Thème 1
Techniques de Preuves

Objectifs

— Connaître quelques techniques de preuve et savoir les mettre en oeuvre.
— Savoir raisonner sur quelques objets “informatiques”.

1.1 Preuve par construction

De nombreux théorèmes stipulent qu'un objet particulier existe. Une technique pour prouver un tel théorème est de montrer comment construire cet objet. Cette technique est appelée preuve par construction.

EXERCICE 1.1 ▸ **Application : graphes**
Un graphe non orienté G est défini par un couple (S, A), S étant un ensemble de sommets, A un ensemble d'arêtes (c'est-à-dire un ensemble de parties de S à deux éléments) . On appelle le degré d'un sommet le nombre d'arêtes reliées à ce sommet. Un graphe est dit k-régulier si tous ses sommets sont de degré k. Prouver le théorème suivant :

THÉORÈME 1. *Pour tout entier n pair, $n > 2$, il existe un graphe 3-régulier composé de n sommets.*

1.2 Preuve par contradiction/l'absurde

Cette technique consiste à supposer que le théorème est faux, puis à montrer que cette hypothèse conduit systématiquement à une conséquence fausse, appelée contradiction.

EXERCICE 1.2 ▸ **Irrationalité de $\sqrt{2}$**
Montrer que $\sqrt{2}$ est irrationnel.

EXERCICE 1.3 ▸ **Relation binaire**
Soit R une relation binaire définie sur un ensemble fini A, et soient a et b deux éléments de A. S'il existe un chemin de a vers b dans R, alors il existe un chemin de longueur au plus $|A|$.

1.3 Preuve par récurrence, ou induction

Cette technique de preuve permet de montrer que tous les éléments d'une suite infinie (ou pas) vérifient une certaine propriété P. Elle comporte deux étapes :

— étape de base : on montre que $P(k)$ est vraie ;
— pas d'induction : pour $i \geq k$, on montre que si $P(i)$ est vraie, alors $P(i+1)$ est également vraie.

EXERCICE 1.4 ▸ **Maths**
Montrer que $n^4 - 4n^2$ est divisible par 3.

En informatique, les récurrences peuvent servir à prouver la correction de certains algorithmes, comme le montre l'exemple suivant :

EXERCICE 1.5 ▸ **Récurrence simple, correction d'un tri de tableaux**
On rappelle l'algorithme du tri-insertion :

```
PROCEDURE tri_selection(t)
D/R : t : Vecteur[N] d'Entiers
L : i, ideb, imin : Entiers
POUR ideb DE 0 A N-2 FAIRE
   imin <- ideb
```

```
   POUR i DE ideb + 1 A N-1 FAIRE
      SI t[i] < t[imin] ALORS
          imin <- i
      FSI
   FPOUR
   permuter(t, ideb, imin)
FPOUR
FPROCEDURE
```

On considère acquis la correction de la procédure `permuter`, c'est à dire que l'appel `permuter(t,i,j)` réalise l'échange entre les éléments d'indice `i` et `j` du tableau `t`.

1. Prouver la propriété suivante :

   P(`ideb`)="à la fin de la boucle (externe) d'indice `ideb`, le sous-tableau `t[0..ideb]` contient les `ideb+1` plus petits éléments du tableau initial et ce, dans l'ordre croissant."

2. Conclure.

L'induction structurelle reprend la technique précédente, en l'adaptant au cas des structures de données récursives.

EXERCICE 1.6 ▸ Induction structurelle, tri sélection sur les listes

On considère la définition des listes suivante : une liste est :

- soit vide;
- soit un élément suivi d'une liste.

Reprenons un exemple de LIF3 (Scheme) :

```
(define maximum ; -> entiers
  (lambda (myl) ; liste de nombres supposee non vide
    (if (null?  (cdr myl))
        (car myl)
        (max (car myl) (maximum (cdr myl))))))

(define supprime ; -> liste
  (lambda (e l) ; e atome, l liste
    (cond ((null? l) '())
          ((eq? e (car l)) (cdr l))
          (else (cons (car l) (supprime e (cdr l)))))))

(define tri-max ; -> liste de nb triee en ordre croissant
  (lambda (l) ; l liste de nb
    (if (or (null? l) (null? (cdr l)))
        l
        (let ((m (maximum l)))
          (append (tri-max (supprime m l)) (list m))))))
```

1. Comment peut-on raisonner par induction sur les listes ?
2. Montrer par induction structurelle sur `myl` que la fonction maximum retourne bien ce que l'on veut.
3. Que fait la fonction `supprime` ?
4. En supposant la fonction `supprime` correcte, montrer par induction structurelle la correction de `trimax`.

Thème 2
Machines de Turing

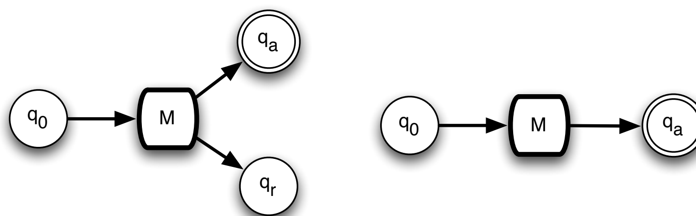
Objectifs

L'objectif de ce sujet est de faire appréhender par la construction “manuelle” la puissance d'expressivité du modèle des machines de Turing.

Compétences :
- Savoir réaliser des exécutions d'une machine de Turing donnée, et savoir caractériser le langage associé.
- Savoir construire des machines de Turing simples.
- Connaître le pouvoir d'expressivité de ces machines.

d'après un TD de B. Salvy et D. Monniaux, dont on reprend une partie du texte.

Conventions, sous-programmes. On s'efforcera de programmer les machines de Turing de façon *modulaire*[1], en réutilisant des machines déjà écrites. Pour cela, une machine sera symbolisée par l'un des deux schémas



selon que la transition vers l'état de refus (si cet état existe) est réutilisée pour pointer vers un autre état ou non.

L'utilisation de machines comme sous-programmes impose une spécification très précise de leurs droits. Ainsi, sauf indication contraire, les machines de Turing n'écriront jamais à gauche de la position initiale de leur tête de lecture, le mot donné en entrée sera toujours borné par un # à gauche et à droite, et lorsqu'une machine de Turing s'arrêtera sur son état d'acceptation, sa tête de lecture sera d'abord revenue à sa position initiale.

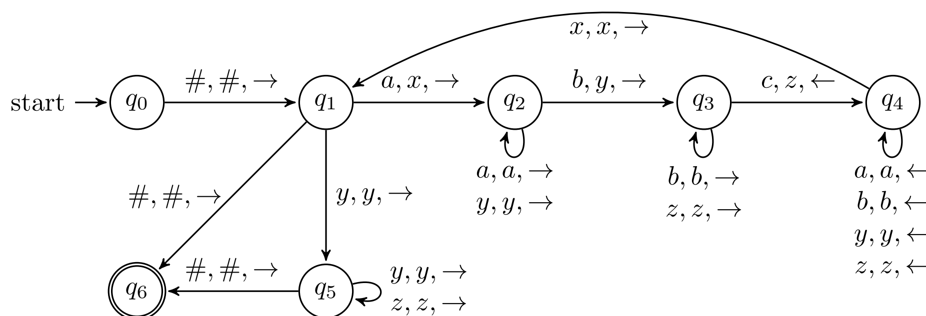
Alphabets Dans la suite, l'alphabet de travail Γ est constitué de l'alphabet Σ (les mots reconnus sont dans Σ^*) et du caractère blanc noté #.

Machines réutilisables Les machines F_x et B_x présentées ci-dessous sont utilisables par “copier coller”. On rappelle leur spécification dans l'exercice 2.2.1.

