Chapter 2
Gammars and attributes

EXERCISE #1 ▶ **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;
```

Solution. There is many possibilities for the grammar, here is one (`prog` is the start symbol):

```
prog: decl+ EOF
decl := type varList TKSCOL
varList := varElt (TKCOL varElt)*
varElt := TKvarName | TKvarName TKEQ aValue
type := TKINT | TKFLOAT
aValue := TKINTVAL | TKFLOATVAL
```

where `TKINT` (resp. `TKFLOAT`) designs the token produced by lexing the chain "int" (resp. "float"), `EOF` the token produced by "end of file", `TKSCOL` ";", ...

`TKvarName` designs the token produced by lexing identifiers (strings). We will denote by `TKvarName.val` the associated chain. In the same manner, `TKINTVAL.val` denotes the int value associated to the token `TKINTVAL` token.

For the attribution, the information we need to collect are:

- The list of all variable names that are "under the same type declaration". This list can be collected though the use of an attribute of type string associated to the non terminal `varElt` (denoted by `varElt.id`) and an attribute of type string list associated to `varList` (denoted by `varList.listid`).
- An attribute that tells whether the non terminal `type` was an int or a float. This attribute, of Boolean type (true if it was an int declaration, else false), will be denoted by `type.isInt`.

Now let us describe how to produce and use these attributes from grammar rules. First, generating `type.isInt` is straightforward:

```
type := TKINT  { type.isInt <- true }
     | TKFLOAT { type.isInt <- false }
```

Keeping track of a given string name is not difficult:

```
varElt := TKvarName              {varElt.id <- TKvarName.val}
        | TKvarName TKEQ aValue  {varElt.id <- TKvarName.val}
```

Then we have to recursively construct a list of id from the recursive grammar rules that defines the non terminal `varList`:

```
varList1 := varElt                { varList1.listid <- [varElt.id] }
         |  varElt TKCOL varList2
                { varList1.listid <- (varElt.id)::varList2.idlist }
```

We now have to merge and print all info at the "decl" level:

```
decl := type varList TKSCOL {
    if type.isInt print "int" else print "bool";
    print_list varList.listid;
    print (";")
}
```

Extensions to be able to print values are left to the reader.

□

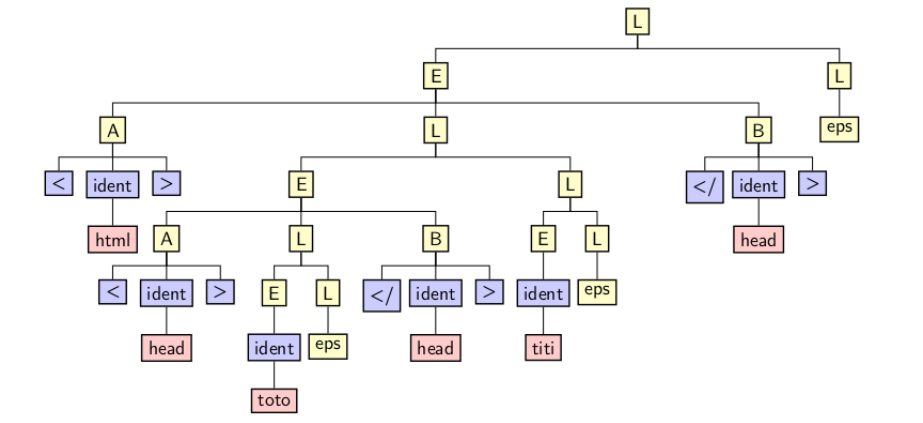
EXERCISE #2 ▶ **XML Files**

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.

Solution. deriv tree:



Attribution:

- ident a un attribut sémantique chaîne (ou id numérique) décrivant l'identifiant vu par l'analyseur lexical
- A et B ont un attribut du même type permettant de propager cet identifiant
- Dans la règle `E->ALB` on vérifie l'égalité des chaînes ou des id.

□