Réfléchi pendant 4m 2s >

Le programme comprend désormais une couche séparée de **syntaxe alternative d’entrée**

et de sortie, sans modifier la représentation interne des termes.

‹/› automates_langages_multisortes.py

Déclaration des syntaxes

Le nouvel argument facultatif `syntaxes` est un dictionnaire indexé par le **nom court** de l'opérateur :

‹/› Python Exécuter

```python
S = Idioma(
    "L=<a,b,h(.,.,.),p(.),m(.,.)>",
    syntaxes={
        "h": "prefixe",
        "p": "postfixe",
        "m": ("infixe", 100, "-"),
    },
)
```

La syntaxe s'applique à tous les opérateurs du système ayant ce même nom court.

Syntaxe préfixe

Elle est réservée aux opérateurs d'arité fixe strictement positive.

‹/› Python Exécuter

```python
"h": "prefixe"
```

Avec `h` ternaire :

```
h(a,b,a)  →  haba
h(a,b,p(a))  →  habap
```

Les arguments sont lus récursivement grâce à l'arité connue de `h`.

‹/› Python Exécuter

```python
S.reconnait("L", "habap")
# True

str(S.analyser("L", "h(a,b,p(a))"))
# 'habap'
```

Syntaxe postfixe

Elle est réservée aux opérateurs unaires.

```python
"p": "postfixe"
```

On obtient :

```
p(a)        → ap
p(p(a))     → app
p(m(a,b))   → (amb)p
```

Les opérateurs postfixes sont plus prioritaires que les opérateurs préfixes et infixes.

Syntaxe infixe

Elle est réservée aux opérateurs binaires :

```python
"m": ("infixe", 100, "-")
```

Le triplet indique :

```
("infixe", priorité, sens)
```

La priorité est un entier compris entre `0` et `255`. Une valeur plus élevée signifie une liaison plus forte.

Le sens vaut :

- `"-"` pour une association à gauche ;
- `"+"` pour une association à droite.

Ainsi :

```
m(a,b)  → amb
```

Les espaces sont facultatifs à l'entrée :

```python
S.reconnait("L", "amb")       # True
S.reconnait("L", "a m b")     # True
```

Associativité

Pour un opérateur associatif à gauche :

Python Exécuter

```python
S = Idioma(
    "L=<a,b,c,m(.,.)>",
    syntaxes={
        "m": ("infixe", 100, "-"),
    },
)
```

l'expression :

```
ambmc
```

est interprétée comme :

```
m(m(a,b),c)
```

Pour une association à droite :

Python Exécuter

```python
S = Idioma(
    "L=<a,b,c,i(.,.)>",
    syntaxes={
        "i": ("infixe", 100, "+"),
    },
)
```

l'expression :

```
aibic
```

est interprétée comme :

```
i(a,i(b,c))
```

L'autre structure exige des parenthèses :

```
(aib)ic
```

Gestion des priorités

Exemple :

```python
S = Idioma(
    "L=<a,b,c,plus(.,.),fois(.,.)>",
    syntaxes={
        "plus": ("infixe", 50, "-"),
        "fois": ("infixe", 100, "-"),
    },
)
```

L'expression :

```
aplusbfoisc
```

correspond à :

```
plus(a,fois(b,c))
```

car `fois` possède une priorité supérieure à `plus`.

Lors de l'affichage, le programme ajoute uniquement les parenthèses nécessaires pour préserver l'arbre interne.

Contraintes vérifiées

Le programme rejette automatiquement :

- une syntaxe préfixe appliquée à un générateur d'arité zéro ;
- une syntaxe préfixe appliquée à un opérateur multi-aire ;
- une syntaxe postfixe appliquée à un opérateur non unaire ;
- une syntaxe infixe appliquée à un opérateur non binaire ;
- une priorité hors de l'intervalle `0..255` ;
- deux opérateurs infixes de même priorité mais de sens différents ;
- des opérateurs homonymes ayant des arités incompatibles avec leur syntaxe commune.

