NOM : PECATTE **Prénom :** Timothée **Jury :**

Algèbre ← Entourez l'épreuve → **Analyse**

Sujet choisi : 923 - Analyses lexicales et syntaxique : applications

Autre sujet :

I. Introduction à la compilation :

But: Pouvoir écrire des programmes dans un langage de haut niveau.

Ex 1: TANT QUE, SI.. ALORS, appels de fonctions,...

Déf 2: (Interpréteur) Un interpréteur peut être vu comme une fonction qui prend en entrée un programme écrit dans un langage L et des données, et qui renvoie les données ou une erreur.

$$I_L : L \times D^* \longrightarrow D^* \cup \{\text{erreur}\}$$

Rmq 3: Il faut donc concevoir un interpréteur pour chaque langage

Déf 4: (Compilateur) Un compilateur d'un langage L vers un langage L' agit comme un "traducteur" qui est fidèle vis-à-vis des interpréteurs, ie : $\forall d \in D^*$, $I_L(p,d) = I_{L'}(C_{L\to L'}(p), d)$

Rmq 5: Ainsi, si on possède un interpréteur pour un langage de référence, il suffit de concevoir un compilateur vers ce langage

Déf 6: On compilateur s'organise en deux parties principales:
- "Le front-end, qui vérifie que l'entrée est bien formée et qui reconstitue le programme
- Le back-end, qui génère le code vers le langage cible, en effectuant éventuellement des optimisations

Question: Comment passer d'un mot qui encode un programme à une représentation sémantique de celui-ci ? Comment détecter les erreurs ? Peut-on les corriger ?

II. L'Analyse lexicale d'un programme:

II.1. But de l'analyse lexicale:

Déf 7: (Symboles) Dans un langage de programmation, les symboles, ou unités lexicales, sont les différents types d'objet qu'une suite de caractère peut représenter.

Ex 8: Identifieurs, parenthèses, mots-clés (SI, TANTQUE), chiffres, commentaires, $=, <, >, :=, \dots$

But 9: Etant donné une chaîne de caractère en entrée, le rôle de l'analyse lexicale est de découper l'entrée en symboles, et de garder ceux qui sont significatifs (les séparateurs sont ignorés).

Rmq 9: Du point de vue d'un compilateur, un programme et son analogue écrit sur une seule ligne sont exactement les mêmes (sauf pour Python). Cela permet donc des libertés quant à l'indentation des programmes.

II.2. Bases théoriques :

Déf 10: (Expressions régulières) Les expressions régulières sont le plus petit ensemble contenant $\emptyset, \varepsilon, a$ pour $a \in \Sigma$, et stables par $+, \cdot, *$ et $(\cdot)$. On associe par induction un langage à une expression régulière.

Ex 11: $L((a|b)^*@(a|b)^*.com)$ est l'ensemble des adresses mail se terminant par .com (a et b comme lettres (à part le "." de ".com" et le @))

Prop 12: L est reconnu par un automate fini déterministe si L est reconnu par une expression régulière.

Rmq 13: Les expressions régulières sont plus lisibles, mais les automates sont plus pratiques pour tester l'appartenance.

Rmq 14 : L'automate obtenu via la preuve est non-déterministe mais on peut le rendre déterministe

Prop 15 : Etant donné une expression régulière, on peut construire de manière effective un automate déterministe minimal qui reconnaît le même langage.

Rmq 16 : Les automates considérés jusqu'à présent permettent seulement de tester l'appartenance, alors que le rôle de l'analyse lexicale est de produire une suite de symboles, il faut donc modifier le comportement de l'automate

Déf 17 : Un automate généralisé est un 5-uplet (Q, q_0, F, δ, O) avec
- Q est l'ensemble des états
- q_0 est l'état initial
- F est l'ensemble des états final
- δ est la fonction de transition
- O est la fonction de production

À l'état q, sur l'entrée a, si $\delta(q,a)$ existe, l'automate passe dans l'état $\delta(q,a)$. Sinon, si $q \in F$, on produit $O(q)$ et on passe dans l'état $\delta(q_0,a)$. Sinon, on arrête la lecture pour renvoyer une erreur.

