Labwork 3 – Code Generation

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 :

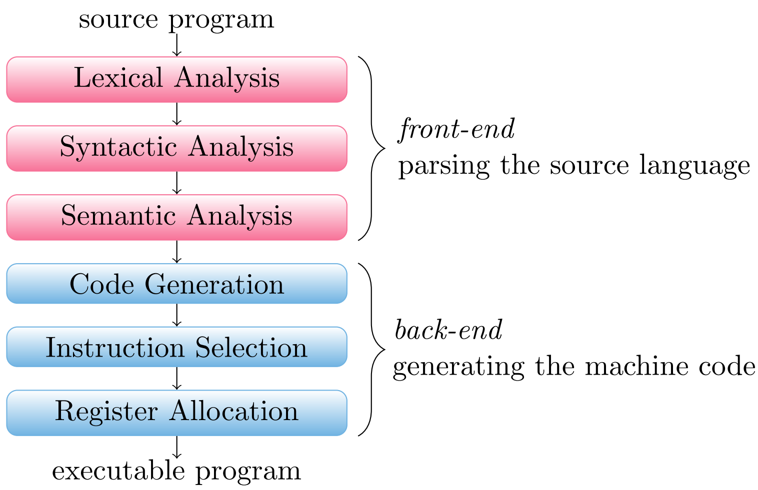
```
> make archive
```

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 aims to generate code for AutoCell : (a) first an intermediate representation of the program is built (tree-based) and (b) this representation is used in the back-end to generate the qadruplets.

1 The Abstract Syntactic Tree

1.1 Description



Referring to the compiler structure above, we have at this point implemented the *Lexical and the Syntactic analyses*. It remains to implement the *Semantic Analysis* that checks non-syntatic properties like identifier existence, typing, etc. The goal of these analyses (forming the front-end) is to ensure to obtain (a) a completely checked program and (b) to provide an easy-to-process representation of the program.

This representation is based on trees (a) that are easy to build when a grammar is parsed bottom-up and (b) that explicitely represent the program without the parsing effort. We could have used derivation trees but they contain lots of elements, useful for the sequential representation of the program, but useless to record the program semantic as trees. This is why this representation is called *Abtract Syntactic Tree* (AST) that benefit from the ability of `OCAML` to manage trees.

In our implementation, the ASTs are declared in `ast.ml`. There are mainly two kinds of ASTs : the expressions `expr` and the statements `stmt`. For now, we will only describe some nodes of these ASTs.

For the statements, we get :

NOP represents no action (useful to represent null statement).

SET_CELL (0, e) represents the assignment of cell $[0, 0] := e$ (the first argument must be 0 for now and e is an expression AST).

SET_VAR (x, e) represents the assignment of variable $x := e$ with x being the number of the register containing the variable and e an expression AST.

SEQ (s_1, s_2) is not visible in the program but is clearly needed : it represents the sequence of two statement, S_1 then s_2, and allows us, by composition, to represent sequences of statements.

For the expressions, we get :

NONE represents no calculation (useful to repesent null expression).

CST (i) represents an expression evaluating to the constant i.

VAR (x) represents an expression that evaluates to the value of the variable which register has number x.

CELL (0, x, y) represents the value of the cell around the current cell at relative position $[x, y]$.

NEG (e) represents the negation operation on the expression AST,

BINOP (ω, e_1, e_2) represents a binary operation like $e_1 + e_2$, $e_1 - e_2$, etc. ω may be one of OP_ADD, OP_SUB ; etc and e_1 and e_2 are expression ASTs.

Beware Some unrequired keywords are not represented. For example, the parentheses are only useful to manage priorities in sequential code but become useless when we have trees.

