Semantique

Q1. Expliquez le principe général de la syntaxe dans les langages de programmation et comment la définir.

La syntaxe est la partie visible du langage
Les règles de syntaxe définissent la forme des phrases admises dans le langage

syntaxe = unités syntaxiques élémentaires(lexèmes : identificateur , séparateur, opérateur , mot-clé) + structure syntaxique

Structure syntaxique : spécification des séquences admissibles d' unités syntaxiques élémentaires
-----> notion commune de grammaire
---> on peut inclure des notation de mise en page (assembleur et langages évolués primitifs)

Analyse syntaxique et lexicale

1. Expressions régulières
2. Automates

But d'analyse lexicale: reconnaître les lexèmes d'une phrase. Il est plus facile de décrire et de lire la description des lexèmes sous la forme d'expressions régulières (approche déclarative) que sous la forme d'un algorithme (approche opérationnelle).

Exemple :

blancs	$[\backslash t \backslash n]^+$
lettre	$[A-Za-z]$
chiffre	$[0-9]$
ident	$\{lettre\}(\_\|\{lettre\}\|\{chiffre\})^*$
entier	$\{chiffre\}^+$

Dans la pratique, l'analyse lexicale est compliquée par divers facteurs, tels que : taille maximale des constantes numériques, comportant des instructions pour le compilateur ,caractères spéciaux dans les chaines de caractères.

L'outil Lex sert à produire un analyseur lexical (il génère un programme source en langage C) `a partir d'expressions régulières ;

Expression régulière :
Les expressions régulière sont des outils très puissants et très utilisés : on peut les retrouver dans de nombreux langages comme le PHP, MySQL, Javascript... ou encore dans des logiciels d'édition de code !
Les expressions régulières peuvent être concaténées pour former de nouvelles expressions régulières; si A et B sont tous deux des expressions régulières, alors AB est également une expression régulière. En général, si une chaîne p correspond à A et une autre chaîne q correspond à B, la chaîne pq correspondra à AB.
Les expressions régulières peuvent contenir des caractères spéciaux et ordinaires. La plupart des caractères ordinaires, comme "A", "a" ou "0", sont les expressions régulières les plus simples.

Voici une syntaxe d'expressions régulières assez commune (syntaxe de l'outil Lex) :

Semantique

$[xz]$	le caractère x ou le caractère z
$[x-z]$	un caractère dans l'intervalle lexicographique x à z
$\backslash n \backslash t$	respectivement la fin de ligne et la tabulation
.	(le point) tout caractère sauf la fin de ligne (n)
x	le caractère x, même si c'est un symbole prédéfini
^ e	e (expression ou caractère) est en début de ligne
e$	e est à la fin d'une ligne
e ?	e est optionnel
e*	e apparait 0 fois ou plus
e+	e apparait 1 fois ou plus
e1|e2	e1 ou e2

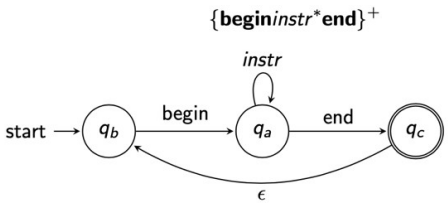
Les automates finis :

Un automate fini est une construction mathématique abstraite, susceptible d'être dans un nombre fini d'*états* mais étant un moment donné dans un seul état à la fois ; l'état dans lequel il se trouve alors est appelé l'« état courant ». Le passage d'un état à un autre est activé par un événement ou une condition ; ce passage est appelé une « transition ».
Un automate particulier est défini par l'ensemble de ses états et l'ensemble de ses transitions.
On rencontre couramment des automates finis dans de nombreux appareils qui réalisent des actions déterminées en fonction des événements qui se présentent. Les automates finis peuvent modéliser un grand nombre de problèmes, par exemple la conception de **l'analyse syntaxique** de langages.

