

Owner Programming Language

Merci de choisir Owner en tant que votre langage de développement !

Owner est un langage de programmation créé par **Aneto Enterprise Inc.** Et publier pour vous aider à développer votre propre langage de programmation Compiler ou Interpréter comme son nom l'indique (**Owner : Propriétaire**).

Il est exploité dans le but éducatif pour permettre aux étudiants licencié d'avoir la connaissance particulière de création d'un langage de programmation Interprété ou Compiler.

Avant de concevoir un langage de programmation, vous devriez savoir le langage cible tels que : **ASM, C ou C++**. A vous de choisir sur quel langage que vous allez utiliser comme langage cible et vos codes sera compiler sur ce langage.

Il n'est pas difficile de créer un langage de programmation. Tout dépend en réalité de sa complexité grammaticale. Aussi créer un langage ne nécessite aucune connaissance technique en programmation. Il faut surtout être logique et très rigoureux. La grammaire d'un langage de programmation peut être enrichie au fil du temps. On peut commencer par créer quelque chose de simple puis faire évoluer notre langage en implémentant des mécanismes de fonctions par exemples, de classes, etc...

Beaucoup de langages informatiques évoluent encore aujourd'hui comme le Java, le C++, le PHP, CSS ou encore le HTML.

Développer son propre langage de programmation nécessite bien évidemment une connaissance dans le langage dans lequel on va l'écrire ainsi que le langage cible. Dans ce cours, nous allons apprendre à écrire notre propre langage de programmation. Il nécessite donc quelques commandes en programmation du langage **Owner**.

Quel est l'intérêt de créer un langage de programmation ?

Il existe plusieurs raisons de vouloir créer un langage de programmation.

- La première est de s'amuser tout simplement. Ensuite, parce qu'on a en tête une syntaxe que l'on souhaite utiliser dans nos programmes. On ne veut pas être soumis à la syntaxe des autres langages et utiliser un vocabulaire qui nous est propre.

- Une autre raison que je vois aussi, c'est la volonté de cacher son code. Un langage que l'on ne comprend pas est synonyme de code secret. Si une seule personne s'invente un langage et qu'il utilise, personne dans son entourage le comprendra. Imaginez une organisation sensible qui souhaiterait protéger ses algorithmes. Elle a à mon avis tout intérêt à développer son propre langage et ne le dévoiler à personne.

Ainsi seulement les personnes connaissant le langage comprendront ce qu'ils écrivent. Et leur code marchera comme n'importe quel autre code parce qu'ils auront le compilateur spécifique au langage. A l'inverse, cela peut être une stratégie purement commerciale. Les langages propriétaires obligent aux utilisateurs d'être dépendants des outils payants pouvant exécuter les programmes.

- Une autre raison que j'ai trouvée est que l'on pense pouvoir améliorer, ou en tout cas apporter des modifications sur la syntaxe d'un langage déjà existant pour le rendre généralement plus simple à écrire.

Mais qu'est-ce que réellement un langage de programmation ?

Le processeur ne comprend en fait qu'un seul langage qui est le langage binaire appelé aussi le langage machine. Ce langage a la particularité d'avoir un alphabet extrêmement faible, le plus faible qu'il puisse y avoir. Il ne contient en effet que 2 lettres : le '1' et le '0'. Tout ce qui se trouve sur votre ordinateur est traduit en 1 et 0. Ainsi quand un processeur exécute un programme, il traite une série plus ou moins immense de 1 et de 0.

Il est évident que programmer en binaire est une tâche assez compliquée. D'une part parce qu'il nécessite de recopier pour une instruction, son code binaire associé. Ensuite le langage machine est différent d'une famille de processeur à l'autre sans parler de l'architecture.