Par exemple, ceci est interdit :

```python
```

```
Idioma(
    "L=<a,m(.,.)>",
    "H=<a,m(.)>",
    syntaxes={
        "m": ("infixe", 100, "-"),
    },
)
```

car la syntaxe alternative de `m` s'applique à tous les opérateurs portant le nom court `m`, alors que l'un est binaire et l'autre unaire.

Séparation du stockage et de l'affichage

L'écriture :

```
habap
```

est analysée et stockée en mémoire sous l'arbre canonique :

```
h(a,b,p(a))
```

La syntaxe condensée n'est utilisée que :

- lors de l'analyse d'une chaîne d'entrée ;
- lors de l'affichage ;
- lors de l'énumération ;
- lors de la génération aléatoire.

La syntaxe canonique reste toujours acceptée :

Python Exécuter

```python
S.reconnait("L", "h(a,b,p(a))")
# True

S.reconnait("L", "habap")
# True
```

L'énumérateur et le générateur aléatoire produisent directement la forme condensée lorsque des syntaxes alternatives sont déclarées. Tous les tests non interactifs réussissent.

automates_langages_multisortes.py
Code

On ajoute une syntaxe alternative :
La syntaxe délimite, pour les opérateurs d'arité variable.
qui comprend un symbole de début et un symbole de fin non déjà utilisé ni utilisé comme nom d'opérateur dans un même langage . Exemple :
Idioma(
  "L=<a,b,f(...)>",
  "H=<c,d,f(...)>",
  syntaxes={
    "f": ("delimite", "|", ">"),
  },
)
Dans L nous avons f(a,b,a,b) = |abab>
Dans H nous avons f(c,d,c,d) = |cdcd>

Afficher moins

Réfléchi pendant 2m 7s

Le programme prend maintenant en charge la **syntaxe délimitée** pour les opérateurs d'arité variable.

automates_langages_multisortes.py

Déclaration

Python Exécuter

```python
S = Idioma(
    "L=<a,b,f(...)>",
    "H=<c,d,f(...)>",
    syntaxes={
        "f": ("delimite", "|", ">"),
    },
)
```

La configuration :

Python Exécuter

```python
("delimite", "|", ">")
```

signifie que :

- `|` remplace le nom de l'opérateur et la parenthèse ouvrante ;
- `>` remplace la parenthèse fermante ;
- les arguments sont écrits successivement entre les deux délimiteurs.

Ainsi :

```
f(a,b,a,b)  =  |abab>
f(c,d,c,d)  =  |cdcd>
```

Reconnaissance

Python Exécuter

```python
S.reconnait("L", "|abab>")   # True
S.reconnait("H", "|cdcd>")   # True
```

Le typage des arguments reste contrôlé par le langage :

Python Exécuter

```python
S.reconnait("L", "|ac>")     # False
S.reconnait("H", "|ac>")     # False
```

Dans `L`, les arguments de `f` doivent être des termes de `L`. Dans `H`, ils doivent être des termes de `H`.

La syntaxe canonique reste acceptée :

Python Exécuter

```python
S.reconnait("L", "f(a,b,a,b)")   # True
S.reconnait("H", "f(c,d,c,d)")   # True
```

Affichage condensé

Même lorsque le terme est saisi sous forme canonique, il est affiché sous sa forme alternative :

Python Exécuter

```python
terme = S.analyser("L", "f(a,b,a,b)")

print(terme)
# |abab>
```

L'arbre conservé en mémoire reste équivalent à :

```
f(a,b,a,b)
```

Emboîtements

Les syntaxes délimitées peuvent être imbriquées :

```
f(f(a)) = ||a>>
```

Python Exécuter

```python
S.reconnait("L", "||a>>")
# True
```

De même :

```
f(a,f(b),a) = |a|b>a>
```

Les délimiteurs rendent explicite la fin de chaque liste d'arguments.

