M1 Computer Science
Language Theory:
Compilation

Labwork 5 – Extensions to CellLang

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.

This labwork is optional.
It will accounted as additional bonus score on the final mark.

Several extensions are proposed in the following where you have to use the method applied until now:

1. Understand the proposed syntax and extension.
2. If needed, add required tokens to `parser.mly`.
3. If needed, add the token scanning to `lexer.mll`.
4. Add the required rules to `parser.mly`.
5. Extend the AST and build them in `parser.mly`.
6. Provide the translation to quads in `comp.ml`.

The proposed extensions are independent and can be implemented in any order.

1 Logical NOT, OR and AND

This extension aims to implement logicial operators in conditions with the following syntax:

- `! x = 3` - logical NOT on the following predicate.
- `x = 3 & x < y` – logical AND on the predicates in both sides.
- `x = 3 | x = 5` – logical OR on the predicates in both side.

- `(x = 3 | x = 5) & x < y` – parentheses in conditions.

Several observations need to be done on these operators:

- These operators applies to predicates (comparisons) and not on variables (there is not Booleans in CellLang).
- `&` and `|` are left-associative.
- `&` has a higher priority than `|`.
- They support shortcut evaluation: for `&`, if the first predicate is false, the second predicate is not evaluated; for `|`, if the first predicate is true, the second predicate is not evaluated.

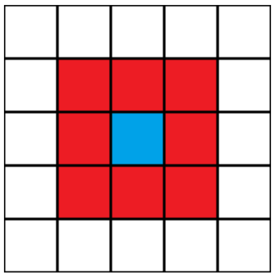
Hint: these operators can benefit a lot from the way the `comp_cond` function is defined. For example, the following translation can be used for the logical NOT:

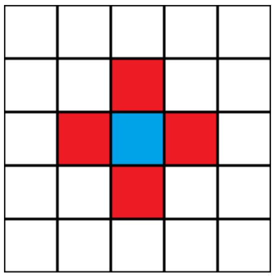
$$\texttt{comp\_cond}\ NOT(c)\ L_{true}\ L_{false} = \texttt{comp\_cond}\ c\ L_{false}\ L_{true}$$

2 Loop and Iterators

An important part of CellLang programs is to perform the same calculation with the neighbor cells that are around the current cell.

Although there are a lot of different ways to select which neighbor cells to work with, there are two main neighborhood traversal policies:


Moore's neighborhood


Von Neumann's neighborhood

This extensions aims to provide loop constructions to implement these neighborhood policies. They will have the following syntax:

- *for* `i` *in* `moore` *do* `s = s + i` *enddo*
- *for* `i` *in* `vonneumann` *do* `s = s + i` *enddo*

The statements between `do` and `enddo` is repeated as many times as there are cells in the neighborhood policy (the order of traversal is undefined). The variable `i` (or any other identifier) gets the value of the current neighbor cell in the loop. There is no constraints on the identifier `i`: it could have been already declared or not.

Hint 1: the constants for the directions – `pNORTH`, `pNORTEAST`, ..., have been cleverly chosen to make easier the implementation of these loops.

Hint 2: with the VM command `cGET`, the lookup direction was determined by a constant direction stored in *b* argument. To implement this extension, we need to be able to pass a direction stored in a register. This is implemented by the command `cGETV` (value 1005): *b* argument is the register number containing the direction.

3 Peephole Optimization

This extension does not concern the syntax but the quality of the produced quads. It specially addresses the problem of condition compilation which generated code is often not very beautiful (example in OCAML below):

```
[
        GOTO_EQ (l_then, 3, 4);
        GOTO (l_else);
        LABEL (l_then)
]
```

Clearly, this quad sequence can be reformulated as (remark the inversion of the condition):

```
[
        GOTO_NE (l_else, 3, 4);
        LABEL (l_then)
]
```

Another interesting pattern for optimization is the following:

```
[
        GOTO_EQ (l);
        LABEL (l)
]
```

That can be reduced to:

```
[
        LABEL (l)
]
```

Implementing such an optimization is quite easy and is called a peephole optimization: the quad program is traversed and a small window of quad is observed at a time in order to find the pattern of optimization. When the pattern is found, the concerned quads are replaced with the optimized version and the program traversal go on with the rest of the quads.

Such an optimization is quite easy to implement in OCAML thanks to the powerful `match` instruction. For example, to detect an addition followed by a subtraction of the same quantity, one could write:

```
match program with
| ADD(x, _, y)::SUB(a, b, c)::rest
        when x = a && x == b && y == c -> ...
| ...
```

To Do: Implements the peephole optimization for the propoposed patterns on `GOTO` quads.