D'autre part, écrire en binaire est très long et source évidente d'erreurs. Car oui l'Homme commet des erreurs. Pour pallier à ce problème, des informaticiens ont inventé un langage qui s'appelle assembleur ou ASM et qui en bref associe des symboles aux combinaisons de bits. La traduction du langage assembleur vers le langage machine est effectuée par un Assembleur.

Le langage assembleur est aussi spécifique à l'architecture du processeur sur lequel on souhaite exécuter le programme mais il a l'avantage d'être largement plus facile à lire et à écrire que le binaire.

Pour répondre précisément à la question du sous-titre, un langage de programmation est un compilateur ou un interpréteur.

Un compilateur est un traducteur d'un langage source vers un langage cible. Le processus de traduction appelé compilation se fait théoriquement en plusieurs phases :

- Une phase d'analyse de composant (Composeur), il identifie les mots et s'assure qu'ils sont compris dans le vocabulaire du langage
- Une phase d'analyse syntaxique, il s'assure que la syntaxe est correcte autrement dit si la suite de mots écrite est dans le bon ordre.
- Une phase d'analyse sémantique, il s'assure du sens des phrases. Par exemple si je dis "Je suis plus vieux que mon grand-père.", syntaxiquement c'est correcte, ma phrase ne comprend aucune erreur grammaticale mais sémantiquement parlant elle n'a aucun sens.

Les langages interprétés :

On exclut les langages scripts qui sont des langages interprétés.

On peut citer : Le PHP, Python, JavaScript, SQL et j'en passe sont des langages qui sont compris par des interpréteurs ou interprètes.

Le processus de traduction est toute fois le même, seulement dans ce cas-là la compilation est faite dynamiquement.

L'interpréteur analyse, traduit et exécute aussi tôt l'instruction. Et il fait ceci instruction par instruction. Ainsi on peut dire que la compilation et l'exécution sont faites simultanément. C'est la différence avec les langages compilés, d'où le programme peut être exécuté une fois seulement que la totalité du code a été compilé. L'avantage des langages scripts est la portabilité des programmes car ils s'exécuteront tant que la machine possède les interpréteurs spécifiques à son architecture. Alors qu'un programme écrit en **C** par exemple, nécessite d'être recompilé si on change de machine ayant une architecture différente.

Les Shell Unix (sh, bash, csh...) sont de très bons exemples d'interpréteurs. Ils interprètent des commandes systèmes généralement depuis un terminal.

On exclut aussi généralement tous les autres langages informatiques qui ne sont pas des langages de programmation. Nous avons notamment les langages de description des données, de représentation des données comme le XML, SVG, MathML, XSL, DTD, Relax NG, HTML, JSON et j'en passe...

Ce sont pour la plupart des standards définis, des formats de données facilitant l'échange et l'interopérabilité. Nous avons aussi des langages de modélisation

comme UML qui a la particularité d'être graphique. Néanmoins, ce n'est pas parce que ce ne sont pas des langages que l'on compile qui ne sont pas compilables. On peut tout à fait imaginer écrire des instructions de programmation en XML puis ensuite créer un compilateur qui va lire le fichier XML et produire un fichier exécutable.

Le cas Java

Le langage Java est assez particulier. C'est un langage compilé qui, à la compilation, donne un fichier exécutable sur n'importe quelle machine. La compilation d'un programme écrit en Java par javac (compilateur Java) donne du byte code java (une sorte de langage assembleur Java on va dire). Et on n'a plus besoin de recompiler le code à chaque fois qu'on change d'architecture à la différence du C ou du C++.

Seulement s'il est si portable que cela, c'est que derrière il y a en fait une machine qui lit ce byte code java généré par le compilateur Java. Cette machine est la machine virtuelle Java, souvent abrégé JVM. En fait on compile du Java en une sorte de langage assembleur (byte code Java) par javac pour que ce soit ensuite interprété par la JVM. JVM, qui elle est bien spécifique à l'architecture de la machine sur laquelle elle se trouve. Java est ainsi cross plate-forme (indépendant de la machine).