Les automates finis (ou automates à états finis) permettent de formaliser les séquences d' états et d'opérations requis pour implémenter des expressions régulières (il y a donc équivalence).

exemple automate fini :

Exemple :



Q2. Expliquez ce qu'est la syntaxe concrète et ce qu'est la syntaxe abstraite d'un langage de programmation.

La syntaxe est un domaine bien compris et formalisé, mais qui ne s'intéresse qu'aux propriétés structurelles et grammaticales du langage. La syntaxe abstraite identifie les composantes significatives des constructions du langage ; elle est liée aux symboles non terminaux de la grammaire.

En informatique, la syntaxe abstraite d'une structure de données représente les types qui définissent ces données sans forcément leur assigner une représentation ou un codage précis. Elle sert en particulier à la représentation du code source de langage de programmation, et est généralement stockée dans un arbre syntaxique abstrait.

Chaque langage de programmation a une syntaxe concrète. De plus, chaque implémentation d'un langage de programmation utilise une syntaxe abstraite. En d'autres termes, la syntaxe concrète fait partie de la définition du langage. La syntaxe abstraite fait partie de la définition d'une implémentation particulière (évaluateur ou compilateur) d'un langage. Si le langage nous permet d'accéder aux structures d'implémentation de l'intérieur de lui-même, alors bien sûr, la syntaxe abstraite fait partie de la définition de ce langage. Par exemple dans Pico, cela est possible car une fonction f peut être traitée comme un tableau de taille 4 afin que nous puissions accéder au code du corps par l'expression f [3].

Semantique

Le résultat de ceci est un arbre qui peut encore être disséqué en utilisant un simple référencement de tableau. Par conséquent, dans Pico, l'apparence des arbres de syntaxe abstraite fait partie de la définition de Pico. Ainsi, dans Pico, la syntaxe abstraite fait partie de la définition de Pico, et pas seulement un détail d'implémentation. "Normalement" cependant, l'accès à l'implémentation de la langage à partir de cette langue n'est pas possible. Ensuite, la syntaxe abstraite est strictement une question d'implémentation.

La syntaxe concrète d'un langage de programmation est définie par une grammaire sans contexte. Il se compose d'un ensemble de règles (productions) qui définissent la façon dont les programmes ressemblent au programmeur. Alors que la syntaxe abstraite spécifie un ensemble d'arbres de syntaxe abstraite légale, la syntaxe concrète sert de base à un analyseur qui traduit le texte en un arbre de syntaxe abstraite. Les spécifications de syntaxe concrètes incluent les jetons et les mots clés qui permettent à un analyseur de construire un arbre de syntaxe abstrait sans ambiguïté. Dans Metal, la partie syntaxique concrète d'une spécification est distincte de la partie syntaxique abstraite. La syntaxe abstraite est plus `` abstraite " en ce qu'elle ne traite que des types, pas des jetons, qui peuvent être modifiés sans changer la partie essentielle de la syntaxe. En effet, une syntaxe abstraite donnée pourrait avoir un nombre quelconque de spécifications de syntaxe concrètes associées, bien que le langage Metal ne le permette pas actuellement.
• La syntaxe abstraite d'une implémentation est l'ensemble d'arbres utilisés pour représenter les programmes dans l'implémentation. C'est-à-dire que la syntaxe abstraite définit la façon dont les programmes ressemblent à l'évaluateur / compilateur.

Q3. Expliquez ce qu'est l'**ambiguïté de la syntaxe** dans un langage de programmation.

Les compilateurs et les interprètes utilisent des grammaires pour construire les structures de données qu'ils utiliseront pour traiter les programmes. Par conséquent, idéalement, un programme donné devrait être décrit par un seul arbre de dérivation. Cependant, selon la conception de la grammaire, des ambiguïtés sont possibles. Une grammaire est ambiguë si une phrase du langage généré par la grammaire a deux arbres de dérivation distincts

exemple pour a-b-c il y'a deux arbres de dérivations .

