegister(program);
      int r_var = get_symbol_location(program, $2, 0);
      gen_add_instruction(program,
            r_save, REG_0, r_var, CG_DIRECT_ALL);
   }
   code_block
   {
      int r_var = get_symbol_location(program, $2, 0);
      gen_add_instruction(program,
            r_var, REG_0, r_save, CG_DIRECT_ALL);
      free($2);
   }
;
```

# Method 1 - Global variables

## Pros and Cons

**Pro** Super-easy to implement

**Con** Doesn't work when the statement is **nestable**

What happens when the **save** statement is **nested**?

```
int a=10, b=20;

save a {
  save b {
    /* ... */
  }
}
```

1. The first action of the first *save* assigns `r_save`
2. The first action of the **second** *save* **overwrites the value of `r_save`**
3. The generated code for restoring `b` is correct
4. The generated code for restoring `a` is **incorrect**: it uses the register with the old value of `b` instead!

# Method 2 - Global stack

We can patch the issues of method 1 by introducing a **global stack**

- Typical way to implement it: the **linked list library in collections.h**

```
t_list *save_stack = NULL;
/* ... */
save_statement :
   SAVE IDENTIFIER
   {
      int r_save = getNewRegister(program);
      int r_var = get_symbol_location(program, $2, 0);
      gen_add_instruction(program, save_stmt_reg, REG_0, r_var, CG_DIRECT_ALL);

      save_stack = addFirst(save_stack, INTDATA(r_save)); // push
   }
   code_block
   {
      int r_save = LINTDATA(save_stack);
      save_stack = removeFirst(save_stack); // pop

      int r_var = get_symbol_location(program, $2, 0);
      gen_add_instruction(program, r_var, REG_0, r_save, CG_DIRECT_ALL);

      free($2);
   }
;
```

# Method 2 - Global stack

## Pros and Cons

**Con** Super-cumbersome to implement

**Pro** Works when the statement is **nestable**

```
int a=10, b=20;

save a {
  save b {
    /* ... */
  }
}
```

1. 1st action of the 1st *save*:
   **push** the register ID for restoring `a`
2. 1st action of the 2nd *save*:
   **push** the register ID for restoring `b`
3. 2nd action of the 2nd *save*:
   **pop** the register ID for restoring `b`
4. 2nd action of the 1st *save*:
   **pop** the register ID for restoring `a`

# Method 3 - Semantic value as a variable

Sometimes you have one or more tokens **without a semantic value** in your syntax:

- In our example the token for the **save** keyword

Idea: use these **unused semantic values** to pass variables from **one semantic action to another**

Steps to follow:

1. Add a type declaration to the token in **Acse.y**
2. **Do <u>NOT</u> assign a semantic value in Acse.lex**
3. Use `$n` to access the variable of the token in the semantic actions

# Method 3 - Semantic value as a variable

Let's use this method in our running example:

```
%token <intval> SAVE

/* ... */

save_statement :
   SAVE IDENTIFIER
   {
      $1 = getNewRegister(program);
      int r_var = get_symbol_location(program, $2, 0);
      gen_add_instruction(program,
            $1, REG_0, r_var, CG_DIRECT_ALL);
   }
   code_block
   {
      int r_var = get_symbol_location(program, $2, 0);
      gen_add_instruction(program,
            r_var, REG_0, $1, CG_DIRECT_ALL);
      free($2);
   }
;
```

# Method 3 - Semantic value as a variable

## Pros and Cons

This is the **recommended method for most cases**:

- **Pro** Fairly simple to implement
- **Pro** Works when the statement is **nestable**
- **Con** Doesn't work when the variable must be accessible from **multiple rules**

It's uncommon that a variable needs to be accessible from multiple rules. But it can still happen!

Example:

- Consider the operator **saved_val()** that returns the previous value of the saved variable
- This operator is implemented as part of the ***exp*** grammar rule, not as part of the ***save_statement*** rule!

The solution to this problem is to use method 2: a global stack.

# Contents

# Control statements

The last part of ACSE we are left with: **control statements**

They are **compound statements** that control the **control flow** of other statements

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

control_statement : if_statement
                  | while_statement
                  | do_while_statement SEMI
                  | return_statement SEMI
;
```

# Branches

### A quick refresh

In **intermediate language** and assembly we modify the control flow with **branches** or **jumps** (same thing, different names).