Bas niveau et haut niveau

Plus le langage de programmation est proche du langage machine, plus il est bas niveau. L'assembleur est donc un langage très bas niveau car c'est le langage machine avec les instructions binaires qui sont substituées par des symboles. Le langage **C** Est un langage bas niveau mais plus haut que l'assembleur.

Le langage que vous allez créer sera quant à lui un langage de plus haut niveau que le langage C.

Pourquoi ?

Le langage que nous allons créer ici sera compiler en **C** qui lui-même devra être recompilé pour obtenir au final un exécutable.

L'exécutable produit sera donc finalement un exécutable provenant du langage C et non pas du langage qu'on va créer. Mais notre langage sera plus facile à écrire pour nous que le **C**.

On retiendra que les langages de haut niveau sont faits pour faciliter le développement en rendant transparent des mécanismes plus ou moins complexes comme la gestion de la mémoire, les registres, les pointeurs, les entrées et sorties que le développeur n'a plus à se soucier.

Cependant un langage de haut niveau gagne en productivité mais pas forcément en performance. Tout dépend en fait de la qualité du code généré par votre propre langage de programmation.

Pour commencer, ce langage de programmation **Owner** est composé de trois étapes (**Composeur, Lexique et Syntaxe**).

NB : Nous avons utilisé quelques couleurs de syntaxe pour vous faciliter la vue et de bien comprendre le langage Owner.

- Les textes en vert ou noir, définit les commandes a utilisé.
- Les textes en rouge entre guillemet définit les caractères que vous pouvez modifier à votre guise.

I. COMPOSEUR

Le composeur Owner est composé des 3 couches qui va nous permettre d'effectuer les appels de nos commandes définit.

1. Flux :

Le flux permet de faire appel à nos fichiers de sortie final. La commande qui nous permet de définir notre flux est **Owner_Flux**.

Exemple :

```
owner_flux="MonFichier";
```

2. Génération :

Définit les méthodes de génération des codes et la commande en question est **Owner_Methode**. Vous pouvez définir les commandes qui seraient générer a l'entête, au centre et en bas de votre langage. Cette méthode utilise deux propriétés qui vont activé ou désactivé une méthode de génération des codes.

- **Definit** : Permet d'activer la méthode
- **NonDefinit** : Désactive la méthode.

Exemple :

```
owner_methode="MethodeDebut",definit;
```

```
owner_methode="MethodeCentrer",nondefinit;  
owner_methode="MethodeFin",definit;
```

3. Séquence :

Définit les séquences de code possibles pour notre l'Arbre de Syntaxe.
C.à.d. Chaque séquence de code est associé à un numérique. Elle utilise la commande **Owner_Sequence**.

NB : Ces numériques peut être commencé par le nombre 2.

Pourquoi commencer par le nombre 2 ?

Parce que par défaut Owner s'occupe des deux nombres qui sont de 0 à 1.

Exemple :

```
owner_sequence= "MonIf          2 "  
owner_sequence= "MonThen       3 "  
owner_sequence= "MonElse       4 "  
owner_sequence= "MonEndIf      5 "  
owner_sequence= "TousCaracteres 6"
```

II. LEXIQUE

Sur ce point nous avons comme chapitres :

II.I Les chaine des caractères :

Les commandes qui permettent de détecter les chaines des caractères sont belle et bien **Owner_Caracteres**, **Valeur**, **Tous_Caracteres** et **Owner_Caracteres_Appel**. Ces commandes permettent à votre langage de détecté une chaine des caractères pour l'enregistrer a la mémoire et de l'utiliser après.

- **Owner_Caracteres** : Permet de définir une variable qui va contenir les caractères definit par le programmeur.
- **Valeur** : Definit la valeur détectée en mémoire.
- **Tous_Caracteres** : Sauvegarde tous les caractères détectés.
- **Owner_Caracteres_Appel** : Appel la variable sauvegardé pour enfin l'utilisé.

