

Proposition de Langage Universel Simplifié

Génèse du Langage Universel Simplifié:

L'intelligence est à la programmation ce que la programmation est à l'électronique. Si on fait l'abstraction de notre familiarité avec les ordinateurs, il est au premier abord extrêmement surprenant et inattendu de pouvoir construire des circuits électroniques universels. Les circuits électroniques sont écrits avec un langage qui ne comporte qu'un seul mot, disons le transistor. On peut avec ce mot, fabriquer tous les circuits électroniques ayant des fonctions précises et contrôlées. Il est naturel de penser que, pour remplir une fonction donnée, il faudra construire un circuit électronique spécifique, ou au moins adapter physiquement un circuit électronique existant à cette fonctionnalité. Être en mesure de concevoir des circuits électroniques génériques (micro processeurs), capables d'exécuter des programmes pour faire un grand nombre de fonctions non spécifiques est une "chance" étonnante.

Si on transpose au niveau de complexité supérieur, nous en sommes actuellement pour la programmation au stade où l'on pense qu'il faut adapter les programmes à chaque fois que l'on veut leur faire remplir une nouvelle fonction. On sait faire des programmes semi génériques (système d'exploitation, tableurs, bases de données), on sait faire des programmes qui s'adaptent un peu (correcteur d'orthographe), mais on a pas encore découvert les programmes universels, qui peuvent servir dans toutes les situations. On peut prendre ce type de programme pour base de définition de l'intelligence artificielle. Il est évident qu'un tel programme échappe à un contrôle total du programmeur, ce que permettent les expériences de Stephen Wolfram.

J'ai poussé l'analogie avec les circuits électroniques et j'ai développé un langage informatique, inspiré du LISP, qui comporte une seule instruction. Je l'appelle Langage Universel. Je cherche encore un acronyme satisfaisant pour ce langage. Ce langage est plus primitif que le LISP, d'un niveau de complexité inférieur. En effet, le LISP est un langage qui utilise la récursivité. Ce langage, lui est purement séquentiel (au départ). Mais ce n'est pas qu'un langage informatique. C'est aussi un langage qui peut être utilisé pour définir avec une grande précision tous les concepts. Il intègre une nature, que n'a pas le langage mathématique par exemple, qui est la nature 'évolutive'. Comme tous les langages informatiques, ce langage est un programme mais il a la particularité de pouvoir se modifier entièrement pendant son exécution. Dans ce langage, l'interprétation d'une "phrase" dépend de ce qui s'est passé avant, de son histoire, son expérience.

Programmation en Langage Universel Simplifié Séquentiel Intelligent (PLUSSI):

Quel est le principe de ce langage? J'ai mis un certain temps à trouver l'instruction unique, persuadé qu'elle devait exister grâce à l'analogie avec les circuits électroniques. Cette instruction unique se note:

$X \rightarrow Y$ ou "écrire" X sur Y.

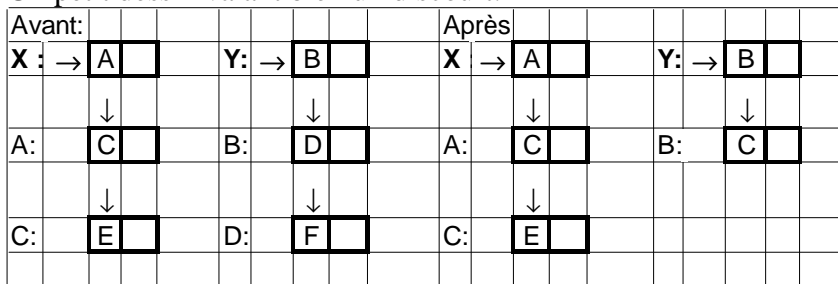
En effet, la seule chose que fait un ordinateur actuellement est de déplacer de l'information d'une adresse mémoire à une autre, et ce, quelle que soit la complexité des opérations effectuées!

Mais attention: elle s'interprète de manière indirecte, c'est à dire que ce n'est pas la valeur de X qui est copiée à l'adresse indiquée par Y, mais la valeur présente à l'adresse mémoire indiquée par X qui est copiée à l'adresse mémoire indiquée par la valeur présente à l'adresse mémoire indiquée par Y. (voir le schéma, c'est bien plus clair qu'un long discours). Donc si l'adresse mémoire indiquée par X contient A et l'adresse mémoire indiquée par Y contient B, on copie la valeur contenue à l'adresse mémoire A (ou valeur de A) à l'adresse mémoire B. Après exécution de l'instruction, on a effectivement $Valeur(B)=Valeur(A)$. En langage informatique 'usuel', cela revient à écrire:

$Mémoire(Mémoire(Y))=Mémoire(Mémoire(X))$

Si Mémoire est un tableau à une entrée, censé représenter la mémoire de l'ordinateur.

