Oetske Leroux – Bachelor en Sciences informatiques 2ème

Sémantique des langages Informatiques – Questions

Examen le 16 juin 2020 à 9:30

Chapitre 5

Question 8 :
Expliquez ce qu'est le contexte, le domaine sémantique et les relations d'évaluations. Pourquoi sont-ils nécessaires dans la construction d'un langage de programmation ?

Le contexte est l'espace dans lequel le programme évolue. Il est nécessaire de le prendre en compte pour la construction d'un langage car le programme va interagir avec un environnement et il doit être capable de répondre à celui-ci.

Le domaine sémantique permet d'associer les valeurs sémantique à chaque catégorie syntaxique.

Les relations d'évaluations relient les programmes avec des résultats.

Question 9 :
Expliquez comment construire la sémantique du véhicule programmable.

Il faut déjà définir les mouvements possibles : M = {L,R,F} (left, right, front)
Un programme est une liste construites avec (concaténation :: , liste vide ε) de telles instructions.
Les programmes sont donc des termes : $T_{\{\varepsilon, ::\}}(\{L,R,F\}) = T_{\{\varepsilon, ::\}}(M)$

Exemple :

Suivre un carré correspond au programme :
square = F :: R :: F :: R :: F :: R :: F :: R :: ε

Pour faire avancer un véhicule programmable on a besoin des coordonnées et d'une direction.
Domaine sémantique : position ∈ Direct x Coord
avec Direct = {-1,0,1} x {1,0,1} = $\{-1,0,1\}^2$ / ?/
et Coord = Z x Z

La relation de transition pour une instruction est donc :
(Direct x Coord) x M x (Direct x Coord) que l'on note <d,p>, m → <d', p'>
Le véhicule est dans une direction *d* et une position *p*, on lui donne un mouvement *m*, et le véhicule se trouve dans une nouvelle direction *d*' et une nouvelle position *p*'.

Les règles pour les actions élémentaires :

i. ----------

<d,p>, L → <[0 -1 ; 1 0] * d, p> Rotation de 90° à gauche

ii. ----------
<d,p>, R → <[0 1 ; -1 0] * d, p> Rotation de 90° à droite

iii. ----------
<d,p>, F → <d, p + d> Unité d'avance

Évaluation récursive du programme :
Cas de base :

Rε : ---------- Pas de changement
<d,p>, ε → <d,p>

R: <d,p>, m → <d',p'>, <d',p'>, prog → <d'',p''> Succession d'instruction :
---------- (Direct x Coord) x $T_{\{\varepsilon, ::\}}$(M) x Direct x
<d,p>, m::prog → <d'',p''> Coord

Question 10 :
Expliquez comment le véhicule programmable pourrait être amélioré pour ajouter des obstacles.

Il est peut être améliorer en prenant compte de l'environnement dans lequel il se trouve. Il faudrait pour ça des capteurs.
On aurait besoin d'ajouter une syntaxe Event Driven qui :
- ajoute des instructions pour la prise en compte des capteurs
- indique ce que fait le véhicule en l'absence d'événement capteur

Ainsi qu'une syntaxe State Drive qui :
- ajoute des gardes sur l'état des variables 'capteur'
- indique ce que fait le véhicule en fonction de l'état

On aurait donc toujours d, p pour la direction et les coordonnées et en plus b, l'occurrence de l'événement {t,f}

La sémantique introduira donc un état supplémentaire indiquant qu'un capteur a été touché.

Syntaxe de l'Event Driven :

<d,p,t>, q → <c',d',b'>, q ∈ P

<d,p,f> → evProg < p',d',b'> avec q le programme réactif contrôlant les mouvements du véhicule

Syntaxe du State Driven :
Soit b le booléen modifiant l'état. On a <p,d,b>.

<p,d>, q → <p',d'>

<p,d, t>, if obstacle then q → <p',d',f>

<p,d, f>, if obstacle then q → <p,d,f>

<p,d>, q → <p',d'>

<p,d,f>, if no obstacle then q → <p',d', f>

<p,d,t>, if no obstacle then q → <p,d, f>

Chapitre 6

Question 11 :

Expliquez le principe d'assignation et de substitution de variables.

Les variables définissent un contexte ('état de la mémoire').
Contexte d'évaluation : assignation
Un contexte d'évaluation est ici un ensemble de substitution de variables par des valeurs. Il faut indiquer les valeurs que vont prendre chaque variable dans son domaine.

