La création d'un langage de programmation

et les différentes fonctions qu'il peut occuper

Sommaire

[Une introduction sur le langage 3](#_Toc158574722)

[Le langage naturel 3](#_Toc158574723)

[Langage bas-niveau / Langage haut-niveau 4](#_Toc158574724)

[Paradigmes de langage 5](#_Toc158574725)

[Compilé / Interprété 7](#_Toc158574726)

[Conclusion 8](#_Toc158574727)

[PowerLine : Début du projet 9](#_Toc158574728)

[RAM 9](#_Toc158574729)

[Registres 10](#_Toc158574730)

[Fonctionnement théorique 11](#_Toc158574731)

[Lexing 12](#_Toc158574732)

[Parsing 12](#_Toc158574733)

[Adaptations de la traduction 14](#_Toc158574734)

[Commencement des programmes 16](#_Toc158574735)

[Lexing 16](#_Toc158574736)

[Parsing 17](#_Toc158574737)

[Traducteur vers l’Assembleur 18](#_Toc158574738)

[Perspectives d’améliorations 19](#_Toc158574739)

[Conclusion 21](#_Toc158574740)

# Une introduction sur le langage

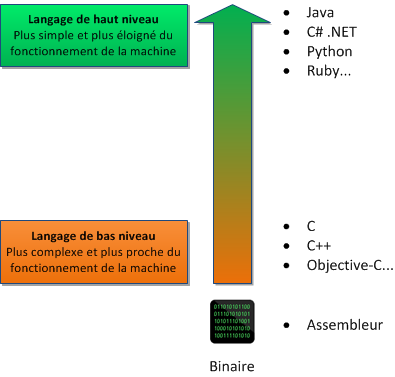
## Le langage naturel

Lorsque l’on parle du langage, nous pensons en priorité au langage humain. Certains chercheurs estiment que le début du langage humain pourrait remonter à plus de 100 000 ans, soit entre la moitié et les deux tiers de l’émergence de l’homo sapiens. Mais le langage ne nous a pas attendu pour se diffuser. Des abeilles aux éléphants, des dauphins aux corbeaux, le langage a pris différentes formes, vocal, gestuel, visuel, olfactif, etc. Le langage est un système de communication complexe et structuré, peu importe sa forme, permettant d’échanger des informations et d’interagir avec les autres membres le comprenant. De tout temps, l’humanité a cherché à imiter le comportement de la Nature. Epaulés de la sélection naturelle ne nous offrant qu’à voir la quintessence de ses optimisations, il était bien normal de se servir d’une technologie qui a été testée depuis des millions d’années. Des ingénieurs en aérospatiale ont examiné de près les techniques de vol des oiseaux et des insectes pour concevoir des aéronefs plus efficaces, réduisant la consommation de carburant et améliorant la manœuvrabilité. Les drones et les robots autonomes ont également bénéficié de la biomimétique pour naviguer dans des environnements complexes et changer leur forme en fonction des besoins. Le langage humain est également une pure imitation du langage qu’ont développé les autres animaux. Nous l’avons amélioré, transformé, divisé, et encore aujourd’hui, il évolue dans la multitude de langues que nous avons inventée. Se lance alors une problématique : imaginons deux chefs d’état alliés ont besoin de se coordonner et de discuter de la situation actuelle le plus vite possible, comment faire alors qu’ils ne parlent pas la même langue ? C’est très simple, soit ils parlent la même langue, soit ils utilisent un traducteur qui lui connait les deux langues.

Mais quel rapport avec le langage de programmation ?

Le langage de programmation, bien que purement artificiel, est une imitation du langage naturel, puisqu’il en a les mêmes buts. Tout comme nous utilisons des structures grammaticales et des règles pour exprimer nos pensées de manière claire et cohérente, les programmeurs utilisent des langages de programmation pour donner des instructions précises aux ordinateurs sans ambigüités. Ces langages, avec leurs syntaxes et leurs sémantiques spécifiques, permettent de traduire des concepts abstraits en une suite d'instructions compréhensibles par les machines. A la manière des chefs d’état, l’ordinateur ne parle pas une langue accessible aux humains. Au lieu de s’embêter à essayer de communiquer en binaire, nous allons utiliser un traducteur. Mais rappelons-le, les instructions que nous souhaitons donner aux machines ne doivent pas être ambiguës contrairement à notre langage humain qui regorge de double sens et d’interprétabilité. Nous avons donc créé des langages que des traducteurs en langage machine pourrait comprendre. Mais ces langages sont plus ou moins compliqués à s’approprier. S’introduit alors le concept de langage haut et bas niveau.

## Langage bas-niveau / Langage haut-niveau

Dans le domaine de la programmation informatique, les langages de programmation sont classés en différents niveaux en fonction de leur proximité avec le langage machine, c'est-à-dire le langage binaire compréhensible par l'ordinateur. Cette classification en niveaux est souvent liée à la complexité des instructions et à la manière dont les tâches sont exprimées dans le code source.

Les langages de haut niveau, tels que Python et JavaScript, se situent au sommet de cette échelle. Ces langages sont conçus pour être plus proches du langage naturel, notamment de l'anglais, facilitant ainsi la compréhension pour les programmeurs. Des constructions syntaxiques claires et des mots-clés intuitifs, comme «print» ou «def» en Python, ou «function» et «const» en JavaScript, rendent le code plus lisible et accessible. Cette proximité avec le langage humain permet aux développeurs de créer des algorithmes de manière plus intuitive, favorisant une compréhension claire et visuelle du code.