Un petit dessin valant bien un discours:



Est c’est tout ?

C’est effectivement la seule “instruction”. Difficile de s’en persuader autrement qu’en essayant de reproduire les fonctions classiques d’un ordinateur dans ce langage. Toutefois, pour fonctionner, ce langage a besoin d’un certain nombre d’éléments prédéfinis, qui sont des adresses mémoires spécifiques que je m’emploie à réduire au minimum. (au même titre qu’un ordinateur a besoin d’une séquence de démarrage hardware puis software très rigoureuse)

La structure physique de la “machine” minimale nécessaire pour interpréter ce langage correspond à une machine de John Von Neumann, avec une mémoire interne, une entrée, une sortie, et un moteur de traitement. Le langage repose sur une structure interne de la mémoire analogue à celle utilisée par le LISP (car et cdr) que j’appelle tête et suite: une adresse mémoire permet de ‘pointer’ 2 valeurs d’une taille correspondant à la taille d’une adresse mémoire. C’est le minimum pour pouvoir construire des architectures de données (de même que l’élément électronique de base a 2 entrées et 2 sorties, au moins), dont principalement des classiques listes chaînées.

Le langage est exécuté par un moteur, dit de niveau 0, qui est un programme de type automate, très basique, systématique, et qui ne sera pas modifié au cours de la vie du programme principal. Ce programme de base peut être écrit dans n’importe quel langage informatique ou créé en électronique pure, cela n’a pas d’importance.

Ce moteur exécute les fonctions suivantes:

Étendre la mémoire accessible de 2 adresses supplémentaires si nécessaire.

Mettre le caractère courant de l’entrée à l’adresse fixe de l’input, après avoir décalé l’input si nécessaire.

Exécuter une instruction “-->” de la liste d’exécution.

Réactualiser le contenu de l’adresse de “Suite” (si le pointeur de suite, Variable suite a été modifié)

Afficher le caractère courant de l’output si nécessaire.

Les fonctions de captures de caractère de l’input , d’affichage et de libération de mémoire sont considérées comme des activités de niveau -1, de complexité inférieure au moteur “circulaire” et répétitif de niveau 0. Elles peuvent exister à partir d’une électronique très usuelle.

Avant de pouvoir lancer le moteur, il faut lancer une séquence de démarrage, si la mémoire n’est pas déjà initialisée, comparable au BIOS d’un ordinateur. Cette séquence de démarrage, dite de niveau 1, et écrite dans un langage autre ou pré codée en mémoire, a pour but de ‘déclarer’ les adresses fixes correspondant à :

- la mémoire disponible, l’input, l’output, la Suite, la chaîne d’exécution

et de préparer le système à son interaction avec l’extérieur:

- créer la liste des objets connus, créer la première chaîne d’exécution qui lit dans un ordre fixe des chaînes de caractères successivement entrées et séparées par une notion prédéfinie de ‘fin’. Cette liste est dite de niveau 2 et permet de choisir les mots clés initiaux du langage (qui pourront être ‘oubliés’ ultérieurement). La séquence de démarrage doit aussi préparer la seconde version de la chaîne d’exécution par une série d’instructions fixes. Cette première chaîne d’exécution contiendra,

en langage universel, un interpréteur de langage universel primaire.

Performance du Langage Universel Simplifié:

Comme le LISP, ce langage est particulièrement lent et peu performant en terme de vitesse d'exécution. Mais ce n'est clairement pas le but recherché ici. Comme le LISP c'est un langage qui apprend, mais à la différence du LISP, il n'est pas supposé toujours repartir de zéro. Il faut continuer à rajouter des acquis supplémentaires et la séquence de démarrage ne doit être exécutée qu'une fois, à la "naissance". Par contre il s'écrit sans aucune parenthèses, purement en séquentiel (tant qu'on ne décide pas de changer l'interpréteur) en beaucoup plus simple et 'naturel'. La programmation obtenue, en fonction des mots choisis, se rapproche déjà beaucoup du langage naturel. Sa conception permet de TOUT modifier pendant l'exécution du programme, y compris le programme lui même et transgresse toutes les règles de "bonne programmation". Je cherche actuellement à condenser au maximum la séquence de démarrage pour "voir" la complexité se construire à partir d'éléments simples. En parallèle, j'apprends à utiliser ce langage qui est beaucoup plus subtile qu'il n'y paraît... Forcément puisqu'il permet de tout écrire avec une seule instruction... L'objectif de la construction de ce langage est bien de décortiquer "visuellement" le mécanisme de construction de la complexité, de manière à le comprendre et à l'extrapoler à l'étage suivant, l'intelligence. Le fait de travailler dans une machine non intelligente permet de s'assurer que l'on ne rajoute pas d'hypothèses involontaires et incontrôlées. L'interprétation "humaine", sur le papier d'un texte en Langage Universel Simplifié est ardue et très souvent fausse.