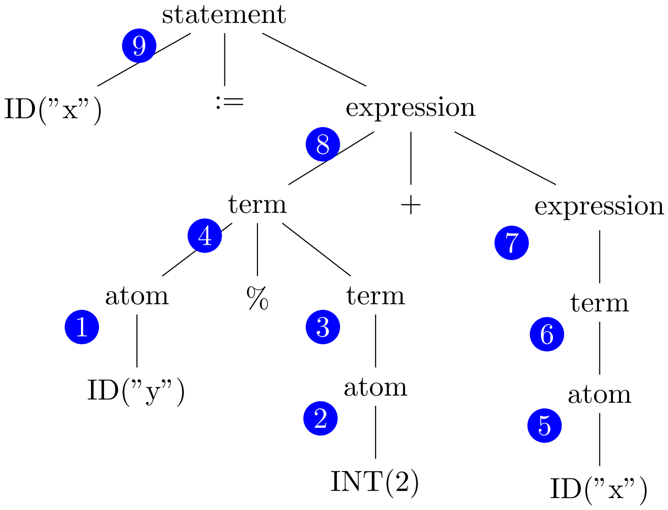
1.2 Generation

How do we generate these ASTs ? The short answer is : from the actions. Each time a grammar rule is parsed (*reduced*), the corresponding action is called and we can use it to build the tree and return it as the result of the action.

A longer and deeper answer is : the bottom-up order implemented by $LALR(k)$ analysis is perfectly adapted to build a tree ! Indeed, the subrule actions of a rule action (producing the subtree) are triggered before the current rule action. That is, when the current action is triggered to build a new tree node, the subtree nodes have already been built ! Let's take the code below :

```
x := y % 2 + x
```

Below is represented a possible corresponding derivation tree :

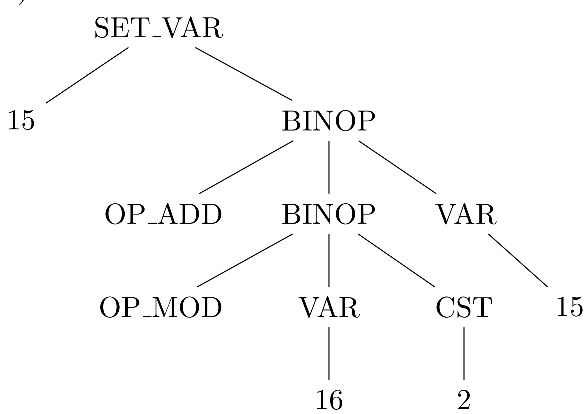


The numbers show in which order the reductions are performed to parse the statement, from (1) to (9) :

- **(1)** *atom* $\rightarrow ID$ is reduced and the AST $VAR(16)$ is built (y is stored in register 16).
- **(2)** *atom* $\rightarrow INT$ is reduced and $CST(2)$ is built.
- **(3)** *term* $\rightarrow$ *atom* is reduced and the AST $CST(2)$ is passed upper.
- **(4)** *term* $\rightarrow$ *atom* $PERCENT$ *term* is reduced, $BINOP(OP\_MOD, VAR(”y”), CST(2))$ can be assembled from subtrees $VAR(16)$ and $CST(2)$ from (1) and (2).
- **(5)** etc

In the end, we get the AST (of type `stmt`) :

```
SET_VAR(15,
      BINOP(OP_ADD,
            BINOP(OP_MOD,
                  VAR(16),
                  CST(2)
            ),
            VAR(15)
      )
)
```



With the register 15 containing x and 16 containing y.

But, how does it work in practice ? In fact, in `ocamlyacc`, receiving a semantic value is not limited to terminales (like `ID` or `INT`) but this applies also to non-terminals. In fact, the `OCAML` result of an action is used as the semantic value of the non-terminal on the left of the rule. Therefore, each symbol of a production is associated with a semantic value (all `OCAML` expression produces a value, `()` by default). To access the semantic value of the symbol of a production, special variables named `$`*i* are used with *i*, starting at 1, reprensenting the number of the symbol, in the production, we wish to get the semantic value from.

The example below shows the example of the modulo operator :

```
term: atom PERCENT term
        { BINOP(OP_MOD, $1, $3) }
```

The `atom` symbol (position 1, variable `$1`) and the `term` symbol (position 3, variable `$3`) are associated with an expression semantic value – type `expr`. These sub-expressions are combined together to form a new `BINOP` expression, result of the action, and representing the semantic value of the left `term` non-terminal.

