Question 1: Expliquer le principe général de la syntaxe dans les langages de programma -tion et comment le définir.

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

La syntaxe **fait référence aux façons dont les symboles peuvent être combinés pour créer des phrases** (ou programmes) bien formées dans la langue. La syntaxe **définit les relations formelles entre les constituants d'une langue**, fournissant ainsi une description structurelle des différentes expressions qui composent les chaînes juridiques de la langue**. La syntaxe traite uniquement de la forme et de la structure des symboles** dans une langage **sans aucune considération pour leur signification.**

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)

Utilise BNF or EBNF ?
Comment définir la syntaxe:
Méthodes formelles de description de la syntaxe
La syntaxe des langages de programmation **utilise des grammaires pour décrire les mécanismes** de langage formels utilisant la grammaire indépendante du contexte.

La grammaire indépendante du contexte provient à l'origine de la théorie formelle du langage. En informatique, **la grammaire** indépendante du **contexte implique la syntaxe du langage de programmation qui est des jetons.**

Formulaire Backus-Naur (BNF)
En informatique, la BNF (Backus Normal Form ou Backus-Naur Form) **est une technique de notation pour les grammaires hors contexte**, souvent **utilisée pour décrire la syntaxe des langages utilisés en informatique**, tels que les langages de programmation informatique, les formats de documents, les jeux d'instructions et la communication. protocoles

Analyse syntaxique et lexicale : L'analyse lexicale est **la première phase de la compilation**, analyse lexicale **reconnaît des *unités lexicales*** (les identificateurs, les caractères spéciaux simples , les caractères spéciaux doubles, les mots-clés , les constantes littérales), qui sont les mots avec lesquels les phrases sont formées . Pour ce faire, on associe des automates finis aux expressions régulières.

1. Expressions régulières
2. Automates

Expression régulière :
Définition : *Soit Σ un alphabet. Les expressions rationnelles (ou régulières) sur Σ et les langages correspondants sont définis récursivement :*
1) • ∅ *est une expression rationnelle, et représente l'ensemble vide*,
   • ε *est une expression rationnelle, et représente l'ensemble* {ε},
   • *pour toute lettre a de Σ, a est une expression rationnelle, et représente l'ensemble* {*a*}.

2) *Si **r** et **s** sont des expressions rationnelles, qui représentent les langages R et S, alors **r + s**, **rs**, et **r*** sont des expressions rationnelles qui représentent les langages $R \cup S$,RS et R^*.*

Les expressions régulière sont des outils très puissants et très utilisés : on peut les retrouver dans de nombreux langages comme le PHP, MySQL, Java script… ou encore dans des logiciels d'édition de code ! Voici une syntaxe d'expressions régulières assez commune (syntaxe de l'outil Lex) :

[xz]	le caractère x ou le caractère z
[x − z]	un caractère dans l'intervalle lexicographique x à z
\n \ 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

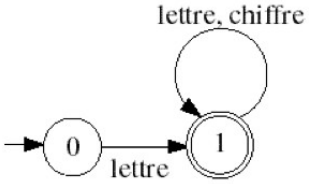
Les automates finis :

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**.
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 : Par exemple, dans un langage comme Pascal, les identificateurs sont définis par le modèle suivant :

lettre → [A-Za-z]
chiffre→ [0-9]
id → **lettre** (**lettre** | **chiffre**)*



L'expression rationnelle définissant l'unité lexicale **id** se traduit par un automate :
Dans le fonctionnement d'un tel automate, **on associe une action à chaque état final** : si on lit par exemple « var1 », on suit le parcours 0111 dans l'automate, qui s'arrête dans l'état final 1. On a alors détecté un identificateur.
Action : l'analyseur **forme l'unité lexicale (id, var1) et ajoute le symbole var1 dans la table des symboles.**

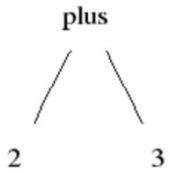
Question 2: Expliquez ce qu'est la **syntaxe concrète** et ce qu'est la **syntaxe abstraite** d'un langage de programmation
La syntaxe : est un domaine bien compris et formalisé, mais qui **ne s'intéresse qu'aux propriétés structurelles et grammaticales du langage** . **La syntaxe établit la structure, pas la signification**, mais la signification d'une phrase ou d'un programme est liée à sa syntaxe.

syntaxe abstraite : identifie les composantes significatives des constructions du langage et 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. La **syntaxe abstraite** d'une implémentation est l'ensemble d'arbres utilisés pour représenter les programmes dans l'implémentation.