Algorithme de l'interpréteur de Langage Universel Simplifié actuel

L'algorithme actuel, une fois la phase d'initialisation passée est le suivant:

Le caractère actuellement dans l'input est inséré en tête de la liste chaînée d'input.

La liste chaînée d'input est comparée à la liste des chaînes de caractères déjà connues.

Si elle ne se termine pas par une chaîne de caractères connues on reprend au début.

Sinon, si la chaîne d'input est exactement la chaîne de caractères connue, on stocke sa référence dans la liste d'instruction.

Si la liste d'instruction comporte 1 seul objet connu on reprends au début.

Si la chaîne d'instruction se termine par une chaîne connue et a au moins un caractère en tête, alors on crée un nouvel objet connu dont la chaîne de caractère caractéristique est le début de la liste d'input. On stocke dans la liste d'instruction les référence du nouvel objet puis de le objet reconnu.

Les deux premiers objets de la liste d'instruction sont insérées dans la liste chaînée d'exécution et supprimés de la liste d'instructions.

On reprend au début.

Cet algorithme est lui même contenu dans la liste chaînée d'exécution dont une instruction est exécutée à chaque "tour" du moteur de niveau 0.

Cette version de l'interpréteur découpe donc l'input en chaînes connues , et est sensible à la distinction majuscule minuscule, ne connaît aucun séparateur sauf des expressions déjà connues.

Exemples de code en Langage Universel Simplifié

Par convention, on mettra en caractères *Italiques* les expressions qui ne sont pas encore connues, donc *nouvelles*. Cette convention est juste là pour faciliter la lecture Humaine du code. On mettra en gras les expressions qui sont des mots clés prédéfinis. La première instruction comporte une flèche de direction qui indique la direction de tout le code. Cette flèche ne fait pas partie du code.

Fin <-- Créer **Fin Fin** Créer Nouvelle valeur Créer Autre nouvelle valeur
Est nouveau --> **Fin** Nouvelle valeur Est nouveau Autre nouvelle valeur Est nouveau

sont deux séquences d'instructions qui font la même chose, à savoir apprendre les nouvelles expressions en italique (le sens des mots ne compte pas bien sûr).

Cela fait d'ailleurs la même chose que la séquence:

A-->**Fin** BACA

Établissement d'une fonction classique en langage universel simplifié:

Comme vous ne me croyez pas quand je dis que cette seule instruction suffit, nous allons commencer par retrouver le test "Variable a=Variable b". C'est une fonction très utilisée en informatique, quel que soit le langage et qui retourne vrai si a et b ont la même valeur et faux sinon. Voilà comment on le code en langage universel simplifié:(le saut de ligne sert juste à la lisibilité humaine)

Est nouveau --> **Fin**

Pour commentaire Est nouveau

Ici on commence la déclaration des constantes Pour commentaire

Adresse vrai Est nouveau

Adresse faux Est nouveau

Adresse identique Est nouveau

Adresse variable identique 1 Est nouveau

Adresse variable identique 2 Est nouveau

Adresse variable a Est nouveau

Adresse variable b Est nouveau

Ici on commence la déclaration des variables Pour commentaire

Vrai Adresse vrai

Faux Adresse faux

Identique Adresse identique

Variable identique 1 Adresse variable identique 1

Variable identique 2 Adresse variable identique 2

Variable a Adresse variable a

Variable b Adresse variable b

Début du code utilisant la fonction Pour commentaire

aa Variable a Affectation de la valeur de la variable a à tester Pour commentaire

bb Variable b Affectation de la valeur de la variable b à tester Pour commentaire

Début de l'appel de la fonction Pour commentaire

Variable a Adresse variable identique 1

Variable b Adresse variable identique 2

Exécution de la fonction Pour commentaire

Adresse faux Variable identique 1

Adresse vrai Variable identique 2

Variable identique 1 Adresse identique

A ce moment identique contient vrai si les deux variables avaient le même contenu (ce qui n'est pas le cas ici) Pour commentaire

Et sous cette forme là, sans même besoin de la notation italique, il s'agit du code directement interprétable. Dans ce code il y a un défaut assez grave ? le voyez vous ?