Ci-dessus : représentation du niveau de langages de programmation

Cependant, à mesure que l'on descend sur cette échelle vers des langages de plus bas niveau, tels que le langage d'assemblage ou même le langage machine, la lisibilité du code diminue considérablement. Ces langages exigent une connaissance approfondie de l'architecture matérielle de l'ordinateur et sont souvent cryptiques pour un observateur non averti. Les instructions de bas niveau peuvent être représentées par des codes numériques et des opérations binaires, ce qui rend la programmation dans ces langages plus ardue et moins intuitive. Quelques exemples ci-dessous :

En C :

#include <stdio.h>

int main() {

int n = 42;

printf(«%d\n», n);

return 0;

}

En Assembleur :

section .data

n db 42

section .text

global \_start

\_start:

mov eax, 4

mov ebx, 1

mov ecx, n

mov edx, 2

int 0x80

mov eax, 1

xor ebx, ebx

int 0x80

En python :

n = 42

print(n)

On se rend vite compte de la complexité que peut avoir un langage bas niveau et la facilité à comprendre un langage haut niveau. Mais cette facilité a un prix, celui de la vitesse d’exécution et de la performance. Le langage haut niveau va avoir besoin de plus de temps pour faire passer l’information puisque sa formulation peut grandement différer de l’information à transcrire. Mais au-dessus de tout, les langages bas niveau peuvent offrir des possibilités supplémentaires, comme la capacité de modifier directement la mémoire et son affectation, ou même la manière de réaliser physiquement chaque instruction. On en conclut donc que nous n’utiliserons pas un langage de programmation pour sa simplicité mais plutôt pour les capacités qu’ils nous offrent. Un langage peut être très bon dans un certain domaine, un autre excellera dans un autre domaine. Et nous voilà dirigés lentement vers la notion de paradigmes.

## Paradigmes de langage

Selon Larousse, le paradigme est un « Modèle Théorique de pensée qui oriente la recherche et la réflexion scientifique ». Dans le contexte de la programmation, il s’agit d’approches ou de styles spécifiques pour concevoir et structurer des programmes informatiques. Chaque paradigme a ses propres règles, conventions et techniques qui guident la manière dont les programmes sont écrits.

Si vous vous trouvez face à un problème vaste, il n’y a pas une seule manière de le résoudre, vous pouvez user de différentes solutions afin d’en venir à bout. Les paradigmes seraient comme les styles de solutions accessibles. Voici la liste de quelques paradigmes de programmation :

* Impératif

L’impératif est centré sur la description de la séquence d’instructions qui modifie l’état du programme. On écrit simplement une suite d’étape pour obtenir le résultat voulu. Les programmes impératifs utilisent des instructions comme les boucles, les conditions et les variables pour décrire sans ambigüité les actions à effectuer.

On peut citer le langage C ou le Pascal qui sont principalement des langages impératifs.

* Déclaratif

Contrairement à l'impératif, le paradigme déclaratif se concentre sur la déclaration des résultats souhaités sans détailler les étapes pour les atteindre. Les programmes déclaratifs décrivent ce que le programme doit accomplir, et le système détermine comment le faire. C’est un peu du « Débrouille toi »

On peut citer le SQL comme langage déclaratif.

* Orienté Objet

Le paradigme orienté objet organise le code autour d'objets, des entités qui regroupent des données et les méthodes qui les manipulent. La notion d’objets est difficile à saisir lorsque l’on commence à travailler avec. Il s’agit en fait d’une sorte d’entité qui a des paramètres propres. Comparons donc cela à un jeu vidéo : vous commencez en tant que barbare. Vous avez donc un bonus de force, de vitesse et de résistance, dans votre inventaire se trouve une épée. En programmation, au lieu de créer une liste pour chaque attribut pour chaque personne qui rejoint votre jeu afin de lui ajouter l’épée et le bonus de force, de vitesse et de résistance, vous allez créer un objet. Cet objet sera la représentation de votre barbare, et lorsque vous l’appellerez pour le créer il commencera avec son épée et ses stats bonus. Vous pourrez alors le modifier directement ou exécuter des fonctions qui lui son propre, comme donner un coup d’épée ou courir par exemple.

Également, si votre barbare prend une spécialité de combat comme viking par exemple et que désormais il peut également se protéger avec son bouclier en plus de son coup d’épée et courir, vous n’allez pas recréer un objet de zéro, vous utiliserez ce qui s’appelle l’héritage. Votre viking sera un barbare basique, mais avec un bouclier en plus. Il hérite de toutes les spécificités de son parent.

C’est une technique va vous permettre de gagner du temps mais n’est pas applicable partout. Il ne sert à rien de créer une classe pour faire une addition, son utilisation est assez spécifique.

Les langages Python et Java en sont des exemples.

* Logique

Le paradigme logique est une approche de programmation qui repose sur la logique formelle et les règles déductives. Dans ce paradigme, les programmes sont définis par des faits et des règles logiques. La mise en œuvre la plus courante de ce paradigme est le langage de programmation Prolog. Voici les principes de base :

Faits : Les faits sont des déclarations qui expriment des vérités sur le domaine du problème. Par exemple, dans un système de planification, un fait pourrait être « Le soleil se lève à l'est. »

Règles : Les règles décrivent les relations entre les faits. Elles sont souvent formulées sous la forme « Si [condition], alors [conclusion] ». Par exemple, « Si c'est l'aube et les oiseaux chantent, alors c'est le matin. »

Inférence : Le moteur d'inférence du langage logique examine les faits et les règles disponibles pour déduire de nouvelles informations. Il peut répondre à des requêtes en tirant des conclusions logiques.