Q4) Expliquez le principe d'induction. Définissez des ensembles par induction.
L'induction est une méthode de recherche scientifique: elle consiste à chercher des lois générales à partir de l'observation de faits particuliers.
Preuves par récurrence : propriété P(x), pour tout x $\in_N$

Technique de Preuve :
Prouver P(x) est vrai pour x = 0 c'est à dire P(0) (cas de base)
Supposant que pour tout k $\in\mathbb{N}$, P(k)est vrai

on prouve que P(k + 1) est vrai (cas inductif)
Alors P(n) est vrai pour tout n $\in\mathbb{N}$

Remarque: La validité de la preuve par induction est liée à l'existence d'un ordre bien fondé sur les entiers.

Exemple : $0+1+2+\ldots+x-1+x=¿\ \frac{x+1}{2}$

Définition inductives d'ensembles

Théorème : soit S cet ensemble S = $\{Xt.q0 \in X \wedge \forall x, x \in X \Longrightarrow x+1 \in X\}$
Démonstration : on utilise $\forall X \in S : P_X(n) = (n \in X)$

- $P_X(0)$
- hyp : $P_X(n)$ est vrai . il faut prouver $P_X(n+1)$

Exemple :

- $0 \in EP$
- X $\in EP \Longrightarrow x+2 \in EP$
-

Q5) En vous aidant de la slide 31 du chapitre 4, expliquez les opérations somme , max , long.

opérations avec les propriétés usuelles :

soit **max(l)** plus grand élément de l
soit **somme(l)** somme des éléments de l
soit **long(l)** longueur de l
on définit un théorème $\forall I \in Liste, somme(I) \leq max(I) * long(I)$
on va le prouver par induction :¨
preuve :$P(x) = somme(x) \leq max(x) * long(x)$
cas de base : $P([\,])$ induction : $P(n::l)$ *sachant que* $p(l)$ *est vrai*

$$\frac{\square}{somme([\,]) = 0} \qquad \frac{n \in N, l \in liste\ somme(l) = m}{somme(n::l) = m+n}$$

$$\frac{\square}{long([\,]) = 0} \qquad \frac{n \in N, l \in liste\ long(l) = k}{long(n::l) = k+1}$$

$$\frac{\square}{max([\,]) = 0} \qquad \frac{n \in N, l \in liste\ somme(l) = m}{max(n::l) = n}$$

$$\frac{n \geq / max(l), max\ (l) = m}{max(n::l) = m}$$

Max(l) somme(l) long(l) ¿ ≤

$$\frac{\square}{0 \in N} \quad \frac{n \in N}{n+1 \in N}$$

$$\frac{n \in N}{0*n = 0} \quad \frac{\square}{0+n = n}$$

$$\frac{m*n = k}{s(m)*n = k+n} \quad \frac{m+n = k}{s(n)+m = s(k)}$$

Semantique

$$\leq \frac{n \in N}{0 \leq n} \quad \frac{x \leq y}{s(x) \leq s(y)}$$

¿ $\leq$ *sont définis sur* $0, s$ *et* $+$¿

$$\frac{x \in N}{x+0=x} \quad \frac{x, y, k \in N, x+y=k}{x+s(y)=s(k)}$$

$$\frac{x \in N}{x*0=0} \quad \frac{x, y, k, l \in N, x*y=l, l+n=k}{x*s(y)=k}$$

$$\frac{x, y \in N, x \leq y}{s(x) \leq s(y)} \quad \frac{x, y \in N, x=y}{y=x} \quad \frac{x \in N}{x=x}, \quad \frac{x, y, z \in N, x=y, y=z}{x=z}$$