2.1 Reconnaissance de langages

EXERCICE 2.1 ▸ **MT mystère**

Quel langage reconnaît la machine de Turing (sur l'alphabet d'entrée $\{a, b, c\}$) ci-dessous ?



1. Attention, nous n'utiliserons pas la composition sous forme de branchements conditionnels évoquée en cours, mais uniquement la composition “copier-coller” décrite ici.

2.1.1 Langages rationnels

Les machines de Turing restreintes dont les transitions sont toutes de la forme $\delta(q_1, \ell) = (q_2, \ell, \rightarrow)$ sont classiquement appelées automates finis déterministes. Elles reconnaissent les langages rationnels, dont l'exercice suivant donne un exemple typique.

EXERCICE 2.2 ► **Sous-mot et co**
Écrire une machine de Turing acceptant les mots sur l'alphabet $\Sigma = \{a, b\}$ qui contiennent le sous-mot *aab* et se terminent par un *b*, et refusant[2] tous les autres mots sur Σ. Les langages rationnels servent à exprimer les "expressions régulières" qui sont utilisés dans de nombreux éditeurs de texte pour effectuer de la recherche avancée. Ne pouvant ni écrire, ni revenir en arrière, ces automates sont incapables de reconnaître des langages dont les mots ont une structure de dépendance à longue portée, mais permettent des recherches très rapides et avec peu de mémoire dans de très grand textes comme le génome.

EXERCICE 2.3 ► **Dans le même genre**
Construire des machines de Turing reconnaissant les langages suivants :

— $L = a^* b a^* b$

— $L = \{w \in \{a, b, c\}^* \mid w \text{ contient } abc \text{ mais pas } bb\}$

2.1.2 Langages hors-contexte

Les langages de programmation ne sont pas des langages rationnels, mais sont souvent algébriques (ou "hors contexte", de l'anglais "context-free"), c'est-à-dire une généralisation des langages rationnels, qui permet de reconnaître des mots bien parenthésés (penser à des *begin...end* ou *if...fi*). Pour les reconnaître, il est indispensable de pouvoir mémoriser ce qu'on a déjà lu. On utilise pour cela une pile (par exemple on empile *begin* et lorsqu'on lit *end* en dépile *begin*), ou la fonctionnalité des machines de Turing à écrire sur le ruban.

EXERCICE 2.4 ► **Parenthèses**
Écrire une machine de Turing acceptant les mots bien parenthésés sur l'alphabet $\{a,), (\}$. Ces mots ont le même nombre de parenthèses fermantes que de parenthèses ouvrantes, et aucun préfixe ne possède plus de parenthèses fermantes que d'ouvrantes. (Indication : on peut effacer le mot initialement présent sur le ruban).

EXERCICE 2.5 ► **Dans le même genre**
Construire des machines de Turing reconnaissant les langages suivants :

— $L = a^n b^n$

— $L = a^{2^n}$

2.2 Machines de Turing qui calculent

2.2.1 Déplacements sur le ruban

La programmation de machines de Turing pousse à déplacer fréquemment la tête de lecture sur le ruban, par exemple jusqu'aux extrémités. Les deux sous-routines suivantes seront donc très utiles pour la suite.

EXERCICE 2.6 ► **Forward/Backward**
Écrire une machine de Turing F_x (pour *Forward*) qui avance la tête de lecture sur le ruban jusqu'à la première occurrence de la lettre $x \in \Gamma$[3] et recule alors sur le caractère précédent. Cette machine refuse les mots qui ne contiennent pas x. Écrire de même la fonction symétrique B_x (pour *Backward*) qui recule et termine sur le premier caractère suivant l'occurrence de x précédente.

EXERCICE 2.7 ► **Forward/Modif**
Écrire une machine de Turing $\mathsf{ModifF}_{a,b}$ qui transforme les a en b, les b en a, de gauche à droite, retourne à la position initiale et s'arrête.

2. Cetta machine doit donc terminer sur toutes les entrées. Chaque mot du langage doit mener vers un état acceptant, mettons q_a, et les mots qui ne sont pas dans le langage doivent mener à un état de refus, mettons, q_r.

3. Oui, $x = \#$ est possible

2.2.2 Copies et effacements

Dans les exercices suivants, on fera bien attention aux conventions. Pour les machines *delete* et *insert*, le début du mot résultat se trouvera impérativement au même endroit sur le ruban que le mot de l'entrée.

EXERCICE 2.8 ▸ **Delete**
Écrire deux machines D et D' (pour *delete*) qui prennent en entrée un mot ℓw où $\ell \in \Sigma = \{a, b\}$ et w est un mot sur Σ, et s'arrêtent avec le mot w sur leur ruban. Il faut donc décaler **vers la gauche** d'une lettre toutes les lettres de w. La machine D termine avec sa tête de lecture sur le premier caractère de w. La machine D' termine avec sa tête de lecture sur le premier caractère suivant w, mais ne devra pas supposer que le premier caractère à gauche de ℓ est un blanc (cela sera utile pour des utilisations comme sous-programme plus loin).

EXERCICE 2.9 ▸ **Delete until**
Écrire une machine D_ℓ qui prend en entrée un mot sur Σ et en efface tous les caractères jusqu'au premier $\ell \in \Sigma$ inclus. Là encore la sortie devra se trouver au même endroit que l'entrée.

EXERCICE 2.10 ▸ **Insert**
(même convention que l'exercice précédent) Écrire trois machines I_ℓ, I'_ℓ et I''_ℓ (pour *insert*) qui prennent en entrée un mot w sur l'alphabet $\{a, b\}$, et s'arrêtent avec respectivement les mots ℓw, $w\ell$, $w\ell$ sur leurs rubans. La différence entre I'_ℓ et I''_ℓ est que cette dernière laisse sa tête de lecture sur le caractère suivant $w\ell$.

EXERCICE 2.11 ▸ **Copy**
Écrire une machine Copy_ℓ^m qui prend un mot de la forme $w_1 \ell w_2$ où w_1 et w_2 sont des mots sur $\Sigma = \{a, b\}$, ℓ et m sont des lettres différentes de **B** dans $\Gamma \setminus \Sigma$, et s'arrête avec le mot $w_1 \ell w_2 m w_1$ sur son ruban.

Dans la suite on s'autorisera d'employer les noms de ces machines pour insérer ou supprimer des lettres, même lorsque les alphabets seront différents de $\{a, b\}$. Ces noms représenteront alors les machines construites selon les mêmes principes sur ces alphabets.

2.2.3 Un peu d'arithmétique

Jusqu'ici les calculs se sont limités à des manipulations élémentaires sur des mots. Cependant, en réutilisant les machines précédentes, et en codant le k-uplet d'entiers $(n_1, \ldots, n_k)$ par le mot $a^{n_1+1} b a^{n_2+1} b \cdots a^{n_k+1}$, il est également possible de leur faire calculer beaucoup d'opérations arithmétiques. (On aurait pu aussi prendre des 0 et des 1, mais il sera plus facile de réutiliser les machines précédentes sur cet alphabet $\{a, b\}$.)

EXERCICE 2.12 ▸ **Zéro**
Écrire une machine de Turing prenant en entrée le codage d'un entier et s'arrêtant avec le codage de 0 (On dira qu'elle *calcule* 0).

EXERCICE 2.13 ▸ **Successeur unaire**
Écrire une machine de Turing qui calcule la fonction prenant en entrée un entier n et renvoyant $n + 1$.

EXERCICE 2.14 ▸ **Addition unaire**
Écrire une machine de Turing qui calcule la fonction prenant en entrée deux entiers n et m et renvoyant $n + m$.