Imaginez un enquêteur résolvant un mystère. Le paradigme logique serait comme la construction d'un tableau de liens et de déductions. Chaque indice découvert est ajouté à la liste des connaissances. Les règles logiques agissent comme des directives pour connecter les indices et tirer des conclusions. Si le détective sait que chaque fois qu'il pleut, le sol est mouillé, et qu'il constate un sol mouillé, il peut en déduire qu'il vient de pleuvoir.

En d'autres termes, c'est comme assembler un puzzle où chaque pièce s'emboîte logiquement avec les autres. Les règles servent de guide pour savoir comment placer les pièces ensemble. C'est une manière de résoudre des problèmes en construisant une compréhension logique du domaine plutôt qu'en séquençant des instructions.

Le paradigme logique est souvent utilisé dans des domaines tels que l'intelligence artificielle, la planification automatisée, les systèmes experts et d'autres applications où la prise de décision basée sur des règles logiques est cruciale.

Le prolog se développe de cette manière. En voici un exemple.

On voit plus facilement émerger le paradigme logique via cet exemple.

% Faits : caractéristiques des animaux

a\_des\_plumes(oiseau).

a\_des\_pattes(oiseau).

peut\_voler(oiseau).

a\_des\_pattes(chien).

a\_un\_pelage(chien).

% Règles : déduction basée sur les caractéristiques

est\_un\_oiseau(X) :- a\_des\_plumes(X), a\_des\_pattes(X), peut\_voler(X).

Il existe bon nombre d’autres paradigmes de langage de programmation. Mais notre histoire de traducteur refait surface pour expliquer une nouvelle notion, celle de l’interprétation et de la compilation.

## Compilé / Interprété

Un alien s’est écrasé sur Terre, mais visiblement il vient en paix. Bien que vous ne sachiez pas ce qu’il dit (vous n’avez pas fait LV2 martien) ses intentions ne vous paraissent pas du tout agressives. En fait, il semblerait qu’il ait besoin de votre aide ! Le moteur de sa soucoupe s’est lourdement détérioré lors de son atterrissage forcé et il ne sait pas comment le réparer. Mais vous, vous êtes le meilleur mécanicien du pays, et en jetant un œil à sa technologie vous avez tout de suite compris comment faire. Le seul problème, c’est que son moteur produit une chaleur extrême, mais l’alien ne semble pas du tout craindre cette chaleur. Par chance, il se trouve que notre martien a une machine qui peut traduire n’importe quelle langue écrite dans la sienne. Comment allez-vous faire pour lui expliquer ?

Si vous décidez d’écrire toutes les étapes à suivre sur une grande feuille et de la passer dans le traducteur, c’est que vous avez choisi la méthode compilée. Un langage compilé, lorsque vous allez lancer l’exécution du programme, va traduire tout votre script en langage machine, l’ordinateur va exécuter toutes les instructions. L’avantage de cette méthode est sa grande rapidité. En effet le martien va faire exactement ce que vous lui avez demandé et enchainer toutes les étapes sans attendre. Mais le gros défaut de cette méthode est l’incapacité de correction. Votre machine est lancée, mais rien ne va l’arrêter désormais. Et s’il y a une erreur, il va falloir faire avec ! La suite d’instruction continuera, en faisant toujours plus d’erreurs et quand il s’arrêtera, il vous dira simplement « ça n’a pas marché ».

Mais si vous décidez d’envoyer chaque instruction sur une feuille que vous passez dans la machine étape par étape, vous avez choisi le langage interprété. Chaque instruction va se traduire en langage machine, puis va s’exécuter, vérifiant s’il y a une erreur, puis passera à la suivante. Si vous avez donné une instruction faussée, alors le programme s’arrête et vous dira « ça n’a pas marché à cette instruction ». Vous aurez donc accès à la moindre correction possible sur votre programme. Mais que c’est long ! Chaque instruction fait un aller-retour entre traduction et vérification d’erreur. Vous allez perdre un temps fou, mais au moins vous n’allez pas avoir à chercher votre erreur.

## Première conclusion

En conclusion, le langage de programmation se présente comme une traduction ingénieuse du langage naturel dans le monde informatique, permettant aux programmeurs de donner vie à leurs idées en un langage compréhensible par les machines. À travers l'exploration des niveaux de langage, des paradigmes de programmation, et des méthodes d'exécution, nous comprenons que le choix du langage revêt une importance cruciale dans la conception de solutions informatiques. C'est une décision qui transcende la simple syntaxe pour toucher la lisibilité, la vitesse d'exécution, et l'adaptabilité aux besoins spécifiques d'un projet. Alors que l'informatique continue d'évoluer, le langage de programmation demeure au cœur de cette transformation, reflétant la constante recherche de moyens plus efficaces et expressifs pour communiquer avec les machines. Ainsi, choisir un langage de programmation devient une démarche stratégique, un acte de création guidé par la vision du programmeur et les exigences du projet.

# PowerLine : Début du projet

PowerLine sera un langage haut niveau dont l’objectif ne sera pas la rapidité mais plutôt la facilité à manier. Nous allons essayer de créer un langage haut niveau, autant que possible, et en français. Ce but est de créer une approche simple pour les jeunes souhaitant débuter le code. Pour commencer, nous essayerons de créer ce langage de manière compilée, de lui implémenter quelques fonctionnalités de bases, puis une fois fonctionnel, on rajoutera le plus possible d’éléments intéressant comme l’affichage de textes. Puis il faudra créer un traducteur vers l’assembleur et lui permettre de s’exécuter. Pour ça, tout un tas d’étapes apparaissent, que j’expliquerai en temps voulu.