1.3 Exercise

Until now, the production actions was implemented using `NONE` and `NOP` ASTs. In this exercise, we want to replace them by ASTs representing the Autolang program. Notice that some of these actions have already been provided in the original sources.

Another important feature, to implement in the actions, is the *Semantic Analysis.* AutoCell's semantic analysis is very simple as we have only to manage variables :
— Each variable is assigned to a register.
— Before being referred to in an expression, a variable has first to be initialized in a previous assignment.

To implement these checks, the following functions are available :

declare_var *id* – assigns a register for variable named *id* and returns its number.

get_var *id* – gets the register number assigned to variable *id* or -1 if there is none.

error *string* – raises an error stopping the compilation and displays the given *string* as message.

The questions below aims to fulfill the different actions in order to build the ASTs for expressions and statements. It is advised to test each added action with the compiler using the option `-ast` that stops the compilation and dumps the ASTs that have been built :

```
> ./autocc -ast autos/shift.auto
```

Notice that some actions have already been implemented and you should not have to change them. Notice also that some actions have only to transmit the tree from one non-terminal to the other one : for instance, a production of the form *expression* → *term* could only have as action { `$1` }.

The questions below are ordered to make easier the development and the testing of the actions. One or several test files are proposed with each item. So you have to implement and test :

1. The assignment to a variable (using `declare_var`) to get the register number that will contain the variable),
   test : `autos/varassign.auto`.
2. The variable used in an expression (getting their register number with `get_var`),
   test : `autos/vars.auto`, `autos/undeclared.auto`.
3. The negation operation (unary -),
   test : `autos/neg.auto`
4. The positive operation (unary +),
   test : no test
5. The binary operations (binary +, -, *, /, %),
   test : `autos/expr.auto`.

2 Performing the translation

The second part of this labwork addresses the problem of the translation, i.e. the compilation to generate quadruplets.

2.1 The compilation module

Everything is done inside the source file `comp.ml`. It contains one function for each kind of AST we have to translate :

comp_stmt compiles the statements,

comp_expr compiles the expressions.

In addition, the compiler provides the background double loops used to process all cells of the map. This code is generated by the function `compile` and the generated code is :

```
        INVOKE 4, 2, 3      // cSize (width in R2, height in R3)
        SETI R4, #1
        SETI R0, #0         // R0 = x
x_lab:
        SETI R1, #0         // R1 = y
y_lab:
        INVOKE 3, 0, 1      // cMOVE to (R0, R1) (x, y)

        // Autocell code

        ADD R1, R1, R4
        GOTO_LT y_lab, R1, R3

        ADD R0, R0, R4
        GOTO_LT x_lab, R0, R2
        STOP
```

This code asks the system for the dimension of the map (command `cSize`) and put them into (R2, R3). Then, it sets up a double loop to process cells of the map with (x, y) coordinates stored in (R0, R1). Using this code, it only remains to generate the Autocell code as the body of the internal loop.

Generating the code means we want to write a function that takes the AST as parameter and return a list of quadruplets, implementing the AST. These quadruplets are represented by the type `quad` in file `quad.ml` :

```
type quad =
        | ADD of int * int * int
        | SUB of int * int * int
        | MUL of int * int * int
        | DIV of int * int * int
        ...
```

So, in order to generate a quadruplet of addition that works on registers R_0, R_2 and R_3, one has to write, in OCAML :

```
        ADD(0, 2, 3)
```

And to put them in a list in order to get a sequence of quadruplets. For example, the translation of quadruplets of expression $x := y\%2 + x$ which AST is given in previous section (with x in R_{15} and y in R_{16}) would be :

As an AST :

```
SET_VAR(15,
    BINOP(OP_ADD,
        BINOP(OP_MOD,
            VAR(16),
            CST(2)
        ),
        VAR(15)
    )
)
```

In quadruplets assembly :

```
# x in R15, y in R16
SET R35, R16
SETI R36, #2
MOD R37, R35, R36
SET R38, R15
ADD R39, R37, R38
SET R15, R39
```