- Soit V l'ensemble des variables et $Exp_V = T_{\{1,-,*,/\}}(N \cup V)$ avec V le nom de la variable.
- Les assignation sont des fonctions des variables dans les valeurs : *assign* : V → N (domaine utilisé dans ce cas : les entiers).

Contexte d'évaluation : substitution

- Soit V l'ensemble des variables et $Exp_V = T_{\{1,-,*,/\}}(N \cup V)$
- Les substitutions sont des fonctions des termes et assignation dans les termes :
  subs : $T_{\{1,-,*,/\}}(N \cup V) \times$ *assign* → $T_{\{1,-,*,/\}}(N \cup V)$

Exemple de substitution :
(ε/[v=2])/[w=5]((3 * v) + w) = (3 * 2) + 5 avec [v=n] signifiant que l'assignation de la variable v prend la valeur n.

Définition inductive de la substitution :
Soit un ensemble d'opérations OP et V un ensemble de variables, une substitution est une relation satisfaisant les propriétés suivantes :

$$\frac{}{\varepsilon \in Subs_{OP,C,V}} \qquad \frac{s \in Subs_{OP,C,V}, v \in V, e\ s \in Top(C \cup V)}{s/[v = e]\ s \in Subs_{OP,C,V}}$$

Sémantique d'évaluation
Relation d'évaluation : *eval* :$(Exp_V \times Subs) \times N$
Notation : $e \in ExpV = T_{\{1,-,*,/\}}(N \cup V)$
On utilise : $S \vdash e \rightarrow n$ pour $(e,S,n) \in$ *eval*.

Definition (Sémantique d'évaluation)

$e \in ExpVar = T_{\{+,-,*,/\}}(\mathbb{N} \cup V)$ et $n \in \mathbb{N}$, $s \in Subs$
$+_{\mathbb{N}}, *_{\mathbb{N}}, -_{\mathbb{N}}, /_{\mathbb{N}}$ sont les fonctions sur $\mathbb{N}$

$$\text{R Constante} : \frac{}{S \vdash n \Longrightarrow n} \qquad \text{R var} : \frac{}{S/[v = n] \vdash v \Longrightarrow n}$$

$$R+ : \frac{S \vdash e \Longrightarrow n, S \vdash e' \Longrightarrow n'}{S \vdash e + e' \Longrightarrow n +_{\mathbb{N}} n'} \qquad R* : \frac{S \vdash e \Longrightarrow n, S \vdash e' \Longrightarrow n'}{S \vdash e * e' \Longrightarrow n *_{\mathbb{N}} n'}$$

$$R- : \frac{S \vdash e \Longrightarrow n, S \vdash e' \Longrightarrow n'}{S \vdash e - e' \Longrightarrow n -_{\mathbb{N}} n'} \qquad R/ : \frac{S \vdash e \Longrightarrow n, S \vdash e' \Longrightarrow n'}{S \vdash e/e' \Longrightarrow n/_{\mathbb{N}} n'}$$

Remarque : la relation d'évaluation est définie pour les expressions dont les variables sont définies dans la substitution, sinon elle est indéfinie.

Question 12 :

Expliquez les structures de contrôles "if then else" et "while do ".

Soit V l'ensemble des variables.
Les expressions $ExprVar_V$ construites sur les nombres et sur les opérateurs usuels.

- $Rel_V = T_{\{<, \leq, >, \geq, =\}}(Expr_V)$ Relation (prédicat)
- $Instr_V = T_{\{if\_then\_else,\ while\_do\}}(Expr_V \cup Rel_V)$
- if_then_else : $Rel_V \times Bloc_V \times Bloc_V \rightarrow Instr_V$
- while_do : $Rel_V \times Bloc_V \times Instr_V$
- (: = /affectation/:V x $Expr_V \rightarrow Instr_V$)

avec $Bloc_V$ étant les séquences d'instructions
- $\varepsilon : \rightarrow Bloc_V$ Bloc vide
- -;- : $Instr_V \times Bloc_V \rightarrow Bloc_V$ Séquence d'instructions

On a donc $Bloc_V = T_{\{-;-,\ \varepsilon\}}(Instr_V)$.