Définitions des opérateurs :

$$\frac{\square}{long([\,])=0} \quad \frac{n \in N, l \in liste, long(l)=k}{long(n::l)=k+1}$$

$$\frac{\square}{somme([\,])=0} \quad \frac{n \in N, l \in liste, somme(l)=k}{somme(n::l)=k+n}$$

$$\frac{\square}{max([\,])=0} \quad \frac{n \in N, l \in liste, max(l)=k, k \leq n}{max(n::l)=n}$$

$$\frac{n \in N, l \in liste, max(l)=k, k \not< n}{max(n::l)=k}$$

Des principes généraux des fonctions

$$\frac{f : D*D*\ldots*D \to D, t_1=t_1', t_2=t_2', \ldots, t_n=t_n'}{f(t_1, t_2, \ldots t_n)=f(t_1', t_2', \ldots, t_n')}$$

En utilisant la transitivité :

$$\frac{f : D*D*\ldots*D \to D, t_1=t_1', t_2=t_2', \ldots, t_n=t_n', f(t_1, t_2, \ldots, t_n)=e}{f(t_1', t_2', \ldots, t_n')=e}$$

P est un prédicat

$$\frac{f : D*D*\ldots*D \to D, t_1=t_1', t_2=t_2', \ldots, t_n=t_n', P(t_1, t_2, \ldots, t_n)=e}{P(t_1', t_2', \ldots, t_n')}$$

Démonstration :

$P([\,])=somme([\,]) \leq max([\,])*long([\,])$

$$\frac{\dfrac{\dfrac{}{0 \leq 0}, \dfrac{\overline{0 \in \mathbb{N}}}{0*0=0}}{0 \leq 0*0}}{somme([\,]) \leq max([\,])*long([\,])}$$

$$P([\,])$$

$induction : P(l) \vdash P(n::l)$
Cas 1 :

$$\dfrac{\dfrac{\dfrac{\dfrac{**1}{somme(l) \leq max(l)*long(l), n \leq max(l) \vdash somme(l)+n \leq max(l)*long(l)+max(l)}}{somme(l) \leq max(l)*long(l), n \leq max(l) \vdash somme(l)+n \leq max(l)*(long(l)+1)}}{somme(l) \leq max(l)*long(l) \vdash somme(l)+n \leq max(n::l)*(long(l)+1)}}{somme(l) \leq max(l)*long(l) \vdash somme(n::l) \leq max(n::l)*long(n::l)}$$
$$P(l) \vdash P(n::l)$$

**1 est vrai à cause de lemme : $\dfrac{a \leq b \vdash c \leq d}{a \leq b \vdash c+a \leq d+b}$

Cas 2 :

$$\dfrac{\dfrac{\dfrac{\dfrac{**2}{somme(l) \leq max(l)*long(l), max(l) \leq n \vdash somme(l) \leq n*long(l)}}{somme(l) \leq max(l)*long(l), max(l) \leq n \vdash somme(l)+n \leq n*long(l)+n}}{somme(l) \leq max(l)*long(l), max(l) \leq n) \vdash somme(l)+n \leq n*(long(l)+1)}}{somme(l) \leq max(l)*long(l) \vdash somme(l)+n \leq max(n::l)*(long(l)+1)}$$
$$\dfrac{somme(l) \leq max(l)*long(l) \vdash somme(n::l) \leq max(n::l)*long(n::l)}{P(l) \vdash P(n::l)}$$

**2 est vrai à cause des lemmes : $\dfrac{a \leq b \vdash c \leq d}{a \leq b \vdash c*a \leq d*b}$ et $\dfrac{\vdash c \leq d, \vdash a \leq e}{\vdash c \leq e}$

Q6) Quelle est la différence entre la sémantique d'évaluation et la sémantique de calcul?

Les expressions doivent être construites sur les nombres et sur les opérateurs habituels
$\exp = T_{¿¿}$
Example:
$\exp = \{3+2, (4*3)+1, 4*3+1, 3+3*5-6+2, \ldots\}$

Note1 : Expressions must be evaluated on numbers.
Note 2: The evaluation relation determines a relation:

$$eval \subseteq \exp \times N$$

$e \in \exp = T_{¿¿}$
$alors\ on\ note : e \Rightarrow n \Longleftrightarrow (e, n) \in eval$