Ex 18 : Annexe

Rmq 19 : L'automate cherche ainsi à reconnaître le plus grand préfixe possible comme un symbole, ce qui autorise la déclaration de variables de la forme `SIte`.

II.3. Souplesse pratique :

Rmq 20 : En pratique, il est laborieux d'écrire 0|1|2|3|4|5|6|7|8|9 à chaque fois qu'on veut parler d'un caractère numérique.

Déf 21 : (Classe de caractère). Une classe de caractère est un identifiant qui décrit un ensemble de caractère fini.

Ex 22 : ca = a-zA-Z, ch = 0-9, ow = |, oper = (, ...

Déf 23 : (Séquence d'expr régulière). Une séquence d'expressions régulière sur Σ est une suite de définition
$$\begin{aligned} A_1 &= R_1 \\ A_2 &= R_2 \\ &\vdots \\ A_n &= R_n \end{aligned}$$
où les A_i sont des identificateurs distincts et les R_i sont expr régulières sur $\Sigma \cup \{A_1, \dots, A_{i-1}\}$

Déf 24 : Opération "sauf" $R_1 \text{ sauf } (R_2) \sim R_1 \overline{\Sigma^* R_2 \Sigma^*} R_2$

Rmq 25 : Les opérations, et d'autres, sont autorisées dans le générateur automatique Flex d'automate généralisé pour analyse lexicale

DEV 1 Construction de l'analyseur lexicale pour l'exemple suivant

ch = 0-9, hex = A-F,
intconst = ch ch*
hexconst = h (ch|(hex)(ch|(hex)*
realconst = intconst. intconst (e ch ch|ε)

DEV 1

DEV 16 Construction directe d'un automate déterministe à partir d'une expr régulière.

DEV 16

III. L'analyse syntaxique d'un programme :

III.1. But de l'analyse syntaxique :

Rmq 26 : Après le passage de l'analyse lexicale, le programme est encore illisible et donc peu pratique à manipuler

But : Structurer le programme et détecter les éventuelles erreurs de syntaxe

Rmq 27 : C'est aussi le rôle de l'analyseur syntaxique de vérifier que certaines règles imposées par le langage sont vérifiées, comme par exemple la déclaration des variables avant leur usage, mais cela ne sera pas abordé dans la suite.

III.2. Grammaires hors-contexte et premiers automates à pile :

Déf 28 : Une grammaire non contextuelle est un quadruplet (V_N, V_T, P, S) où V_N est l'ensemble des symboles non terminaux, V_T est l'ensemble des terminaux, $P \subseteq V_N \times (V_N \cup V_T)^*$ est l'ensemble des productions, et $S \in V_N$ est le symbole de départ

Ex 29: $G_0 = (\{E,T,F\}, \{+,*,(,),id\}, \{E \to E+T \mid T,\ T \to T*F \mid F,\ F \to (E) \mid id\}, E)$

Déf 30: (Dérivations). $\varphi \xrightarrow{G} \psi$ s'il existe des mots σ, τ, α et un non-terminal A tel que $\varphi = \sigma A \tau$ et $\psi = \sigma\alpha\tau$ et $A \to \alpha \in P$.
On définit $\xrightarrow{G}{}^*$ comme la clôture transitive réflexive de $\xrightarrow{G}$

Déf 31: (Arbre syntaxique). Un arbre syntaxique est une représentation d'une dérivation qui ne tient pas compte de l'ordre dans lequel les non-terminaux sont remplacés. Il est défini par induction sur la dérivation.

Ex 32: Annexe

Déf 33: (Ambiguïté). Une phrase $x \in L(G)$ est ambigüe si elle admet plusieurs arbre syntaxiques. Une grammaire est ambigüe lorsqu'elle contient au moins une phrase ambigüe.

Rmq 34: L'ambiguïté d'une grammaire n'est pas forcement gênante en soi, mais induit du non-déterminisme que nous devons résoudre en faisant un choix.