EXERCICE 2.15 ▸ **Soustraction unaire (CC 2015)**
Soit f la fonction $\mathbb{N}^+ \times \mathbb{N}^+ \to \mathbb{N}$ telle que $f(n,m) = \begin{cases} n - m & \text{si } n \geq m \\ 0 & \text{sinon.} \end{cases}$.

1. Concevez une machine de Turing qui calcule cette fonction f, en unaire (en unaire, 5 est représenté par $IIIII$ et le chiffre 0 par le mot vide).
   Par exemple, si $n = 5$ et $m = 3$, $(q_0, \underline{\#}IIIII\#III) \models^* (q_f, \underline{\#}II\#)$.
2. (bonus) Modifiez votre machine pour étendre la fonction aux cas $n = 0$ et/ou $m = 0$.

2.2.4 Calcul binaire

EXERCICE 2.16 ▸ **Successeur binaire**
Écrire une machine de Turing qui calcule la fonction prenant en entrée un entier n (en base 2) et renvoyant $n+1$.

EXERCICE 2.17 ▸ **Multiplication par 2**
Écrire une machine de Turing qui calcule la fonction prenant en entrée un entier n (en base 2) et renvoyant $2n$.

EXERCICE 2.18 ▸ **Taille d'un mot**
Écrire une machine de Turing qui prend un mot $w \in \{0, 1\}^*$ et renvoie la longueur du mot écrite en binaire.

2.3 Puissance d'expressivité : l'exemple des carrés

Les machines de Turing peuvent calculer/reconnaître plus que les langages algébriques.

Un mot W est un *carré* s'il existe un mot w tel que $W = ww$. L'ensemble des carrés sur un alphabet Σ n'est pas un langage algébrique (donc non reconnaissable avec un automate à pile, ce qui se démontre avec le lemme de la double étoile). Cependant, il est possible de les reconnaître par une machine de Turing effectuant des aller-retours.

EXERCICE 2.19 ► **MT pour les carrés avec #**
Écrire une machine de Turing reconnaissant les mots de la forme $w\#w$ où w est un mot de l'alphabet $\Sigma = \{a, b\}$.

EXERCICE 2.20 ► **Découper un mot en deux**
Écrire une machine de Turing qui n'accepte que des mots de longueur paire, et laisse sur son ruban le mot qui lui a été donné en entrée avec un caractère $\# \not\in \Sigma$ inséré entre ses deux moitiés.

EXERCICE 2.21 ► **Finalement...**
Combiner ces deux machines pour écrire une machine reconnaissant les carrés.

EXERCICE 2.22 ► **Miroir**
Écrire une machine de Turing qui reconnaît les mots de la forme $w\widetilde{w}$, où $\widetilde{w}$ est le mot miroir de w.

EXERCICE 2.23 ► **Calcul du miroir (CC2016)**
Construisez une machine de Turing calculant la fonction $f : \{a, b\}^* \to a, b^*$ qui, à tout mot $w \in \{a, b\}^*$ de longueur paire, associe le mot miroir w^R.

Si, par exemple en entrée le ruban contient $\underline{\#}aababb\#$, le ruban de sortie sera $\underline{\#}bbabaa$.

Indication : un algorithme possible est de permuter le premier et le dernier symbole, puis le second et l'avant dernier ... jusqu'à ce que toutes les permutations soient faites.

2.4 Sources

Les auteurs, et aussi `http://www.enseignement.polytechnique.fr/informatique/INF412/`.

Thème 3
Fonctions numériques

Objectifs

Les fonctions primitives récursives ont été introduites par Gödel dans son travail sur l'incomplétude. Elles permettent de décrire des fonctions dont il est clair que le calcul termine toujours. Elles correspondent ainsi aux fonctions qui peuvent être calculées sans l'instruction "while" dans un langage typé comme Pascal, c'est-à-dire avec des "if-then-else" et des "for i :=1 to n". L'objectif de ce TD est de se convaincre de cette expressivité et d'observer une partie de ses limitations.

3.1 Fonctions primitives récursives

Informellement, ces fonctions sont celles que l'on peut définir sur l'ensemble $\mathbb{N}$ des entiers naturels par récurrence. L'ensemble de ces fonctions est défini par induction.

Définition. Une fonction $f : \mathbb{N}^n \to \mathbb{N}$ (avec $n \geq 0$) est *primitive récursive* si elle est :

- Une des fonctions renvoyant 0 : $\mathsf{zero_n} : (x_1 \dots x_n) \mapsto 0$. La fonction constante $zero_0 : () \mapsto 0$ sera par abus de notation souvent notée 0 ;
- $\mathsf{succ} : x \mapsto x+1$ la fonction successeur (alors $n = 1$) ;
- $\mathsf{id}_n^i : (x_1, \dots, x_n) \mapsto x_i$ les fonctions de projection, pour $1 \leq i \leq n$

et stable par les opérations de base :

- $\mathsf{Comp}_n(g, h_1, \dots, h_m) : (x_1, \dots, x_n) \mapsto g(h_1(x_1, \dots, x_n), \dots, h_m(x_1, \dots, x_n))$ la composition. Notons que n désigne ici l'arité de la fonction obtenue (le nombre d'arguments, qui peut être 0), et m désigne l'arité de la fonction g. Le cas $m = 0$ permet de construire des fonctions avec g d'arité 0 ;
- $\mathsf{Rec}(g, h)$ la fonction définie par récurrence (à droite) comme

$$\begin{cases} f(x_1, \dots, x_{n-1}, 0) = g(x_1, \dots, x_{n-1}), \\ f(x_1, \dots, x_{n-1}, m+1) = h(x_1, \dots, x_{n-1}, m, f(x_1, \dots, x_{n-1}, m)). \end{cases}$$

Ces règles simples permettent de définir des fonctions nouvelles. Par exemple, on peut définir une fonction Pred (prédécesseur) par la règle de récurrence :

$$\mathsf{Pred}(0) = 0, \quad \mathsf{Pred}(x+1) = x,$$

ce qui s'exprime plus formellement par $\mathsf{Rec}(0, h)$, avec $h : (x_1, x_2) \mapsto x_1$, ie $\mathsf{Pred} = \mathsf{Rec}(0, \mathsf{id}_2^1)$.

3.1.1 Quelques résultats simples

EXERCICE 3.1 ▸ **Addition**
Montrer que l'addition $+ : \mathbb{N}^2 \to \mathbb{N}$ est primitive récursive. Pour ce premier exemple, bien détailler les règles employées en explicitant les fonctions $g, h, h_1, \dots, h_m$ éventuellement en jeu.

EXERCICE 3.2 ▸ **Zero**
Montrer que l'on pourrait se restreindre à uniquement la fonction constante 0 d'arité 0 dans la définition des primitives récursives, c'est-à-dire que l'on peut construire $\mathsf{zero_n}$.

EXERCICE 3.3 ▸ **Constantes**
Montrer que les fonctions constantes $() \mapsto n$, $n \in \mathbf{N}$, sont primitives récursives

EXERCICE 3.4 ▸ **Swap**
Montrer que si $f : \mathbb{N}^2 \to \mathbb{N}$ est primitive récursive, alors $\mathsf{Swap(f)} = g$ définie par $g(x, y) = f(y, x)$ est aussi primitive récursive.

EXERCICE 3.5 ▸ **Fonctions arithmétiques**
Montrer directement à l'aide des définitions uniquement (zéro, id, succ, Comp et Rec), et éventuellement la fonction Swap, que les fonctions suivantes sont primitives récursives :

— pred(n) qui vaut 0 si $n = 0$ et $n - 1$ sinon.