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, a**lors 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]. 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 et pas seulement un détail d'implémentation**.

syntaxe concrète : La syntaxe concrète d'écrit complètement la représentation écrite (placement des parenthèses, ponctuation, etc). Elle est définie par une grammaire hors 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.

syntaxe abstraite : quelles sont les parties significatives de l'expression?
Exemple :une expression de somme a ses deux expressions d'opérande comme parties significatives

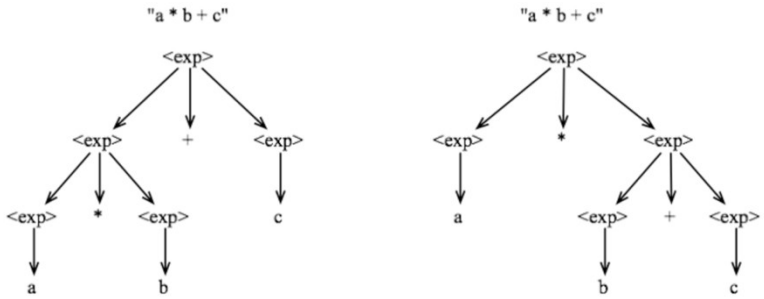


Syntaxe concrète : à quoi ressemble l'expression?
Exemple: la même expression de somme peut se présenter de différentes manières:

```
2 + 3              -- infix
(+ 2 3)            -- prefix
(2 3 +)            -- postfixe
sum of 2 and 3     -- English
```

Question 3: Explain what the ambiguity of syntax is in a programming language

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**.
Par exemple : a * b + c



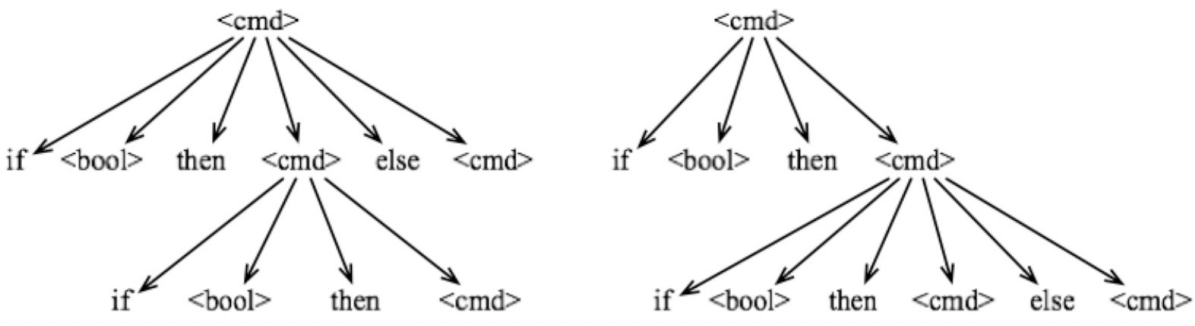
Afin de voir que cette grammaire est ambiguë, nous pouvons observer qu'il est possible de dériver deux arbres de syntaxe différents pour la chaîne "a * b + c".
Une façon d'éviter toute ambiguïté dans une langue est d'utiliser des parenthèses :
(a* b) + c ou a*(b + c)
Un exemple particulièrement célèbre d'ambiguïté dans les compilateurs se produit dans la construction if-then-else. L'ambiguïté se produit car de nombreux langages autorisent la clause conditionnelle sans la partie "else". Considérons un ensemble typique de règles de production que nous pouvons utiliser pour dériver des instructions conditionnelles:

$$¿\,cmd\,>::=if\,<bool>then<cmd>¿if\,<bool>then<cmd>else<cmd>¿$$

En tombant sur un programme comme le code ci-dessous, nous ne savons pas si la clause "else" est associée à la plus externe ou à la plus interne "alors". En C, ainsi que dans la grande majorité des langages, les compilateurs résolvent cette ambiguïté en associant un "else" avec le "then" le plus proche. Par conséquent, selon cette sémantique, le programme ci-dessous affichera la valeur 2 chaque fois que a > b et c <= d:

```
if (a > b) then
 if (c > d) then
   print(1)
 else
   print(2)
```