Relations d'évaluations : il faut les déinir pour chaque domaine syntaxique.
- Les instructions : Subs $\vdash Instr_V \rightarrow_I$ Subs
- Les blocs : Subs $\vdash Bloc_V \rightarrow_B$ Subs

/Ne concerne pas if then else et while do, mais on les utilise, je les ai donc mis pour la culture/
[
Évaluation d'une instruction : affectation

Definition (Sémantique d'évaluation : Règle affectation)

$e \in ExprVar_V$ et $v \in V$, $S, S', S'' \in Subs$

$$Raffectation : \frac{S \vdash e \Longrightarrow n}{S \vdash v := e \Longrightarrow_I S/[v = n]}$$

Évaluation d'un bloc :

Definition (Sémantique d'évaluation)

$i \in Instr_V$ et $p \in Bloc_V$, $S, S', S'' \in Subs$

$$\text{R Prog vide} : \frac{}{S \vdash \epsilon \Longrightarrow_B S}$$

$$Rsequence : \frac{S \vdash i \Longrightarrow_I S', S' \vdash p \Longrightarrow_B S''}{S \vdash i; p \Longrightarrow_B S''}$$

]

Évaluation d'une instruction : *if then else*

Definition (Sémantique d'évaluation : Règles IFTHENELSE)

$p, p' \in Bloc_V$ et $r \in Rel_V$, $S, S' \in Subs$

$$RIFTHEN : \frac{S \vdash r, S \vdash p \Longrightarrow_B S'}{S \vdash if\ r\ then\ p\ else\ p' \Longrightarrow_I S'}$$

$$RIFELSE : \frac{S \nvdash r, S \vdash p' \Longrightarrow_B S'}{S \vdash if\ r\ then\ p\ else\ p' \Longrightarrow_I S'}$$

Évaluation d'une instruction : *while do*

Definition (Sémantique d'évaluation : Règles WHILE)

$p \in Bloc_V$ et $r \in Rel_V$, $S, S' \in Subs$

$$RWHILEDO : \frac{S \vdash r, S \vdash p; while\ r\ do\ p \Longrightarrow_B S'}{S \vdash while\ r\ do\ p \Longrightarrow_I S'}$$

$$RDONE : \frac{S \nvdash r}{S \vdash while\ r\ do\ p \Longrightarrow_I S}$$

Question 13 :

Comment modifiez la règle ”if then else” pour ne plus prendre en compte le ”else” ? Comment modifier la règle ”while do ” pour en faire un ”do while ” ?

Question 14 :

Expliquez comment les fonctions ont été introduites dans le langage.

- Une fonction est un morceau de code qui peut être appelé n'importe où dans les expressions.
- Une fonction a un nom
- Une fonction à des paramètres et retourne un résultat
- Les paramètres d'une fonction sont les paramètres formels
- Lors de l'appel ils deviennent les paramètres effectifs à l'intérieur de la fonction
- La visibilité des variables est limitée à la fonction (pas de variables globales!)

Les expressions sont étendues avec l'appel de fonctions :
Soit V l'ensemble des variables et $Exp_{F,V} = T_{\{+,-,*,/\}uF}(N \cup V)$ et les relations sur les entiers.

Syntaxe :

- $Rel_{F,V} = T_{\{<, \leq, >, \geq, =\}}(Expr_{F,V})$ Relation (prédicat)
- $Instr_{F,V} = T_{\{if\_then\_else,\ while\_do,:=,\ return\}}(Expr_{F,V} \cup Rel_{F,V})$
- if_then_else : $Rel_{F,V} \times Bloc_{F,V} \times Bloc_{F,V} \rightarrow Instr_{F,V}$
- while_do : $Rel_{F,V} \nrightarrow Bloc_{F,V} \times Instr_{F,V}$
- : = /affectation/: $V \times Expr_{F,V} \rightarrow Instr_{F,V}$
- return : $Exp_{F,V} \rightarrow Instr_{F,V}$ Nouvelle isntruction

avec $Bloc_V$ étant les séquences d'instructions

- $\varepsilon : \rightarrow Bloc_{F,V}$ Bloc vide
- -;- : $Instr_{F,V} \times Bloc_{F,V} \rightarrow Bloc_{F,V}$ Séquence d'instructions

Définition de Fonction :

- Les fonctions $Func_{F,V}$ construites sur les blocs et le nom de la fonction avec ses paramètres. $Func_{F,V} = T_F(V) \times Bloc_{F,V}$

Les programmes sont donc composées de fonctions et d'un corps

$$Prog_{F,V} = P(Func_{F,V}) \times Bloc_{F,V}$$

Le contexte inclus les définitions de fonctions
Évaluer une fonction nécessite d'associer les paramètres formels avec les paramètres effectifs, la visibilité des variables est limitée à la fonction, stopper l'évaluation lorsqu'il y a un *return*.

