Université de Paris
M1 Informatique

Programmation Fonctionnelle Avancée
Année 2019-2020

Projet

Programmer par des exemples[1]

Concat est un petit langage de programmation qui permet d'accomplir des tâches simples de transformations de chaînes de caractères. L'objectif de ce projet est de faire de la *programmation par l'exemple* : Au lieu de faire écrire un programme Concat par un programmeur, un utilisateur va donner quelques exemples d'une entrée et de la sortie attendue du programme. Puis, le programme que vous allez écrire va produire un programme Concat qui est cohérent avec les exemples fournis (ou une erreur quand un tel programme n'existe pas).

Puisque votre tâche sera d'engendrer des programmes Concat à partir d'exemples, il est inutile de réfléchir à une syntaxe concrète. Le programme que vous écrirez va construire le programme Concat directement sous la forme d'une syntaxe abstraite, et puis l'afficher sous une forme que vous pouvez choisir. En particulier il est inutile de créer un parseur pour ce langage.

1 Notations

Si s est une chaîne de caractères de longueur n, et $0 \leq i < j \leq n$, alors nous écrivons $s[i..j[$ pour la sous-chaîne de s qui va de l'index i *inclus* à l'index j *exclus*. Par exemple, $"hello"[2..4[= "ll"$. On écrit $s[..j[$ pour $s[0..j[$, et $s[i..[$ pour $s[i..n[$. On remarque que $s[i..j[\cdot s[j..k[= s[i..k[$.

Quand on peut découper une chaîne s en $s = s_1 \cdot s_2 \cdot s_3$, où chacun des s_1, s_2, s_3 peut être vide, on appelle s_1 un préfixe, s_2 un facteur, et s_3 un suffixe de s.

2 Étape 1

Un programme Concat décrit une fonction qui envoie une chaîne de caractères vers une chaîne de caractères. La syntaxe abstraite est donnée à la figure 1, et la sémantique est résumée à la figure 2.

$$
\begin{aligned}
\texttt{program} &\rightarrow \texttt{expression} \ \cdots \ \texttt{expression} \\
\texttt{expression} &\rightarrow \underline{\text{const}}(\textit{string}) \mid \underline{\text{extract}}(\texttt{pos\_expression}, \texttt{pos\_expression}) \\
\texttt{pos\_expression} &\rightarrow \underline{\text{forward}}(\textit{nat}) \mid \underline{\text{backward}}(\textit{nat})
\end{aligned}
$$

Figure 1: Syntaxe abstraite de Concat - Étape 1

[1] Version 1 – 26/02/2020

$$
\begin{aligned}
[\![e_1 \cdots e_n]\!](s) &= [\![e_1]\!](s) \cdot \ldots \cdot [\![e_n]\!](s) \\
[\![\underline{\text{const}}(sc)]\!](s) &= sc \\
[\![\underline{\text{extract}}(p_1, p_2)]\!](s) &= s[\ [\![p_1]\!](s)..[\![p_2]\!](s)\ [\\
[\![\underline{\text{forward}}(i)]\!](s) &= i \\
[\![\underline{\text{backward}}(i)]\!](s) &= \mathit{length(s)} - i
\end{aligned}
$$

Figure 2: Sémantique de Concat - Étape 1

- Un programme est une séquence d'expressions, la sémantique d'un programme $e_1 \ldots e_n$ appliqué à une entrée s est la concaténation des sémantiques de chaque expression e_i appliquée à s.
- Une expression est soit $\underline{\text{const}}$ appliqué à une chaîne sc, soit $\underline{\text{extract}}$ de deux pos_expressions p_1 et p_2. Dans le premier cas la sémantique est la chaîne sc. Dans le deuxième cas, quand la sémantique de p_1 appliquée à s est i_1 et la sémantique de p_2 appliquée à s est i_2, alors la sémantique de l'expression entière appliquée à s est $s[i_1..i_2[$.
- Une pos_expression est une expression qui dénote une position dans une chaîne de caractères. À l'étape 1 c'est soit $\underline{\text{forward}}(i)$, qui dénote l'index i, ou bien $\underline{\text{backward}}(i)$ qui dénote l'index $n - i$ lors d'une application à une chaîne de longueur n.

