Projet IN406 - Partie 2

FQ & YS

21 avril 2020

1 Analyse descendante

Le problème de l'analyse est celui de prendre une grammaire G, et de construire un algorithme capable, pour toute chaîne w de terminaux, de décider si elle appartient ou non à G. De plus, on demande de reconstruire un arbre de dérivation pour w dans G en cas de réponse affirmative.

Certains langages permettent de résoudre ce problème a l'aide de l'analyse dite descendante.

Cette méthode cherche a construire une dérivation gauche directement a partir du symbole initial, ou de façon équivalente, cherche a construire un arbre de dérivation a partir de la racine avec une exploration en profondeur d'abord de gauche a droite.

Pour cela, l'analyseur doit pouvoir décider, en s'aidant éventuellement avec les prochains k symboles en entrée, quelle est la prochaine production dans la dérivation gauche que l'on reconstruit.

2 La grammaire des regexp

Soit la grammaire $G = (\Sigma, V, S, P)$ avec $\Sigma = \{\texttt{+}, \texttt{.}, \texttt{*}, \texttt{(}, \texttt{)}\} \cup$ l'alphabet noté `char`, avec $V = \{S, A, C, E\}$ et avec les règles de production :

$$P = \left\{ \begin{array}{lcl} S \longrightarrow A\texttt{+}S & | & A \\ A \longrightarrow C\texttt{.}A & | & C \\ C \longrightarrow E\texttt{*} & | & E \\ E \longrightarrow \texttt{char} & | & \texttt{(}S\texttt{)} \end{array} \right\}$$

2.1 Modification de la grammaire

Pour faire l'analyse descendante, il faut transformer la grammaire en récursive droite et avec en premier caractère un terminal. par exemple les deux règles : $S \longrightarrow A\texttt{+}S$ et $S \longrightarrow A$ se transforment en trois règles, en ajoutant le non terminal B :

$$\left\{ \begin{array}{l} S \longrightarrow A\texttt{+}S \\ S \longrightarrow A \end{array} \right. \Longrightarrow \left\{ \begin{array}{l} S \longrightarrow AB \\ B \longrightarrow \texttt{+}AB \\ B \longrightarrow \varepsilon \end{array} \right.$$

De même, en rajoutant le non terminal D :

$$\left\{ \begin{array}{l} A \longrightarrow C\texttt{.}A \\ A \longrightarrow C \end{array} \right. \Longrightarrow \left\{ \begin{array}{l} A \longrightarrow CD \\ D \longrightarrow \texttt{.}CD \\ D \longrightarrow \varepsilon \end{array} \right.$$

De même, en rajoutant le non terminal F :

$$\left\{ \begin{array}{l} C \longrightarrow E\texttt{*} \\ C \longrightarrow E \end{array} \right. \Longrightarrow \left\{ \begin{array}{l} C \longrightarrow EF \\ F \longrightarrow \texttt{*}F \\ F \longrightarrow \varepsilon \end{array} \right.$$

Ce qui donne comme ensemble de règles de production :

$$P = \left\{ \begin{array}{ll} S \longrightarrow AB & \mathbf{1} \\ B \longrightarrow \texttt{+}AB & \mathbf{2} \\ B \longrightarrow \varepsilon & \mathbf{3} \\ A \longrightarrow CD & \mathbf{4} \\ D \longrightarrow \texttt{.}CD & \mathbf{5} \\ D \longrightarrow \varepsilon & \mathbf{6} \\ C \longrightarrow EF & \mathbf{7} \\ F \longrightarrow \texttt{*}F & \mathbf{8} \\ F \longrightarrow \varepsilon & \mathbf{9} \\ E \longrightarrow \texttt{char} & \mathbf{10} \\ E \longrightarrow \texttt{(}S\texttt{)} & \mathbf{11} \end{array} \right\}$$

2.2 Table de transitions

On peut alors construire la table des transitions. On suppose que toutes les expressions régulières se terminent par le caractère # qui indique donc la fin de mot.