En fait les variables testées sont détruites par le test. On sait si Variable a était égale avant le test à Variable b. Pour y remédier il faut compliquer un peu le code, pour sauvegarder les valeurs des variables avant et les restituer après, ce qui donne:

Est nouveau --> **Fin**

Pour commentaire Est nouveau
Ici on commence la déclaration des constantes Pour commentaire
Adresse vrai Est nouveau
Adresse faux Est nouveau
Adresse identique Est nouveau
Adresse variable identique 1 Est nouveau
Adresse variable identique 2 Est nouveau
Adresse sauvegarde 1 Est nouveau
Adresse sauvegarde 2 Est nouveau
Adresse variable a Est nouveau
Adresse variable b Est nouveau
Ici on commence la déclaration et l'initialisation des variables Pour commentaire
Vrai Adresse vrai
Faux Adresse faux
Identique Adresse identique
Variable identique 1 Adresse variable identique 1
Variable identique 2 Adresse variable identique 2
Sauvegarde 1 Adresse sauvegarde 1
Sauvegarde 2 Adresse sauvegarde 2
Variable a Adresse variable a
Variable b Adresse variable b
Début du code utilisant la fonction Pour commentaire
aa Variable a *Affectation de la valeur de la variable a à tester* Pour commentaire
aa Variable b *Affectation de la valeur de la variable b à tester* Pour commentaire
Début de l'appel de la fonction Pour commentaire
Variable a Adresse variable identique 1
Variable b Adresse variable identique 2
Exécution de la fonction Pour commentaire
Adresse variable identique 1 Adresse sauvegarde 1
Adresse variable identique 2 Adresse sauvegarde 2
Adresse faux Variable identique 1
Adresse vrai Variable identique 2
Variable identique 1 Adresse identique
Adresse sauvegarde 1 Variable identique 1
Adresse sauvegarde 2 Variable identique 2

Ici Identique contiendra vrai à la fin de l'exécution et les valeurs initiales de a et b sont restituées. En résumé ce que fait cet algorithme c'est de définir de manière absolue le test qui permet de comparer deux valeurs et voir si elles sont identiques. On écrit ce que l'on considère comme la réponse négative sur la première variable considérée comme une adresse. On écrit ensuite sur l'adresse indiquée par la seconde variable la réponse positive. On comprend que si les deux variables pointent vers la même adresse, on vient de réécrire au même endroit, donc d'effacer la valeur précédente. La première variable contient donc la réponse cherchée.

Il reste un (tout petit) bug dans cette version du code. Pouvez-vous le découvrir ?

Bien sûr, une fois la 'fonction' identique définie, on a plus besoin de répéter toutes les déclarations pour la réutiliser.

Cette fonction est d'un niveau de complexité très bas car elle n'utilise pas de référence à la chaîne d'exécution et pas la notion de suite.

Affichage de données:

L'output est un élément prédéfinis du système. Si on appelle **Afficher** sa chaîne caractéristique alors

A-->Afficher

a le double effet de faire apprendre A comme nouvel objet si il n'est pas connu et d'afficher sur le média d'output la valeur actuellement contenue à l'adresse de A. Par convention on considérera que le langage d'input et d'output sont les mêmes, c'est à dire que si A a une valeur de caractère, c'est un caractère qui sera affiché, sinon une adresse mémoire.

Notion de Suite:

La notion de suite est nécessaire au fonctionnement complet du langage et je n'ai pas trouvé de moyen de l'exprimer autrement. Toute adresse pointe vers 2 adresses mémoires et suite désigne l'autre adresse mémoire. Cela est réalisé grâce à une adresse mémoire particulière 'Adresse suite' qui est pointée par le moteur de niveau 0 en permanence vers la deuxième adresse pointée par l'adresse mémoire 'Variable suite'.

Si je fais:

Variable a Adresse variable suite

Alors après l'exécution de cette instruction, Suite contiendra la seconde adresse pointée par la valeur de Variable a.

Pour clarifier, le code suivant:

ABCDAdresse variable suite

Variable suite Afficher

Suite Adresse variable suite

Variable suite Afficher

Suite Adresse variable suite

Variable suite Afficher

Suite Adresse variable suite

Variable suite Afficher

A pour effet d'afficher DCBA sur le média d'output.