Normally, instructions are executed in the order they appear...

- We use **branches** to **change** the next instruction executed
- This instruction is called the **destination** of the branch

At execution time, a branch can either be **taken** or **not taken**

- This depends on the **condition** of the branch

If you forgot how branches (or jumps) work, go back to the first set of ACSE slides!

# Branches

The branch's destination is identified by a **label**.

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**

- 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 labels

**Forward branch**

1. Create the label with *newLabel()*
2. Generate the branch passing the newly created label structure
3. Insert the label in the program by using *assignLabel()* just before generating the destination statement

**Compiler code**

```
t_axe_label *label = newLabel(program);
gen_bt_instruction(program, label, 0);

// ...
// more code generation
// ...

assignLabel(program, label);
```

**Compiler output**

```

    BT L0

    // ...
    // more code
    // ...

L0:
```

# Creating labels

**Backward branch**

1. Create and insert the label in the program with *assignNewLabel()*
2. Generate the branch instruction when needed

**Compiler code**

```
t_axe_label *label
label = assignNewLabel(program);

// ...
// more code generation
// ...

gen_bt_instruction(program, label, 0);
```

**Compiler output**

```
L0:

    // ...
    // more code
    // ...

    BT L0
```

# Today's agenda

All semantic actions for control structures do the same thing:

- They **add branches** around a code block
- These branch implement the effect of the loop condition

Today we look at how to generate code for conditional and looping structures, using as an example the three control structures in ACSE:

- *if*
- *while*
- *do-while*

# Contents

# Semantics of the statement

Let's start by analyzing 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**

*We will ignore the else part in the lecture to save some time. Look at the extra slides to know more.

# Translation to intermediate language

Let us look at the following example program:

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

The existing semantic actions generate the following **partial** code:

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

SUBI R0 R1 #10
SEQ R2 0           // a == 10

ADDI R3 R0 #10
WRITE R3 0         // write(10);
```

Let us simulate by hand the work the compiler does:

- Where do we insert the **branch instructions?**
- Which ones do we insert?
- Do we need to insert other instructions that are not branches?

# Translation to intermediate language

Note that the condition expression value is in the **R2 register**.

- We need to check if the value of R2 is **zero**
- If it is, **we skip over the body of the statement**

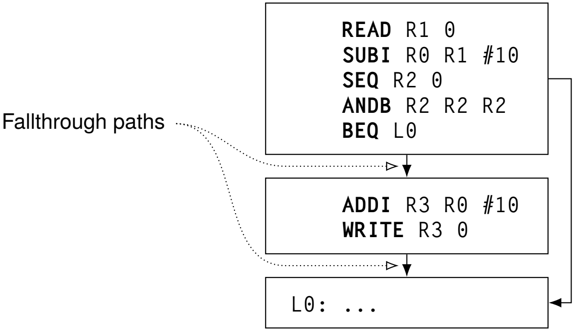
```
READ R1 0          // read(a);

SUBI R0 R1 #10
SEQ R2 0           // a == 10

ANDB R2 R2 R2      // update the Z flag
BEQ L0             // if Z flag is set, go to L0

ADDI R3 R0 #10
WRITE R3 0         // write(10);

