# Guillaume Lay

Mémoire

# En quoi le cycle de vie d’un langage de programmation influe t’il sur les choix technologiques des projets et de leur gestion au sein d’une PME ?

2020-2021

Table des matières

[Guillaume Lay 1](#_Toc70066298)

[En quoi le cycle de vie d’un langage de programmation influe t’il sur les projets ? 1](#_Toc70066299)

[Remerciements 3](#_Toc70066300)

[Résumé 4](#_Toc70066301)

[Abstract 5](#_Toc70066302)

[Introduction 6](#_Toc70066303)

[Partie 1 : Le cycle de vie d’un langage 6](#_Toc70066304)

[Introduction 6](#_Toc70066305)

[Leek Wars 7](#_Toc70066306)

[Résumé 7](#_Toc70066307)

[Le LeekScript 7](#_Toc70066308)

[Sous-partie 1 : La création d’un langage de programmation 9](#_Toc70066309)

[Le contexte 9](#_Toc70066310)

[Pourquoi le point-virgule ? 9](#_Toc70066311)

[Sous-partie 2 : Comment faire évoluer son langage 9](#_Toc70066312)

[Sous-partie 3 : Comment anticiper le déclin 9](#_Toc70066313)

[Partie 2 : Comment choisir son langage 10](#_Toc70066314)

[Sous-partie 1 : Etude 10](#_Toc70066315)

[Sous-partie 2 : Prise de décision 10](#_Toc70066316)

[Sous-partie 3 : Les problèmes d’un retour en arrière (Kendo et le Lierre) 10](#_Toc70066317)

[Partie 3 : L’impact d’un changement sur un projet 10](#_Toc70066318)

[Sous-partie 1 : La formation des équipes 10](#_Toc70066319)

[Sous-partie 2 : La cohabitation des deux versions 10](#_Toc70066320)

[Sous-partie 3 : La transition client 10](#_Toc70066321)

[Conclusion 11](#_Toc70066322)

[Bibliographie 12](#_Toc70066323)

[Livre 1 12](#_Toc70066324)

[Livre 2 12](#_Toc70066325)

[Livre 3 12](#_Toc70066326)

[Déroulé méthodologique 13](#_Toc70066327)

[Présentation du thème : 13](#_Toc70066328)

[Liste des personnes rencontrées : 13](#_Toc70066329)

[Calendrier M2 13](#_Toc70066330)

## Remerciements

Je tiens à remercier ILTR bidule machin.

## Résumé

## Abstract

# Introduction

### Présentation de l’entreprise

Fondée en 2002 par Yann Gobrait, ILTR est une entreprise d’édition de logiciels située à Angers dans le Maine-et Loire. C’est une PME d’une quarantaine d’employés dont l’activité principale est l’édition de logiciels de gestion de l’espace public.

### Présentation de GEODP

Le logiciel GEODP est le produit principal que propose la société ILTR. C’est un outil puissant permettant de gérer et de facturer l’utilisation de l’espace public. Cela peut aussi bien englober la gestion des emplacements d’exposants lors d’un marché hebdomadaire que le recensement des enseignes et affiches publicitaires qui occupent la voie publique.

Le logiciel est séparé en modules et chacun d’eux représente un secteur métier différent, tous orientés vers la gestion de l’espace public. Historiquement, le module le plus abouti et distribué est l’outil placier qui sert à la gestion des emplacements d’exposants sur les marchés. De nombreux autres secteurs d’activités sont disponibles mais ils sont en grande partie similaire au placier dans la manière de se présenter. Seul quelques spécificités métier différent d’un module à l’autre.

# Partie 1 : Le cycle de vie d’un langage

Parler d’Angular !!!

Solidity

GoLang

GameMaker

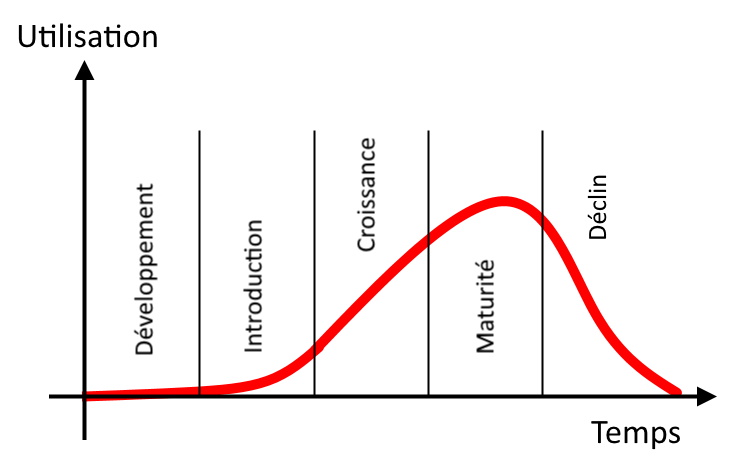
## Introduction

Aujourd’hui, on dénombre au moins sept cent langages de programmation différents. Au fil du temps, certains cessent d’être utilisés tandis que de nouveaux apparaissent chaque année. La course effrénée du progrès informatique permet la création de technologies d’une efficacité sans cesse améliorée et cela passe par l’actualisation des outils.