Exemple :

Dans cet exemple, la variable qui va contenir nos caractères est **MesCaractes** et pour nos commentaires est **Commentaires**.

Exemple :

```
owner_caracteres="MesCaractes", valeur=tous_caracteres  
owner_caracteres="Commentaires", valeur=owner_commentaires
```

II.II Appel de la variable sauvegardé :

1. Exemple (Analyse de caractères a utilisé):

Nous venons de voir comment définir et sauvegardé une chaine des caractères dans une variable. Maintenant il est temps de la vérifié et l'utilisé à notre langage. Dans cet exemple on va récupérer tous les caractères qui se trouvent entre petit guillemet.

```
owner_caracteres_appel="MesCaractes"  
/* Si on détecte le premier petit guillemet, on débute la récupération des  
caractères. */  
si: "" alors:  
debut("MesCaractes");  
finisi;  
/* Tant que nous n'avons pas arrivé au dernier petit guillemet, ne rien  
faire et on continue la récupération. */  
si: "<MesCaractes>" alors:  
nerienfaire;  
finisi;  
/* Si nous arrivons à détecter les caractères et le dernier petit guillemet,  
on initialise la récupération. */  
si: "<MesCaractes>"+"" alors:  
debut:(owner_initialise);  
finisi;  
/* Si Ya rien à détecter, avant de sauter la ligne on récupère nos  
caractères. */  
si: "<MesCaractes>"+"." alors:  
owner_recupere_tous;  
/* On retourne les caractères dans la variable qu'on à definit  
précédemment. */  
retour:"TousCaracteres";
```

finsi;

2. Exemple (Définition de commentaire du langage):

On utilisera le même exemple pour définir les commentaires. La définition de commentaire permettra aux programmeurs à qui utilisera votre nouveau langage de se référencé à une ligne des codes facilement.

```
owner_caracteres_appel="Commentaires"
/* Si on détecte le <!--, on sait que c'est le debut d'un commentaire. */
si: "<!--" alors:
debut:("Commentaires");
finsi;
/* Tant que nous n'avons pas arrivé au dernier >, on passe à la ligne
suivante. */
si: "<Commentaires>" alors:
owner_ligne ++;
finsi;
/* Si nous arrivons à détecter les caractères et le dernier >, on initialise
les commentaires. */
si: "<Commentaires>"+">" alors:
debut:(owner_initialise);
finsi;
/* S'il Ya rien à détecter, avant de sauter la ligne on saute directement.
*/
si: "<Commentaires>"+"." alors:
finsi;
```

II.III Ouverture, Vérification et Fermeture des commandes :

Dans ce stade, nous allons maintenant définir nos commandes du langage que les programmeurs vont commencer a utilisé. Si le programmeur saisi une commande non structuré à ce niveau, directement un message d'erreur se déclenche et s'affiche au console. Ce stade autrement appelé un Débogueur(Débogue).

Exemple :

```
owner_verification:
{"If"}      {retour: owner_ecoute: "MonIf;"}
```



```

{"Then"} {retour: owner_ecoute: "MonThen;"}
{"Else"} {retour: owner_ecoute: "MonElse;"}
{"End If"} {retour: owner_ecoute: "MonEndIf;"}
owner_verification;

```

NB : On voit que, nous avons défini les vérifications de tous nos commandes sauf ceux de **MesCaractes** et **Commentaires** parce qu'elles sont utilisées automatiquement.

II.VI Réponse :

Maintenant il est temps de définir le message d'erreur qui se déclenchera lorsque les programmeurs vont saisir une commande qui ne sera pas à la portée de notre nouveau langage. Cette étape possède comme commandes :

- **Owner_Reponse** : Elle permet d'ouvrir ou de fermer la réponse.
- **Owner_Message** : Permet d'afficher un message sur la console.
- **Owner_Ligne** : Cette commande détecte la ligne d'où une erreur est survenue.
- **Owner_Code** : Affiche le code d'erreur qu'elle a détecté.
- **Owner_Nombre_Caracteres** : Affiche le nombre des caractères qu'on a détecté à cette erreur.