— vabs(m, n) valeur absolue de la différence. On pourra utiliser la différence tronquée : difftronc$(m, n) = m - n$ si $m \geq n, 0$ sinon, que l'on montrera primitive récursive.

— eq(m, n) qui vaut 1 si $m = n$ et 0 sinon. On pourra définir un prédicat isZero, que l'on montrera primitif récursif.

EXERCICE 3.6 ▸ **Un prédicat primitif récursif**
Montrer que le prédicat leq2 : $x \mapsto \begin{cases} 1 & \text{si } x \leq 2 \\ 0 & \text{sinon.} \end{cases}$ est primitif récursif. On pourra utiliser le fait que IsZero est primitif récursif.

EXERCICE 3.7 ▸ **Un deuxième prédicat**
Montrer que gt2even : $x \mapsto \begin{cases} 1 & \text{si } 2 < x \text{ et } x \text{ pair} \\ 0 & \text{sinon.} \end{cases}$ est primitif récursif.

À partir de cette étape, on pourra écrire des définitions de fonctions primitives récursives par cas (si les cas sont des tests primitifs récursifs) et en utilisant des récursions qui font strictement décroître un certain paramètre (en montrant que la récursion termine).

EXERCICE 3.8 ▸ **Fonctions arithmétiques, un peu plus difficiles**
On ne demande pas de montrer que mod et pow sont primitives récursives. Montrer que la fonction sqr(m, n) racine n-ième de m, est primitive récursive. On commencera par définir la racine carrée comme fonction entière, on généralisera à la racine n-ième, avant toute tentative de résolution de l'exercice.

3.1.2 Boucles / Test de Primalité

L'exercice suivant montre qu'une certaine forme de "boucle for" est également possible.

EXERCICE 3.9 ▸ **For**
Montrer que si $g : \mathbf{N}^2 \to \mathbf{N}$ est primitive récursive, alors les fonctions $S_g : \mathbf{N}^2 \to \mathbf{N}$ et $P_g : \mathbf{N}^2 \to \mathbf{N}$ définies ci-dessous le sont aussi :

$$S_g : (z, y) \mapsto \sum_{x=0}^{z} g(x, y), \qquad P_g : (z, y) \mapsto \prod_{x=0}^{z} g(x, y).$$

EXERCICE 3.10 ▸ **Prédicats**
Montrer que si R est un prédicat primitif récursif (à deux variables), alors le sont aussi :

$$E_R : (z, y) \mapsto \exists x \leq z, R(x, y) \quad \text{et} \quad A_R : (z, y) \mapsto \forall x \leq z, R(x, y).$$

On construira judicieusement un prédicat g pour la question précédente, à partir de R.

EXERCICE 3.11 ▸ **Prime**
Montrer que le prédicat "x est un nombre premier" est primitif récursif.

EXERCICE 3.12 ▸ **Max de p éléments - CC 2015**
Soit $p \geq 1$. Montrer que la fonction max_p qui à $(x_1, \ldots x_p)$ associe le maximum de $x_1, \ldots x_p$ est primitive récursive. *On pourra utiliser* pred, *la fonction* difftronc$(x, y) \mapsto \max(0, x - y)$ *supposée primitive récursive, et une définition par cas* ite$(x, y, b) = if\,(b > 0)$ *then x else y supposée aussi primitive récursive.*

EXERCICE 3.13 ▸ **Une fonction définie par cas - CC 2016**
Montrer que $f : x \mapsto \begin{cases} \frac{x}{2} & \text{si } x \text{ pair} \\ \frac{x+1}{2} & \text{sinon.} \end{cases}$ est primitive récursive.

On pourra encore utiliser la fonction ite *supposée primitive récursive pour définir un prédicat* pair *qui sera ensuite utilisé pour définir la fonction* f.

EXERCICE 3.14 ▸ **Définition par cas**
Montrer que si f et g sont des fonctions primitives récursives et φ_1 et φ_2 des prédicats primitifs récursifs, alors la fonction $\mathsf{ite3}_{f,g,\varphi_1,\varphi_2}$ définie par :

$$\mathsf{ite3}_{f,g,\varphi_1,\varphi_2} : (x,y) \mapsto \begin{cases} f(x) & \text{si } \varphi_1(x) \\ g(x) & \text{sinon si } \varphi_2(x) \\ y & \text{sinon.} \end{cases}$$

est récursive primitive. +, *x, not sont supposés récursifs primitifs.*

EXERCICE 3.15 ▸ **Une deuxième définition par cas**

En appliquant l'exercice précédent, montrer que $\mathsf{tagada} : x \mapsto \begin{cases} 42 & \text{si } x \leq 2 \\ x+2 & \text{si } x > 2 \text{ et } x \text{ pair} \\ \big(\mathsf{tagada}(x-1)\big)^2 & \text{sinon.} \end{cases}$ est primitive récursive.

On pourra considérer que les fonctions constantes sont primitives récursives sans le redémontrer.

3.1.3 Exploration bornée

Soit $R(y, x_1, \ldots, x_n)$ un prédicat. On définit un opérateur Min de "minimisation bornée" par :

$$\operatorname*{Min}_{y=0}^{x} R(y, x_1, \ldots, x_n) : (x, x_1, \ldots, x_n) \mapsto \begin{cases} \text{le plus petit } y \leq x \text{ tel que } R(y, x_1, \ldots, x_n)\text{, s'il en existe un} \\ x \text{ sinon.} \end{cases}$$

EXERCICE 3.16 ▸ **Min Bornée**
Montrer que si R est primitive récursive, alors $\operatorname{Min}_{y=0}^{x} R(y, x_1, \ldots, x_n)$ aussi. Cette opération permet de définir des fonctions de manière implicite par exploration bornée. En voici deux exemples.

EXERCICE 3.17 ▸ **Nombre premier**
En utilisant le résultat précédent, montrer que la fonction $n \mapsto n$ième nombre premier est primitive récursive.

EXERCICE 3.18 ▸ **Cantor**
La fonction de Cantor $\langle x, y\rangle : (x, y) \mapsto ((x+y)^2 + 3x + y)/2$ envoie bijectivement $\mathbb{N}^2$ sur $\mathbb{N}$. Montrer que $\langle x, y\rangle$ est primitive récursive, ainsi que les deux fonctions K et L telles que $K(\langle x, y\rangle) = x$ et $L(\langle x, y\rangle) = y$.

EXERCICE 3.19 ▸ **Fibonacci**
En utilisant l'exercice précédent, montrer que la suite de Fibonacci, définie par récurrence, est primitive récursive :

$$f(0) = 1, \quad f(1) = 1, \quad f(n+2) = f(n+1) + f(n).$$

3.1.4 La fonction d'Ackermann n'est pas primitive récursive

Pour montrer qu'une fonction n'est pas primitive récursive, il ne suffit pas de disposer d'une définition de cette fonction qui viole une des contraintes.

EXERCICE 3.20 ▸ **Minimum**
La fonction min peut être définie par

$$\min(0, y) = 0, \quad \min(x+1, 0) = 0, \quad \min(x+1, y+1) = \min(x, y) + 1.$$

qui n'est pas une définition de fonction primitive récursive. Donner une preuve que min est quand même primitive récursive.

La fonction d'Ackermann est un des premiers exemples historiques de fonction "calculable" qui ne soit pas primitive récursive. Les exercices qui suivent prouvent ce résultat.
Définition. Soit A_n la suite de fonctions définie par

$$A_0(m) = m+1, \quad A_{n+1}(0) = A_n(1), \quad A_{n+1}(m+1) = A_n(A_{n+1}(m)).$$

La *fonction d'Ackermann* est définie [1] par $A(n, m) = A_n(m)$.