Face à cet amoncellement de langages de programmation divers et variés, nous pouvons nous interroger sur les tenants et les aboutissants de chacune de leur existence. Comment sont-ils créés ? Comment, de leur berceau à leur tombeau, influent-ils sur le paysage informatique ? Pourquoi sont-ils sans cesse renouvelés ? C’est à toutes ces questions que nous apporterons des réponses.

Comme tout produit, chaque langage informatique suit un cycle de vie composé de cinq phases distinctes. Chacune de ces périodes possède ses propres questionnements et enjeux pour atteindre les objectifs de ventes ou de taux d’utilisation.

Ces cinq étapes sont les suivantes :

* Développement : Toute la partie conception et création du produit, l’établissement de ses règles et propriétés.
* Introduction : Le moment où le produit est révélé aux utilisateurs.
* Croissance : La période pendant laquelle le produit gagne en notoriété et se voit évoluer rapidement.
* Maturité : Lorsque le produit est installé et profite d’un taux d’utilisation maximal.
* Déclin : Le produit cesse d’être utilisé et tombe en désuétude.

Dans cette partie, nous détaillerons ces différentes étapes du cycle de vie d’un langage informatique en se concentrant sur les points les plus importants afin d’observer l’impact du temps sur une technologie informatique. Nous nous baserons sur des cas concrets et nous comparerons leur historique afin d’en tirer des enseignements.

## Sous-partie 1 : La création d’un langage de programmation

La création d’un langage de programmation passe par plusieurs étapes indispensables. Pour comprendre la logique de base à adopter, nous allons nous aider d’un cas concret, celui du projet Leek Wars.

### Leek Wars

#### Résumé

Leek Wars est un jeu vidéo français créé en 2013 par des étudiants en informatique. Son objectif est d’initier les joueurs à la conception d’intelligences artificielles et à la programmation en général. Le principe est que chaque joueur dispose d’un personnage à l’effigie d’un poireau et doit affronter les personnages adverses. Pour cela, les actions que le poireau devra effectuer lors de l’affrontement devront être définie au préalable par le biais d’un script écrit par le joueur.

Sachant que les combats se déroulent au format tour par tour, à la manière d’un jeu d’échecs, le comportement du poireau doit prévoir toutes les situations afin d’espérer vaincre son ennemi. Toute une panoplie d’armes et de sortilèges sont à la disposition des combattants. Tout cet attirail leur laisse le choix de la stratégie à aborder. Ils ont donc une totale liberté sur le comportement de leur poireau mais ils devront user de tactique et de fins calculs pour qu’ils aient le plus de chances de se sortir de toutes les situations.

Même si le projet est réservé à un public très ciblé, à savoir les curieux au sujet de l’informatique et les développeurs en quête d’expérimentation, le jeu a su gagner en notoriété au fil des ans et la communauté ne cesse de s’agrandir. La barre des cinquante mille inscrits a été dépassée et l’application est en constante évolution grâce à des mises à jour régulières apportant équilibrage et nouveau contenu.

C’est donc un jeu de programmation ayant un contexte et un objectif propre à lui-même. Le code source du projet étant en accès libre, nous allons pouvoir étudier en détail le langage qui a été créé pour le jeu afin de mieux saisir les bases de ce qui fait un langage informatique.

Etant donné que ce projet présente dans son histoire à la fois la naissance, l’évolution et la quasi-disparition d’un langage, il nous servira de cas d’étude pour le sujet principal de cette première partie. La retranscription d’un entretien avec son créateur Pierre Laupêtre est jointe en annexe.

#### Le LeekScript

Afin de permettre aux joueurs de définir le comportement de leur poireau, un langage de programmation a été créé spécifiquement pour le jeu. Syntaxiquement proche du JavaScript, il a été pensé pour être totalement lié au système du jeu et pour être abordable pour les néophytes. Il est accompagné par un IDE intégré au jeu ainsi que son compilateur dédié.

Le LeekScript V1 est la version actuelle de ce langage et est celle qui est accessible aux joueurs utilisant l’application. Cependant, une nouvelle version, la V2, plus légère, plus performante et offrant de nouvelles possibilités est en cours de développement à l’heure de la rédaction de ce document. Elle doit notamment ajouter la programmation orientée objet (POO).

Suite à l’annonce relativement précoce de ce nouveau langage et de ses fonctionnalités, l’attente des joueurs a poussé le créateur du jeu à implémenter dans la V1 la POO. Cette évolution majeure de la technologie, causant une rétrocompatibilité difficile, a lancé la version 1.1 du LeekScript. La V2 est donc toujours en développement mais le projet entier étant open-source, il nous a été possible d’en prendre connaissance.