Cependant, la décision d'associer le "else" au "then" le plus proche est arbitraire. Les concepteurs de langage auraient pu choisir de coupler le bloc "else" avec le bloc "then" le plus externe, par exemple. En fait, la grammaire que nous avons vue ci-dessus est ambiguë. Nous démontrons cette ambiguïté en produisant deux arbres de dérivation pour la même phrase, comme nous le faisons dans l'exemple de la figure ci-dessous:
L'ambiguïté syntaxique peut être gérée de deux manières. Elle peut être supprimée dans la grammaire.
La syntaxe non ambiguë conduit à une sémantique non ambiguë; ou il peut être géré par



la phase sémantique du compilateur, en appliquant une règle. Étant donné que de nombreux détails des langues sont traités au niveau sémantique, en particulier ceux qui ne peuvent pas être exprimés par les grammaires BNF, il semblerait approprié de prendre la deuxième voie. Cependant, de nombreux concepteurs de langage estiment que toute ambiguïté syntaxique doit être supprimée. La section 3.3.1.7 décrit la suppression standard de l'ambiguïté dans les expressions en appliquant une notion de priorité d'opérateur dans les règles BNF elles-mêmes. Cela se fait en supprimant la double récursivité et en créant deux règles sur une. La section 3.3.1.10 fait de même avec le formulaire if-then-else en C. Nous pouvons également changer la syntaxe pour simplement éviter le problème. Modula 2, par exemple, insiste sur et FIN le mot réservé pour terminer chaque instruction if. c'est à dire. Ce qui est ambigu en C:

```
if (x > 0)
  if (y = 0)
  {
    x = 1;
    y = 2;
  }
else
  y = 3;
```

devient sans ambiguïté dans le module 2:

```
IF X > 0 THEN
  IF y = 0 THEN
    x := 1;
    y := 2
  END
ELSE
  y := 3
END;
```

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 inductif et minimalité

Définition (des nombres naturels)

- **0** $\in N$
- **x** $\in N \Longrightarrow x+1 \in N$

les ensemble Q et R , … satisfaitent aussi cette propriété

Définition inductives d'ensembles

Théorème : soit S cet ensemble S = $\{ Xt.q\, 0 \in X \wedge \forall x, x \in X \Longrightarrow x+1 \in X \}$
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 $\in EP \Longrightarrow x+2 \in EP$

Les entier pairs sont ensemble minimum satisfaisant ces propriétés mais il y' a des autres ensembles satisfaisants ces propriétés

$$EP_1 = [0, 2, 4, 6, \ldots]$$
$$EP_2 = [0, 0.5, 2, 2.5, 4, 4.5, 6, \ldots]$$
$$EP_3 = [0, 0.5, 0.7, 2, 2.5, 2.7, 4, 4.5, 4.7, 6, \ldots]$$

Remarque :
La propriété ∀k ∈ℕ‹P(k) = 2k ∈EP est valable pour tous les EP sans être minimal.

(en fait la propriété porte sur ℕ uniquement. Par contre ℕ est minimal)
Récurrence 2 ∗0 = 0 ∈EP et 2k ∈EP implique 2 ∗(k + 1) ∈EP, ici 2 ∗K + 2.

La propriété∀n ∈EP‹P(n) = ∀m ∈EP‹n + m ∈EP nécessite l'hypothèse de minimalité . 0▷5 + 2▷5 = 3 $\notin EP_2$

Les jugements sur les prédicats sont construits sur le même modèle que les jugements en logique . Définissons d'ensembles : $\frac{\square}{\vdash 0 \in EP}$ $\forall k, \frac{\vdash k \in EP}{\vdash k+2 \in EP}$

$$\forall k, \forall n \frac{\vdash 0 \in N}{\vdash 0 \in DIV_0}, \forall k, \forall n \frac{\vdash k \in N \wedge \vdash n \in DIV_k}{\vdash n \in DIV_{n+k}}$$

Si le jugement n'a pas de partie gauche, la forme permisses ⇒ conclusion est notée : $\frac{permission}{conclusion}$

$$\frac{\square}{0 \in EP}, \frac{k \in EP}{k+2 \in EP}$$

$$\frac{n \in N}{n \in DIV_0} \quad \frac{k \in N, n \in DIV_k}{n \in DIV_{n+k}}$$

déduction :Une déduction est une série d'application de règles de définition inductives

- Les règles se composent 'verticalement' ou une prémisse doit être la conclusion d'une autre règle.
- Les règles des plus haut niveaux sont les règles de bases sans prémisses
- Les variables peuvent être instanciées dans leurs domaines de définitions.

Exemple : prouver que 4 ∈ EP

$$\frac{\frac{\overline{0 \in EP}}{0+2 \in EP}}{2+2 \in EP}$$

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

Soit les 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*

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

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

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

$$\frac{n \not\geq max(l), max\ (l) = m}{max(n :: l) = m}$$

Max(l) somme(l) long(l) ¿ ≤

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

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

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

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

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

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

$$induction: P(l)\vdash P(n::l)N$$

Cas 1 :

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

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

Cas 2 :

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

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

Q6) What is the difference between evaluation semantics and computational semantics?
L'idée de sémantique d'évaluation est de prends quel part un syntaxe et nous fourni un objet sémantique qui représente la situation après l'exécution de l'expression .

