Projet de compilateur

Partie 1 : Analyse lexicale

Cours Turing

Introduction

Pour cette première véritable étape de la conception de votre compilateur pour le langage `ALAN`, vous allez mettre au point ce qui est généralement la première phase de tout compilateur ou interpréteur, un analyseur lexical, ou simple simplement *lexer*.

Un lexer est un programme qui traduit un texte, c'est-à-dire une séquence de caractères, vers une séquence de *tokens*. Les tokens correspondent au mots du langage source, ce sont ces différents composants lexicaux de base comme les mots-clés, les identifiants, les symboles de ponctuation, etc.

Par exemple, dans le programme `ALAN` suivant :

```
{
    def fact(n: int): int = if (n == 0) 1 else n * fact(n + -1);

    print("Affichage de \"fact(15)\":\n");
    print(fact(15))
}
```

l'on retrouve la séquence de tokens suivante :

```
<OpenBrace at 1:1>
<DefKeyword at 2:5>
<Identifier(name='fact') at 2:9>
<OpenParenthesis at 2:13>
<Identifier(name='n') at 2:14>
<ColonSymbol at 2:15>
<Identifier(name='int') at 2:17>
<CloseParenthesis at 2:20>
<ColonSymbol at 2:21>
<Identifier(name='int') at 2:23>
<EqualSymbol at 2:27>
<IfKeyword at 2:29>
<OpenParenthesis at 2:32>
<Identifier(name='n') at 2:33>
```

```
<ComparisonOperator(op='==') at 2:35>
<IntLiteral(value=0) at 2:38>
<CloseParenthesis at 2:39>
<IntLiteral(value=1) at 2:41>
<ElseKeyword at 2:43>
<Identifier(name='n') at 2:48>
<MultiplicativeOperator(op='*') at 2:50>
<Identifier(name='fact') at 2:52>
<OpenParenthesis at 2:56>
<Identifier(name='n') at 2:57>
<AdditiveOperator(op='+') at 2:59>
<IntLiteral(value=-1) at 2:61>
<CloseParenthesis at 2:63>
<SemicolonSymbol at 2:64>
<Identifier(name='print') at 4:5>
<OpenParenthesis at 4:10>
<StrLiteral(value='Affichage de "fact(15)":\n') at 4:11>
<CloseParenthesis at 4:41>
<SemicolonSymbol at 4:42>
<Identifier(name='print') at 5:5>
<OpenParenthesis at 5:10>
<Identifier(name='fact') at 5:11>
<OpenParenthesis at 5:15>
<IntLiteral(value=15) at 5:16>
<CloseParenthesis at 5:18>
<CloseParenthesis at 5:19>
<CloseBrace at 6:1>
```

Chaque token généré correspond à une sous-séquence du texte d'entrée. De plus, chaque caractère du texte d'entrée contribue au plus à un unique token. Dans l'exemple ci-dessus, les espaces et saut de lignes ne donnent pas lieu à la création de tokens. Chaque token a une position dans le texte d'entrée et qui est mémorisée. De plus, certains tokens contiennent des informations supplémentaires, comme par exemple `IntLiteral` qui contient la valeur du nombre, ou `Identifier` qui contient le nom de l'identifiant.

La séquence de tokens ainsi générée sera ensuite analysée par la phase suivante, la phase d'analyse syntaxique, afin d'en extraire la structure et ainsi d'en créer un AST. Cette phase sera le sujet du prochain cours et l'étape suivante du projet. Mais avant cela, commençons par réaliser notre lexer !

1 Conception du lexer

1.1 Fichiers de base

Vous trouverez sur Moodle les fichiers de base de cette étape. Ces fichiers sont au nombre de deux :

- `tokens.py` contient la définition des différents types de tokens, chacun représenté par une classe.
- `lexing.py` contient la définition de la fonction principale du lexer, appelée `lex`. Le fichier est exécutable, ce qui permet d'expérimenter avec le lexer.

Commencez par télécharger ces deux fichiers et à les placer à la racine de votre projet de compilateur, à côté des autres fichiers Python du compilateur.

1.2 Aperçu de la fonction `lex`

La finalité de l'étape du jour est la conception d'une fonction appelée `lex` qui est définie dans le fichier `lexing.py`. La fonction vous est fournie complètement implémentée, mais a besoin pour fonctionner correctement que chaque type de token soit décrit par une expression régulière. Nous regarderons ceci dans plus de détails dans quelques instants, mais avant cela, analysons le fonctionnement de la fonction.

La fonction `lex` procède de manière itérative et maintient une variable qui contient la position à lire dans le texte. Au début de l'exécution de la fonction, la position à lire se situe tout au début du texte. À chaque itération de la boucle principale, la fonction passe en revue toutes les expressions régulières fournies pour trouver la *plus longue* correspondance possible en commençant exactement à la position courante dans le texte. En cas d'égalité de longueur, l'expression régulière essayée en premier aura la priorité. Si aucune correspondance ne peut être trouvée, alors une exception est lancée. Dans le cas contraire, un token est construit, puis émis par la fonction.

1.3 Écriture des expressions régulières

Votre tâche principale pour l'étape d'aujourd'hui sera de donner, pour chaque type de token du langage, une expression régulière qui décrit la forme des tokens. Le fonction `lex` vous est fournie déjà implémentée, mais elle a besoin pour fonctionner correctement que les expressions régulières de chaque sorte de token soient fournies. Pour cela, il vous faudra compléter la définition de la variable `TOKENS` en indiquant les expressions régulières manquantes.

