Exercise session 2
AST, Attributions, Types

2.1 Derivation trees and attributions

EXERCISE #1 ▶ **Arithmetic expressions**
Let us consider the following grammar (the end of an expression is a semicolon):

$$
\begin{aligned}
Z &\rightarrow E; \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow F \\
F &\rightarrow (E) \\
F &\rightarrow i
\end{aligned}
$$

- What are the derivation trees for $1+2+3;$, $1+2*3;$, $(1+2)*3;$?
- Attribute the grammar to evaluate arithmetic expressions.

EXERCISE #2 ▶ **Declarations of variables**
Write a grammar that accepts declarations of variables like:

```
int x=1;
float y,z;
int t;
float u,v=0;
```

and rejects:

```
int x, int y;
```

Then write an attribution that prints individual declarations (of the first case) like:

```
int x=1; float y; float z; int t; float u; float v=0;
```

EXERCISE #3 ▶ **Prefixed expressions**
Consider prefixed expressions like `* + * 3 4 5 7` (or `* + 1 2 * 3 4`) and assigments of such expressions to variables:
`a=* + * 3 4 5  7`. Identifiers are allowed in expressions.

- Give a grammar that recognizes lists of such assigments.
- Write derivations trees.
- Write grammar rules to compute the values of the expressions.
- Write grammar rules to construct infix assigments during parsing: the former assigment will be transformed into the *string* `a=(3 * 4 + 5)*7`. Be careful to avoid useless parentheses.
- Modify the attribution to verify that the use of each identifier is done after his first definition.

2.2 The Mu-language

The objective here is to be familiar with the grammar of the language we will compile.

EXERCISE #4 ▶ **Mu-grammar**
Here is the (simplified) grammar for the Mu language (expr are numerical or boolean expressions):

```
grammar Mu;

prog: vardecl_l block EOF #progRule;

vardecl_l: vardecl* #varDeclList;

vardecl: VAR id_l COL typee SCOL #varDecl;

id_l
    : ID              #idListBase
    | ID COM id_l     #idList
    ;

block: stat*   #statList;

stat
    : assignment
    | if_stat
    | while_stat
    | log
    | OTHER {print("unknown␣char:␣{}".format($OTHER.text))}
    ;

assignment: ID ASSIGN expr SCOL #assignStat;

if_stat: IF condition_block (ELSE IF condition_block)* (ELSE stat_block)? #
   ifStat;

condition_block: expr stat_block  #condBlock;

stat_block
    : OBRACE block CBRACE
    | stat
    ;

while_stat: WHILE expr stat_block #whileStat;

log: LOG expr SCOL #logStat;

expr
    : <assoc=right> expr POW expr          #powExpr
    | MINUS expr                           #unaryMinusExpr
    | NOT expr                             #notExpr
```

Write two valid programs for this grammar.

2.3 Typing

We recall (some of) the rules of the course for Typing:

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma(x) = t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x := e : \texttt{void}} \qquad \frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}}$$

EXERCISE #5 ▶ **Well typed**
Type the mini-while program:

```
var x1:int;
var x3:bool;
x1 := 3 ;
while (not x3) do
  x1 := x2 + 1 ;
  x3 := x3 and true
done
```