In OCAML :

```
[
    SET(35, 16);
    SETI(36, 2);
    MOD(37, 35, 36);
    SET(38, 15);
    ADD(39, 37, 38);
    SET(15, 39)
]
```

The `comp_stmt` looks like :

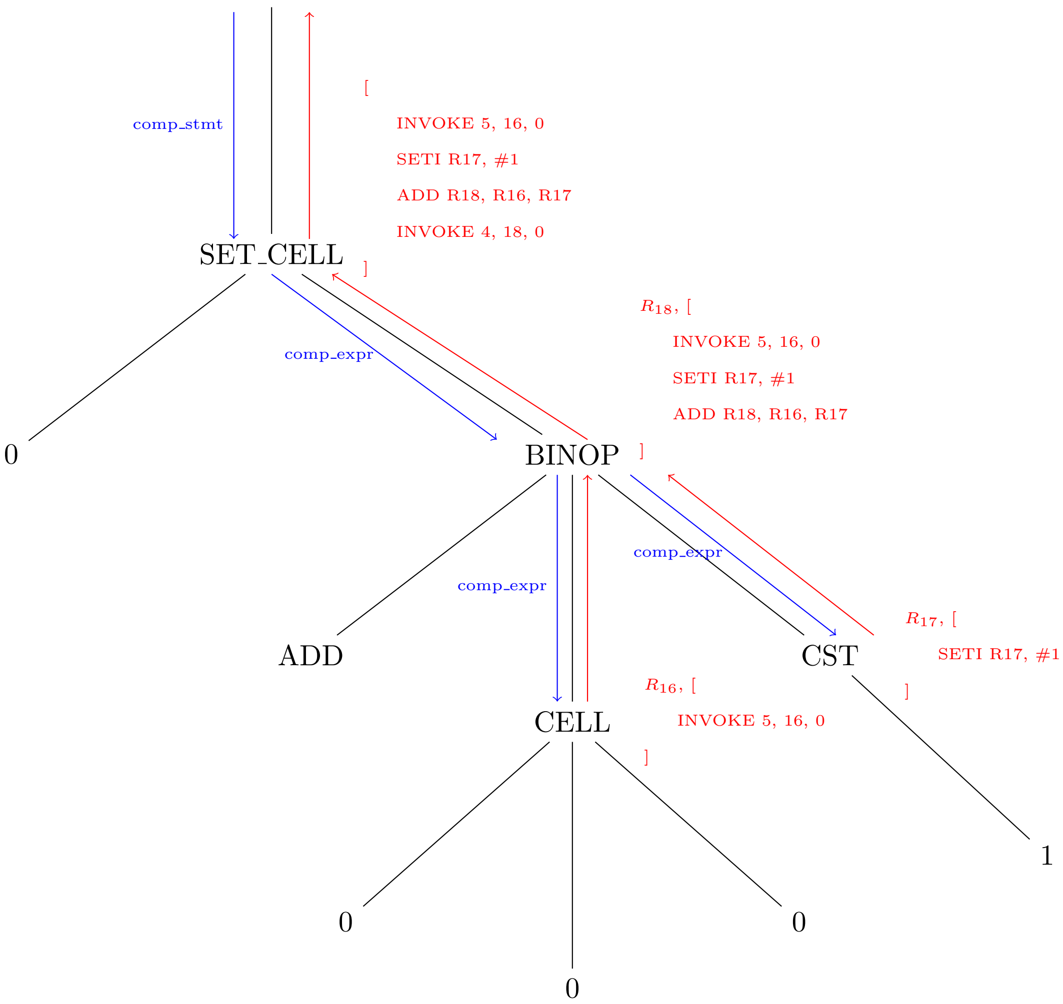
```
let rec comp_stmt s =
        match s with
        | NOP ->
                []
        | SET_CELL (f, e) ->
                let (v, q) = comp_expr e in
                q @ [
                        INVOKE (cSET, v, f)
                ]
        | SEQ (s1, s2) ->
                (comp_stmt s1) @ (comp_stmt s2)
```

The compilation of the statement `s` depends on its AST nature, that is tested with a `match`. If `s` is a `NOP`, it is translated as an empty list of quadruplets, `[]` : nothing to do. If `s` is a sequence `SEQ(`s_1, s_2`)` made of the statements `s1` then `s2` : the result in term of quadruplets is the concatenation of quadruplets implementing `s1` and `s2`.