Pour rendre notre programme fonctionnel, il faut d’abord trouver une syntaxe unique, qui lui permettra de ne connaître aucune ambiguïté. Commençons par des commandes simples et pourtant qui renferment une certaine difficulté. D’abord, nous allons essayer de faire des calculs arithmétiques. Notre ordinateur n’était qu’une grosse machine de calcul, cela va nous occuper un petit temps. Puis en avançant sur les différentes syntaxes, on se rendra compte que nous pourrons réutiliser les différents éléments, comme le fait de mettre : après une fonction. Mais commençons donc par les calculs.

Si nous demandions à la machine de nous faire le calcul suivant :  
(3 + 2) \* 6 + 12

La première étape serait de déterminer 3+2 Puis nous devons multiplier 6 avec … avec l’instruction précédente, soit 3+2. Le souci est que la machine doit se souvenir des étapes précédentes. Et pour ça on va utiliser un outil de l’ordinateur qui se nomme …

## RAM

La RAM, ou mémoire vive (Random Access Memory), est un composant essentiel de l’ordinateur qui offre une mémoire temporaire et rapide pour stocker et accéder aux données nécessaires pendant l'exécution des programmes. Contrairement au stockage permanent comme les disques durs ou les SSD, la RAM est « volatile », ce qui signifie que son contenu est effacé lorsque l'ordinateur est éteint. La RAM joue un rôle crucial dans les performances d'un système, car elle permet le chargement rapide des applications et des données fréquemment utilisées par le processeur. L'accès rapide à la RAM permet à l'ordinateur d'exécuter plusieurs tâches simultanément de manière efficace. Les capacités et la vitesse de la RAM influencent directement les performances globales d'un système informatique, et un montant adéquat de RAM est crucial pour assurer une expérience informatique fluide et réactive.

On pourrait la représenter comme un grand ruban contenant des millions de cases. Chaque cas a un numéro, pour savoir où elle se trouve, et une valeur, soit notre stockage.

1 2 3 4 5 6 7 8 9 …

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 15 | 6565 | 78 | 2 | 66 | 22 | 1 | 2 |  |

Donc, la RAM agit comme un espace de stockage pour le processeur, fournissant une mémoire rapide et temporaire qui facilite l'exécution rapide des opérations et améliore les performances globales d'un ordinateur. Et nous allons nous en servir énormément.

Imaginons donc que nous avons notre case vide du ruban de mémoire ram, disons la case 215. Notre calcul se ferait donc comme ça :

* (215) <- const 3
* (215) <- add 2
* (215) <- mul 6
* (215) <- add 12

Et on peut se dire qu’on a fini, et que c’est vraiment rapide. Notre case 215 a donc le nombre que nous souhaitions stocker en elle. Il ne manque plus que de créer une commande pour pouvoir l’afficher.

Mais une information vient tout chambouler. En voici une représentation sous forme de tableau.

|  |  |  |
| --- | --- | --- |
| Stockage | Espace | Temps (nano seconde) |
| Registre | 16 | 1 |
| RAM | 1Go | 100 |

Nous avons énormément de RAM et peu de ce qu’on appelle les registres. Mais s’ils sont si peu, c’est parce qu’ils sont excessivement plus rapides que la RAM, ce qui coûte donc plus cher à la fabrication de l’ordinateur.

## Registres

Les registres sont de petits espaces de stockage ultra-rapides situés directement dans le processeur d'un ordinateur. Ils sont utilisés pour stocker temporairement des données et des instructions pendant l'exécution d'un programme. Puisqu'ils sont intégrés au cœur du processeur, les registres permettent un accès très rapide aux informations, beaucoup plus rapide que l'accès à la mémoire principale de l'ordinateur. Les données temporaires, comme les résultats intermédiaires de calculs, sont stockées dans les registres pour que le processeur puisse les manipuler rapidement pendant l'exécution des programmes, contribuant ainsi à accélérer les opérations informatiques. En résumé, les registres sont des zones de stockage ultra-rapides utilisées par le processeur pour travailler rapidement avec les données nécessaires à l'instant précis.

Ils vont donc nous servir à faire fonctionner notre programme à peu près 100 fois plus vite que si nous utilisions de la RAM pour tout faire. On nomme nos deux registres comme ceci : %rax et %rbx

* %rax <- const 3
* %rbx <- const 2
* %rax <- add %rbx
* %rbx <- const 6
* %rax <- mul %rbx
* %rbx <- const 12
* %rax <- add %rbx

Et grâce à ça, on vient théoriquement de rendre notre programme 50 fois plus rapide.

# Fonctionnement théorique

Implémentons les premières fonctionnalités comme la commande d’affichage. Nous choisirons par « Montrer » pour ne pas reprendre le si commun « Afficher » et sera une commande intégrée. Elle aura cette première syntaxe :

Montrer : [Ce que vous voulez afficher], [Ce que vous voulez afficher], [Ce que vous voulez afficher] .

Elle se termine par un . pour montrer que c’est la fin de notre fonction.

Et si nous reprenons notre calcul et que nous souhaitons le Montrer, voici ce que ça donne.

Montrer : (3 + 2) \* 6 + 12 .