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 : les expression doivent etre évalué sur les nombres.
Note 2: La relation d'évaluation détermine une relation

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

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

$eval = (\exp, n)\ eval = (3+2, 5)\ eval = (3*4+1, 15)\ eval = (3*4+1, 13)$

Définition : pour la definition on a une ensemble de règle
$e \in \exp = T_{(+,-,*,/,)}(N)\ et\ n \in N$
$sont +¿_N, ¿_N, -¿_N, /❑_N\ ¿¿$ des fonctions dans N

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

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

Example: $on\ veut\ évaluer\ (3*4)+1 = 13$

$$\frac{\frac{❑}{3 \to 3}, \frac{❑}{4 \to 4}}{3*4 \to 12}, \frac{❑}{1 \to 1}$$
$$(3*4)+1 \to 13$$

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

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

par induction on va prouver pour

- $P(e+e')\ est\ unique\ ?\ n+n'\ est\ unique\ ?$
- $P(e-e')$
- $P(e*e')$
- $P(e/e')$

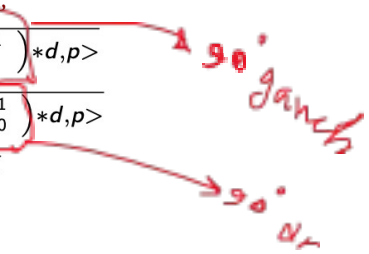
L'unicité pour les fonction :
les fonctions soit déterministes càd -> f : $N \to \ldots$ f(3) est déterministe càd il y'a un seul résultat -> la fonction est bijective
un prédicat n'est pas une fonction mais on peut forcer les prédicats à être déterministe selon leurs définition .

Théorème d'existence de la sémantique d'évaluation des expressions arithmétique :
$\forall e \in \exp = T_{¿¿}$
Démonstration :
$P(x) = \{\exists k \in N, x \Rightarrow k\}$
Exemple : sémantique d'un véhicule programmable
Les mouvment possibles : M = {L , R , F}

Les programmes sont de termes $T_{¿¿}$)
Par exemple suivre une carré corresponde au programme
Square = F :: R :: F :: R :: F :: R :: F :: R :: ε
D'aord on doit fixer une demaine sémantique **Direction** ×**coordonné** où coordonné = $Z \times Z$
et direction = {-1 , 0 , 1} ×{-1 , 0 , 1} = {-1 , 0 , 1}^{2}
et après définir des règles pour définir une sémantique d'évaluation .

$¿d, p>, m = ¿d', p'>¿$

- $\dfrac{}{<d,p>, L \Rightarrow <\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} * d, p>}$
- $\dfrac{}{<d,p>, R \Rightarrow <\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} * d, p>}$
- $\dfrac{}{<d,p>, F \Rightarrow <d, p+d>}$



Avec la sémantique d'évalution : j'ai une position <d,p> je prends un programme « prog » et finalement avec la sémantique d'évalution je me retrouve dans un nouvel position <d',p'>
$¿d, p>, prog \Rightarrow <d', p'>¿$

le problème avec ça c'est on sait pas les étapes intermédiaire
$¿d, p>, mov \Rightarrow <d', p'>, prog \Rightarrow <d'', p''> \dfrac{¿}{¿d, p>, mov :: prog \Rightarrow <d'', p''>¿¿}¿$

Typage :
Sémantique d'évaluation Typé On s'intéresse de type de valeur qui sont calculé

$T \vdash e \Longrightarrow v : t$ où $t \in T, v \in Dom(t), e \in \exp$

Si je connait le type T une expression va s'évaluer on une valeur du domaine qui aura un certain type t
Dans un compilateur en général on sépare est-ce qui est ***typage statique*** qui est fait à la compilation et le ***typage dynamique*** qui est liée à l'exécution.
p.x dans le cas de java il y'a une partie qui est fait à la compilation et une partie fait à l'exécution .

Definition (Sémantique d'évaluation typée)
$e \in Exp = T_{OP}(\emptyset)$
op sont les fonctions sur les types

R Constante : $\dfrac{n \in OP, \mu(n) = \epsilon t, t \in T}{T \vdash n \Longrightarrow n_t : t}$

Rop2 : $\dfrac{op \in OP, \mu(op) = t_1 t_2 t, T \vdash e \Longrightarrow n : t_1, T \vdash e' \Longrightarrow n' : t_2}{T \vdash e\ op\ e' \Longrightarrow n\ op_t\ n' : t}$

je prends un opérateur op et je sait que cette opérateur a un profil t_1t_2 dans t.
j'évalue e à n avec certain type t_1 et j'évalue
e' à n' avec certain type t_2 alors à ce moment j'évalue e op é à n op_t n' .
op_t : l'opérateur effective du type t

on accepte que des objet à type indiqué -> x'est un typage fort.