When `s` is a cell assignment, `SET_CELL (f, e)` then an `INVOKE` to command `cSET` has to be performed. This call takes two arguments : the assigned field (`f` is provided by `s` as is) but the value v must be the number of the register containing the result of the evaluation of the expression AST `e`. This means that (a) we have to generate the code for the expression AST `e` and (b) that the result has to be stored in a register. This is why the compilation function for expressions, `comp_expr`, returns a pair of values, `(v, q)` where `v` is the register number that gets the result from the sequence of quadruplet `q`, corresponding to the translation of expression `e` (this effect will be detailed latter). In the end, the resulting quadruplet is the concatenation of the quadruplets producing v and the `INVOKE` quadruplet.

2.2 Example

To illustrate the whole picture, let's take the example below, that shows an AST, the `comp_XXX` functions and the generated quadruplets and register numbers (corresponding to code `[0, 0] := [0, 0] + 1`) :



In blue are depicted the recursive calls between `comp_`*XXX* functions : `comp_comp` is called on `SET_CELL` that, in turn, performs a call on `comp_expr` for `BINOP`. The latter performs two calls to `comp_expr`, one for each of its arguments, `CELL` and `CST`.

In red are the return of recursive calls with the returned value. The call to `comp_expr` for `CELL` returns 16, the register that contains the result of the translation and the quadruplets used to translate `CELL` : [`INVOKE 5, 16, 0`]. Notice, in this instruction, that the register number 16 is passed as argument to get the result of the call in and that 5 (first argument) is the code of command `cGET` system call.

The same is performed with expression `CST`. The quadruplets resulting from the translation stores the result in register 17 and the quadruplet sequence setting R_{17} is : [`SETI R17, #1`]. When an expression is a constant, its role is just to store the constant in a register to use it thereafter.

Then, the code for the `BINOP` can be generated based on the code produced for its arguments. The result is stored in register number 18. The code is obtained by concatenating code of both operands and by appending the addition itself. As the first argument has stored its result in register 16 and the second argument in register 17, the quadruplet `ADD R18, R16, R17` is added to the code obtained so far.

Finally, `comp_stmt` can produce its own code based on the expression on the right of the assignment and knowing that the result of the expression is stored in the register numbered 18. An `INVOKE` to command `cSET` (code 4) is generated using 18 (to get the assigned value) as its argument and 0 for the default field number. This instruction is added to the code produced so far. Notice the difference between translating an expression and a statement : the statement is not supposed to produce a value and, therefore, returns only a sequence of quadruplets.

The only problem that remains is : how tp obtain a register number, to store the result of the expression code, that is not already in use. The function called `new_reg ()` does the trick (in an ugly way – don't look to this code). This is illustrated in the `CELL` expression AST below :

```
let rec comp_expr e =
        match e with
        | CELL (f, x, y) ->
                let v = new_reg () in
                (v, [
                        INVOKE (cGET + f, v, pos x y)
                ])
```

In this code, a new free register is obtained by a call to `new_reg ()` and the resulting number is stored in `v` and used in the `INVOKE` quadruplet generation. One can also observe that the `comp_expr` function returns a pair (register number containing the result, sequence of quadruplet implementing the expression).

2.3 The exercise

To Do We will now complete the functions `comp_stmt` and `comp_expr` to generate the code for statement and expression ASTs that are not already supported. Namely, this include the AST constructions :

— `CST` (expression)
— `SET_VAR` (statement)
— `VAR` (expression)
— `NEG` (expression)
— `BINOP` (expression)

To check if your generation is valid, you can look to the generated `.s` files. For each file `autos/FILE.auto`, `autocc` generate an assembly file `autos/FILE.s`.

For testing, you can reuse the source files of the first part of the labwork :

— `autos/vars.auto`
— `autos/neg.auto`
— `autos/expr.auto`
— `autos/varassign.auto`