	+	.	*	(	)	char	#
S				$S \longrightarrow AB$		$S \longrightarrow AB$	
A				$A \longrightarrow CD$		$A \longrightarrow CD$	
B	$B \longrightarrow \texttt{+}AB$				$B \longrightarrow \varepsilon$		$B \longrightarrow \texttt{\#}$
C				$C \longrightarrow EF$		$C \longrightarrow EF$	
D	$F \longrightarrow \varepsilon$	$D \longrightarrow \texttt{.}CD$			$D \longrightarrow \varepsilon$		$D \longrightarrow \varepsilon$
E	$F \longrightarrow \varepsilon$	$F \longrightarrow \varepsilon$	$F \longrightarrow \varepsilon$	$E \longrightarrow \texttt{(}S\texttt{)}$		$E \longrightarrow \texttt{char}$	
F	$F \longrightarrow \varepsilon$	$F \longrightarrow \varepsilon$	$F \longrightarrow \texttt{*}F$		$F \longrightarrow \varepsilon$		$F \longrightarrow \varepsilon$

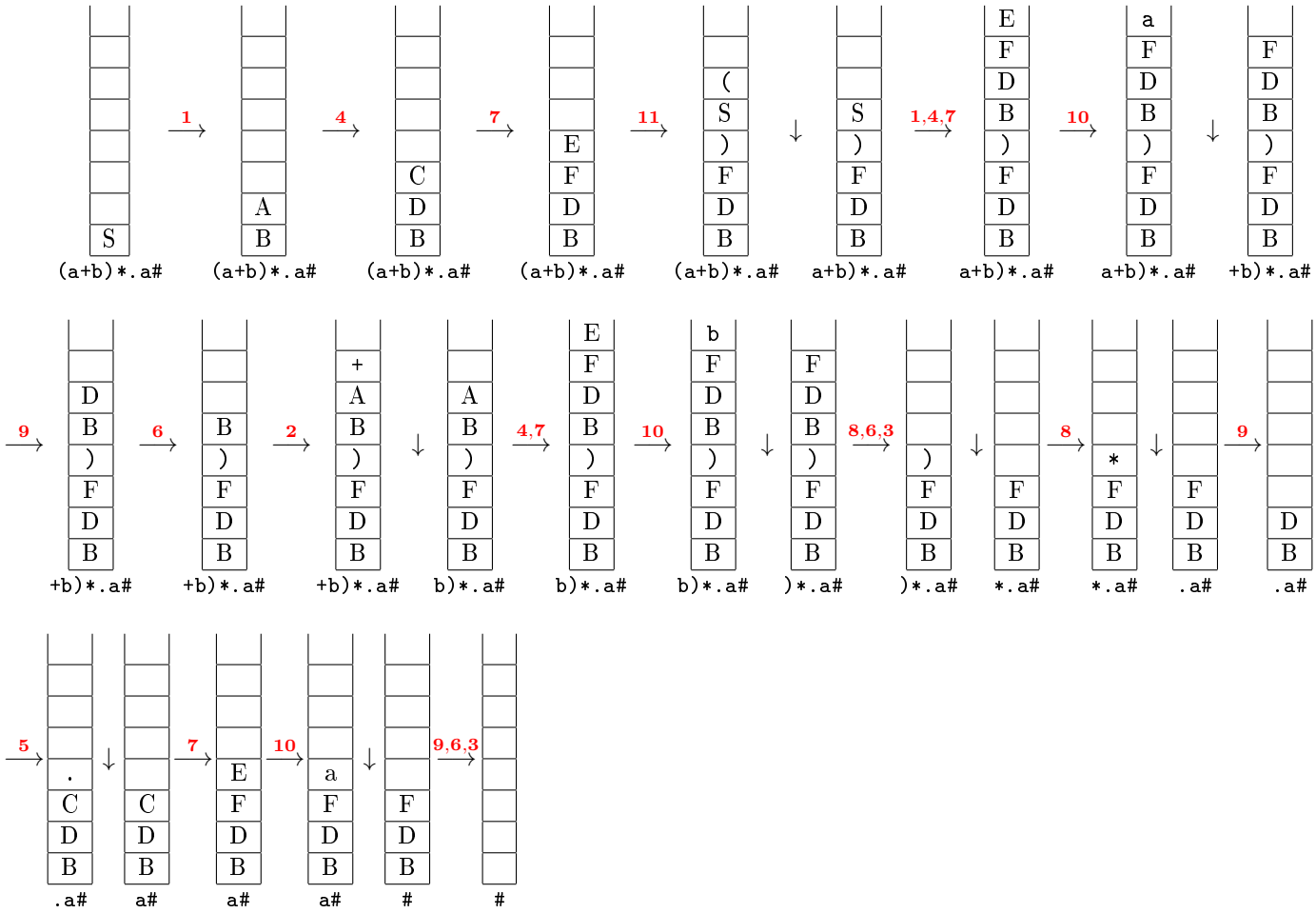
Pour reconnaître un mot, on utilise un automate à pile initialisé avec S en fond de pile