On retranscrit donc les informations en langage plus bas niveau :

* %rax <- const 3
* %rbx <- const 2
* %rax <- add %rbx
* %rbx <- const 6
* %rax <- mul %rbx
* %rbx <- const 12
* %rax <- add %rbx
* Print %rax

Ce qui donnerait en assembleur

global \_start

section .text

\_start:

mov %rax, 3 ; %rax <- const 3

mov %rbx, 2 ; %rbx <- const 2

add %rax, %rbx ; %rax <- add %rbx

mov %rbx, 6 ; %rbx <- const 6

imul %rax, %rbx ; %rax <- mul %rbx

mov %rbx, 12 ; %rbx <- const 12

add %rax, %rbx ; %rax <- add %rbx

mov rdi, 1 ; Descripteur de fichier pour la sortie standard (stdout)

mov rsi, %rax ; Valeur à imprimer (dans %rax)

mov rdx, 10 ; Nombre d'octets à imprimer

mov %rax, 1 ; Code système pour l'appel système write

syscall

mov %rax, 60 ; Code système pour l'appel système exit

xor rdi, rdi ; Code de retour 0

syscall

Comme on peut le remarquer, une simple opération en assembleur prend autant de ligne. Pour commencer, on va juste traduire notre programme dans le pseudo assembleur, laissons le véritable calvaire pour la partie pratique. Alors comment faire pour notre traducteur ? Nous entrons dans la partie de la compilation.

## Lexing

Le lexing, en français analyse lexicale, est la première étape du processus d'analyse d'un programme informatique. Son rôle principal est de convertir une séquence de caractères, représentant le code source d'un programme, en une série de "jetons" ou "tokens". Ces tokens sont des unités lexicales, des morceaux plus petits et significatifs du code, tels que des mots-clés, des identificateurs, des opérateurs, des nombres et des symboles.

L'analyse lexicale simplifie le traitement ultérieur du code source en fournissant une représentation structurée et simplifiée du texte. Les lexèmes ainsi obtenus servent de base à l'étape suivante du processus de compilation ou d'interprétation. Un analyseur lexical (lexer) parcourt le code source caractère par caractère, reconnaît les motifs lexicaux et génère une séquence de tokens associés à ces motifs.

Prenons par exemple notre commande que nous avions créée :

Montrer : (3 + 2) \* 6 + 12 .

D’abord, le Lexer va d’abord décomposer notre chaine en une liste de caractères, soit

[‘M’, ‘o’, ‘n’, ‘t’, ‘r’, ‘e’, ‘r’, ‘ ‘, ‘:’, ‘ (‘, ‘3’, ‘+’, ‘2’, ‘)’ , ‘\*’, ‘6’, ‘+’ , ‘1’, ‘2’, ‘ ‘, ‘.’]

Première étape, il va retirer les espaces, deuxième étape, il va assembler les suites de caractères qu’il trouve pour essayer de créer un token. Il va également vérifier si deux chiffres se trouvent l’un à coté de l’autre ou si deux chiffres entourent un point pour former un nombre décimal. Et on obtient ça :

[‘Montrer’, ‘:’, ‘ (‘, ‘3’, ‘+’, ‘2’, ‘)’ , ‘\*’, ‘6’, ‘+’ , ‘12’, ‘.’]

## Parsing

Le parsing, ou analyse syntaxique, constitue la deuxième étape du processus d'analyse d'un programme informatique, succédant à l'analyse lexicale. Son objectif est de prendre la séquence de tokens générée lors de l'analyse lexicale et de les organiser en une structure hiérarchique qui représente la syntaxe du programme conformément à la grammaire du langage de programmation. L'analyse syntaxique utilise des règles grammaticales définies pour déterminer la manière dont les tokens peuvent être combinés pour former des constructions syntaxiques valides. Ces règles décrivent la structure grammaticale du langage, indiquant par exemple comment les déclarations de fonctions, les boucles, les conditions et d'autres constructions doivent être formées. L'objectif final de l'analyse syntaxique est de créer un arbre syntaxique, également appelé arbre d'analyse, qui représente la structure hiérarchique du programme. Cet arbre est utilisé comme base pour l'étape suivante du processus, généralement l'analyse sémantique ou la génération de code.

Revenons donc à notre exemple :

[‘Montrer’, ‘:’, ‘ (‘, ‘3’, ‘+’, ‘2’, ‘)’ , ‘\*’, ‘6’, ‘+’ , ‘12’, ‘.’]

D’abord on a la fonction montrer qui va englober des caractères avec : et . On en déduit donc qu’il faut afficher l’intérieur du montrer. Puis on a des parenthèses avec un calcul à l’intérieur. On doit commencer par exécuter notre calcul puis le suivant, puis le suivant. Notre arbre syntaxique devrait ressembler au suivant :

Montrer

+

\* 12

+ 6

3 2

3 + 2 \* 6 + 12 Montrer

Voici l’ordre de nos actions à réaliser selon la phrase que nous avions donnée au début du Lexing (sans tenir compte des priorités de calcul). On peut donc en déduire le code qui y sera associé en pseudoAssembleur.

* %rax <- const 3
* %rbx <- const 2
* %rax <- add %rbx
* %rbx <- const 6
* %rax <- mul %rbx
* %rbx <- const 12
* %rax <- add %rbx
* Print %rax

Et avec cela, on peut en déduire une première étape de traduction après le parsing, un peu à la manière du Prolog, on écrit une règle et en ressort une affirmation.

