TDM4 - LAAS

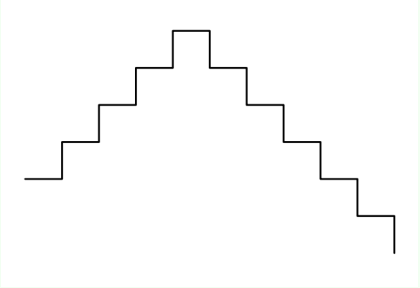
mars 2022

Le langage **« Jade »** permet de commander un traceur se déplaçant verticalement ou horizontalement sur un plan. Les instructions de base du langage sont :

- déplacements : `north`, `south`, `east`, `west` : le traceur se déplace d'un « pas » respectivement vers le haut, le bas, la gauche ou la droite.
- manipulation du stylo : `pendown`, `penup` baisse ou lève le stylo. Un déplacement effectué avec le stylo baissé effectue un tracé.
- la longueur du pas est un entier qui vaut initialement 10 . Elle peut être modifiée par l'instruction `setstep` *<entier positif>*

Le langage dispose d'une instruction conditionnelle `if`, et d'une boucle `while`. Une condition repose sur des comparaisons (`< <= > >= == !=`) entre valeurs entières pouvant être combinées par les opérateurs logiques. La position courante du traceur est désignée par les 2 valeurs entières `posx` et `posy` et la longueur du pas par `step`.
Des instructions peuvent être regroupées en un seul « bloc » en les encadrant par des accolades. Exemple :

```
pendown
setstep 20
while (posx<200){
 east
 if (posx<100)
    north
 else south
}
```



Voici la grammaire définissant ce langage

programme	→	*sequence*
sequence	→	*sequence instruction* | *instruction*
instruction	→	*simple* | *bloc* | *inst_if* | *inst_while*
simple	→	*inst0* | *inst1 entier*
bloc	→	`{` *sequence* `}`
inst_if	→	`if` *cond instruction else_part*
else_part	→	`else` *instruction* | ε
inst_while	→	`while` *cond instruction*
cond	→	`(` *expr_bool* `)`
expr_bool	→	*expr_bool* `||` *term* | *term*
term	→	*term* `&&` *factor* | *factor*
factor	→	`!` *F* | `(` *E* `)` | *const_bool* | *comparison*
comparison	→	*entier op_comp entier*
entier	→	*const_entier* | *variable*
		Tokens
inst0	→	`north` | `south` | `east` | `west` | `penup` | `pendown`
inst1	→	`setstep`
op_comp	→	`<` | `>` | `<=` | `>=` | `==` | `!=`
variable	→	`posx` | `posy` | `step`
const_entier	→	*integer*
const_bool	→	`true` | `false`

Exercice 1 :
Le but de l'exercice est de concevoir un traducteur de Jade vers python. Il lira donc en entrée un texte source Jade et produira un code source python.
L'exécution du code source résultat exécute les mouvements décrits par le rogramme jade au moyen de la tortue (module `turtle` de python)
Ce qui vous est fourni :

- l'analyseur lexical `JadeLexer`
- le parser `JadeParser` implémentant la grammaire ci-dessus, sans action sémantique (ou presque).

Par ailleurs, vous est également fournie une classe python `JadeMachine` capable d'exécuter les instructions simples Jade et d'informer sur la position de la tortue et la valeur du pas.
À titre d'exemple, voici ce que devrait produire votre tarducteur pour le source Jade ci-dessus :

```
import jademachine
jm = jademachine.JadeMachine()
s = jm.myturtle.getscreen()
jm.exec0("pendown")
jm.exec1("setstep", 20)
while jm.posx < 200 :
        jm.exec0("east")
        if jm.posx < 100 :
                jm.exec0("north")
        else :
                jm.exec0("south")

jm.myturtle.hideturtle()
s.exitonclick()
```

Il vous reste à compléter `JadeParser` pour ajouter les calculs d'attributs synthétisés nécessaires à la production du programme python, et bien entendu à tester sur le plus d'exemples possibles.
Indications :

- pour les variables *cond*, *expr_bool*, *term*, *factor*, *comparison*, *entier* :
  l'attribut associé doit être de type `chaîne`. L'attribut associé à `cond` doit représenter l'écriture en syntaxe python de la condition analysée.
- pour les variables *sequence*, *instruction*, *simple*, *bloc*, *inst_if*, *else_part*, *inst_while* :
  l'attribut associé doit être une `liste d'instructions python`, donc une liste de chaînes (chaque instruction python est rangée dans une chaîne). La fonction (fournie) `indent()` prend en argument une liste d'instructions et renvoie la même liste mais avec un niveau d'indentation supplémentaire (ce qui est nécessaire pour les parties internes des *if* ou des *while*)
- l'action sémantique de *programme* est fournie : elle structure en lignes la liste d'instructions associées à `sequence` et l'encadre entre l'initialisation et la conclusion du programme.

NB : les blocs Jade `{ ... }` ont pour but de grouper des instructions. Ils n'induisent **pas** une indentation supplémentaire en python. Par exemple `east { north west } south`

se traduit par

```
jm.exec0("east")
jm.exec0("north")
jm.exec0("west")
jm.exec0("south")
```

et NON par :

```
jm.exec0("east")
    jm.exec0("north")
    jm.exec0("west")
jm.exec0("south")
```