L0: // rest of the program's code
```

- The `ANDB R2 R2 R2` instruction is a common trick to update the Z flag without changing any other register

# Control flow graphs

We can visualize the possible execution paths using **control flow graphs**

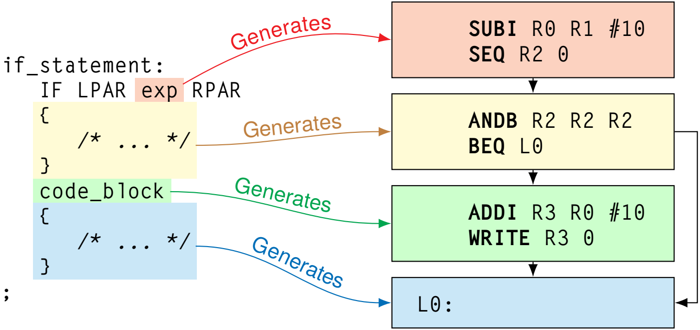
- They are simply a form of flow charts
- **Fallthrough:** when a basic block flows into another because of the natural order of the instructions (and not because of an explicit branch)



# Semantic actions

Now, let's write the code for handling the code generation:

- We need two semantic actions:
  - One for the code **before** the body
  - One for assigning the label **after** the body




**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:

- 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         {  /* ... */  }
;
```

# Semantic actions

```
/* ... */
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 = gen_load_immediate(program, $3.value);

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

# Contents

# A common task

Let us focus for a moment on this piece of code found in the implementation of the *if* statement:

```
t_axe_label *label = newLabel(program);
t_axe_expression exp = /* ... */;

int r;
if ($3.expression_type == REGISTER)
  r = $3.value;
else
  r = gen_load_immediate(program, $3.value);

gen_andb_instruction(program, r, r, r, CG_DIRECT_ALL);
gen_beq_instruction(program, label, 0);
/* ... */
assignLabel(program, label);
```

When the exp. type is **IMMEDIATE**, we **materialize the expression**

- Its value is tested at **runtime** by the ANDB instruction
- This is needed for the **conditional branch** to work

# The code that gets generated

Let's look at the IR code that we generate when the exp. is IMMEDIATE (constant):

- In this example, the expression's value is 5555:

```
    ADDI R1 R0 #5555  /* materialize the constant */
    ANDB R1 R1 R1     /* update the Z flag */
    BEQ L0            /* branch if Z is set */
    /* ... */
L0:
```

We can immediately see that this code is **inefficient**:

- The ADDI instruction already updates the Z flag
  - The extra ANDB is unnecessary
- R1 is always assigned to the same value (which is $\neq 0$)

$\Rightarrow$ **The Z flag is always clear** (i.e. set to zero)

$\Rightarrow$ **The BEQ branch is <u>never</u> taken**

# Pre-computed branches

Since the branch condition is **constant**, we already know in advance if the branch will be taken or not

- This means we can check the condition at **compile time**
- Instead of generating a conditional branch, we generate an **unconditional** branch

```
    ADDI R1 R0 #5555
    BEQ L0
    /* code */
L0:
```

Never branches →

```
    /* code */
```

```
    ADDI R1 R0 #0
    BEQ L0
    /* code */
L0:
```

Always branches →

```
    BT L0
    /* code */
L0:
```

# Back to the code

Let’s implement this optimization in the ACSE compiler:

```
/* ... */
if_stmt:
  IF LPAR exp RPAR
  {
    $1 = newLabel(program);

    if ($3.expression_type == IMMEDIATE) {
      if ($3.value == 0) {                    Constant condition
        gen_bt_instruction(program, $1, 0);
      } else {
        // do nothing, no branch is required
      }
    } else {
      int r = $3.value;                       Non-constant condition
      gen_andb_instruction(program,
          r, r, r, CG_DIRECT_ALL);
      gen_beq_instruction(program, $1, 0);
    }
  }
  /* ... */
```

# Contents

# *while*: Grammar

Let's continue our exploration by analyzing the *while* statement.

- This statement **repeats** the statements it contains until a given condition becomes false
- It is a **looping construct**

The grammar is very simple:

```
while_statement: WHILE LPAR exp RPAR code_block;
```

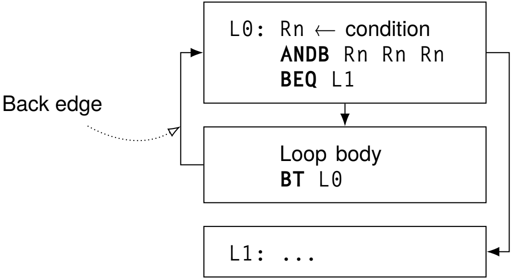
# The code we want to generate

A loop is implemented using a **branch** that goes in **backwards direction** with respect to the normal control flow

- In the control flow graph, the arrow representing this particular branch is called the **back edge**

The condition is checked at the beginning of the loop.

- If it is false, a BEQ instruction breaks the loop.



# Semantic actions

Again we need to share information between semantic actions:

- The label for breaking out of the loop
- The label at the beginning of the loop for continuing it

We use the semantic action of the WHILE terminal to store this information.

**axe_struct.h**

```
typedef struct {
   t_axe_label *label_condition;
   t_axe_label *label_end;
} t_while_statement;
```

**Acse.y**

```
%union {
   /* ... */
   t_while_statement while_stmt;
}
/* ... */
%token <while_stmt> WHILE
```

# Semantic actions

```
while_statement:
  WHILE
  {
    $1.label_condition = assignNewLabel(program);
  }
  LPAR exp RPAR
  {
    if ($4.expression_type == IMMEDIATE)
      gen_load_immediate(program, $4.value);
    else
      gen_andb_instruction(program,
        $4.value, $4.value, $4.value, CG_DIRECT_ALL);

    $1.label_end = newLabel(program);
    gen_beq_instruction(program, $1.label_end, 0);
  }
  code_block
  {
    gen_bt_instruction(program, $1.label_condition, 0);
    assignLabel(program, $1.label_end);
  }
;
```

# Grammar & control flow

*do-while* is similar to *while* but the condition check is done at the end

- It simplifies the control flow graph, now we need just one label
- The branch instruction is BNE because it shall be taken when the condition is **not zero**
- The loop body always executes at least once

```
do_while_statement: DO code_block WHILE LPAR exp RPAR;
```

```
L0: Loop body
    Rn ← condition
    ANDB Rn Rn Rn
    BNE L0
```

```
...
```

# Semantic actions

```
%token <label> DO

/* ... */

do_while_statement:
  DO
  {
    $1 = assignNewLabel(program);
  }
  code_block WHILE LPAR exp RPAR
  {
    if ($4.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);
  }
;
```

# Contents

# A simple exercise

Let's implement the **execute-when statement**:

- A code block is **executed** only **when** an expression is true
- The expression is specified **after** the code block...
- ...but it must be evaluated **before** the block is executed

Basically the same as an **if** statement but **reversed**

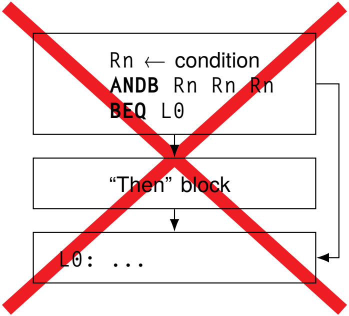
```
int a;                          int a;
read(a);                        read(a);
execute {              =        if (a < 0) {
  write(a);                       write(a);
} when (a < 0);                 }
```

The grammar of *execute-when* is:

```
exec_when_statement:
  EXECUTE code_block WHEN LPAR exp RPAR SEMI;
```

# The control flow we wish

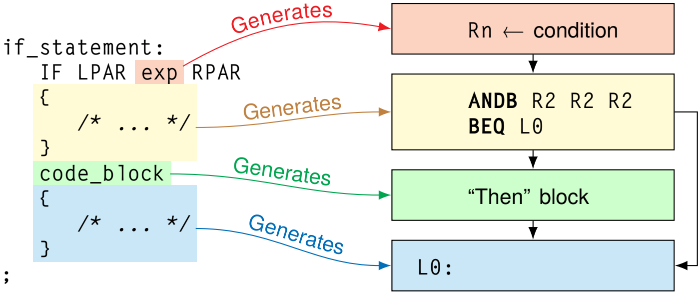
- In theory *execute-when*'s control flow graph is the same as the one used for *if* statements.
- In practice we **cannot generate this kind of code from an execute-when statement!**
- Why?



# The control flow we wish

ACSE is a **syntactic-directed translator**

- The code appears in the program in the order it is generated




The various semantic actions and non-terminals must appear in the right order

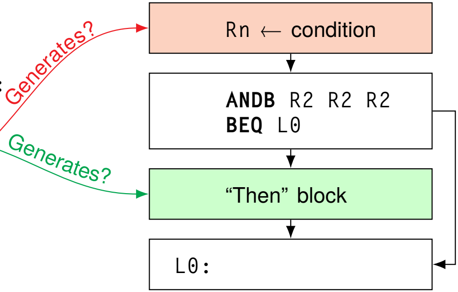
- Notice that **the arrows do not cross**

# The control flow we cannot achieve

For the **execute-when** statement, this is not the case!

- The code for computing the condition should be **before** the block
- But the non-terminal responsible for it appears **after**
- Remember: it's the **reduction of non-terminals** that triggers semantic actions and in turn **code generation**

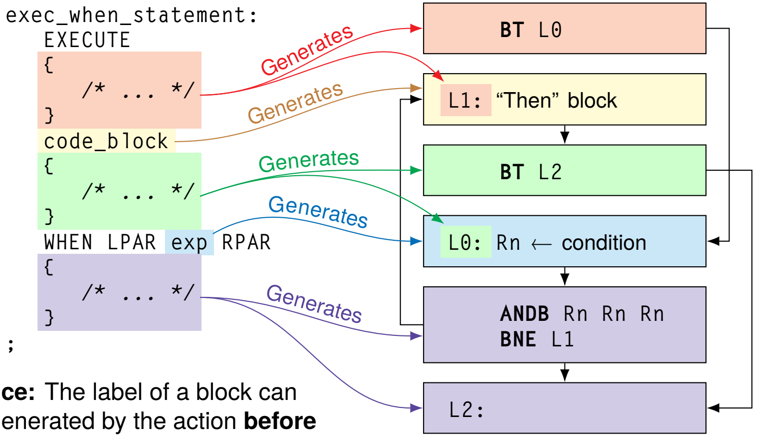
```
exec_when_statement:
    EXECUTE
    code_block
    WHEN LPAR
    exp
    RPAR SEMI
;
```



# The solution of the puzzle

This doesn’t mean we can’t implement this statement...

- But we need to work around the issue by **adding extra branches**




**Notice:** The label of a block can be generated by the action **before** the one which generates its body

# Tokens and semantic actions

This time we need **3 labels** shared amongst the actions.

- We associate them with the EXECUTE terminal symbol

**Acse.lex**

```
"execute"   { return EXECUTE; }
"when"      { return WHEN;    }
```

**axe_struct.h**

```
typedef struct {
   t_axe_label *l_block;
   t_axe_label *l_exp;
   t_axe_label *l_exit;
} t_exec_when;
```

L1 in the previous picture

L0 in the previous picture

L2 in the previous picture

**Acse.y**

```
%union {
   /* ... */
   t_exec_when exec_when_data;
}
/* ... */
%token <exec_when_data> EXECUTE
%token WHEN
```

# Semantic actions