2.3 Algo de reconnaissance d'un mot

Afin de reconnaitre un mot w on va utiliser un automate à pile utilisant la table de transitions ci-dessus :
- La pile est initialisée avec le symbole S dans la pile et le caractère courant du mot qui est le premier caractère
- Si le symbole en haut de la pile est un **non terminal**, on regarde si dans la table il y a une règle pour le symbole en haut de la pile et le caractère courant :
  - Si NON : mot non reconnu
  - Si OUI : on dépile le symbole et on empile les symboles correspondant à la règle trouvée dans la table de transition (le caractère courant n'est pas modifié).
- Si le symbole en haut de la pile est un **terminal**, on regarde si il est égal au caractère courant :
  - Si NON : mot non reconnu
  - Si OUI : on dépile le symbole et on ne rempile rien et le caractère courant devient le caractère suivant du mot à reconnaître.
- Si on a lu tout le mot et que la pile n'est pas vide : mot non reconnu.
- Si on a lu tout le mot et que la pile est vide : mot reconnu.

2.4 Exemple

Exemple de construction de l'arbre de dérivation pour `(a+b)*.a#`. Dans l'exemple, les caractères du mot déjà lus ne sont plus écrits :



3 Questions

1. Pour l'exemple ci-dessus, donner l'arbre de dérivation.
2. Expliquer comment construire l'arbre de dérivation à partir de l'algorithme de reconnaissance.
3. Expliquer comment construire l'arbre de l'expression régulière à partir de l'arbre de dérivation.
4. Expliquer comment construire l'arbre de l'expression régulière directement à partir de l'algorithme de reconnaissance.

4 Code

Vous avez à votre disposition des structures de données et des fonctions, pour vous aider à coder.

4.1 Pile

Dans pile.h, vous trouverez la définition des états non terminaux et une pile d'états avec les opérations usuelles de pile. Les états sont définis par un type enum, voir par exemple https://zestedesavoir.com/tutoriels/755/le-langage-c-1/notions-avancees/les-enumerations/.

```
typedef enum{ //les différents type de symboles dans la grammaire
S,A,B,C,D,E,F,CAR,PARO,PARF,PLUS,POINT,ETOILE
}STATE;
```

```
void affiche_state(STATE s);//affiche le nom d'un état

typedef struct{
STATE *contenu;//contenu de la pile
int sommet;//position du sommet de la pile
int taille_max;//taille maximum de la pile
}PILE;


int est_vide(PILE p);
PILE nouvelle_pile(int taille_max);
PILE empiler(PILE p, STATE s);
STATE depiler(PILE *p);
void affiche_pile(PILE p);
void liberer_pile(PILE p);
```