Par exemple :

- le programme $\underline{\text{extract}}(\underline{\text{forward}}(0), \underline{\text{backward}}(0))$ est la transformation identique qui transforme une chaîne s en s.
- le programme suivant:

$$\underline{\text{const}}(\text{"}Hello,\text{"});\ \underline{\text{extract}}(\underline{\text{forward}}(3), \underline{\text{backward}}(7))$$

  transforme la chaîne "Mr Smith junior" en "Hello, Smith".

Le programme que vous devrez écrire aura la fonctionnalité suivante : il lit d'abord dans un fichier une séquence de paires associant à une chaîne d'entrée une chaîne de résultat. Chaque paire constitue une ligne de votre fichier, et entrée et résultat seront séparés par le symbole tabulation. Votre programme doit ensuite calculer un programme Concat p tel que pour toute paire exemple (in, out), on ait $[\![p]\!](in) = out$. Par exemple, étant donné les exemples suivants :

Entrée	Résultat
10/10/2017	10
05-15-2015	15
20.02.2020	02

votre programme peut trouver le programme Concat suivant :

$$\underline{\text{extract}}(\underline{\text{forward}}(3), \underline{\text{forward}}(5))$$

Sur cet exemple, on voit déjà qu'il peut y avoir plusieurs programmes Concat possibles pour un même problème, par exemple en utilisant ici des $\underline{\text{backward}}$ à la place des $\underline{\text{forward}}$.

L'idée du programme est la suivante : pour chacune des k paires (in_i, out_i) on construit l'ensemble $P_i = \{p_i \mid [\![p_i]\!](in_i) = out_i\}$, puis on calcule l'intersection $\bigcap_{i=1}^{k} P_i$, et finalement on choisit un programme dans cette intersection. Si l'intersection est vide on peut signaler une erreur. Dans l'exemple précédent, l'ensemble P_1 correspondant à la première paire va contenir, parmi d'autres, les programmes suivants :

- const("10")
- extract(forward(3), forward(5))
- extract(forward(3), forward(4)); const("0")

et beaucoup d'autres encore. On se rend rapidement compte du fait qu'il ne va pas être possible d'énumérer *explicitement* tous les éléments d'un tel ensemble P_i, car si la chaîne résultat a une longueur n alors il y a déjà 2^n possibilités de la découper en morceaux, et pour chaque morceau du découpage il y aura plusieurs possibilités de l'obtenir soit comme constante, soit par l'extraction de l'entrée à des positions différentes.

Pour cette raison, au lieu d'énumérer explicitement tous les éléments d'un ensemble P_i, on va plutôt en construire une représentation compacte. Cette représentation doit permettre les deux opérations suivantes :

- calculer l'intersection de deux ensembles, nécessaire pour combiner les ensembles obtenus pour des paires différentes ;
- extraire un programme concret, si possible le programme le plus simple qui se trouve dans un ensemble.

2.1 La représentation d'un ensemble de programmes

Nous allons utiliser ici une représentation par des DAG (*directed acyclic graphs* - graphes orientés non cycliques) afin de représenter tous les ensembles de programmes P_i vus ci-dessous, et de représenter leurs intersections. Notez que cette représentation est spécifique à notre problème: elle ne permettra pas d'encoder un ensemble arbitraire de programmes. On décrit d'abord le DAG qui représente l'ensemble P qui correspond à une chaîne d'entrée in de longueur n et un résultat out de longueur m.

L'ensemble de nœuds du graphe est $\{0, \ldots, m\}$, et il y a une arête de i vers j quand $i < j$. Le nœud 0 est la source, et m est le puits de ce graphe (voir la figure 3).