* Si on a un nombre n, alors r?x <- const n

*(r?x car on ne sait pas pour l’instant s’il faut la mettre dans %rax ou %rbx)*

* Si on a un opérateur, alors dans un premier temps, il faut placer la branche de gauche dans %rax et la branche de droite dans %rbx puis on fait %rax <- [opérateur] %rbx
* Si on doit montrer, alors print %rax

Et maintenant, voyons si notre déduction est bonne ou s’il faut l’ajuster un peu ! Imaginons que je veuille …

Montrer : 6 \* 4 + 3.

1. Lexing

['M', 'o', 'n', 't', 'r', 'e', 'r', ' ', ':', ' ', '6', ' ', '\*', ' ', '4', ' ', '+', ' ', '3', '.']

['Montrer', ':', '6', ' ', '\*', ' ', '4', ' ', '+', ' ', '3', '.']

1. Parsing

Montrer

\*

6 +

4 3

On suit donc notre règle. On commence par la branche de gauche, on a un 6, mettons donc %rax à 6.

* %rax <- const 6

Puis on descend dans la branche de droite en parcourant l’arbre, on a un 4 et un 3, mettons les dans %rax et %rbx.

* %rax <- const 4
* %rbx <- const 3

On remonte d’une étape, %rbx est égal à la somme de %rax et %rbx.

* %rbx <- add %rax

Et enfin, en remontant, on obtient un \* donc

* %rax <- mul %rbx

Et on obtient alors la suite d’instruction suivante :

(4 + 3) \* 4… Ce n’est pas ce qu’on avait demandé, en effet, notre %rax = 6 a été écrasé par %rax = 4… ça signifie qu’il nous faudrait un troisième registre. Mais si alors on fait une étape de plus il nous faudrait une infinité de registres et leur prix étant élevé, il vaut mieux trouver une autre solution… Une solution que nous avions plus tôt mais qui était trop lente. Pourquoi ne pas mélanger la ram et les registres ?

## Adaptations de la traduction

D’abord, on a remarqué que ça ne fonctionnait pas comme on le souhaitait parce qu’il faut rajouter de la mémoire. C’est exactement ce que l’on va faire avec de nouvelles fonctions pseudoAssembleur que l’on va appeler store et load. Store permet de charger de la mémoire dans un emplacement ram vide indiqué par son numéro, load permet de le récupérer dans un registre spécifié.

Donc on peut désormais tenter d’adapter nos règles à nos nouvelles fonctionnalités.

* Si on a un nombre n, alors

%rax <- const n

(?) <- store %rax

*( (?) car on ne sait pas pour l’instant l’emplacement de notre case de ram)*

* Si on a un opérateur :

(X) <- branche de gauche

(Y) <- branche de droite

Rax <- load (X)

Rbx <- load (Y)

Rax <- [opérateur] %rbx

(Z) <- store %rax

* Si on doit montrer, alors print %rax

Rax <- load (Z)

Print %rax

Et en mélangeant ram et registre, on obtient un programme performant qui permet d’avoir assez de mémoire pour fonctionner. Mais le problème est de pour déterminer X, Y et Z. Ce qu’on va faire, c’est qu’on va sélectionner la première case libre dans notre ram. Imaginons notre ruban comme le suivant :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 1 | 4 | 5 | 5 | 5 | 4 | 4 | 477 |

L’ordinateur a réservé les deux premières cases mais on peut utiliser la suite. Reprenons notre calcul.

Montrer

\*

6 +

4 3

En suivant les règles voilà ce qu’il se passe :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 1 | 6 | 5 | 5 | 5 | 4 | 4 | 477 |
| 7 | 1 | 6 | 4 | 3 | 5 | 4 | 4 | 477 |

Puis on fait l’addition :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 1 | 6 | 7 | 3 | 5 | 4 | 4 | 477 |

Et enfin la multiplication :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 1 | 42 | 7 | 3 | 5 | 4 | 4 | 477 |

Ainsi on a notre résultat. Au fur et à mesure que nos résultats se forment, on libère la mémoire dont on n’a plus l’utilité. Une fois 42 affiché, on le libèrera aussi, et un autre programme viendra sûrement un jour le remplacer. C’est d’ailleurs ce que fait notre programme lorsqu’il écrase 4 pour former 7.

On s’aperçoit rapidement que la ram fonctionne comme une pile. La modification se forme toujours au plus haut de la pile, que ce soit un ajout ou une suppression. Seulement, le programme ne peut pas se souvenir tout seul de chaque emplacement de mémoire, au lieu de ça, on va créer un index de pile. C’est-à-dire qu’on ne touchera plus jamais à un élément à l’intérieur mais seulement celui qui se trouve au- dessus. Ça permet donc de ne pas avoir à se soucier de l’emplacement de notre élément 4. On connaitra le dernier élément, et c’est lui qui nous intéresse. Cet index va nous servir tellement souvent qu’on va en faire un registre. Bienvenue à %rsp, le stack pointer ou pointeur de pile.

Ainsi, on aura juste à implémenter notre pile de cette façon :

(%rsp) <- store %rax

%rsp <- add 1

Ça signifie que nous chargeons la case mémoire de numéro %rsp de la valeur de %rax.

Pour dépiler, il faut donc faire l’inverse :

%rsp <- sub 1

%rax <- load (%rsp)

Et pour ainsi clore cette partie de compilation, voici comment fonctionne notre traducteur vers pseudassembleur.