### Qu’est-ce qu’un langage de programmation ?

Avant de commencer toute analyse, il est nécessaire de définir ce qu’est un langage de programmation. Par définition, il s’agit d’un moyen de communication entre l’homme et la machine. Il permet à un être humain d’interagir avec un outil électronique afin d’en tirer le comportement voulu.

Comme le dialecte initial de l’ordinateur ne se compose que de suites d’uns et de zéros, la question est uniquement affaire de traduction. Un langage de programmation est donc un dialecte qui fait office d’intermédiaire entre ces deux mondes. L’humain peut ainsi donner des tâches à faire à la machine.

Afin d’exécuter son œuvre, la machine dispose d’un élément fondamental : la mémoire. Tout composant électronique destiné à exécuter des calculs dispose d’un espace où il peut entreposer ses informations. C’est la ressource principale utilisée dans un calcul informatique.

### Pourquoi créer un langage ?

En règle générale, la création d’un nouveau langage de programmation survient pour combler un besoin technologique. Par exemple, le langage C a été créé afin de réécrire UNIX, le système d’exploitation à l’origine de Linux, afin de le rendre plus performant, les langages existants n’étant pas à la hauteur des ambitions de ses fondateurs.

Plus récemment, le langage Go a été créé par Google afin de proposer un langage aussi puissant que le C avec la syntaxe la plus facile à apprendre possible. Le but est d’encourager et de rendre accessible la programmation pour tous.

Dans un autre contexte, l’objectif du langage PUG est de simplifier la rédaction de code HTML en remplaçant les balises par une syntaxe basée sur l’indentation. Le but est donc simplement d’offrir un confort visuel au développeur.

Pour aller plus loin, certains développeurs s’amusent à créer des langages dits ésotériques. Ce sont des langages volontairement complexes et se basant parfois sur des principes allant à l’encontre de l’efficacité. Ils n’ont pour but que d’expérimenter ou d’amuser. Bon nombre de ces langages expérimentaux sont consultables sur le site web Esolang.