1. Il s'agit de la définition moderne de cette fonction. Ackermann avait défini une variante assez semblable, mais à trois arguments. Sa construction a été simplifiée plus tard.

EXERCICE 3.21 ▸ **Suite de fonctions**
Montrer que pour n fixé, la fonction A_n est primitive récursive. Comme on l'a vu avec min, il faut une idée supplémentaire pour prouver qu'une fonction n'est pas primitive récursive. La construction d'Ackermann vise à produire une fonction à croissance "trop rapide" pour être primitive récursive. Par exemple, pour les premières valeurs de l'indice, on observe facilement par récurrence que

$$A_1(m) = m+2, \quad A_2(m) = 2m+3, \quad A_3(m) = 8\cdot 2^m - 3, \quad A_4(m) = 2^{2^{\cdot^{\cdot^{2}}}} - 3.$$

L'exercice suivant mesure cette croissance ; dans un premier temps, elle peut être sautée et son résultat admis.

EXERCICE 3.22 ▸ **Ackermann**
Montrer que pour tous n, m, $A_n(m) > m$, que les fonctions A_n sont strictement croissantes, et que pour tous n, m, $A_n(m) \leq A_{n+1}(m)$. Enfin, montrer que $A_k(A_m(n)) \leq A_{2+\max(k,m)}(n)$. Le point clé est le suivant.

EXERCICE 3.23 ▸ **Résultat sur les fonctions primrec**
Pour toute fonction primitive récursive $f(x_1, \ldots, x_k)$, montrer qu'il existe n tel que

$$\forall (x_1, \ldots, x_k), \quad f(x_1, \ldots, x_k) \leq A_n(x_1 + \cdots + x_k).$$

EXERCICE 3.24 ▸ **Conclusion**
Conclure que la fonction d'Ackermann n'est pas primitive récursive.

3.1.5 Des machines de Turing pour les fonctions primitives récursives

L'objectif de cette dernière section est de montrer que les machines de Turing ont une expressivité suffisante pour calculer (toutes) les fonctions primitives récursives.

EXERCICE 3.25 ▸ **Projection**
Écrire une machine de Turing Proj_i qui calcule la fonction prenant en entrée un k-uplet d'entiers et renvoyant le i^{e}, avec $1 \leq i \leq k$.

EXERCICE 3.26 ▸ **Composition**
À partir de k machines $\mathsf{G}_1, \ldots, \mathsf{G}_k$ calculant des fonctions $g_1, \ldots, g_k$ de $\mathbb{N}^n$ dans $\mathbb{N}$ et d'une machine F calculant une fonction f de $\mathbb{N}^k$ dans $\mathbb{N}$, construire une machine calculant l'application :

$$(x_1, \ldots, x_n) \mapsto f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$$

EXERCICE 3.27 ▸ **Récursivité**
À partir d'une machine G calculant une fonction g de $\mathbb{N}^{n-1}$ dans $\mathbb{N}$ et d'une machine H calculant une fonction h de $\mathbb{N}^{n+1}$ dans $\mathbb{N}$, construire une machine calculant la fonction de $\mathbb{N}^n$ dans $\mathbb{N}$ définie récursivement par

$$f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n),$$
$$f(x_1+1, x_2, \ldots, x_n) = h(f(x_1, \ldots, x_n), x_1, \ldots, x_n).$$

Conclusion Malgré leur puissance d'expressivité, les fonctions primitives récursives ne capturent pas toute la richesse des fonctions que peut calculer une machine de Turing (fonctions *récursives*). Il leur manque peu : si on rajoute à leur définition l'opération de *minimisation non-bornée* (correspondant au "while"), définie comme la minimisation bornée en Section 3.1.3, sauf qu'on ne restreint pas à $y \leq x$, alors on obtient des fonctions partielles (il est possible que la condition ne soit jamais satisfaite) appelées *fonctions récursives partielles* ou fonctions μ-récursives.

Certaines de ces fonctions récursives partielles sont totales, définies donc pour toute valeur de leur argument, ce sont exactement les fonctions *récursives*, donc toutes les fonctions calculables par une machine de Turing.

3.2 Fonctions μ-récursives

Définition. Soit une fonction $g : \mathbb{N}^{n+1} \to \mathbb{N}$ (avec $n \geq 0$). La *minimisation non bornée*, ou plus simplement la *minimisation* de g est la fonction $f : \mathbb{N}^n \to \mathbb{N}$ définie par :

$$f(x_1, \ldots, x_n) = \begin{cases} \text{le plus petit } m \text{ tel que } g(x_1, \ldots, x_n, m) = 1, \text{ s'il en existe un,} \\ 0 \text{ sinon.} \end{cases}$$

La minimisation n'est pas toujours une fonction calculable. En effet, l'algorithme

```
m <- 0
tant que g(x1, ... ,xn, m) <> 1
  m <- m + 1
retrourner m
```

peut ne pas terminer.

On dit qu'une fonction g est *minimisable* si sa minimisation est calculable avec l'algorithme décrit ci-dessus, c'est-à-dire si $\forall x_1, \ldots, x_n \in \mathbb{N}, \exists m \in \mathbb{N}$ tel que $g(x_1, \ldots, x_n, m) = 1$.

On note alors $\mu m[g(x_1, \ldots, x_n, m)] = f(x_1, \ldots, x_n)$.

Définition. Les fonctions $\mu\text{-}récursives$ sont les fonctions obtenues à partir

— des fonctions primitives récursives,

— de l'opérateur de minimisation pour les fonctions minimisables.

EXERCICE 3.28 ▸ **Log**
Montrer que la fonction logarithme est μ-récursive.

EXERCICE 3.29 ▸ **Quotient, modulo**
Utilisez l'opérateur de minimisation pour définir les fonctions :

— $\mathsf{quo}(m, n) = \begin{cases} \lfloor \frac{m}{n} \rfloor & \text{si } n > 0 \\ 0 & \text{sinon.} \end{cases}$;

— $\mathsf{mod}(m, n) = \begin{cases} m - n.\lfloor \frac{m}{n} \rfloor & \text{si } n > 0 \\ 0 & \text{sinon.} \end{cases}$.

EXERCICE 3.30 ▸ **Premiers**
On suppose connu le prédicat (supposé μ-récursif) $\mathsf{prem}(n)$ qui vaut 1 si n est premier, 0 sinon. Montrer que la fonction $\mathsf{nprem}(n)$ qui renvoie le n-ième nombre premier est une fonction μ-récursive.

EXERCICE 3.31 ▸ **Codage**
L'objectif de cet exercice est de décider des langages avec des prédicats (récursifs primitifs ou μ-récursifs). Tout d'abord, nous allons coder les mots dans des entiers, ensuite la notion de reconnaissance sera définie à l'aide d'une fonction qui retourne 1 si le codage du mot en entrée appartient au langage, et 0 sinon.

— Soit un alphabet Σ. Comment coder les mots de Σ^* ? le mot vide ?

— Ecrire des prédicats primitifs récursifs décidant les langages suivants :

  — $L_1 = \{w | w \text{ contient un nombre impair de } a\}$

  — $L_2 = a^*$.

  — $L_3 = (a + b)^* ab(a + b)*$.

EXERCICE 3.32 ▸ **Minimisation non-bornée**
Nous avons vu que les machines de Turing ont au moins l'expressivité des fonctions primitives récursives.

Nous venons de voir que ce qui manquait aux fonctions primitives récursives pour avoir toute l'expressivité des machines de Turing était la minimisation non-bornée (correspondant au 'while' des langages de programmation). C'est par contre une opération facile à implanter sur une machine de Turing.