* Si on a un nombre n, alors

%rax <- const n

(%rsp) <- store %rax

%rsp <- add 1

* Si on a un opérateur :

%rsp <- sub 1

%rbx <- load (%rsp)

%rsp <- sub 1

%rax <- load (%rsp)

%rax <- [opérateur] %rbx

(%rsp) <- store %rax

%rsp <- add 1

* Si on doit montrer, alors

%rsp <- sub 1

%rax <- load (%rsp)

Print %rax

# Commencement des programmes

En annexe, vous trouverez trois programmes python dans le dossier « premiers essais ». Le premier se nomme lexing.py. En voici la description de fonctionnement.

## Lexing

Le fichier Lexing est assez court. Il a pour seul but de récupérer en entrée une chaine de caractère, soit celle de notre programme et il fait ressortir une chaine récupérable par le parser. Voici comment fonctionne l’algorithme de la fonction :

* tokken = ["Montrer"]: Définit une liste de tokens que la fonction recherche dans la chaîne d'entrée. Dans cet exemple, il cherche le token "Montrer".
* s1 = list(var.replace(" ", "")): Convertit la chaîne de caractères en une liste s1 où les espaces sont supprimés.
* La boucle while suivante parcourt la liste s1 pour fusionner les chiffres adjacents en un seul élément, permettant ainsi de traiter les nombres à plusieurs chiffres.
* La boucle for suivante parcourt la liste de tokens (tokken). Pour chaque token, elle cherche une correspondance à la fin de la liste s1 et fusionne les caractères si une correspondance est trouvée.
* Si un token est trouvé à la fin de s1, les caractères correspondants sont fusionnés.
* La boucle while à l'intérieur de la boucle for permet de gérer les cas où plusieurs occurrences du token peuvent se chevaucher.
* La fonction retourne la liste s1 modifiée, qui contient maintenant les tokens fusionnés et les caractères restants.
* La chaîne d'entrée est définie comme 'Montrer : (3 + 2)\*5^2-6.', et la fonction lexing1 est appelée avec cette chaîne. La sortie de la fonction est affichée.
* La sortie du programme est [Montrer, :, (, 3, +, 2, ), \*, 5, ^, 2, -, 6, .]

Ainsi, on va convertir notre langage powerline en langage parsable. Les prochains mots clés, tokens, seront à ajouter dans la liste de début de programme ce qui simplifie grandement la mutabilité et l’évolutivité du programme lorsqu’il faudra ajouter des fonctionnalités de base.

## Parsing

La fonction du parser est un peu plus longue mais a été condensée au maximum pour être efficace. Elle a pour entrée la chaine du lexer et pour sortie un arbre syntaxique sous forme de liste python. Voici les étapes à atteindre pour obtenir le résultat souhaité.

* Convertir une expression mathématique en notation infixée en notation postfixée.
* Construire un arbre d'expression à partir de l'expression postfixée.
* Analyser une liste de tokens pour identifier et traiter les instructions "Montrer" ou "Calculer".

Voyons chaque fonction individuellement :

Fonction infix\_to\_postfix

Cette fonction prend une expression mathématique en notation infixée en tant qu'entrée et la convertit en notation postfixée en utilisant une approche de l'algorithme de la file d'attente (Shunting Yard).

Fonction postfix\_to\_tree

Cette fonction prend une expression postfixée en tant qu'entrée et construit un arbre d'expression à partir de cette expression.

Fonction parsing

Cette fonction prend une liste de tokens en entrée et analyse la structure de ces tokens pour identifier et traiter les instructions "Montrer" ou "Calculer". Elle utilise les fonctions précédentes pour accomplir cela.

Fonction display\_tree

Cette fonction prend un arbre d'expression en tant qu'entrée et l'affiche visuellement en utilisant des caractères spéciaux pour représenter les branches. La fonction utilise une approche récursive pour parcourir l'arbre. Chaque niveau d'indentation est représenté par l'espacement, et les caractères ├── et └── sont utilisés pour indiquer les branches.

*La fonction display a pour seul but de construire plus joliment les arbres et est totalement facultative au projet. En revanche voici les designs qu’elle permet de voir.*

|  |  |  |
| --- | --- | --- |
| Montrer  └── -  ├── \*  ├── +  ├── 3  └── 2  └── ^  ├── 5  └── 2  └── 6 | Montrer  └── +  ├── +  ├── +  ├── +  ├── +  ├── 2  └── 2  └── 2  └── 2  └── 2  └── 2 | Montrer  └── -  ├── +  ├── 2  └── \*  ├── 5  └── 6  └── +  ├── 3  └── /  ├── 4  └── 2 |
| Montrer : (3 + 2)\*5^2-6. | Montrer : 2+2+2+2+2+2. | Montrer : 2+ 5\*6-(3+4/2). |

On voit que grâce au système de priorité, les opérations se font dans le bon sens, laissant par exemple sur le 3 -ème arbre la soustraction pour la fin. Mais en réalité, cette fonction n’est pas si facultative. Elle va permettre d’arriver sur la phase 3, celle de la traduction en pseudoAssembleur. C’est cette fonction qui détermine l’ordre de visionnage des différentes branches.

## Traducteur vers l’Assembleur

Pour les précédentes explications, il était important de rendre les commandes assembleur un peu plus lisibles, ce qui justifiait l’utilisation du pseudo assembleur. Désormais nous n’avons plus besoin de simplifier. De plus l’ordinateur ne peut comprendre que l’assembleur pur dont le fichier doit être compilé lui-même avec tout un tas d’étapes que nous évoquerons plus tard. Seulement, comme dit plus tôt, l’assembleur est d’une complexité folle et même après avoir essayé de faire fonctionner de simples programmes, il me fut presque impossible de les faire fonctionner. En revanche, il y a un langage plus haut niveau que nous pourrions utiliser, basé sur de l’assembleur et qui a besoin d’une compilation pour fonctionner… Vous pouvez vous munir du fichier traductor.py, nous allons baser le PowerLine sur du C.