### Comment créer un langage de programmation ?

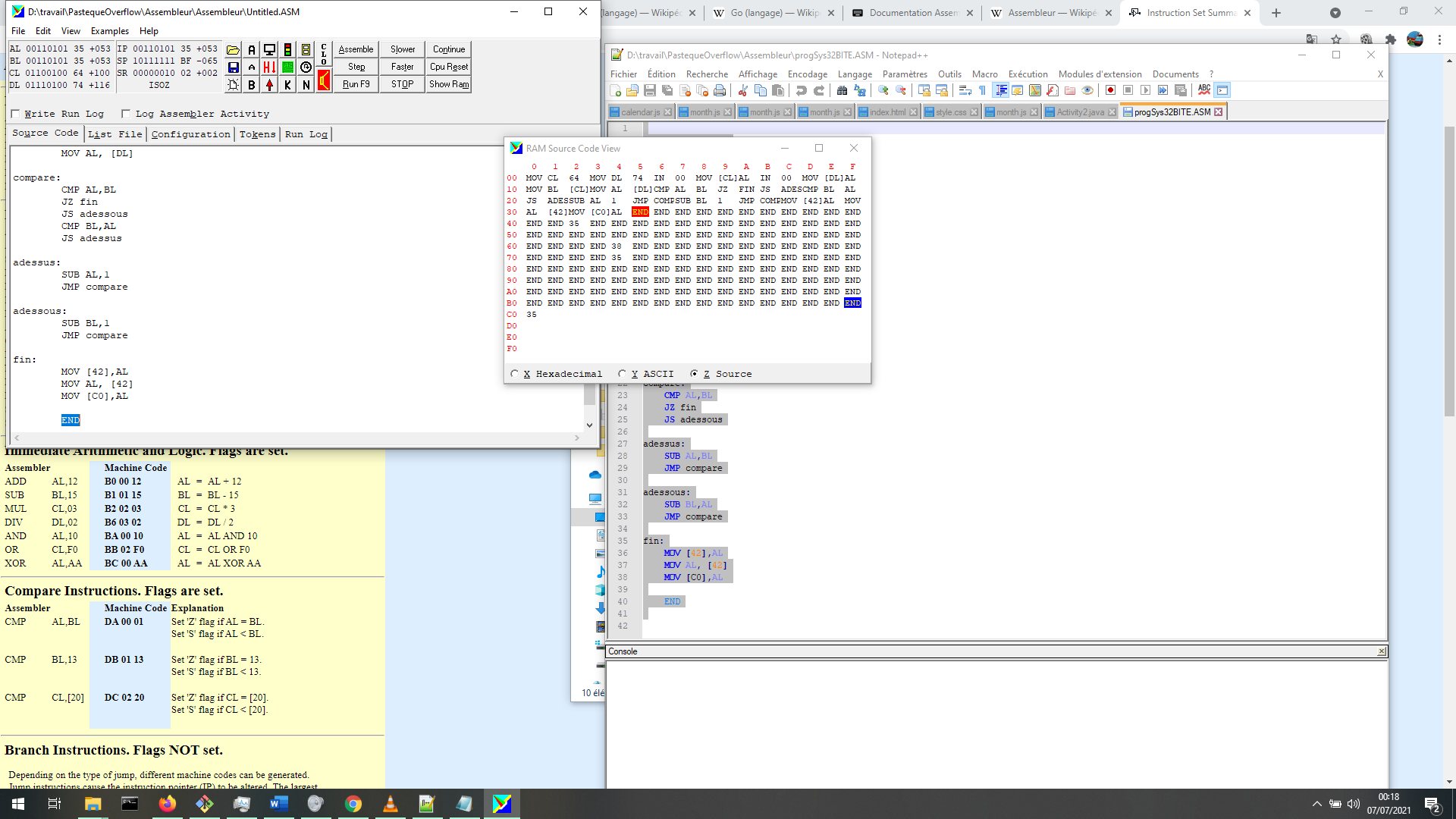
#### Choisir un langage

Aussi étrange que cela puisse paraître, pour créer un langage, il faut en choisir un avec lequel nous développerons notre propre langage. En effet, afin de traduire les instructions en véritables opérations pour l’ordinateur, il faut choisir un premier intermédiaire. Pour un langage théoriquement plus efficace, le choix doit se porter sur un langage dit « proche de la machine » ou « de bas niveau ». Cela signifie que nous utilisons un langage qui doit passer par peu de traductions différentes avant d’être compris par l’ordinateur.

Le langage le plus basique existant est l’assembleur et lui-même doit déjà être traduit deux fois avant d’être compris par la machine. D’abord converti en caractères ASCII (du chiffre 0 à la lettre f, soit 16 caractères différents), il est ensuite traduit en bits (0 ou 1), le langage courant de l’ordinateur qu’on appelle aussi le langage machine.

Comme il est inévitablement le parent de tous les langages de programmation existants, nous allons expliquer succinctement son fonctionnement afin de mieux comprendre ce dont tous les langages héritent.

Tout d’abord, pour l’assembleur, la mémoire se présente sous la forme de cases et chacune d’elle dispose d’une adresse. Cette adresse servira d’index de référence lorsqu’on voudra se servir de cette case en particulier.

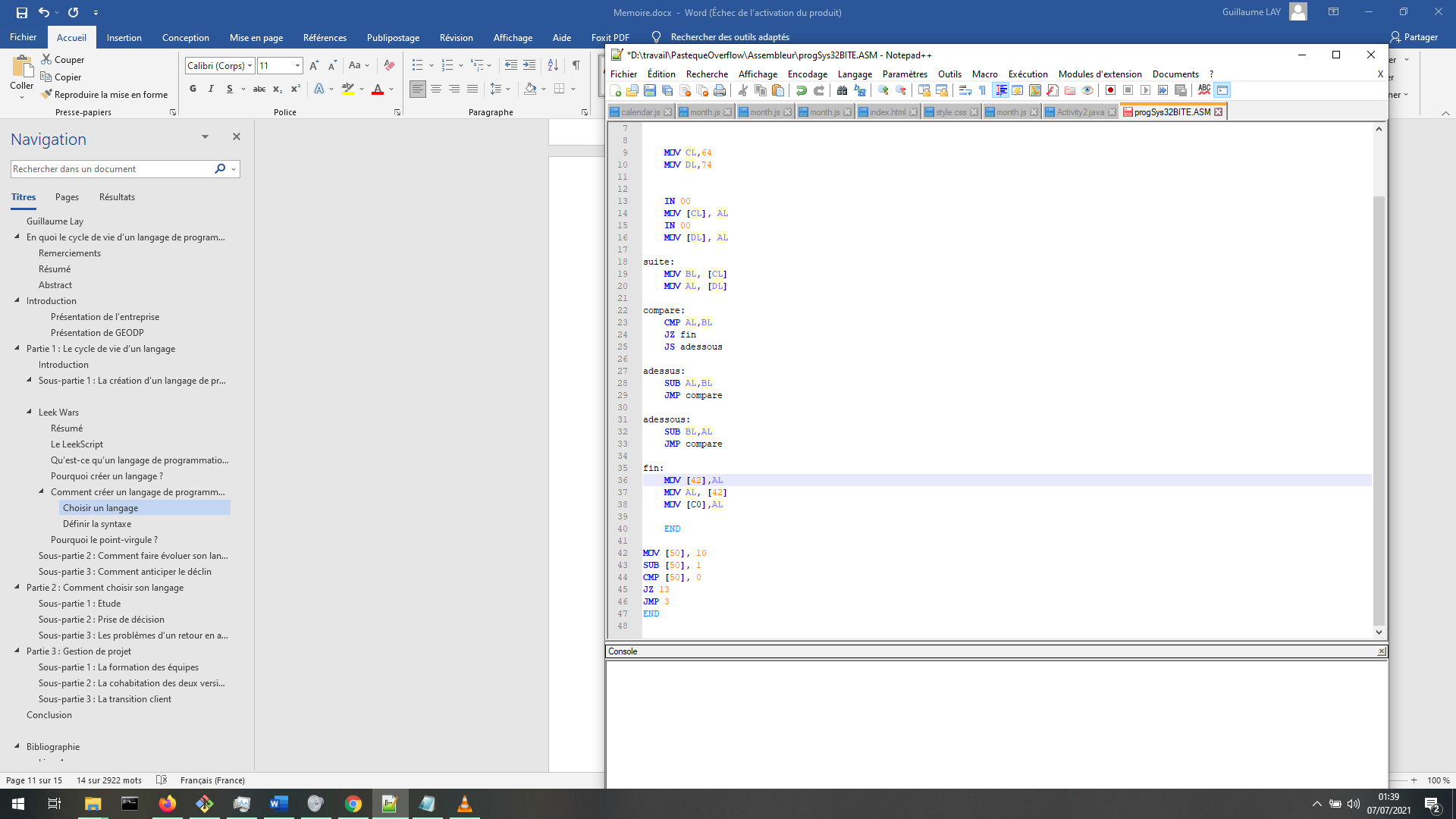
En assembleur, le programme à exécuter occupe lui aussi des emlacements de la mémoire. Dans l’exemple ci-dessous, le programme utilise les emplacements (0) à (35). Le reste des cases est une zone de mémoire disponible pour le stockage des valeurs durant l’éxécution.