À partir d'une machine G calculant une fonction $g : \mathbb{N}^{n+1} \to \{0, 1\}$, construire une machine, qui prend en entrée le codage de $(x_1, \ldots, x_n)$ et s'arrête avec sur son ruban le codage du plus petit $m \in \mathbb{N}$ tel que $g(, x_1, \ldots, x_n, m) = 1$ s'il en existe un, et boucle sinon.

Thème 4
Indécidabilité

Objectifs

- Savoir prouver qu'un problème est indécidable.
- Découvrir de nouveaux modèles de calculs de puissance équivalente aux machines de Turing.

Rappel Pour montrer qu'un problème est indécidable, soit on montre de manière directe (cf. cours), soit on effectue une réduction, dont on rappelle la définition :

DÉFINITION 1. Soient L_1 et L_2 deux langages de Σ^*. Une *réduction* de L_1 à L_2 est une fonction récursive (*i.e.* calculable par machine de Turing) $\rho : \Sigma^* \to \Sigma^*$ telle que :

$$w \in L_1 \text{ ssi } \rho(w) \in L_2.$$

PROPOSITION 1. *Pour montrer qu'un langage L n'est pas récursif (i.e est indécidable), il suffit d'exhiber un langage L' non récursif tel que L' se réduit à L.*

En général, au lieu d'exhiber une machine de Turing qui réalise la réduction, on se contente de fournir un algorithme dans un langage "raisonnable". Ceci sera justifié par la section 4.3.

4.1 Trois problèmes indécidables

Le Xième problème de Hilbert peut s'énoncer de la façon suivante :

HILBERTX Soit p un polynôme à n variables à coefficients dans $\mathbb{Z}$, existe-t-il une racine entière, c'est à dire $x_1, \ldots x_n \in \mathbb{Z}^n$ tels que $p(x_1, \ldots, x_n) = 0$?

Il a été montré indécidable par Matiiassevitch en 1970.

EXERCICE 4.1 ▶ **Xth Hilbert**
Montrer par réduction à partir du Xième problème de Hilbert que le problème de satisfiabilité sur $\mathbb{R}$ des formules construites sur $\{0, 1, +, *, \sin\}$ est indécidable.

On rappelle $LU = \{< M, w > | M \text{ accepte } w\}$. Ce problème a été montré indécidable dans le cours.

EXERCICE 4.2 ▶ **Machines de Turing**
Soient M_1 et M_2 deux machines de Turing. Montrer que le problème $L(M_1) =?L(M_2)$. est indécidable.

EXERCICE 4.3 ▶ **Miroir**
Soit L un langage accepté par une machine de Turing et soit L^R le langage constitué des mots de L vus dans un miroir. Par exemple, si $L = \{a, ab, abb, abbb\}$, alors $L^R = \{a, ba, bba, bbba\}$. Montrer que le problème "est-ce que $L = L^R$?" est indécidable.

4.2 Indécidabilité du Problème de Correspondance de Post (PCP)

Soit Σ un alphabet fini [1], et $P \subseteq \Sigma^\star \times \Sigma^\star$ un ensemble fini de dominos étiquetés par des mots sur l'alphabet Σ (des paires de mots, donc). Le Problème de Correspondance de Post (PCP), introduit par Emil Post en 1946, consiste à déterminer s'il existe une séquence de dominos de P tels que le mot obtenu par la concaténation des premières composantes est identique à celui formé par la concaténation des secondes composantes. Plus formellement, on cherche à déterminer l'existence d'une suite $(u_i, v_i)_{0 \leqslant i \leqslant n}$ telle que : $\forall 0 \leqslant i \leqslant n, (u_i, v_i) \in P$ et $u_0 \cdot u_1 \cdots u_n = v_0 \cdot v_1 \cdots v_n$.

EXERCICE 4.4 ▶ **PCP**

1. *Rédaction par P. Brunet pour l'examen de MIF15 de déc. 2014*

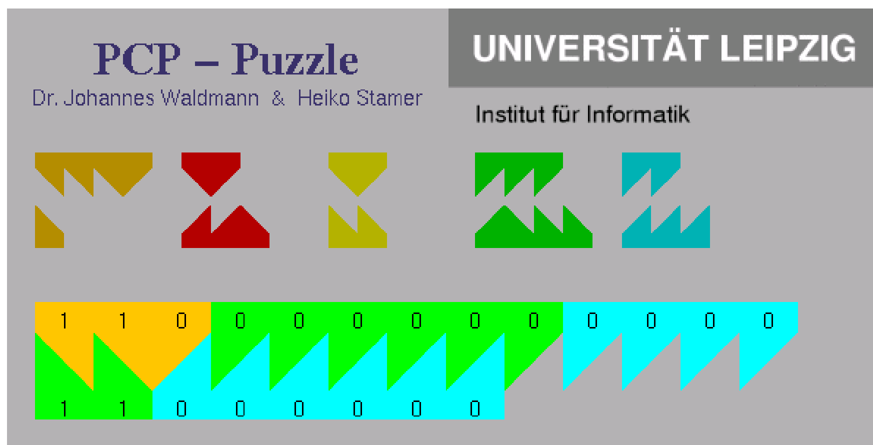


FIGURE 4.1 – PCP, version graphique, `http://freecode.com/projects/pcp-puzzle`

1. Résoudre PCP **“à la main”** pour les instances suivantes. *Si une correspondance existe, on la donnera explicitement. Sinon, on justifiera soigneusement qu'une telle correspondance n'existe pas.*
   (a) $P_0 = \{(a, aaa), (aaaa, a)\}$;
   (b) $P_1 = \{(aab, ab), (bab, ba), (aab, abab)\}$;
   (c) $P_2 = \{(a, ab), (ba, aba), (b, aba), (bba, b)\}$;
   (d) $P_3 = \{(ab, bb), (aa, ba), (ab, abb), (bb, bab)\}$.
2. Donner un algorithme pour décider PCP dans le cas où Σ ne contient qu'une seule lettre.

On considère maintenant une variante de PCP, le Problème de Correspondance de Post Modifié (PCPM). Pour ce problème, on considère un ensemble fini de dominos P et un domino initial $(u_0, v_0) \in P$, et on cherche à déterminer l'existence d'une correspondance qui commence par (u_0, v_0).

On considère comme acquis que PCPM est indécidable : en effet il existe une réduction du problème de l'arrêt d'une machine de Turing vers PCPM. On va maintenant montrer que PCP est indécidable, en réduisant depuis PCPM.

EXERCICE 4.5 ▸ **Réduction**
On commence par introduire deux fonctions p et s. Soit \$ un nouveau symbole n'appartenant pas à Σ, pour tout mot $w = a_1 a_2 \cdots a_n$ (les a_i sont les lettres), on définit :

$$\begin{cases} p(w) = \$a_1\$a_2\cdots\$a_n \\ s(w) = a_1\$a_2\$\cdots a_n\$ \end{cases}$$

1. Montrer que les fonctions p et s vérifient les propriétés suivantes :
   — pour deux mots v et w, $p(vw) = p(v)p(w)$ et $s(vw) = s(v)s(w)$;
   — pour tout mot w, $p(w)\$ = \$s(w)$.

   Soit $P, (u_0, v_0)$ une instance une PCPM, on définit $P' = \{(p(u_0), \$s(v_0))\} \cup P_1 \cup P_2$ avec :

$$\begin{cases} P_1 = \{(p(u), s(v)) \mid (u, v) \in P\} \\ P_2 = \{(p(u)\$, s(v)) \mid (u, v) \in P\} \end{cases}$$

2. Montrer que $P, (u_0, v_0)$ admet une solution pour PCPM si et seulement si P' admet une solution pour PCP.
3. En déduire que PCP est indécidable.