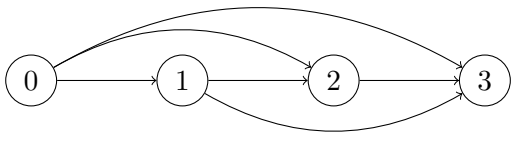

Figure 3: Le graphe des positions pour $m = 3$, ici montré sans les étiquettes

Une arête de i vers j va être étiquetée par une représentation de l'ensemble des expressions e (au sens de la figure 1) tel que $[\![e]\!](in) = out[i..j[$. Par conséquent, tout programme p avec $[\![p]\!](in) = out$ correspond à un chemin dans le graphe de la source vers le puits, et les expressions qui constituent le programme p sont des expressions qu'on peut trouver sur les arêtes de ce graphe.

L'arête de i vers j est décorée avec un ensemble. Dans cet ensemble on trouvera $\underline{\text{const}}(out[i..j[)$, les autres éléments sont des $\underline{\text{extract}}$ appliqués à deux *ensembles* de position. Si la chaîne $out[i..j[$ apparaît plusieurs fois dans la chaîne entrée alors vous construisez un $\underline{\text{extract}}$ pour chaque occurrence de $out[i..j[$ dans in. Et ce $\underline{\text{extract}}$ aura comme arguments les ensembles de pos_expressions p donnant respectivement les index du début et de la fin de cette occurrence de $out[i..j[$ dans in. Notez que pour l'instant ces ensembles de positions sont de taille 2 (un $\underline{\text{forward}}$ et un $\underline{\text{backward}}$) mais cela changera par la suite. En résumé, la décoration de l'arête de i vers j est

$$\{\underline{\text{const}}(out[i..j[)\} \cup \bigcup_{in[k..l[=out[i..j[} \underline{\text{extract}}(\{p \mid [\![p]\!](in) = k\}, \{p \mid [\![p]\!](in) = l\})$$

Note : dans la syntaxe abstraite de Concat, $\underline{\text{extract}}$ prend deux pos_expressions en arguments, mais dans les étiquettes d'un DAG, $\underline{\text{extract}}$ prend deux *ensembles* de pos_expressions en arguments.

2.1.1 Exemple

Pour rendre les exemples plus lisibles nous écrivons ici $\underline{\text{f}}$ et $\underline{\text{b}}$ au lieu de $\underline{\text{forward}}$ et $\underline{\text{backward}}$.

1. $in =$ "$abad$", $out =$ "dxa". Le graphe a les 4 nœuds $0, 1, 2, 3$ (figure 3), avec les arêtes suivantes :

de	vers	étiquette
0	1	$\underline{\text{const}}(\text{"}d\text{"}), \underline{\text{extract}}(\{\underline{\text{f}}(3), \underline{\text{b}}(1)\}, \{\underline{\text{f}}(4), \underline{\text{b}}(0)\})$
0	2	$\underline{\text{const}}(\text{"}dx\text{"})$
0	3	$\underline{\text{const}}(\text{"}dxa\text{"})$
1	2	$\underline{\text{const}}(\text{"}x\text{"})$
1	3	$\underline{\text{const}}(\text{"}xa\text{"})$
2	3	$\underline{\text{const}}(\text{"}a\text{"}), \underline{\text{extract}}(\{\underline{\text{f}}(0), \underline{\text{b}}(4)\}, \{\underline{\text{f}}(1), \underline{\text{b}}(3)\}), \underline{\text{extract}}(\{\underline{\text{f}}(2), \underline{\text{b}}(2)\}, \{\underline{\text{f}}(3), \underline{\text{b}}(1)\})$

2. $in =$ "$efegh$", $out =$ "$ghxe$" Le graphe a les 5 nœuds $0, 1, 2, 3, 4$, avec les arêtes suivantes :