On obesrve que chaque fragment d’instruction occupe une case mémoire. Par exemple, l’instruction « MOV CL 64 » (qui sert à enregistrer la valeur 64 dans une variable au nom « CL ») occupera trois cases. Le déroulement d’un programme se fait case par case. Un curseur se place sur une case et y circule une par une afin de réaliser les opérations dans l’ordre défini. L’ordinateur ne contrôle pas le débordement. Il est donc tout à fait possible d’enregistrer une valeur à un emplacement qui contient un fragment du programme.

A noter que certaines cases ne sont pas vides (64 et 74). C’est quelque chose qui arrive couramment en temps normal. En effet, lorsque l’ordinateur assigne un espace mémoire libre pour l’éxécution d’un programme, il peut y résider les restes de programmes précédents. Ce phénomène est très courant. Sur une autre échelle, lorsque l’on supprime un fichier du disque dur, les bits qu’il occupe physiquement sont simplement déclarés disponibles sans voir leur valeur changée. La zone de mémoire devient un espace réutilisable même si elle contient encore les informations du fichier. La réécriture est possible sur ces cases puisque la valeur qu’ils contiennent n’est plus utilisée.

Dans un langage à bas niveau comme l’assembleur, il est donc primordial de contrôler le contenu de chacun des emplacements mémoire que l’on utilise avant l’éxécution du programme. Il n’y a pas que l’assembleur qui requiert une telle précaution. En C par exemple, cette problématique est également présente.

Nous n’allons pas détailler l’intégralité des opérations possibles en assembleur mais il est primordial de connaître les plus fondamentales afin d’étudier les autres langages qui en héritent. Voici sonc quelques unes des opérations les plus importantes en assembleur :

* MOV {p0} {p1}: Permet de changer la valeur d’une case en fonction des deux paramètres (p0 et p1). Un paramètre peut être aussi bien une valeur qu’une adresse mémoire. Par exemple :
  + « MOV [42] 12 » enregistre la valeur 12 dans l’emplacement de mémoire 42
  + « MOV [42] [12] » enregistre la valeur de l’emplacement mémoire 12 dans l’emplacement mémoire 42
* ADD, SUB, DIV, MUL : Ce sont les opérateurs arithmétiques de base (addition, soustraction, division, multiplication).
* AND, OR, XOR : Ce sont les opérateurs logiques fondamentaux. Ils retournent soit vrai, soit faux, en fonction de leurs paramètres (p0 et p1).
  + {p0} AND {p1} : Si p0 et p1 sont vrais, alors il renvoie vrai, sinon il renvoie faux.
  + {p0} OR {p1} : Si p0 ou p1 est vrai, alors il renvoie vrai, si les deux sont faux il renvoie faux.
  + {p0} XOR {p1} : Si soit p0 soit p1 est vrai alors il renvoie vrai, sinon il renvoie faux. Il renvoie donc faux si les deux sont vrai.
* CMP : C’est l’opérateur de comparaison. Il compare la grandeur de deux éléments. Le résultat est soit Z soit S de cette façon :
  + CMP {p0} {p1} : Si p0 = p1 , renvoie Z. Si p0 < p1, renvoie S.
* JMP : C’est un opérateur qui permet de déplacer le curseur d’éxécution du programme à une certaine case mémoire.
* JZ, JS : Ces opérateurs sont directemepnt liés à CMP. Ils ont le même but que JMP mais ils tiennent compte du résultat de CMP enregistré en cache.
* END : Cet opérateur stoppe l’éxécution du programme.