Exemple :

```

owner_reponse:
owner_message("\n\tErreur a la ligne %d :\n\t'%s' n'est pas reconnu en
tant que commande interne ou externe, un programme executable ou
un fichier des commandes.\n\tComporte %d
lettre(s)\n\t",owner_ligne,owner_code,owner_nombre_caracteres);
owner_erreur_composeur=activer;
owner_reponse;

```

III. SYNTAXE

Maintenant nous allons nous assurer du sens des phrases. Par exemple comme nous l'avons dit aux explications précédentes, si on saisit le texte "**Je suis plus vieux que mon grand-père.**", syntaxiquement c'est correcte, mais la phrase ne comprend aucune erreur grammaticale mais sémantiquement parlant elle n'a aucun sens.

Alors c'est ici qu'on va définir nos grammaires syntaxiques et nos sémantiques parce que nous venons déjà de définir nos composeurs.

1. **Les Variables** : Nous avons 2 types des variables Owner qui sont la variable interne et externe. Les variables internes sont les variables qu'on va nous même définir pour l'utiliser à notre nouveau langage de programmation. Tandis que les variables externes sont les variables déjà utilisables par Owner.

NB : Les variables externes sont obligatoires parce que si vous ne l'utilise pas, la détection des lignes et la détection d'une erreur du composeur ne marchera pas.

a) **Exemple** :

```
owner_bool "mavariante1=false";  
owner_bool "mavariante2=false";
```

b) **Exemple** :

```
owner_bool_externe=owner_ligne;  
owner_bool_externe=owner_erreur_composeur;
```

2. **Table de hachage** :

Notre table de hachage a comme structure des Variables qui ont comme membre le type et un pointeur générique vers les valeurs qu'on va définir. Les commandes qui facilitent l'appel des hachages sont :

- **Owner_Table** : Définit le debut ou la fin d'une table de hachage.
- **Owner_Table_Type** : Type des caractères a utilisé.
- **Owner_Table_Taille** : Obtient la taille des caractères détectés.
- **Owner_Variables** : La valeur a utilisé à notre type de table.

Exemple :

```
owner_table:  
owner_table_type=owner_variable;  
owner_table_taille=owner_caracteres;  
owner_table;
```

3. Union :

L'union dans Owner est utilisée pour typer nos composeurs, syntaxes ainsi que nos sémantiques. Ici nous allons déclarer une union avec trois types : nombre de type Int, texte de type pointeur pour les caractères (char*) et nœud d'arbre syntaxique (AST).

- **Owner_Union_Longeurs** : Definit le type de la taille.
- **Owner_Union_Caracteres** : Type de caractères d'union.
- **Owner_Union_Methodes** : La méthode a utilisé pour unir nos phases.
- **Owner_Union_Texte** : Type des caractères texte pour une union.
- **Owner_Union_Noeud** : Valeur a utilisé pour la méthode d'une union.

Exemple :

```
owner_union:  
owner_union_longeurs=owner_nombre;  
owner_union_caracteres=owner_texte;  
owner_union_methodes=owner_noeud;  
owner_union;
```

4. Importation des expressions :

Ce stade est très facile et il va nous permettre d'importer nos composeurs pour enfin les utiliser par ordre syntaxique ou sémantique.

- **Owner_Expressions** : Definit le debut ou la fin d'une (des) expression(s).
- **Owner_Expressions_Valeur** : Definit la valeur à ajouter à l'expression.