de	vers	étiquette
0	1	$\underline{\text{const}}(\text{"}g\text{"}), \underline{\text{extract}}(\{\underline{\text{f}}(3), \underline{\text{b}}(2)\}, \{\underline{\text{f}}(4), \underline{\text{b}}(1)\})$
0	2	$\underline{\text{const}}(\text{"}gh\text{"}), \underline{\text{extract}}(\{\underline{\text{f}}(3), \underline{\text{b}}(2)\}, \{\underline{\text{f}}(5), \underline{\text{b}}(0)\})$
0	3	$\underline{\text{const}}(\text{"}ghx\text{"})$
0	4	$\underline{\text{const}}(\text{"}ghxe\text{"})$
1	2	$\underline{\text{const}}(\text{"}h\text{"}), \underline{\text{extract}}(\{\underline{\text{f}}(4), \underline{\text{b}}(1)\}, \{\underline{\text{f}}(5), \underline{\text{b}}(0)\})$
1	3	$\underline{\text{const}}(\text{"}hx\text{"})$
1	4	$\underline{\text{const}}(\text{"}hxe\text{"})$
2	3	$\underline{\text{const}}(\text{"}x\text{"})$
2	4	$\underline{\text{const}}(\text{"}xe\text{"})$
3	4	$\underline{\text{const}}(\text{"}e\text{"}), \underline{\text{extract}}(\{\underline{\text{f}}(0), \underline{\text{b}}(5)\}, \{\underline{\text{f}}(1), \underline{\text{b}}(4)\}), \underline{\text{extract}}(\{\underline{\text{f}}(2), \underline{\text{b}}(3)\}, \{\underline{\text{f}}(3), \underline{\text{b}}(2)\})$

2.2 Intersection de deux ensembles

Considérons un DAG D_1 avec un ensemble de nœuds N_1, source s_1 et puits t_1, et un DAG D_2 avec ensemble de nœuds N_2, source s_2 et puits t_2. Leur intersection est définie ainsi :

- L'ensemble de nœuds est $N_1 \times N_2 = \{(n_1, n_2) \mid n_1 \in N_1, n_2 \in N_2\}$.
- La source est (s_1, s_2) et le puits est (t_1, t_2).

- Il y a une arête de (i_1, i_2) vers (j_1, j_2) quand D_1 a une arête de i_1 vers j_1 et D_2 a une arête de i_2 vers j_2.
- Soit E_1 l'ensemble de décorations de l'arête de i_1 vers j_1 dans le graphe D_1, et E_2 l'ensemble de décorations de l'arête de i_2 vers j_2 dans le graphe D_2. L'arête de (i_1, i_2) vers (j_1, j_2) est alors décorée par

$$\begin{array}{ll} & \{\underline{\text{const}}(w) \mid \underline{\text{const}}(w) \in E_1 \cap E_2\} \\ \cup & \{\underline{\text{extract}}(p_1 \cap p_2, q_1 \cap q_2) \mid \underline{\text{extract}}(p_1, q_1) \in E_1, \underline{\text{extract}}(p_2, q_2) \in E_2\} \end{array}$$

Vous pouvez évidement "nettoyer" le graphe obtenu, en supprimant :

- dans les décorations d'arêtes, les $\underline{\text{extract}}$ dont un argument au moins est l'ensemble vide
- les arêtes du graphe qui sont décorées par un ensemble vide
- les nœuds du graphe qui ne sont pas accessibles à partir de la source
- les nœuds du graphe à partir desquels on ne peut pas atteindre le puits.

Pour obtenir de nouveau un graphe dont les nœuds sont des entiers positifs ou nuls vous pouvez par exemple représenter une paire (i_1, i_2) par $i_1 * (t_2 + 1) + i_2$ où t_2 est le puits du DAG D_2, c'est-à-dire en pratique son nombre de noeuds moins 1. Une meilleure solution consiste à renuméroter les nœuds après suppression de tous les nœuds inutiles.

2.2.1 Exemple (suite de 2.1.1)

Le graphe représentant l'intersection des deux graphes de la section 2.1.1 contient en principe $4 * 5 = 20$ nœuds, de $(0, 0)$ à $(3, 4)$. Pourtant il y a certains de ces nœuds qui ne sont pas accessibles à partir de $(0, 0)$: tout les nœuds $(0, j)$ avec $j > 0$ ou $(i, 0)$ avec $i > 0$. Puis certains de ces nœuds ne permettent pas d'atteindre $(3, 4)$: tout les nœuds $(3, j)$ avec $j < 4$ ou $(i, 4)$ avec $i < 3$. En plus, certains nœuds peuvent s'avérer inutiles quand on trouve des arêtes étiquetés avec un ensemble vide. Pour cette raison nous construisons le graphe avec les étiquettes sur les arêtes à partir de la source $(0, 0)$.