Voici ci-contre un exemple de programme en assembleur. Il réalise un décompte de 10 à 0 en utilisant l’emplacement mémoire 50 pour stocker la valeur du décompte. Il est composé de six instructions différentes.

Si le résultat de la comparaison à 0 est Z (le décompte est égal à 0) alors on saute à l’instruction en case 13, soit l’instruction END. Sinon, on continue l’éxécution et on arrive au JMP qui nous renvoie à la case 3 du programme, la soustraction de 1 au décompte.

La manière d’écrire du code en assembleur est donc très austère. On reste très proche du langage machine et il est donc assez difficile d’envisager de coder de lourdes opérations avec ceci. C’est pourquoi lorsqu’on utilise un langage plus élevé comme le C, toutes ces questions d’allocation de cases de mémoire sont simplifiées. Même si on a à faire aux mêmes problématiques, l’écriture et la lecture du code sont plus accessibles.

#### Les variables et le typage

Lorsque l’on développe un code, on peut être amené à manier toutes sortes de données. Des entiers, des mots ou même des paragraphes entiers, l’information à traiter peut être de différentes formes. Pour faciliter la gestion de la mémoire, il existe ce qu’on appelle le typage. Un chiffre n’a pas besoin d’autant d’espace qu’un mot en mémoire.

On parle de variable lorsqu’on désigne un espace de mémoire identifié par un nom. Lorsque nous avons abordé l’assembleur, nous identifions déjà les espaces mémoire par une référence dans le tableau de dimensions 16 par 16. Le nommage permet de donner des noms plus parlants qu’un simple entier.

Il existe plusieurs types de données que l’on peut stocker dans une variable. Bien évidemment, la gestion de la mémoire peut varier d’un type à l’autre. Si on se réfère au C, il en existe deux catégories, les types primitifs et les types par référence.

Les types primitifs sont les types les plus basiques des données. Ils désignent des propriétés qui occupent relativement peu d’espace mémoire et qui peuvent être maniés facilement.

* Le booléen : Il n’a pour valeur que vrai ou faux. C’est le type le plus simple qui existe. En effet, il peut être représenté en binaire ( 0 = faux, 1 = true)
* Le caractère : Il a pour valeur une lettre, un symbole ou tout autre caractère alphanumérique.
* L’entier : Il a pour valeur un nombre entier.
* Le nombre flottant : Il a pour valeur un nombre avec des décimales.

Lorsque l’on traite avec des types primitifs, on utilise un accès par valeur. C’est-à-dire que on influe directement sur le contenu de la case mémoire. C’est à différencier avec l’utilisation de variables dite par pointeur ou par référence. Celle-ci concerne les formes de données plus complexes comme les tableaux et les listes. Dans ces cas d’usage, le traitement s’effectue sur l’adresse de la variable.

Par exemple :

Int entier\_1 = 1 ;

Int entier\_2 = entier\_1 ;

(Ajouter schéma de mémoire)

Int[] tab\_1 = [1,2,3] ;

Int[tab\_2 = tab\_1 ;

On différencie donc la variable de son adresse. D’ailleurs, la gestion des valeurs inutiles peut différer selon les langages. Lorsqu’une valeur n’est plus référencée par aucun pointeur, son espace mémoire doit être libéré. La plupart des langages le font automatiquement via ce qu’on appelle le ramasse-miette mais pour d’autres, il faut penser à libérer la mémoire manuellement. En cas d’oubli, cela peut entraîner une grosse latence du programme voire pire, une fuite de mémoire. C’est donc quelque chose à surveiller lorsque l’on traite des valeurs volumineuses.

En fonction du langage, il peut y avoir un certain degré de typage. Pour certains langages, on doit déclarer le type de données que l’on va ranger dans une variable et pour d’autre il n’en est pas question. Globalement, il y a trois degrés de typage : fort, modéré et faible.

Dans le typage fort, le type doit être déclaré à chaque déclaration de variable et il est impossible d’en changer par la suite.

Int entier = ‘’Bonjour’’ ; // Impossible

Int entier = 1 ;

Entier = ‘’Bonjour’’ ; // Impossible

Un typage fort permet d’éviter de nombreuses erreurs mais peut paraître assez rigide à l’utilisation. Ce système est utilisé par le C et le C++ qui y ajoutent une petite subtilité, le type void. Une variable de type void ne connaît son type que lorsqu’elle reçoit son contenu. Ce fonctionnement, c’est celui du typage modéré.

Dans le typage modéré donc, le préfixe pour toutes les variables est le même mais une fois qu’elles reçoivent un contenu, elles adoptent son type et ne peuvent en changer.

Var entier = 12 ;

Var chaine = ‘’Bonjour’’ ;

Chaine = 12 ; //Impossible

Entier = chaine ; // Impossible

Ce mode de typage a été adopté par le C#. Son avantage par rapport au typage fort est principalement lié au confort de rédaction du code.