Sémantique computationnelle des expressions arithmétiques :
l'idée est pas de donner un sens à une expression directement mais de dire comment je veux calculé des différentes expressions . introduire la possibilité de choisir la facon d'évaluer une expression .

Definition de la sémantique computationelle:

$e, e', e'' \in \exp = T_{¿¿}$
une relation comp $\subseteq \exp \times \exp$
On calcule la valeur final par une succession d'état
les règle qui sont nécéssaire sont :

$+¿_N$: *plus effective de naturel* $¿ \quad RC+: \dfrac{\square}{n+n' \to n+¿_N n'} \quad RC*: \dfrac{\square}{n*n' \to n ¿_N n'} ¿$

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

$\forall op \in \{+,-,*,/\}$
les cas non terminaux :

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

Example

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

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

je veut donner un exemple qui pourrait poser un problème de non déterministcàd. que on peut réduire soit gauche soit droit.
Exemple : (3 *4) + (4 / 2)

$$RCR\,\frac{4/2 \to 2,}{(3*4)+(4/2) \to} \quad RCL\,\frac{3*4 \to 12}{(3*4)+2 \to 12+2} \quad RC+\,\frac{\square}{\to 14}$$

Il y' a une autre vois d'évaluation qu'on fait d'abord RCL
Pour les expression plus complexe on a plusieurs façon de faire .
Le cas où les différentes façon pourrait ne se pas être égal -> g(y) + f(x) éventuelle effet de beurre si j'évalue f(x) peut avoir **effet de beurre** ou la même chose pour g(y)

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 computationelle : en plusieurs pas $e \Longrightarrow\Longrightarrow\Longrightarrow n$

pour comparer la sémantique d'évaluation et la sémantique de calcul on commence par observer la forme canonique , la forme canonique a une propriété particulière dans notre système de règle c'est qu'**elle est Confluence** càd si j'ai une forme j'obtient toujours une forme canonique(lors que il n'y a plus de réduction possibles : $\dfrac{e \to e', e' \to / e''}{e \rightsquigarrow e'}$) c'est le fait que j'atteint une valeur .
$\forall e \in \exp = T_{¿¿}$

$e \rightsquigarrow e' \Longleftrightarrow e' = n$
c'est évident car si e est une entier à ce moment là c'est une forme canonique liée à la règle **RBase** .par la contraposé si ce n'est pas un entier ce n'est obligatoirement une forme canonique .

2'ème théorème c'est que si je suis en train de calculé au moyen de notre sémantique computationelle c'est équivalent à le faire au moyen de sémantique évalutionnelle.

Quelque soit l'expression que je veut prendre il va exister un entier t.q si j'évalue e par la sémantique d'évaluation c'est la même chose que calculé la forme canonique.
$\forall e \in \exp = T_{¿¿}$
$e \Rightarrow n \Longleftrightarrow e \rightsquigarrow n$

La preuve est systématique . On preuve ça par induction sur les expréssions
Cas de base : = $P(e) = (\exists n, e \Rightarrow n \Longleftrightarrow e \rightsquigarrow n)$ c'est le prédiat que je doit prouver pour tous les e , on commence par constatant(cas de base) et on prend tous les expressions $¿, +, -¿$ .on suppose que c'est vrai pour e , e' et quand on les combien on réussi à prouver que c'est vrai.

Exemple : programmer un véhicule
Avec la sémantique d'évalution : j'ai une position <d,p> je prends un programme « prog » et finalement avec la sémantique d'évalution je me retrouve dans un nouvel position <d',p'>
$¿ d, p>, prog \Rightarrow < d', p' > ¿$

le problème avec ça c'est on sait pas les étapes intermédiaire
$¿ d, p>, mov \Rightarrow < d', p' >, prog \Rightarrow < d'', p'' > \dfrac{¿}{¿ d, p>, mov :: prog \Rightarrow < d'', p'' > ¿¿} ¿$
Pour savoir les étapes intermédiaires on change la règle, comme ça on voit l'effet de chaque mouvement .
$¿ d, p>, mov \Rightarrow < d', p' > \dfrac{¿}{¿ d, p>, mov :: prog \Rightarrow < d', p' >, prog} ¿$

Dans ce cas avec une position $¿ d, p>, prog\ p \rightarrow \rightarrow \rightarrow < d'', p'' >$,on voit le déplacement de véhicule pas à pas .