1. On commence avec les nœuds directement accessibles à partir de la source. Parmi eux, les nœuds suivants ont déjà été éliminés (ils ne permettent pas d'atteindre le puits) : $(1, 4)$, $(2, 4)$, $(3, 1)$, $(3, 2)$, $(3, 3)$. On calcule les étiquettes sur les arêtes de $(0, 0)$ vers les nœuds qui restent :

de	vers	étiquette
$(0,0)$	$(1,1)$	$\underline{\text{extract}}(\{\underline{\text{f}}(3)\}, \{\underline{\text{f}}(4)\})$
$(0,0)$	$(1,2)$	$\underline{\text{extract}}(\{\underline{\text{f}}(3)\}, \{\underline{\text{b}}(0)\})$
$(0,0)$	$(1,3)$	*vide*
$(0,0)$	$(2,1)$	*vide*
$(0,0)$	$(2,2)$	*vide*
$(0,0)$	$(2,3)$	*vide*
$(0,0)$	$(3,4)$	*vide*

2. Nous continuons avec les nœuds utiles trouvés à l'étape précédente :

de	vers	étiquette
$(1,1)$	$(2,2)$	*vide*
$(1,1)$	$(2,3)$	*vide*
$(1,1)$	$(3,4)$	*vide*
$(1,2)$	$(2,3)$	$\underline{\text{const}}(\text{"}x\text{"})$
$(1,2)$	$(3,4)$	*vide*

3. Il nous reste un seul nœud utile pour continuer :

de	vers	étiquette
$(2,3)$	$(3,4)$	$\underline{\text{extract}}(\{\underline{\text{f}}(0)\},\{\underline{\text{f}}(1)\}),\underline{\text{extract}}(\{\underline{\text{f}}(2)\},\{\underline{\text{f}}(3)\})$

2.3 Extraire un programme d'un ensemble

Extraire un programme d'un DAG est maintenant trivial. Pour chaque arête, décorée avec un ensemble de const et de extract, vous gardez seulement l'expression la plus simple selon un critère qui vous semble pertinent. Puis, pour chaque arête vous calculez un poids basé sur l'unique décoration, de sorte que ce poids soit plus important pour des décorations plus complexes. Encore une fois, vous pouvez choisir librement votre fonction de poids. Ce choix deviendra par contre plus important à l'étape 2. Enfin, vous extrayez de ce graphe un plus court chemin de la source au puits selon l'algorithme bien connu de Dijkstra. Le programme Concat choisi est alors la séquence des expressions sur les arêtes de ce chemin.

2.3.1 Exemple (suite de 2.2.1)

Dans notre exemple, il y a un seul chemin de la source au puits : $(0,0) \to (1,2) \to (2,3) \to (3,4)$. Un programme Concat possible est donc

$$\underline{\text{extract}}(\underline{\text{forward}}(3),\underline{\text{backward}}(0));\ \underline{\text{const}}(\text{"}x\text{"});\ \underline{\text{extract}}(\underline{\text{forward}}(0),\underline{\text{forward}}(1))$$

Il y a une autre possibilité :

$$\underline{\text{extract}}(\underline{\text{forward}}(3),\underline{\text{backward}}(0));\ \underline{\text{const}}(\text{"}x\text{"});\ \underline{\text{extract}}(\underline{\text{forward}}(2),\underline{\text{forward}}(3))$$

3 Étape 2

La seule chose qui change à l'étape 2 est la définition des pos_expressions: on doit maintenant pouvoir spécifier également des positions dans l'entrée par une forme restreinte d'expressions régulières.