Exemple :

```
owner_expressions:  
owner_expressions_valeur="MonIf"  
owner_expressions_valeur="MonThen"  
owner_expressions_valeur="MonElse"  
owner_expressions_valeur="MonEndIf"  
owner_expressions_valeur="MesCaractes"  
owner_expressions;
```

NB : Tous nos composeurs ont été importé sauf celui de commentaire parce qu'on ne l'utilisera pas. Les commentaires dans un langage de programmation sont inutilisables !

5. Définition des grammaires syntaxiques et sémantiques :

Maintenant il temps que nous définition nos syntaxes et sémantiques.

1. Exemple (Définition des syntaxes) :

```
owner_ss:
owner_ss_code="MonIf" {
  "mavariab1=true;"
owner_message(owner_fichier_sorti "MonFichier", valeur "if(");
}
owner_ss_code="MonThen" {
owner_message(owner_fichier_sorti "MonFichier", valeur "{");
  "mavariab2=true;"
}
owner_ss_code="MonElse" {owner_message(owner_fichier_sorti
  "MonFichier", valeur "}else{");}
owner_ss_code="MonEndIf" {owner_message(owner_fichier_sorti
  "MonFichier", valeur "}");}
owner_ss;
```

Pourquoi nous avons donné nos variables mavariab1 et mavariab2 la valeur de true ?

Parce qu'elles nous aideront à effectuer de vérification sémantique.

2. Exemple (Définition des sémantiques) :

```
owner_ss:
owner_ss_code="MonIf"
+ "MesCaractes"
+ "MonThen"
{
si: "mavariab1==true || mavariab2==true" alors:
owner_erreur_semantique=desactiver;
autres:
owner_erreur_semantique=activer;
```

```
finsi;  
}  
owner_ss;
```

6. Définition des fichiers et extensions de sortie :

Nous sommes arrivés à la fin des phases Owner. Alors nous devrions indiquer sur quel répertoire de sortie notre langage va générer les sources codes compilés. Comme commandes nous avons :

- **Owner_Sortie** : Définit le début ou la fin de(s) fichier(s) de sortie.
- **Owner_Ouvrir** : Permet d'ouvrir ou de créer un fichier sur le répertoire local.
- **Owner_Fermer** : Effectue la fermeture d'un fichier une fois ouvert ou créé.
- **Owner_Fichier** : Permet la fermeture complète d'un flux de fichier.
- **strdup(argv[1])** : Cette commande est modifiable par son numéro d'argument. Vous pouvez changer le nombre d'argument comme bon vous semble mais ici nous allons définir comme argument [1]. Owner a 3 arguments qui sont le [1], [2], [3].

NB : La commande d'argument nous permet de détecter le chemin complet relatif ou absolu d'un fichier qu'on a ouvert avec notre langage.

Exemple :

```
owner_sortie:  
owner_variable="MonFichierEntree";  
"MonFichierEntree", valeur ("strdup(argv[1])");  
owner_variable="MonFichierSorti";  
"MonFichierSorti", valeur ("C:/Resultat.c");  
"MonFichier", valeur owner_ouvrir ("MonFichierSorti");  
owner_fermer("MonFichier");  
owner_fichier();  
owner_sortie;
```

Exemple complet :

Notre exemple utilisera le langage cible **C**. C.à.d. il va recompiler nos codes du langage crée vers le langage **C**.