Déf 35: (Dérivation gauche). Une dérivation est dite dérivation gauche si à chaque étape on remplace le non-terminal le plus à gauche.

Prop 36: Un langage est algébrique si il est reconnu par un automate à pile.

Rmq 37: Contrairement aux automates simples, on ne peut pas déterminiser un automate à pile dans le cas général, et donc on aura pas de manière efficace pour des grammaires algébriques quelconque.

III.3. Grammaires LL(k) et analyse syntaxique descendante:

Rmq 38: Si on autorise l'automate à lire les k prochains symboles au lieu du 1er, on ne change pas la complexité de la reconnaissance, mais cela peut rendre l'automate déterministe

Déf 39: (PREMIER). On définit PREMIER(X) comme étant l'ensemble des caractères qui commencent une chaîne dérivée à partir de X.

Ex 40: PREMIER(T) = {(, id}

Déf 41: (SUIVANT). On définit SUIVANT(X), pour X non-terminal, comme l'ensemble des caractères qui peuvent suivre immédiatement X dans une proto-phrase.

Ex 42: SUIVANT(T) = {+,)}

Rmq 43: On peut définir de la même manière les fonctions PREMIER$_k$ et SUIVANT$_k$

Prop 44: PREMIER et SUIVANT peut être calculé de manière efficace — DEV2

Déf 45: (Grammaire LL(1)). Une grammaire LL(1) est une grammaire telle que pour toute règle de production $A \to \alpha \mid \beta$, on a : * PREMIER(α) $\cap$ PREMIER(β) $= \emptyset$
* si $\varepsilon \in$ PREMIER(β), alors PREMIER(α) $\cap$ SUIVANT(β) $= \emptyset$

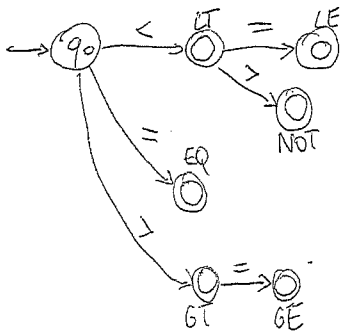
Rmq 46: On peut définir de même les grammaires LL(k) avec PREMIER$_k$ et SUIVANT$_k$

Prop 47: Si G est une grammaire LL(1), alors on peut construire un automate déterministe prédictif qui construit l'arbre syntaxique du mot donné en entrée ou qui renvoie erreur.

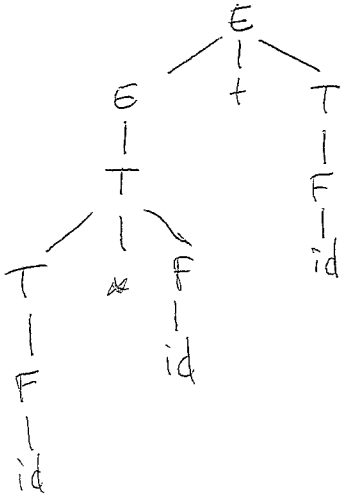
Rmq 48: Cet automate déterministe peut avoir des ε-transitions qui ne sont pas gênantes car l'automate décide de la transition en fonction du prochain caractère.

Déf 49: (Table prédictive). Une table prédictive, pour une grammaire LL(1), est un tableau dans lequel pour chaque non-terminal E, et chaque symbole ω d'entrée possible, on a l'unique règle de production $E \to \alpha$ tq $\alpha \to^* \omega u$ si $\omega \in$ PREMIER(E) et "erreur" si $\omega \notin$ PREMIER(E)

DEV2 ~~Construction de l'automate prédictif d'une grammaire LL(1) sur un exemple.~~



Exemple 18 : Automate généralisé



Exemple 3.2 : Arbre syntaxique de id*id+id (dans G_0).

$S \to E$ $\quad E' \to +E$ $\quad T' \to *T$

$E \to TE'$ $\quad T \to FT'$ $\quad F \to (E)$

$E' \to \varepsilon$ $\quad T' \to \varepsilon$ $\quad F \to id$

Exemple de grammaire LL(1)