`pos_expression`	→	forward(*nat*) | backward(*nat*) | after(`regexp`) | before(`regexp`)
`regexp`	→	`ini token...token fin`
`ini`	→	ϵ | Start
`fin`	→	ϵ | End
`token`	→	plus(`class`) | plusComp(`class`)
`class`	→	alphanumeric | numeric | alpha | lower | upper | special(*char*)

Figure 4: Extension de la syntaxe abstraite à l'étape 2

3.1 Les classes de caractères

Chaque `class` décrit une classe de caractères : les alpha-numériques, les chiffres, les lettres de l'alphabet, les lettres minuscules et les lettres majuscules. Une classe de la forme special(c), où le caractère c est un caractère spécial, est une classe constituée de cet unique caractère c. Vous pouvez choisir quel ensemble de caractères spéciaux vous traitez, par exemple {/,-,(,)}, mais attention à l'explosion combinatoire si cet ensemble est trop grand.

3.2 Les tokens

Un token plus(cl) filtre un mot m quand m est non-vide, et tous les caractères de m sont dans cl ; un token plusComp(cl) filtre un mot m quand m est non-vide, et *aucun* caractère de m n'est dans cl.

3.3 Les expressions régulières

Une expression régulière est une séquence de tokens, éventuellement précédée par Start ou terminée par End. L'expression régulière vide (i.e. constituée de 0 token) ne sera pas acceptée dans nos programmes, même si la définition suivante la considère pour des raisons de simplicité. Le filtrage d'un mot par une expression régulière est défini par l'algorithme glouton suivant :

- On ignore pour l'instant les éventuels Start et End initiaux et finaux
- Une expression régulière vide filtre seulement le mot vide
- Une expression régulière $t\,r$, où t est un token, filtre un mot m si on peut décomposer m en $m = m_1 \cdot m_2$ tels que :
  - t filtre m_1
  - t ne filtre aucun préfixe de m_2
  - r filtre m_2

3.4 Extension des pos_expressions

Étant donnée une expression régulière r et une chaîne de caractères s, soit i un index minimal de s tel que r filtre un préfixe de $s[i..[$, et pour ce i soit j maximal tel que r filtre $s[i..j[$. En d'autres mots, on cherche le premier facteur de s qui est filtré par r, et ce facteur est prolongé

autant que possible. Quand r commence sur Start il faut en plus que $i = 0$, et quand r se termine par End il faut en plus que $j = length(s)$.

Une pos_expression before(re), quand appliquée à une chaîne s, dénote la position i, et la pos_expression after(re), quand appliquée à une chaîne s, dénote la position j.

Par conséquent, extract(before(re), after(re)) produit donc le facteur du mot d'entrée qui est filtré par l'expression régulière re.

3.5 Conseils

Dans votre programme vous auriez besoin de connaître, pour tous i et j, l'ensemble des expressions régulières qui filtrent $s[i..j[$. Vous calculerez cette information de façon ascendante (bottom-up) en vous inspirant de la programmation dynamique.

Vous pouvez gagner en efficacité en déterminant d'avance un ensemble pertinent de classes de caractères en fonction du problème à traiter. Par exemple, une classe qui contient tous les caractères d'entrée, ou qui n'en contient aucun, ne nous sert à rien. De plus, deux classes de caractères cl_1 et cl_2 seront redondantes pour notre problème lorsque pour tout caractère c de l'entrée, $c \in cl_1$ ssi $c \in cl_2$. Il suffit alors de considérer une seule des deux classes. Comme on dispose de tokens négatifs, la situation $c \in cl_1$ ssi $c \notin cl_2$ pour tout caractère c de l'entrée mènent également à une redondance de classes.

4 Pour aller encore plus loin

L'idée de ce projet est tiré de l'article [Gul11]. Vous y trouverez des idées d'extensions possibles. Une première extension possible est de considérer des pos_expressions de la forme after(`regexp`, n) ou before(`regexp`, n), où n est un nombre naturel, qui dénotent le n-ième facteur du mot filtré par l'expression régulière. Vous trouverez dans cet article en plus des autres extensions comme par exemple des programmes avec des boucles, ou des distinctions de cas.

5 Organisation

Voir le fichier `projet/README.md` dans le dépôt git du cours.

References

[Gul11] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 317–330, Austin, TX, USA, January 2011. ACM.