Pour finir, un langage au typage faible signifie un langage ou toutes les variables peuvent prendre n’importe quelle valeur. On peut très bien ranger un entier puis un tableau et ensuite un objet dans le même nom de variable. Il permet une très grande souplesse dans le code ce qui peut amener des risques de corruption des informations. Il est notamment utilisé par le PHP, le JavaScript et d’autres langages de scripting comme le Shell.

Le LeekScript que nous prenons comme exemple dans cette partie adopte un typage faible. Le but du jeu étant de faire découvrir la programmation, ce mode peut être considéré comme le plus simple à appréhender au début. Malgré cela, il reste celui qui sécurise le moins la cohérence des données et peut être source de plus d’erreurs.

#### Fonctions et blocs de code

Lorsqu’un programme est amené à répéter une même tâche, il est judicieux de se servir de fonctions. Une fonction (ou méthode) est une suite d’instructions, comme un sous-programme finalement. Il pourrait s’apparenter à notre décompte fait en assembleur précédemment puisque nous répétions une même tâche jusqu’à un certain point. En fait, cela correspondait davantage à une boucle « while ».

Là ou une boucle ou un bloc conditionnel (groupe d’instructions accessibles si une condition est valide) pourrait s’apparenter à une sorte d’amélioration du JMP de l’assembleur, une fonction est plus complexe. Elle peut notamment disposer de ses propres paramètres.

A la création d’un langage, il faut donc penser à implémenter des fonctions natives pour en permettre l’utilisation. Si on prend l’exemple du LeekScript, toute une panoplie de fonctions est disponible au joueur afin qu’il puisse récupérer les informations capitales sur le plateau de jeu. Les fonctions implémentées d’office dans un langage peuvent être plus ou moins nombreuses.

Dans de nombreux cas, on préfèrera simplifier le nombre de fonctions natives et ranger les autres dans des paquets joignables au programme appelés librairies. Cela permet au développeur de ne prendre que ce dont il a besoin lorsqu’il crée son programme.

#### POO

Un des concepts fondamentaux du développement de logiciels est la programmation orientée objet (POO). Cela consiste en l’utilisation de composants, les objets (ou classes), qui sont des variables avec de multiples propriétés. Un objet peut disposer de variables et des fonctions propres à lui-même que l’on appelle ses attributs. C’est donc quelque chose de plus complexe à manier que des types primitifs. Il est donc bien évidemment à classer dans les types par référence.

Tous les langages ne supportent pas forcément la programmation orientée objet. C’est d’ailleurs la différence fondamentale entre le C et le C++.

La programmation orientée objet ajoute aussi la notion d’encapsulation. C’est le principe de restreindre l’accès aux attributs d’une classe par une autre. Cette pratique permet de sécuriser leur utilisation afin que des valeurs ne soient modifiées que par les composants voulus.

#### Définir le mode d’éxécution

Il y a majoritairement deux sortes de langages de programmation. Soit ils sont compilés, soit ils sont interprétés. La différence se situe dans leur mode d’éxécution et il en découle une divergence majeure dans leur utilisation.

Un langage compilé a besoin d’un compilateur qui convertira le code en un éxécutable utilisable de façon autonome. C’est-à-dire que le code est traduit en langage machine et l’ordinateur sera donc capable d’en éxécuter les instructions sans avoir à passer par un traducteur tierce par la suite. Un compilateur effectue donc une vérification de la conformité du code avant de le transformer. Il est donc un outil idéal pour le développeur pour détecter ses erreurs.

Par contre, là ou le compilateur est tout à fait capable de détecter des erreurs de syntaxe ou de typage, il n’est en revanche pas imperméable aux erreurs d’éxécution. Si une instruction est syntaxiquement correcte mais est erronée dans sa logique, elle passera l’étape de la compilation.

Un langage interprété a lui besoin d’un interpréteur pour être éxécuté. La différence entre un compilateur et un interpréteur est que l’interpréteur, lui, compile le code à la volée. C’est-à-dire que les instructions ne sont vérifiées que lorsque le programme est lancé.

Pour illustrer de façon concrète, nous allons prendre la situation suivante : « Georges utilise la porte d’entrée d’un restaurant ». Voici plusieurs cas illustrés entre un langage compilé et un langage interprété.

|  |  |  |
| --- | --- | --- |
| Suite d’instructions | Résultat si compilé | Résultat si interprété |
| Georges ouvre la porte  Georges passe la prote | Erreur à la compilation. La variable « prote » est inconnue | La porte est ouverte puis il y a une erreur. La variable « prote » est inconnue |
| Georges passe la porte  Georges ouvre la porte | Pas d’erreur à la compilation mais plantage lors de l’éxécution. Georges ne peut pas passer la porte car elle est fermée | Erreur à l’éxécution, Georges ne peut pas passer la porte car elle est fermée |
|  |  |  |
|  |  |  |

En règle générale, comme un code compilé est déjà traduit en un langage de bas niveau, il est plus performant à l’éxécution qu’un code interprété qui lui doit démarrer son interpréteur à chaque lancement. Avec l’amélioration des processeurs, cela est de moins en moins vrai. De plus, les langages interprétés, d’abord boudés pour leur lenteur, se retrouvent mis en avant ces dernières années grâce à leur rapidité de mise en place.