Relations d'évaluations :
Il faut les définir pour chaque domaine syntaxique :

- Les expressions : $p(Func_{F,V}), Subs \vdash Expr_{F,V} \rightarrow N$
- Les instructions : $p(Func_{F,V}), Subs \vdash InstrF,V \rightarrow_I Subs \times (N \cup \{ \_|\_ \})$
- Les blocs : $p(Func_{F,V}), Subs \vdash Bloc_{F,V} \rightarrow_B Subs \times (N \cup \{ \_|\_ \})$
- Les fonctions : L'évaluation est intégrée dans l'évaluation des expressions
- Les programmes : $p(Func_{F,V}), Subs \vdash Bloc_V \rightarrow_P Subs$

avec $(N \cup \{ \_|\_ \})$ la valeur de retour. //_|_ correspond à un I sans la bar du haut, mais pas trouvé le symbole.

Évaluation d'un bloc :

Definition (Sémantique d'évaluation)

$i \in Instr_V$ et $p \in Bloc_V$, $S, S', S'' \in Subs$

$$\text{R Prog vide} : \frac{}{S \vdash \epsilon \Longrightarrow_B < S, \perp >}$$

$$Rsequence : \frac{S \vdash i \Longrightarrow_I < S', \perp >, S' \vdash p \Longrightarrow_B < S'', m >}{S \vdash i; p \Longrightarrow_B < S'', m >}$$

$$Rsequenceret : \frac{n \neq \perp, S \vdash i \Longrightarrow_I < S', n >}{S \vdash i; p \Longrightarrow_B < S', n >}$$

Évaluation d'une fonction dans une expression :

Definition (Sémantique d'évaluation de l'appel de fonction)

$$\frac{< f(x_1, .., x_n), b > \in F, \quad F, S \vdash e_1 \Longrightarrow m_1, .., F, S \vdash e_n \Longrightarrow m_n, \epsilon/[x_1 = m_1]/.../[x_n = m_n] \vdash b \Longrightarrow_B < S', m >}{F, S \vdash f(e_1, ..., e_n) \Longrightarrow m}$$

Explication :

- $f(e_1, ..., e_n)$ est l'appel de la fonction f dans une expression
- $< f(x_1, ..., x_n), b > \in F$ est la définition de la fonction f à appeler. $x_1, ..., x_n$ sont les paramètres formels et b est le corps de la fonction.
- $F, S \vdash e_1 \Longrightarrow m_1, ..., F, S \vdash e_n \Longrightarrow m_n$, calcule les paramètres effectifs
- $\epsilon/[x_1 = m_1]/.../[x_n = m_n]$ construit l'assignation des paramètres formels aux paramètres effectifs
- $\epsilon/[x_1 = m_1]/.../[x_n = m_n] \vdash b \Longrightarrow_B < S', m >$ évalue le bloc b pour les valeurs prises par les paramètres formels, le résultat est m.

L'absence de 'return' rend la fonction indéfinie.

Chapitre 7

Question 15 :
Expliquez ce qui a été échangé par rapport au langage du chapitre 6 pour prendre en compte l'évaluation paresseuse.

L'**évaluation paresseuse** : on évalue une expression si et seulement si une décision doit être prise, sinon rien n'est calculé.
Dans la sémantique du chapitre 6, on évaluait tout de suite l'expression tandis qu'avec celle-ci, on fait une affectation sur l'expression mais on ne l'évalue qu'en cas de décision à prendre → on manipule les expressions, on ne les évalue pas.
Il existe donc deux types de règles dans cette sémantique :
Quand les manipule-t-on ?
Règles paresseuses : Affectation et Appel de fonction (indirectement)
Quand les évalue-t-on ?
Si une décision doit être prise. Règle immédiate : Condtion *if then else* ou dans *while*
L'évaluation paresseuse permet d'augmenter l'espace de définition des programmes.

Exemple :

if x > 0 *then* r:= 3/x *else* r:= 0

n'est pas équivalent à :

func f(b,x,y)

{*switch* b *do* *true : return* x *false : return* y}

r:= f(x > 0, 3/x, 0) → Supposons que x vaut 0. Avec la **eager semantics** on commence par vérifié si x > 0 → 0 > 0 : *false*, ensuite on substitue et on calcule 3/x → 3/0 or 3/0 n'est pas défini et on est donc bloqué.

