TD 2
AST, Attributions

2.1 Derivation trees and attributions

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

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

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

EXERCISE #2 ▸ **XML Files**

We give the following grammar:

```
L : E L
  |
E : A L B
  | ident
A : '<' ident '>'
B : '</' ident '>'
```

1. Give the derivation tree for the chain `<html><head>toto</head>titi</foo>`.
2. Attribute this grammar to verify that opening and closing tags refer to the same identifiers.

EXERCISE #3 ▸ **Variable declarations**
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; float y; float z; int t; float u; float v;
```

If time allows, extend it to print the initializations like:

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

EXERCISE #4 ▸ **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.
- If time allows: 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 its first definition.

2.2 The MiniC language

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

EXERCISE #5 ▸ **MiniC-grammar**

Here is the (simplified) grammar for the MiniC language (expr are numerical or boolean expressions):

```
grammar MiniC;

prog: function* EOF #progRule;

// For now, we don't have "real" functions, just the main() function
// that is the main program, with a hardcoded profile and final
// 'return 0'.
function: INTTYPE ID OPAR CPAR OBRACE vardecl_l block
       RETURN INT SCOL CBRACE #funcDef;

vardecl_l: vardecl* #varDeclList;

vardecl: typee id_l SCOL #varDecl;


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

block: stat* #statList;

stat
    : assignment SCOL
    | if_stat
    | while_stat
    | print_stat
    ;

assignment: ID ASSIGN expr #assignStat;

if_stat: IF OPAR expr CPAR then_block=stat_block
       (ELSE else_block=stat_block)? #ifStat;

stat_block
    : OBRACE block CBRACE
    | stat
    ;

while_stat: WHILE OPAR expr CPAR body=stat_block #whileStat;


print_stat
    : PRINTLN_INT OPAR expr CPAR SCOL #printlnintStat
    | PRINTLN_FLOAT OPAR expr CPAR SCOL #printlnfloatStat
    | PRINTLN_BOOL OPAR expr CPAR SCOL #printlnboolStat
    | PRINTLN_STRING OPAR expr CPAR SCOL #printlnstringStat
    ;

expr
    : MINUS expr #unaryMinusExpr
    | NOT expr #notExpr
    | expr myop=(MULT|DIV|MOD) expr #multiplicativeExpr
    | expr myop=(PLUS|MINUS) expr #additiveExpr
    | expr myop=(GT|LT|GTEQ|LTEQ) expr #relationalExpr
```

```
    | expr myop=(EQ|NEQ) expr #equalityExpr
    | expr AND expr #andExpr
    | expr OR expr #orExpr
    | atom #atomExpr
    ;

atom
    : OPAR expr CPAR #parExpr
    | INT #intAtom
    | FLOAT #floatAtom
    | (TRUE | FALSE) #booleanAtom
    | ID #idAtom
    | STRING #stringAtom
    ;

typee
    : mytype=(INTTYPE|FLOATTYPE|BOOLTYPE|STRINGTYPE) #basicType
    ;

OR : '||';
AND : '&&';
EQ : '==';
NEQ : '!=';
GT : '>';
LT : '<';
GTEQ : '>=';
LTEQ : '<=';
PLUS : '+';
MINUS : '-';
MULT : '*';
DIV : '/';
MOD : '%';
NOT : '!';

COL: ':';
SCOL : ';';
COM : ',';
ASSIGN : '=';
OPAR : '(';
CPAR : ')';
OBRACE : '{';
CBRACE : '}';

TRUE : 'true';
FALSE : 'false';
IF : 'if';
ELSE : 'else';
WHILE : 'while';
RETURN : 'return';
PRINTLN_INT : 'println_int';
PRINTLN_BOOL : 'println_bool';
PRINTLN_STRING : 'println_string';
PRINTLN_FLOAT : 'println_float';

INTTYPE: 'int';
FLOATTYPE: 'float';
STRINGTYPE: 'string';
BOOLTYPE : 'bool';

ID
 : [a-zA-Z_] [a-zA-Z_0-9]*
 ;

INT
 : [0-9]+
 ;

FLOAT
 : [0-9]+ '.' [0-9]*
 | '.' [0-9]+
 ;

STRING
 : '"' (~["\r\n]␣|␣'""')*␣'"'
 ;


COMMENT
// # is a comment in Mini-C, and used for #include in real C so that we ignore #include statements
 : ('#' | '//') ~[\r\n]* -> skip
 ;

SPACE
```

```
: [ \t\r\n] -> skip
;
```

Write a valid program for this grammar.