Le nombre de groupes de capture dans l'expression régulière doit correspondre au nombre de paramètres du constructeur de la classe associée.

Vous trouverez ci-dessous des instructions plus précises pour chaque expression régulière manquante. Dans chaque cas, des exemples et des contre-exemples sont fournis. N'hésitez pas à tester votre implémentation en donnant ces exemples et contre-exemples à votre lexer. Pour rappel, il est possible d'exécuter directement votre lexer en lançant le script `lexing.py`.

1.3.1 `NoneLiteral`

Dans `ALAN`, on notera `none` l'expression correspondant au `None` de Python. Ainsi, l'expression régulière doit simplement décrire l'unique mot `none`. Il n'y a pas besoin de groupes de capture dans l'expression. En effet, le constructeur de `NoneLiteral` n'a pas de paramètre à spécifier.

Exemples

`none`

Contre-exemples

`None`

1.3.2 `BoolLiteral`

Les booléens dans `ALAN` se notent `true` et `false`.

Attention, comme le constructeur de la classe `BoolLiteral` a un unique paramètre, l'expression régulière devra avoir un unique groupe de capture qui englobe toute l'expression.

Exemples

`true`, `false`

Contre-exemples

`TRUE`, `False`, `0`, `Faux`

1.3.3 `IntLiteral`

Les entiers dans `ALAN` se notent en base 10. Les nombres commencent par un signe - (moins) optionnel (pour entrer des nombres négatifs). Suivent ensuite les chiffres (au moins un). Le premier chiffre ne peut pas être 0, sauf dans le cas où l'on représente le nombre 0 (ou -0).

Attention, comme le constructeur de la classe `IntLiteral` a un unique paramètre, l'expression régulière devra avoir un unique groupe de capture qui englobe toute l'expression.

Exemples

`1`, `-1`, `0`, `-0`, `267`, `-1456`, `-41441422475388210`

Contre-exemples

`01`, `1A`, `--2`, `00`, `3.14`

1.3.4 `StrLiteral`

Les chaînes de caractères dans `ALAN` se notent à l'aide d'un nombre arbitraire de caractères entre `"` (doubles apostrophes). Les caractères qui forment une chaîne sont notés d'une des deux façons suivantes :

- À l'aide d'un caractère Unicode quelconque, à l'exception de `"` (doubles apostrophes) et de `\` (barre oblique inversée, *backslash*).
- À l'aide d'un caractère échappé. Les caractères échappés s'écrivent avec deux caractères Unicode : le premier est toujours `\` (barre oblique inversée), le deuxième est un caractère quelconque. Le sens des caractères échappés est le même qu'en Python. Ainsi, `\n` représente un saut de ligne, `\t` une tabulation, `\"` les doubles apostrophes et `\\` une barre oblique inversée.

Attention, comme le constructeur de la classe `StrLiteral` a un unique paramètre, l'expression régulière devra avoir un unique groupe de capture. Le groupe doit capturer les caractères de la chaîne **sans les doubles apostrophes qui la délimitent**.

Exemples

```
"Hello world!"
"Bonjour !\nAu revoir !"
"Il m'a dit \"Hello !\" l'autre jour."
```

Contre-exemples

```
Salut
"Bonjour \"
"Il dit "Au revoir !" à ses amis."
```

1.3.5 `Identifier`

Les identifiants en `ALAN` sont formés à l'aide d'une séquence de caractères alphanumériques (lettres majuscules, minuscules ou chiffres) avec éventuellement des `_` (tiret bas, *underscore*). De plus, le premier caractère ne peut pas être un chiffre. Par souci de simplicité, on se contentera des 26 lettres majuscules et des 26 lettres minuscules de notre alphabet, sans accents, sans espaces ni caractères spéciaux.

Notez que, comme l'expression régulière apparaît après toutes les expressions régulières pour les mots-clés, vous n'avez pas besoin d'exclure ces mots-clés de votre expression régulière pour les identifiants. En effet, comme les expressions régulières des mots-clés sont testées avant celle pour les identifiants, les mots-clés ont la priorité. Ainsi, `if` sera lu comme un mot-clé et non comme un identifiant, bien que la séquence de caractères soit aussi reconnue comme un identifiant.

Attention, comme le constructeur de la classe `Identifier` a un unique paramètre, l'expression régulière devra avoir un unique groupe de capture qui englobe toute l'expression.

Exemples

```
n
foo
bar_baz
f22a
___
__SUPER__
```

Contre-exemples

```
2a
éléphant
mon message
```

2 Extensions au lexer

Dans le cas où vous auriez terminé la conception de votre lexer, voici quelques idées d'améliorations :

— La possibilité de rentrer des entiers en base 2.
— La possibilité de rentrer des entiers en base 16.
— La possibilité d'utiliser des caractères spéciaux dans les identifiants, comme par exemples des caractères accentués ou encore des caractères Unicode tels que des émojis.

Sentez-vous libres d'apporter des améliorations au lexer, tout en gardant à l'esprit que tout ce que vous implémenterez comme modifications aura potentiellement un impact sur la suite du projet !