Eager semantics :

Definition (Sémantique d'évaluation : Règle affectation)

$e \in ExprVar_V$ et $v \in V$, $S, S', S'' \in Subs$

$$Raffectation : \frac{S \vdash e \Longrightarrow n}{S \vdash v := e \Longrightarrow_I S/[v = n]}$$

Lazy semantics :

Definition (Sémantique d'évaluation : Règle affectation)

$e \in ExprVar_V$ et $v \in V$, $S \in Subs$

$$Rlaffectation : \frac{e \Longrightarrow_S e'}{S \vdash v := e \Longrightarrow_I S/[v = e']}$$

On transforme une expression par une autre expression par le biais d'une substitution (e →$_S$ e').

On évalue donc *e* en *e'* et depuis la substitution *S*, *v* prend la valeur de *e'*.
On a besoin des règles suivantes :

$\frac{v \notin Dom(S)}{v \underset{s}{\Longrightarrow} v}$ *v* n'appartient pas au domaine de définition, *v* n'est donc pas bien défini. On ne change pas le résultat de l'expression.

$\frac{v \in Dom(S)}{v \underset{S/[v=e]}{\Longrightarrow} e}$ *v* appartient au domaine, *e* est donc le résultat de l'expression.

$\frac{}{n \underset{s}{\Longrightarrow} n}$ Valeurs constantes

$\frac{e \underset{s}{\Longrightarrow} e'', e' \underset{s}{\Longrightarrow} e'''}{e+e' \underset{s}{\Longrightarrow} e''+e'''}$ expressions

$\frac{e_1 \underset{s}{\Longrightarrow} e_1', \ldots, e_n \underset{s}{\Longrightarrow} e_n'}{f(e_1,\ldots,e_n) \underset{s}{\Longrightarrow} f(e_1',\ldots,e_n')}$ fonctions

Les règles *if then else* sont identiques au langage 'eager' car elles appliquent les règles immédiates.

Concernant le programme, il faut simplement s'assurer qu'il y a une évaluation complète finale et qu'il ne reste plus d'instructions à exécuter une fois le

programme terminé.

Definition (Sémantique d'évaluation d'un programme)

$p \in Bloc_V$, $S, S', S'' \in Subs$

$$\frac{S \vdash p \Longrightarrow_B S'', S'' \Longrightarrow S'}{S \vdash p \Longrightarrow_P S'}$$

NB : La relation ⇒ est deterministe.
La relation ⇒ 'lazy' produit un résultat que la relation 'eager' produit.
La relation ⇒ 'lazy' produit plus de résultats que la relation 'eager' ne produit (fonction partielle).

Question 16 :

Expliquez toutes les étapes nécessaires à la création d'une sémantique pour un langage de programmation.

1. Définir une syntaxe
   Définir un ensemble de règles précisant la manière d'écrire et/ou de disposer les choses
   - Définir les éléments de base : variables, fonctions etc
   - Définir les noms des domaines syntaxique(ensemble dans lequel on a des objets syntaxiques) : par exemple F pour les fonctions, X pour les variables...
   - Définir les domaines syntaxiques : inductivement, EBNF(ne correspond pas exactement à une syntaxe abstraite)
     Exemple : (TP3 – Le donjon de BEUKNAHEUL)
     Langage des humains : LH
     - Élément de base : attaques, manger, dormir
     - Nom des domaines syntaxiques : {AtGa, AtDr, AtFr, DeGa, DeDr, DeFr, M, S}
     - Définition des domaines syntaxique :
       EBNF :
       L := E L | ε
       E := AtGa | AtDr | AtFr | DeGa | DeDr | DeFr | M | S |
       avec:
       L ∈ LH E ∈ ExpressionsHumaines
       ExpressionsHumaines = {AtGa, AtDr, AtFr, DeGa, DeDr, DeFr, M, S}
       ε ∈ LH

       Par définition inductive:
       Cas de base:

       ε ∈ LH

       Décomposition:
       E ∈ ExpressionsHumaines, L ∈ LH

       E L ∈ LH

---------- AtGa ∈ ExpressionsHumaines
AtDr ∈ ExpressionsHumaines

---------- AtFr ∈ ExpressionsHumaines
DeGa ∈ ExpressionsHumaines

---------- DeDr ∈ ExpressionsHumaines
DeFr ∈ ExpressionsHumaines

---------- M ∈ ExpressionsHumaines
S ∈ ExpressionsHumaines

2. Définir un domaine d'interprétation sémantique
   Comment je vais interpréter et donner un sens à mon langage ?
   Exemple : si on travaille sur les entiers, les réels sont importants pour comprendre la sémantique du langage.