Arité minimale

La syntaxe délimitée fonctionne également avec les opérateurs multi-aires ayant une arité minimale :

Python Exécuter

```python
S = Idioma(
    "L=<a,b,f(.,...)>",
    syntaxes={
        "f": ("delimite", "[", "]"),
    },
)
```

Alors :

Python Exécuter

```
S.reconnait("L", "[]")       # False : au moins un argument
S.reconnait("L", "[a]")      # True
S.reconnait("L", "[aba]")    # True
```

Pour :

```python
"L=<a,b,f(.,.,...)>"
```

il faut au moins deux arguments :

```python
S.reconnait("L", "[a]")      # False
S.reconnait("L", "[ab]")     # True
```

Contraintes sur les délimiteurs

Les délimiteurs doivent :

- être des chaînes non vides ;
- être différents l'un de l'autre ;
- ne pas contenir d'espace ;
- ne pas être déjà utilisés par une autre syntaxe délimitée ;
- ne pas être le nom d'un opérateur du système.

Par exemple, cette déclaration est refusée si `a` est un opérateur :

```python
syntaxes={
    "f": ("delimite", "a", ">"),
}
```

La syntaxe délimitée s'applique, comme les autres syntaxes alternatives, à tous les opérateurs portant le même nom court `f`. Tous les tests non interactifs réussissent.

automates_langages_multisortes.py
Code

1- Une autre syntaxe délimitée peut se différentié par seulement le délimiteur de début (le délimiteur de fin étant identique dans les deux syntaxes délimitées).
2- juste une exception, le délimiteur de début peut correspondre au nom de l'opérateur. Exemple
syntaxes={
  "f": ("delimite", "f", ">"),
  "g": ("delimite", "g", ">"),
}
3- juste une exception, le délimiteur de fin peut être " ", le caractère blanc.

Réfléchi pendant 1m 5s >

Les trois règles ont été intégrées au programme :

automates_langages_multisortes.py

1. Délimiteur final commun

Deux syntaxes délimitées peuvent maintenant se distinguer uniquement par leur délimiteur initial :

Python Exécuter

```python
S = Idioma(
    "L=<a,f(...),g(...)>",
    syntaxes={
        "f": ("delimite", "|", ">"),
        "g": ("delimite", "[", ">"),
    },
)
```

Les expressions suivantes sont distinctes :

Python Exécuter

```python
S.reconnait("L", "|aa>")   # f(a,a)
S.reconnait("L", "[aa>")   # g(a,a)
```

Le délimiteur final `>` peut être partagé, car le choix de la syntaxe est déjà déterminé par le délimiteur initial.

En revanche, deux syntaxes délimitées ne peuvent pas avoir le même délimiteur initial.

2. Début égal au nom de l'opérateur

Le délimiteur initial peut être exactement le nom court de l'opérateur concerné :

Python Exécuter

```python
S = Idioma(
    "L=<a,f(...),g(...)>",
    syntaxes={
        "f": ("delimite", "f", ">"),
        "g": ("delimite", "g", ">"),
    },
)
```

On obtient :

```
f(a,a) = faa>
g(a,a) = gaa>
```

Et :

Python Exécuter

```python
S.reconnait("L", "faa>")   # True
S.reconnait("L", "gaa>")   # True
```

Cette exception ne vaut que pour le nom de l'opérateur lui-même. Par exemple, le délimiteur initial de `f` ne peut pas être `g` si `g` est un autre opérateur du système.

3. Espace comme délimiteur final

Le caractère espace peut maintenant être utilisé comme délimiteur final :

Python Exécuter

```python
S = Idioma(
    "L=<a,b,f(...)>",
    syntaxes={
        "f": ("delimite", "|", " "),
    },
)
```

L'espace final fait alors partie de la syntaxe :

```
f(a,b) = |ab␣
```

En Python :

```python
S.reconnait("L", "|ab ")   # True

terme = S.analyser("L", "f(a,b)")
print(repr(str(terme)))
# '|ab '
```

La représentation avec `repr()` permet de rendre visible l'espace final.

Les espaces ordinaires continuent d'être ignorés ailleurs, mais lorsqu'un espace est déclaré comme délimiteur final, celui qui ferme l'expression est conservé et interprété syntaxiquement.

Les nouveaux tests non interactifs vérifient les trois cas et réussissent.

automates_langages_multisortes.py
Code

ChatGPT peut faire des erreurs. Vérifiez les informations importantes.