En effet, il y a beaucoup de langage basé sur le C, puisqu’il a des syntaxes plutôt simples à s’approprier et se compile facilement. Quelques exemples de langages basés sur C :

* C++ : Une extension du langage C qui ajoute le concept de classes et d'objets pour la programmation orientée objet. C++ conserve également la compatibilité avec le code C existant.
* Objective-C : Un langage de programmation qui ajoute des fonctionnalités de programmation orientée objet à C. Il a été principalement utilisé pour le développement d'applications macOS et iOS.
* C# (C Sharp) : Un langage développé par Microsoft qui combine des éléments du langage C et du langage C++, avec un accent particulier sur la programmation orientée objet. C# est largement utilisé pour le développement d'applications Windows et de logiciels basés sur le framework .NET.
* D : Un langage de programmation avec un nom qui me fait bien rire qui tente d'améliorer certains aspects du langage C++, tout en offrant une performance élevée et une syntaxe plus propre. Il intègre des fonctionnalités de programmation fonctionnelle et orientée objet.
* Rust : Un langage de programmation système conçu pour être sûr, concurrent et pratique. Rust a une syntaxe qui rappelle celle du C et est axé sur la prévention des erreurs de mémoire telles que les pointeurs nuls et les déréférencements non sécurisés.

Explications du fonctionnement

reacting\_trad function :

Cette fonction prend en charge la traduction des actions spécifiques en langage source vers du code C. Par exemple, elle traite les nombres, les opérateurs arithmétiques, les chaînes de caractères, et les instructions spécifiques comme "Montrer".

navigate function :

Cette fonction navigue dans la structure arborescente et applique la traduction à chaque nœud de l'arbre.

Variables head et body :

head contient la partie initiale du programme C, y compris les déclarations de variables et la fonction main.

body contient le corps du programme C généré à partir de la structure arborescente.

Génération du fichier C :

Le programme génère le code C en écrivant dans un fichier appelé "output.c".

Compilation et exécution :

Le programme utilise le compilateur GCC pour compiler le code C généré et crée un exécutable appelé "srt.exe".

Ensuite, il exécute "srt.exe" à l'aide de subprocess.Popen et capture la sortie standard et d'erreur.

Nettoyage :

Enfin, le programme supprime le fichier exécutable "srt.exe".

# Perspectives d’améliorations

Ce projet est selon moi, un projet sur le long terme. Avec la base que nous avons ici, nous avons la capacité de faire bon nombre de fonctions basée sur la détection de tokens et l’implémentation de nouvelles fonctionnalités. Le Lexer, Parser et Traductor sont des programmes facilement modifiables, avec des manières claires et une construction réfléchie pour augmenter les capacités. Voici quelques idées de pistes d’améliorations.

**Gestion des Erreurs :**

Implémenter un système de gestion des erreurs plus sophistiqué pour fournir des messages d'erreur détaillés et faciliter le débogage.

**Élargissement des Fonctionnalités :**

Ajouter la prise en charge de nouvelles fonctionnalités du langage source, comme des structures de contrôle de flux, des fonctions définies par l'utilisateur, et des opérations avancées.

**Optimisation du Code Généré :**

Travailler sur l'optimisation du code C généré, en recherchant des moyens d'améliorer l'efficacité et la lisibilité.

**Documentation et Commentaires :**

Ajouter des commentaires dans le code généré pour expliquer la correspondance entre les éléments du langage source et le code C résultant.

**Modularité et Tests Unitaires :**

Organiser le code en fonctions modulaires pour faciliter la maintenance et ajouter des tests unitaires pour garantir la stabilité du programme.

**Interface Utilisateur :**

Développer une interface utilisateur graphique (GUI) pour simplifier l'interaction avec le programme.

**Gestion des Entrées Utilisateur :**

Permettre au programme d'accepter le code source directement à partir de fichiers ou d'une interface utilisateur pour une utilisation plus conviviale.

**Optimisation de la Pile :**

Optimiser l'utilisation de la pile, explorant des techniques pour minimiser son utilisation ou améliorer certaines opérations.

**Extension du Langage Source :**

Étendre la grammaire du langage source pour inclure des fonctionnalités plus avancées et prendre en charge un ensemble plus large d'instructions.

En mettant en œuvre ces améliorations, notre projet pourrait devenir une ressource encore plus puissante et flexible pour la traduction de structures arborescentes entre différents langages de programmation.

# Conclusion

Notre projet vise à fournir une solution de traduction simple mais puissante, permettant de convertir des structures arborescentes d'un langage de programmation spécifique en code C exécutable. La réalisation de ce projet a nécessité la conception d'un traducteur capable de comprendre la grammaire du langage source et de générer du code C équivalent.

Nous avons réussi à créer un prototype fonctionnel qui prend en charge des éléments de base tels que les opérations arithmétiques, l'affichage de valeurs, et la gestion de la pile. Ce prototype peut être étendu pour prendre en charge des fonctionnalités plus avancées dans le futur.

J’ai été obligé par la dead line qui se rapprochait dangereusement d’écourter le projet. Il aurait été bien plus intéressant de vous présenter d’autres fonctionnalités. Peut-être un jour verrez-vous émerger le PowerLine dans les langages de débutants. Merci de m’avoir lu !