3. Définir le contexte global d'évaluation
   « On sait qu'on va avoir des differents categories syntaxiques (les blocs, les instructions, les expressions) et on aimerait savoir dans quel contexte on doit évaluer ces differentes constructions. Par exemples, pour les expressions souvent le context c'est les substitutions. Mais pour un bloc on doit pas seulement connaitre les expressions mais aussi le nom de la fonction, donc si on évalue un bloc il nous faut un contexte qui est lié aux fonctions et aux substitutions. »
   Exemple :

4. Définir les relations d'évaluation sémantique pour chaque catégorie syntaxique
   Relation entre le domaine syntaxique et le domaine sémantique. (Sémantique d'évaluation – big step, sémantique computationnelle – small steps)
   C |- a ⇒ r . En fonction du contexte 'C ', on construit une relation (de 'a' on produit un résultat 'r') avec 'a' une catégorie syntaxique.

5. Définir les règles d'inférences
   La dernière étape est donc de construire les règles d'inférence pour décrire comment on construit cette relation.
   Exemple : c.f. TP3

Chapitre 8

Question 17 :
Expliquez le processus pour interpréter un programme.

Note : Utiliser la question 16.

Question 18 :

Expliquez l'intérêt et la démarche pour l'analyse de programmes.

Que faut-il analyser ?
Différents types.
Si je connais un programme -> déterminer un certains nombre de propriété.

Analyse programme :
Trouver pour un programme donnée, quelques propriétés.
Détecter les chemins d'executions possibles.

Par exemple chemin potentiellement possible car il y a des tests mais n'est jamais executable :
code mort. Variables non initalisées etc.

Comment le faire ?
1) Construction d'un interprete -> 1. l'executer directement sur un certains nombres de données choisies = **faire du test**
Probleme :
On ne peut jamais être exhaustif car si on utilise des chaines de caratere ou des entiers : infini.

2)Il faut trouver des tests bien choisi : pouvoir observer des execution possible du prgramme : **Execution symbolique**

Exemple ch 8 slide 25 :
(eval(e), n) et on enumère toutes les possibilités.

Comment mieux faire ?
Idée : à la place d'avoir des instances spécifiques, essayer d'avoir des termes qui vont donner des classes d'objets qui représente ces objets. (lien avec la lazy evalutation car on ne calcule pas avec des valeurs directes mais des expressions).
Cela nous permet :
Avoir une représentation finie d'une infinité d'instances.

Question 19 :
Expliquez comment et pourquoi il peut être intéressant de changer la représentation des types de données.

Cf p 28
Représenter des données différemment : pourquoi ?
L'Implication va seulement dans un sens
(à cause de is en prolog, c'est une fct stricte : fct qui nécessite la connaissance des paramètres à droite)
On aimerait que is ne soit pas strict en fait. Il faut donc rendre inversibles les opérations et donc représenter les entiers par des générateur (par exemple 3 est successeur de succ de succ de 0) avantage on peut écrire succ de x.

Ch 8 p.25

Axiomatisation de Peano (qui est une définition inductive)
Suffit pour l'addition :
0 + N = N
S(N) + N' = S(N+N')
problème en Prolog : + est une fonction, et en Prolog on a des prédicats.
Ce qui donne : peanoadd(0 : (nat), NN, NN). //(equivalent à 0+NN=NN)
peanoadd(s(N) : (nat->nat), NN, s(NN):(nat->nat)):-
peanoadd(N,NN, NNN).
0: (nat) c'est le type , on l'introduit dans le cas où il y a des problèmes de surcahge.
Typage est le profile de l'opération. Nécessaire de connaître les générateurs disponibles pour chaque opérateur.

Chapitre 9

Question 20 :
Expliquez comment représenter la sémantique d'un langage de programmation logique. Quelles sont les limites de la résolution SLD de Prolog.

Chapitre 10

Question 21 :
Expliquez ce qu'est la notion de type et de typage. En quoi est-ce utile ? Quelle est la différence entre un typage statique et dynamique ? Quelle est la différence entre type faible et fort ?

Question 22 :
Expliquez ce qu'est la compatibilité de types, l'´équivalence de types et l'inférence de types.

Question 23 :
Expliquez ce qu'est la portée, visibilité et durée de vie d'une variable.