4.3 Machines alternatives

4.3.1 Machines à piles

Une machine à k piles possède un nombre fini k de piles $r_1, \ldots r_k$ qui correspondent à des piles d'éléments de Σ. Les instructions d'une machine à piles permettent seulement d'empiler un symbole sur l'une des piles, tester

la valeur du sommet d'une pile, ou dépiler le symbole au sommet d'une pile. Si l'on préfère, on peut voir une pile d'éléments de Σ comme un mot w sur l'alphabet Σ. Empiler (*push*) le symbole a correspond à remplacer w par aw. Tester la valeur du sommet d'une pile (*top*) correspond à tester la première lettre du mot w. Dépiler (*pop*) le symbole au sommet de la pile correspond à supprimer la première lettre de w.

EXERCICE 4.6 ▶ **Simulation**
Montrer que tout machine de Turing peut être simulée par une machine à 2 piles. En déduire un problème indécidable pour les machines à 2 piles.

4.3.2 Machines à compteur

Une machine à n compteurs a plusieurs registres $r_1, \dots r_n$, appelés *compteurs*, chacun capable de stocker un nombre naturel. Un programme est une suite d'instructions de la forme :

1. `q:x++;goto p`
2. `q:x-;goto p`
3. `q:if x=0 then goto p else goto r`
4. `q:stop`

Calculer `0-` produit une erreur. Une configuration de la machine à n compteurs est le $n+1$-uplet $(q, x_1, \dots, x_n)$ qui représente l'étiquette de l'instruction courante et la valeur actuelle des compteurs. Pour l'entrée x, la configuration initiale est $(init, x, 0, \dots, 0)$, ou "init" est une instruction particulière, et qu'elle calcule jusqu'à *stop*. Le résultat est le contenu du registre x_1 après l'arrêt.

EXERCICE 4.7 ▶ **Calculons**
Montrer comment calculer $x \mapsto 2x$, et $x \mapsto x \mod 2$ avec une telle machine.

EXERCICE 4.8 ▶ **Indécidabilité de l'arrêt des machines à compteur**
Prouver que le problème de l'arrêt pour les machines à 4 compteurs est indécidable. On utilisera les étapes suivantes :

— Comment représenter un contenu de pile par un entier ?
— Comment réaliser les opérations Empiler et Depiler ?
— Comment finalement simuler le fonctionnement d'une machine à deux piles par une machine à 3 compteurs ?

Ce résultat est encore valable pour une machine à deux compteurs, via un encodage adhoc. En conséquence, les problèmes d'arrêt, calcul d'invariant, d'accessibilité d'un état mauvais, … sont indécidables pour les automates à deux compteurs. Ce modèle de calcul plus pratique que les Machines de Turing est beaucoup utilisé en analyse de programme.

4.4 Sources

Les auteurs, et aussi

— `http://www.liafa.univ-paris-diderot.fr/~asarin/calc2k3/calcul_cours.pdf`
— `http://www.enseignement.polytechnique.fr/informatique/INF412/uploads/Main/chap7-good.pdf`
— `http://www-verimag.imag.fr/~perin/enseignement/L3/mcal/cours/MCAL-MT.pdf`

Thème 5

Langages P, langages NP, NP-complétude

Objectifs

- Connaître la notion de complexité de problème, les classes P et NP.
- Savoir utiliser les réductions polynomiales.

5.1 Retour aux machines de Turing

EXERCICE 5.1 ▸ **Shift**
Écrire une machine de Turing qui “shifte” son entrée d'une case vers la droite, et évaluer sa complexité. Quelle est la complexité du problème ?

5.2 Quelques problèmes P

Soient les deux problèmes suivants :

- 2-SAT : Soit une forme normale conjonctive F dans laquelle chaque clause a exactement 2 littéraux. F est-elle satisfaisable ?
- 2-COLOR : Soit un graphe G = (V, E). Existe-t-il une fonction c : $V \to \{0,1\}$ telle que pour toute arête (u, v) de G, $c(u) \neq c(v)$?

EXERCICE 5.2 ▸ **2-SAT dans P**
Prouver directement que 2-SAT est dans P. On prendra comme exemple la formule

$$\phi = (x_1 \lor x_2) \land (x_3 \lor \neg x_4) \land (x_1 \lor x_4) \land (x_1 \lor x_3) \land (\neg x_2 \lor x_4)$$

et on essaiera de construire un graphe.

EXERCICE 5.3 ▸ **2-COLOR dans P**
Prouver directement que 2-COLOR est dans P.

EXERCICE 5.4 ▸ **Complémentation**
Montrer que la classe P est close par complémentation.

5.3 Appartenance à P via réduction polynomiale

On rappelle la définition d'une réduction polynomiale :

DÉFINITION 2. Soient L_1 et L_2 deux langages de Σ^*. Une réduction polynomiale de L_1 *vers* L_2 est une fonction $f : \Sigma^* \to \Sigma^*$ *calculable en temps polynomial* telle que $x \in L_1$ ssi $f(x) \in L_2$. On note alors $L_1 \leq^P L_2$.

Informellement, cela signifie que L_1 n'est pas plus difficile que L_2.
On a ensuite le résultat suivant

PROPOSITION 2. *Si $L_1 \leq^P L_2$ et $L_2 \in P$, alors $L_1 \in P$.*

EXERCICE 5.5 ▸ **2-COLOR dans P, par réduction**
Démontrer que le langage 2-COLOR se réduit polynomialement à 2-SAT. En déduire qu'il est dans P.

5.4 NP-Complétude

EXERCICE 5.6 ▸ **Questions de cours**
Soient P_1 et P_2 deux problèmes de décision, et supposons qu'on connaisse une transformation polynomiale (une réduction) de P_1 en P_2. Répondre aux sept questions suivantes avec un maximum de deux lignes de justification par question.

1. Si $P_1 \in \mathsf{P}$, a-t-on $P_2 \in \mathsf{P}$?
2. Si $P_2 \in \mathsf{P}$, a-t-on $P_1 \in \mathsf{P}$?
3. Si P_1 est NP-complet, P_2 est-il NP-complet ?
4. Si P_2 est NP-complet, P_1 est-il NP-complet ?
5. Si on connaît une transformation polynomiale de P_2 en P_1, P_1 et P_2 sont-ils NP-complets ?
6. Si P_1 et P_2 sont NP-complets, existe-t-il une transformation polynomiale de P_2 en P_1 ?
7. Si $P_1 \in \mathsf{NP}$, P_2 est-il NP-complet ?

5.5 Quelques petites réductions “faciles”

On rappelle la proposition qui permet de prouver qu'un problème donné est NP-complet :

PROPOSITION 3. *Si $L_1 \leq^P L_2$, $L2 \in NP$, et L_1 NP-complet, alors L_2 est NP-complet.*

Il s'agit donc de réduire polynomialement le problème connu vers notre problème à montrer NP-complet. Dans la suite, nous allons prouver NP-complets des variantes de “problèmes connus”.

EXERCICE 5.7 ▸ **3-NAE “not all equal”**

— *Instance :* Un ensemble de clauses $C_1, \ldots, C_m$, chacune contenant exactement 3 littéraux.
— *Question :* Existe-t-il une instantiation des variables telle que chaque clause contient un littéral évalué à vrai et un littéral évalué à faux?

Montrer que 3-NAE est NP-complet. *Indice : On pourra créer m+1 nouvelles variables, et construire une instance avec 2m clauses.* Pour l'exercice suivant, on rappelle la définition du problème 2-PARTITION, qui est NP-complet (on ne demande pas de preuve).