$eval = (\exp, n)\ eval = (3+2, 5)\ eval = (3*4+1, 15)\ eval = (3*4+1, 13)$
What is Relation in math:
$R = A \times B = \{(a, b) \mid a \in A, b \in b\}\ denoted\ by\ R : A \to B$
Example:
$A = (m, n), B = (1,2)\ R_1 = A \times B = \{(m,1)(n,2)\}\ R_2 = \{(m,2)(n,1)\}\ R_2(m) = 2\ R_2(n) = 1$
$f(x) = y\ f : A \to B\ A = (1,2)\ B = (3,4)\ f(1) = 3, f(2) = 4\ f = \{(1,3), (2,4)\}$

Définition :

Semantique

$e \in \exp = T_{(+,-,*,/,)}(N) \wedge n \in N$
$¿ +¿_N, ¿_N, -¿_N, /\square_N ¿¿$ are function on N

$$R\,Constante : \frac{\square}{n \to n} \quad R+ : \frac{e \to n, e' \to n'}{e+e' \to n+¿_N n'} \quad R* : \frac{e \to n, e' \to n'}{e*e' \to n ¿_N n'} ¿$$

$$R- : \frac{e \to n, e' \to n'}{e-e' \to n-¿_N n'} \quad R/ : \frac{e \to n, e' \to n'}{e/e' \to n ¿_N n'} ¿$$

Example: $(3*4)+1=13$ $\dfrac{\dfrac{\square}{3 \to 3}, \dfrac{\square}{4 \to 4}}{3*4 \to 12}, \dfrac{\square}{1 \to 1}}{(3*4)+1 \to 13}$

Théorème d'unicité de la sémantique d'évaluation des expressions arithmétique :
$\forall e \in \exp = T_{¿¿} e \to n, e \to n' \Rightarrow n = n'$
Démonstration: on va démontrer par induction

- $P(x) = \{x \to n \wedge x \to n' \Rightarrow n = n'\}$
- $P(n)$
- $hyp : P(e)\ et\ P(é)\ sont\ vérifiée\ alors$
- $P(e+e') = \{e+e' \to n \wedge e+e' \to n' \Rightarrow n = n'\} = ?$
- $P(e-e')$
- $P(e*e')$
- $P(e/e')$

Théorème d'existence de la sémantique d'évaluation des expressions arithmétique :
$\forall e \in \exp = T_{¿¿}$
Démonstration :
$P(x) = \{\exists k \in N, x \Rightarrow k\}$

Sémantique computationnelle des expressions arithmétiques

- Calcul de l'effet de chaque étapes intermédiaires
- Forme de sémantique mettant en évidence les calculs effectifs

- La stratégie devient explicite

Nous allons:

- Donner les règles pour l'exemple des expressions arithmétiques
- Montrer leur validité et complétude

Definition de la sémantique computationelle:

$e, e', e'' \in \exp = T_{¿¿}$

$$RC+: \frac{}{n+n' \to n+¿_N n'} \quad RC*: \frac{}{n*n' \to n ¿_N n'} ¿$$

$$RC-: \frac{}{n-n' \to n-¿_N n'} \quad RC/: \frac{}{n/n' \to n ¿_N n'} ¿$$

$\forall op \in \{+, -, *, /\}$

$$RC\,L: \frac{e \to e''}{e\,op\,e' \to e''\,op\,e'} \quad RCR: \frac{e' \to e''}{e\,op\,e' \to e\,op\,e''}$$

Example

Soit l'expression (3* 4) +1 à calculer

Solution :

$$\frac{\overline{3*4 \longrightarrow 12}}{(3*4)+1 \longrightarrow 12+1'} \quad \frac{}{12+1 \longrightarrow 13}$$

Q7) comment comparer la sémantique d'évaluation et la sémantique de calcul (ou simplement deux sémantiques du même langage).

sémantique d'évaluation : en un pas $e \Longrightarrow n$

sémantique computationnelle : en plusieurs pas $e \Longrightarrow\Longrightarrow\Longrightarrow n$

Semantique