M1 Computer Science
Language Theory:
Compilation

Labwork 4 – Conditional Statement

These labworks are automatically assessed based on an archive you have to deliver on time on the Moodle webpage. To make the archive, you have to type the command:

```
> make archive
```

This produces a file named `archive.tgz` that you have to deposit. The compilation labwork are roughly held each 2 weeks and the delivery date is usually on sunday before the next labwork week.

The three first labworks have introduced the whole procedure to extend our language:

1. declare the new required tokens in `parser.mly`,
2. generate these tokens in `lexer.mll`,
3. write the rules required by the extension in `parser.mly`,
4. add, in the rule action, the checks for the Semantic Analysis and the generation of the ASTs (in `parser.mly`),
5. update the module `comp.ml` with the code generation for the extended ASTs.

In addition, these labworks have dermonstrated the success of an incremental strategy to develop a compiler. A real programming language is often big and it is mostly impossible to detect all issues coming with the language. Instead, we start developing a small working core and then we add extensions step by step.

In this labwork, we will perform new steps in order to add conditional construction to AutoCell and therefore we will still apply the same strategy and method.

1 Simple Condition

In AutoCell, a condition has the following form:

```
if sum = 3 then
    [0, 0] := 1
end

if [0, 0] = 1 then
    if sum < 2 then
        [0, 0] := 0
```

```
            sum := 0
        end
    end
```

The condition is very to usual languages: an `if` keyword, a conditional expression, a keyword `then`, a sequence (possibly empty) of statements and the keyword `end` to complete the condition construction. Clearly, a condition is a statement, not an expression as it does not produce any value. Its role is mainly to manage the execution flow according to the condition.

The condition is a comparison between two expressions. Thhe supported comparators are: "=", "! =", "<", "<=", "<" and ">=".

To Do modify the Lexical Analysis and te Syntactic Analysis to support the conditional statement. Test it with `autos/simpleif.auto`.

To Do The AST module `ast.ml` already contains ASTs for conditional statement, `IF_THEN`, and ASTs for conditions, `cond`. Change the actions in `parser.mly` to build the AST.

To Do Add the support for the `else` keyword to our condition statement.

```
if sum < 2 then
    [0, 0] := 0
else
    [0, 0] := 1
end
```

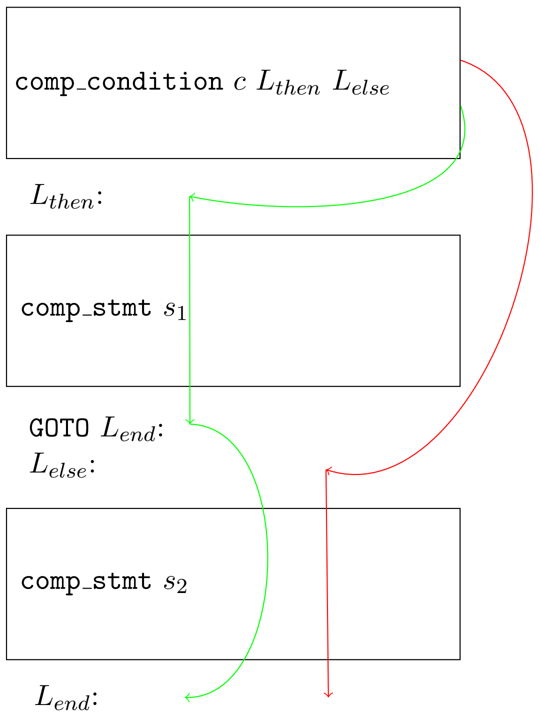
Test it with `autos/ifelse.auto`.

2 Generation

In order to generate quadruplets for a conditional statement, we will to implement the condition and the only way is to use the conditional branches in the quadruplets: `GOTO_EQ`, `GOTO_NE`, etc. Notice that a `GOTO_`*XXX* can be associated with each comparator `COMP_`*XXX*.

We need also to visualize several execution paths (when the condition true, when the condition is false) that will be rendered as a linear sequence of quadruplets. To help visualize the different execution flows, we advise to first represent them with a generation model. This model shows the quadruplets as a sequence (including sequences that are unknown corresponding to substatements) and with arrows showing the different execution paths.

The generation model below is proposed for the AST `IF_THEN`(c, s_1, s_2):



The boxes figures out the quadruplets result of the compilation of c, s_1 and s_2. The green edges represent the execution path taken when the condition is true, the red edges when the condition is false. Observe that:

- We need three labels L_{then}, L_{else} and L_{end},
- The branch quadruplet is generated from `comp_cond` and this is why L_{then} and L_{else} as passed as arguments.
- A `GOTO` quadruplet has been inserted between s_1 and s_2 quadruplets to avoid to execute the *else* part after the execution of the *then* part.

We are mostly done, we only need:

- a way to insert labels in the quadruplet: done with the pseudo quadruplet `LABEL` *number*;
- a way to generate unique label numbers: function `new_lab ()`.

To Do Implement the quadruplet generation fot the conditions in the function `comp_cond`. Notice that only the condition `COMP` has to be implemented: the other types of condition might be implemented in a subsequent labwork. You have to keep in mind the goal of the quadruplets you have to generate: if the condition is true, a branch has to be done to the label L_{then}; otherwise, a branch to the label L_{else} is performed. It is important to generate both branch quadruplets.

Note The quadruplets generated by `comp_cond` are not very beautiful but this works. Yet, in a compiler, a subsequent very simple optimization (called *peephole optimization*) can nicefully fix this problem.

To Do Now, helped by the generation model proposed above, and the function `comp_cond`, you can generate the quadruplets for a condition, `IF_THEN`(c, s_1, s_2) in the function `comp_stmt`. You can look and check the generated quadruplets with `autos/simpleif.auto` and `autos/ifelse.auto`.

3 Multiple Conditions

In this last exercise, we will extend the conditional statemenet with the `elseif` keyword, as exposed in the example below:

```
if [0, 0] = 1 then
    if sum < 2 then
        [0, 0] := 0
    elsif sum > 3 then
        [0, 0] := 0
    end
else
    if sum = 3 then
        [0, 0] := 1
    end
end
```

The *then* part of a conditional statement may be followed by `else` (achieved in the previous section) but also by a sequence of `elsif`. Each `elsif` has a condition, a `then` keyword and its own sequence of statements. Ultimately, this sequence of `elseif` can be completed by an `else` (but it is not mandatory).

To Do Modify the front-end of `autocc` to parse the `elsif` construction. Test if with `autos/conway.auto`.

To Do Modify `parser.mly` to generate the AST for supporting `elsif`: *you do not need to modify the definition of ASTs for this!*. Test if with `autos/conway.auto`.

To Do Now you can run the *Conway's Game Life* cellular automaton written in AutoCell:

```
> ./autocc autos/conway.auto
> ./autoas autos/conway.s
> ./autocell autos/conway.exe maps/MAP.map
```

You can test with maps: `conway-1.map`, `conway-2.map`, ..., `conway-5.map` and enjoy.