DÉFINITION 3 (2-PARTITION). *Instance* $S = \{a_1, \ldots, a_n\}$ des entiers
Question : Existe-t-il $I \subset S$ tel que $\sum_{i\in I} a_i = \sum_{i\notin I} a_i$?

EXERCICE 5.8 ▸ **Deux variantes de 2-PARTITION**
Montrer que les deux variantes suivantes de 2-PARTITION sont NP-complètes.

1. **2-Part-Even :**
Instance : Un ensemble V de $2n$ entiers strictement positifs $a_1, a_2, \ldots, a_{2n}$.
Question : Existe-t-il un sous-ensemble $I \subset [1..2n]$ tel que $\sum_{i\in I} a_i = \sum_{i\notin I} a_i$?
2. 2-PART-EQ :
Instance : Un ensemble V de $2n$ entiers strictement positifs $a_1, a_2, \ldots, a_{2n}$.
Question : Existe-t-il un sous-ensemble $I \subset [1..2n]$ tel que $\sum_{i\in I} a_i = \sum_{i\notin I} a_i$ et $|I| = n$.

5.6 Deux réductions plus compliquées

Dans les deux cas il s'agit de réduction à partir de 3-SAT.

EXERCICE 5.9 ▸ **SUBSET-SUM**
Montrer que le problème suivant SUBSET-SUM est NP-complet.

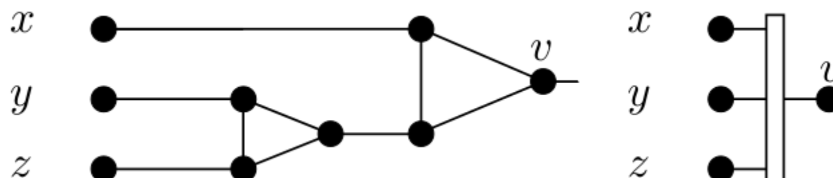
SUBSET-SUM
Instance : un ensemble fini S d'entiers positifs et un entier objectif t.
Question : existe-t-il un sous-ensemble $S' \subseteq S$ tel que $\sum_{x\in S'} x = t$?

Indication : vous pouvez par exemple effectuer une réduction à partir de 3-SAT. A partir d'un ensemble de clauses $C_0, ..., C_{m-1}$ sur les variables $x_0, ..., x_{n-1}$, considérer S l'ensemble des entiers $v_i = 10^{m+i} + \sum_{j=0}^{m-1} b_{ij} 10^j$

et $v'_i = 10^{m+i} + \sum_{j=0}^{m-1} b'_{ij} 10^j$, $0 \le i \le n-1$, où b_{ij} (resp. b'_{ij}) vaut 1 si le littéral x_i (resp. $\overline{x_i}$) apparaît dans C_j et 0 sinon, et des entiers $s_j = 10^j$ et $s'_j = 2.10^j$, $0 \le j \le m-1$. Trouver alors un entier objectif t tel qu'il existe un sous-ensemble $S' \subseteq S$ de somme t si et seulement si l'ensemble initial de clauses est satisfiable. Conclure. Quels autres entiers auraient aussi marché ?

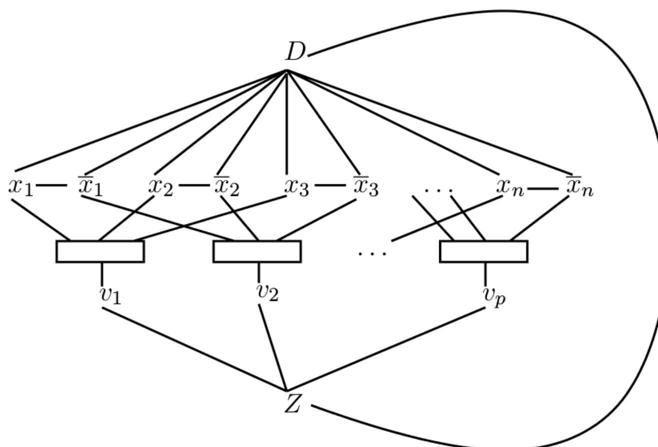
Exercice 5.10 ▸ **NP-complétude de 3-COLOR**

On va faire une réduction à partir de 3-SAT, montré NP-complet dans le cours.
Soit le gadget suivant (à droite son symbole dans la suite) :



1. Montrer que 3-COL est dans la classe NP.
2. Montrer que le gadget satisfait les 2 propriétés suivantes [1] :
   (a) Si $x = y = z = 0$ alors v est *nécessairement* colorié à 0.
   (b) Toute autre entrée (x, y, z) permet de colorier v à 1 ou 2 (au sens où l'on est capable d'exhiber un coloriage *tel que...*)

   Soit I une instance de 3-SAT avec p clauses C_k (numérotées de 1 à p) et n variables x_i. On construit le graphe G suivant :



3. Vérifier que ce graphe a une taille polynomiale en l'instance I considérée.
4. Prouver que I a une solution ssi G est 3 coloriable.

Exercice 5.11 ▸ **REGISTER-ALLOC**

Nous allons montrer que le problème suivant est NP-complet. **REGISTER-ALLOC**

Instance : un graphe de flot de contrôle (général) avec des temporaires, $k \ge 4$ registres.
Question : existe-t-il une allocation des variables aux registres sans conflit ?

On supposera dans la suite que les variables ne sont pas splittées et que la construction du graphe de conflit est polynomiale. On va montrer la NP-complétude par réduction à partir de la NP-complétude de $k-1$-COLOR.

1. Montrer que k-REGISTERALLOC est dans NP.

Soit un graphe $G = (V, E)$. Construisons le graphe de flot de contrôle sur les variables $V \cup \{x\}$ suivant :

— Pour chaque arête (u, v), on définit un bloc $B_{u,v}$ qui définit (au sens *def* du cours de compilation) les variables u et v (en les initialisant à une constante) puis x=u+v.

1. On pourra utiliser la notation suivante : $x, y \in \{0, 1, 2\}$ étant distincts, $\varphi(x, y)$ désigne le troisième élément de $\{0, 1, 2\}$.

— Pour chaque sommet u de V, on construit un bloc B_u qui lit u et x et retourne une nouvelle valeur déduite de u et x (par exemple `return u+x`)
— Chaque bloc $B_{u,v}$ est un prédécesseur direct de B_u et de B_v.
— Les blocs $B_{u,v}$ ont un unique prédécesseur commun vide appelé B_{switch}.

2. Construire le graphe de flot de contrôle produit par la construction précédente pour le graphe cyclique de taille 4 dont les arêtes sont (a, b), (b, c), (c, d) et (d, a).
3. Quel est le graphe d'interférence associé à ce graphe de flot ?
4. Si le graphe initial est k coloriable, que peut-on dire du graphe de conflit du programme après transformation ?
5. Finir la preuve.
6. (Bonus) Que se passe-t-il si on autorise le *split* de variables ? On pourra raisonner sur le graphe précédent en essayant de découper la durée de vie de la variable dans B_a avant que a soit utilisée.

Remarques Source de la preuve : `www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2006/RR2006-13.pdf`, dans ce rapport il est aussi montré les résultats intéressants suivants :
— Si on peut "splitter" les variables, le problème reste NP-complet.
— Si le graphe n'a pas d'arc critique et que l'on peut splitter les variables uniquement à des points fixés, le problème reste NP complet.
— Le problème reste NP complet même si l'on peut splitter les variables, en ajoutant l'hypothèse suivante : *seulement si il existe des instructions machine qui peuvent créer deux variables à la fois.*

5.7 Sources

Les auteurs, et aussi `http://www.di.ens.fr/~fouque/articles/poly-algo.pdf`