Exécution d'un saut incondionnel:

Première apparition de la chaîne d'exécution. Si Exécuter est la chaîne caractéristique de l'adresse de la chaîne d'exécution alors:

Adresse procédure --> Exécuter

A pour effet de changer la chaîne d'exécution (qui contient entre autres l'interpréteur) vers la chaîne pointée par Adresse procédure. Il vaut mieux avoir préparé correctement cette nouvelle chaîne d'exécution si on ne veut pas tuer le programme...En général cela se fait plutôt en copiant la chaîne d'exécution actuelle et en la modifiant.

Exécution d'un saut conditionnel, fonction si alors sinon.

On sait déjà réaliser au moins un test. SI on donne à la variable Vrai une adresse de procédure déjà préparée sous forme de chaîne d'objet 'A exécuter si vrai'. On donne à la variable Faux la procédure 'A exécuter si faux'. Il suffit alors de faire un saut incondionnel vers le résultat de Condition:

Adresse vrai<-- Adresse si vrai

Adresse faux Adresse si faux

Exécuter Condition

Si condition pointe vers vrai, on transfère vers Exécuter le contenu de vrai donc A exécuter vrai

Avec ces fonctions de base, il me paraît clair que l'on peut réaliser n'importe quel programme dans ce langage.

Un petit mot à l'intention des scientifiques et chercheurs en informatique théorique officiels (si ils me lisent) :

Mon activité est une activité de 'loisir' et je ne prétends pas connaître l'état actuel des travaux en informatique théorique. Toutefois, je suis persuadé que ceci présente un intérêt plus profond qu'une lecture rapide et amusée peut vous faire penser. Je serais très heureux de recevoir des informations sur la bibliographie à ce sujet, l'état actuel des connaissances, les précédents, et suis à votre disposition pour toute discussion par e-mail. Cela me ferait sans doute beaucoup progresser d'avoir des retours d'information (même très critiques).

Pour les non chercheurs en informatique théorique:

Il ne s'agit pas que d'un langage informatique, c'est un langage tout court, qui devrait permettre de définir toutes les notions avec beaucoup plus de précision, en mathématiques par exemple, je suis toujours resté un peu frustré du départ de la théorie des ensembles, qui sert de base à toute la construction mathématique.

État actuel des recherches sur le Langage Universel Simplifié.

Je ne peux hélas pas progresser très vite sur le développement de ce langage et suis prêt à collaborer avec toutes les bonnes volontés qui se présenteront pour y travailler. J'en suis actuellement à comprendre comment manipuler ce langage pour comprendre comment se construit la complexité. J'ai écrit un premier interpréteur, qui n'est pas encore assez intégré pour être satisfaisant et donc pas encore en mesure développer un programme intelligent.

Pourquoi un langage universel ?

Comme on peut le constater tous les jours, la communication entre humains est un exercice délicat. Il est très difficile de se faire comprendre de ses interlocuteurs. Le langage que nous manipulons, croyons-nous aisément, est en fait fondé sur des mots dont les définitions sont assez floues. Pour s'en convaincre, il suffit d'ouvrir un dictionnaire et de constater que les définitions des mots sont auto-référencées. Mêmes les mathématiques dépendent pour leur fondation de mots à la définition vague: ensembles, éléments. Dans le cadre d'essais de démonstrations ou de raisonnements, je me suis rendu compte de la fragilité de la réflexion du cerveau humain, très prompt à admettre des hypothèses non vérifiées, sans même sans rendre compte. C'est pourquoi j'ai décidé de me tourner vers un support qui, tout en pouvant processor des équations, exécuter des calculs et des démonstrations, n'introduisait pas d'hypothèses de lui même: l'ordinateur. C'est ainsi que j'ai essayé de voir si je pouvais construire mes raisonnements en déclarant de manière explicite toutes les hypothèses à une machine.

Conclusion

L'objectif n'est évidemment pas d'écrire un nouveau langage informatique mais bien grâce à un langage très dépouillé d'accéder à la compréhension intime de la construction de la complexité. La simplicité de ce langage permet d'envisager par exemple la génération de tous les programmes possibles d'un certain nombre d'instructions pour juger de leur 'intérêt'. On peut aussi les mettre en compétition 'naturelle' pour observer leur comportement. Il me semble en effet, qu'un certain nombre des algorithmes qui régissent notre pensée sont trop simples pour qu'on les imagine. La porte vers l'intelligence artificielle est entr'ouverte et elle conduit vers bien plus que des machines intelligentes. La compréhension du monde est à la clé.