Tout comme le C, le C++ ou le Java, le LeekScript est un langage compilé. En effet, il a été codé en Java et dispose donc d’un compilateur dans ce même langage. Le choix du Java comme technologie pour créer un langage peut sembler inadapté pour un tel projet étant donné qu’il s’agit d’un langage plutôt lourd. Même s’il est compilé, le Java doit faire appel à un programme tierce lors de son éxécution ce qui pourrait provoquer des lenteurs.

Cela dit, le fait que le compilateur du LeekScript soit conçu dans un langage qui demeure facile à lire pour un développeur va nous faciliter sa compréhension. Nous allons pouvoir analyser son fonctionnement avec plus d’aisance.

#### Définir la syntaxe

Nous avons défini tous les traits de notre langage mais il nous reste à savoir comment nous l’écrivons, comment le code va s’articuler visuellement. C’est ce que nous appelons la syntaxe du code. La plupart des langages de programmation sont pensés de la même manière, avec des variables, des instructions et des fonctions. Leur différence se situe souvent dans leur syntaxe.

Etant donné que les langages de programmation sont créés par des humains et pour des humains, il n’est pas étonnant de constater la grande similitude entre la construction syntaxique d’une instruction à un ordinateur et une phrase destinée à un humain. On peut comparer la structure d’une instruction à la construction d’une phrase sujet verbe complément.

Var entier\_1, entier\_2 = 1 + 1 ; // Sujet verbe et compléments

C’est maintenant une convention dans la plupart des langages de programmation, le point-virgule termine une instruction. Le choix de ce caractère n’a rien d’un hasard. En effet, pour éviter toute ambigüité sémantique, il était nécessaire d’employer un caractère n’ayant aucun sens mathématique. Ensuite, il fallait qu’il remplisse la fonction de terminateur d’état. Là où la virgule, dans le langage courant, lie deux expressions sans forcément les séparer (Exemple : « Deux individus, l’un d’eux est roux »), le point-virgule a davantage le rôle de séparateur. Aussi, le point était inutilisable puisque servant déjà à désigner les décimaux.

Le choix du point-virgule comme terminateur d’instruction est un débat qui eut notamment lieu aux débuts de l’informatique. Cette décision était notamment défendue par les chercheurs Gannon et Horning. Aujourd’hui, cette règle s’est installée comme convention syntaxique. Même si elle n’est pas appliquée dans tous les langages et est parfois remplacée par le saut de ligne, elle fait partie des habitudes des développeurs.

La syntaxe du LeekScript étant très semblable à celle du JavaScript, nous n’aurons pas besoin de l’analyser en profondeur. Cependant, elle aura toute son importance lorsqu’il faudra aborder le compilateur.

#### Compilateur

Enfin, lorsque

## Sous-partie 2 : Comment faire évoluer son langage

Etaler les mises à jour

Gérer la Retro compatibilité

## Sous-partie 3 : Comment anticiper le déclin

Angular

# Partie 2 : Comment choisir son langage

## Sous-partie 1 : Etude

Competences de l’équipe

Etude de marché

Etude du langage

Plateforme cible

Contexte du projet et cible

## Sous-partie 2 : Prise de décision

## Sous-partie 3 : Les problèmes d’un retour en arrière (Kendo et le Lierre)

Les raisons d’un retour arrière

Décrocher Kendo

Conséquences

# Partie 3 : Gestion de projet

Cas d’étude GEODP V1 et V2

La transition entre la V1 et la V2

Gérer la transition (Client/ Technique)

Faire le parallèle entre LeekWars et GEODP

## Sous-partie 1 : La formation des équipes

Bien communiquer sur les avantages de la nouvelle version et au BON MOMENT

Migrer le personnel de la V1 à la V2

Former techniquement

## Sous-partie 2 : La cohabitation des deux versions

Différence d’approche entre la gestion des versions

## Sous-partie 3 : La transition client

# Conclusion

# Bibliographie

## Livre 1

## Livre 2

## Livre 3

Déroulé méthodologique :

### Présentation du thème :

Durant tout mon parcours informatique, dès que je parlais de langages de programmation à des non-initiés, j’ai toujours eu droit à cette interrogation : « Mais pourquoi y en a-t-il autant ? ». Même si j’ai une très vague idée de cette raison, j’ai trouvé intéressant de répondre une bonne fois pour toutes à cette question. C’est un sujet qui me passionne et en l’abordant, il me permettra de faire un bilan complet de mes compétences acquises jusqu’à présent.

Ma thématique est assez vague ce qui me permettra d’aborder un large panel de sujets différents.

Liste des personnes rencontrées : 0

### Calendrier M2