4.2 Arbre de dérivation

Dans regexp.h, vous trouverez la définition du type d'un arbre de dérivation : ADERIV, et des fonctions de manipulation d'arbre de dérivation.

```
struct aderiv{
    STATE s;// symbole de la grammaire contenu dans le sommet
    char caractere;//si s est CAR, contient la valeur du caractère, 0 sinon
    struct aderiv *fils[3];//un noeud a au plus trois enfants (cas du +, du * et de ())
    //si le noeud a un fils c'est fils[0], deux fils fils[0] et fils[1] ...
};

typedef struct aderiv *ADERIV;//arbre de dérivation

ADERIV nouvel_arbre(STATE s, char c);//mettre la valeur du caractère pour c si l'état est CAR, 0 sinon
void affiche_aderiv(ADERIV a, int space);//affiche un arbre de dérivation
void liberer_arbre(ADERIV a);//libère la mémoire dun arbre de dérivation
```

Pour stocker la table de transition de l'automate, on utilise le type STATELISTE qui contient au plus 3 états correpondant à la production d'une règle de la grammaire et donc à ce qu'on doit empiler quand on est dans l'état correspondant.

```
typedef struct{ //liste d'états pour la table de transition
int taille;//au plus trois symboles dans la liste
STATE liste[3];
}STATELISTE;

STATELISTE table[7][7] = {//cette table représente la table des transitions de l'énoncé
{{-1},{-1},{-1},{2,{A,B}},{-1},{2,{A,B}},{-1}}, // transition quand le STATE S est lu
{{-1},{-1},{-1},{2,{C,D}},{-1},{2,{C,D}},{-1}},//STATE A
{{3,{PLUS,A,B}},{-1},{-1},{-1},{0},{-1},{1,{CAR}}},//STATE B
{{-1},{-1},{-1},{2,{E,F}},{-1},{2,{E,F}},{-1}},//STATE C
{{0},{3,{POINT,C,D}},{-1},{-1},{0},{-1},{0}},//STATE D
{{0},{0},{0},{3,{PARO,S,PARF}},{-1},{1,{CAR}},{-1}},//STATE E
{{0},{0},{2,{ETOILE,F}},{-1},{0},{-1},{0}}//STATE F
};
```

1. En utilisant le code fourni, implémenter la fonction

`ADERIV construire_arbre_derivation(char *expr)`

qui contruit l'arbre de l'expression régulière à partir d'une expression régulière. À chaque fois que vous utilisez une règle (une transition de l'automate) vous devez ajouter des sommets dans votre arbre qui correspondent à ce que vous avez empilé. Vous vérifierez que l'arbre obtenu pour les différents tests est correct et que les erreurs sont bien détectées.

2. (Optionnel) L'automate fourni ne vérifie pas si les parenthésages sont cohérents. Il faudrait en plus gérer une autre pile pour vérifier que les parenthèses ouvrantes sont cohérentes (Mot de Dyck). Implémenter cette vérification supplémentaire.
3. (Optionnel) Proposez une amélioration à la correction de la partie 1 ou au code fourni pour la partie 2 (simplification, clarification, correction de bogue ...).

5 Modalités pratiques

Merci de respecter les consignes suivantes :
— le projet est à faire en seul ou en binôme ;
— le projet est à rendre dans MOODLE, la deuxième partie doit être rendue au plus tard le **mercredi 6 mai 2020, 23h59** ;
— vous devez déposer un fichier `XY_NOM1_Prenom1-NOM2_Prenom2.zip` qui est le zip du dossier `XY_NOM1_Prenom1-NOM2_Prenom2` contenant les fichiers donnés avec l'énoncé complétés par vos soins.
`XY` sont les initiales de votre chargé de TD (`XB`, `LD`, `FQ`, `GS`, `YS`). Si les deux étudiants viennent d'un groupe différent, choisissez les initiales du chargé de TD de NOM1. Pour être évalué, votre travail doit compiler et s'exécuter quand on tape `make` dans le terminal. Si le nom de fichier remis sur MOODLE ne respecte pas ces règles, le projet ne sera pas évalué.
Le retard de la remise du projet entraîne 1 point de moins par heure de retard.