```
exec_when_statement:
  EXECUTE
  {
    $1.l_exp = newLabel(program);
    gen_bt_instruction(program, $1.l_exp, 0);
    $1.l_block = assignNewLabel(program);
  }
  code_block
  {
    $1.l_exit = newLabel(program);
    gen_bt_instruction(program, $1.l_exit, 0);
    assignLabel(program, $1.l_exp);
  }
  WHEN LPAR exp RPAR
  {
    if ($7.expression_type == IMMEDIATE) {
      if ($7.value != 0)
        gen_bt_instruction(program, $1.l_block, 0);
    } else {
      gen_andb_instruction(program,
          $7.value, $7.value, $7.value, CG_DIRECT_ALL);
      gen_bne_instruction(program, $1.l_block, 0);
    }
    assignLabel(program, $1.l_exit);
  }
;
```

# Contents

# Homework 1/3

### Exam term 2009-09-11

Implement the *write_array* statement:

- Only parameter: array identifier
- Prints out all items in the array
- **Compilation error** if the parameter is not an array*

```
int a[3];

a[0] = 11;
a[1] = 2;
a[2] = 4;
write_array(a);
```

Prints 11, 2, 4

Tips:

- Use *getVariable()* to get the array size, you can assume it is $> 0$
- The generated code is a do-while loop with a predefined body

*Not in the original exam term, but extremely common in recent exam terms!

# Homework 2/3

Implement the *forall* loop:

- The loop syntax specifies a variable and an initial and final value
- The variable starts at the initial value, and is incremented or decremented by 1 at each loop iteration
- The loop continues as long as the variable is $<$ the final value
- **Note:** The statement must allow nesting!
- **Note:** Ignore that the final value may change at each iteration

```
int i, j, x;
read(x);

forall (i = 0 to 10 + x)
  forall (j = 10 + x downto 0) {
    write(i);
    write(j);
  }
```

# Homework 3/3

**Exam term 2010-02-01**

Implement a C-style **for** loop:

```
int i, j;

for (i=0, j=100; i<10 && j>85; i=i+1, j=j-2) {
  write(i + j);
  write(i - j);
}
```

You can assume that:

- the first and last clauses are **lists of assignments** used to initialize and update variables respectively
- the second clause is the **optional loop condition** (no loop condition means it's always true)

**Hint**: a *for*-loop is equivalent to a while loop...

# How to tackle the exam

**Before the exam:**

1. Have a lot of **exercise** with past exam terms
2. Do not solve them on paper only! **Use your PC**, and test the modified ACSE to check if the solution is right
3. Try to look at the solution only **after** you managed to solve the exam by yourself
   - If your solution is different than the official one, try to understand why it is different
   - The official solution might not always be the best one!
4. Participate in the tutoring sessions (held before each exam, dates still pending)
   - Legends say that good things come to those who participate in the tutoring
   - But you'll have to see my face again

# How to tackle the exam

**At the exam:**

1. Read the text **very well**
2. Understand the code you need to generate by **rewriting the statement in terms of the basic LANCE syntax**
3. Draw the control-flow-graph and decide which semantic action generates which block (if needed)
4. Forgot how to translate some construct to assembly? **Look at Acse.y for tips**

Do not fear! Every exam can be solved by applying the concepts we have seen in these 5 lectures.

- For deeper treatment of some topics, see the **Toolchain Manual**
- Also in the toolchain manual: **list of common mistakes**

# Conclusion

Exam terms tend to fall in the following categories:

- New expression operators
  - Often bit manipulations
  - Tend to be simpler
- New statements
  - Some might be disguised as expressions!
  - Typically syntactic sugar for a moderately complex procedure
  - Often involve generating an **internal** loop that iterates over an array
- Wildcard (exercises not fitting in the above patterns)
  - More frequent in the old days, rare today to limit bloodbaths

**Next Lecture:**

Solutions of selected exam terms

# Contents

# Let us add the else part

Now let's add the option to have an *else* part in *if* statements.

Since the *else* part is **optional**, we now have two non-terminals:

```
if_statement : if_stmt
             | if_stmt ELSE code_block
             ;
if_stmt      : IF LPAR exp RPAR code_block
             ;
```

**Tricky question...**

Could the grammar have been this one?

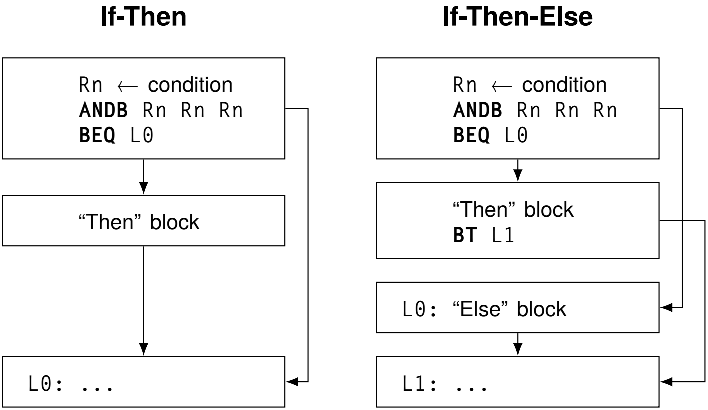
```
if_statement : IF LPAR exp RPAR code_block
             | IF LPAR exp RPAR code_block ELSE code_block
             ;
```

**Important**: Consider what happens when we add the semantic actions. Does any ambiguity arise?

# Modifications to the semantic actions

New semantics:

- When the condition is **false**, the code branches to the *else* block
- After the *then* block, we need to **skip** the *else* block



# Modifications to the semantic actions

The code generation is split amongst the various rules...

- **if_stmt**:
  - Generates the common code to both cases
  - The “condition = 0” label is **left unassigned**
  - **The semantic value of if_stmt stores the label**
- **if_statement**:
  - No else part?
    - No additional code is generated
    - The “condition = 0” label points to first instruction after the loop
  - There is an else part?
    - We generate a jump over the *else* block
    - The “condition = 0” label points to first instruction before the else block

# Semantic actions

```
%type <label> if_stmt
%token <label> ELSE
%token <label> IF

/* ... */

if_stmt:
  IF LPAR exp RPAR
  {
    int r =
      $3.expression_type == IMMEDIATE
        ? $3.value
        : gen_load_immediate(
            program, $3.value);

    gen_andb_instruction(program,
        r, r, r, CG_DIRECT_ALL);

    $1 = newLabel(program);
    gen_beq_instruction(program, $1, 0);
  }
  code_block
  {
    $$ = $1;
  }
;
```

**The ternary operator in C**

The ternary operator chooses between two values depending on a condition:

$\langle$condition$\rangle$ ? $\langle$val. if true$\rangle$ : $\langle$val. if false$\rangle$

```
if_statement:
  if_stmt
  {
    assignLabel(program, $1);
  }
  | if_stmt ELSE
  {
    $2 = newLabel(program);
    gen_bt_instruction(program, $2, 0);
    assignLabel(program, $1);
  }
  code_block
  {
    assignLabel(program, $2);
  }
;
```

# Final note about ambiguities

The syntax of C-like *if* statements are **inherently ambiguous**:

- This notorious problem is called **dangling else**

To which *if* statement does the final *else* belong?

```
int a, b;
if (a == 0)
  if (b == 10)
    write(a);
  else
    write(b);
```

```
int a, b;
if (a == 0)
  if (b == 10)
    write(a);
else
  write(b);
```

Two ways to solve the issue:

- Add precedence declarations that set a higher precedence for the rule with ELSE
- Add an **%expect** declaration ← what ACSE does

# Expect declarations

The **%expect** declaration tells Bison that a certain number of shift-reduce conflicts must be **expected** in the grammar.

- The conflicts will not generate any warning
- Bison solves these conflicts automatically (typically, the generated parser always reduces)
- If the number of shift-reduce conflicts is wrong, Bison halts with an error!

The expect declaration is found at the beginning of **Acse.y**.