Source Codes :

```
/* Debut Définition de flux de notre fichier de sortie final */
```

```
owner_flux="MonFichier";
```

```
/* Fin Définition de flux de notre fichier de sortie final */
```

```
/* Debut Définition des méthodes */
```

```
owner_methode="MethodeDebut",definit;
```

```
owner_methode="MethodeCentrer",nondefinit;
```

```
owner_methode="MethodeFin",definit;
```

```
/* Fin Définition des méthodes */
```

```
/* Debut Définition des séquences */
```

```
owner_sequence="MonIf      2"
```

```
owner_sequence="MonThen    3"
```

```
owner_sequence="MonElse    4"
```

```
owner_sequence="MonEndIf   5"
```

```
owner_sequence="TousCaracteres 6"
```

```
/* Fin Définition des séquences */
```

```
/* Debut Définition de chaine des caractères */
```

```
owner_caracteres="MesCaractes", valeur=tous_caracteres
```

```
owner_caracteres="Commentaires", valeur=owner_commentaires
```

```
/* Fin Définition de chaine des caractères */
```

```
/* Debut Appel de la variable sauvegarder et vérification */
```

```
owner_caracteres_appel="MesCaractes"
```

```
owner_caracteres_appel="Commentaires"
```

```
/* Si on détecte le premier petit guillemet, on débute la récupération des caractères. */
```

```
owner_debut_corps:
```

```
si: "" alors:
```

```
debut("MesCaractes");
```

```
finsi;
```

```
/* Tant que nous n'avons pas arrivé au dernier petit guillemet, ne rien faire et on continue la récupération. */
```

```
si: "<MesCaractes>" alors:
```

```
nerienfaire;
```

```
finsi;
```

```
/* Si nous arrivons à détecter les caractères et le dernier petit guillemet, on initialise la récupération. */
```

```
si: "<MesCaractes>"+"" alors:
```

```
debut:(owner_initialise);
```

```
finsi;
```

```
/* S'il Ya rien à détecter, avant de sauter la ligne on récupère nos caractères. */
```

```
si: "<MesCaractes>"+"." alors:
```

```
owner_recupere_tous;
```

```
/* On retourne les caractères dans la variable qu'on à definit précédemment. */
```

```
retour:"TousCaracteres";
fini;

/* Fin Appel de la variable sauvegarder et vérification */


/* Debut Vérification d'une ligne commenté */
/* Si on détecte le <!--, on sait que c'est le debut d'un commentaire. */
si: "<!--" alors:
debut:("Commentaires");
fini;

/* Tant que nous n'avons pas arrivé au dernier >, on passe à la ligne suivante.
*/
si: "<Commentaires>" alors:
owner_ligne ++;
fini;

/* Si nous arrivons à détecter les caractères et le dernier >, on initialise les
commentaires. */
si: "<Commentaires>"+ ">" alors:
debut:(owner_initialise);
fini;

/* S'il Ya rien à détecter avant de sauter la ligne, on saute directement. */
si: "<Commentaires>"+ "." alors:
fini;

/* Fin Vérification d'une ligne commenté */
```



```
/* Debut ouverture de vérification */  
owner_verification:  
{ "If" } { retour: owner_ecoute: "MonIf;" }  
{ "Then" } { retour: owner_ecoute: "MonThen;" }  
{ "Else" } { retour: owner_ecoute: "MonElse;" }  
{ "End If" } { retour: owner_ecoute: "MonEndIf;" }  
owner_verification;  
/* Fin fermeture de vérification */
```

```
/* Debut réponse sur la console */  
owner_reponse:  
owner_message("\n\tErreur a la ligne %d :\n\t'%s' n'est pas reconnu en tant  
que commande interne ou externe, un programme executable ou un fichier des  
commandes.\n\tComporte %d  
lettre(s)\n\t", owner_ligne, owner_code, owner_nombre_caracteres);  
owner_erreur_composeur=activer;  
owner_reponse;  
/* Debut réponse sur la console */
```

```
owner_fin_corps;
```

```
/* Debut définition de variable interne */
```

```
owner_bool "mavariab1=false";
```

```
owner_bool "mavariab2=false";
```

```
/* Fin définition de variable interne */
```

```
/* Debut définition de variable externe */
```

```
owner_bool_externe=owner_ligne;
```

```
owner_bool_externe=owner_erreur_composeur;
```

```
/* Fin définition de variable externe */
```

```
/* Debut table de hachage */
```

```
owner_table:
```

```
owner_table_type=owner_variable;
```

```
owner_table_taille=owner_caracteres;
```

```
owner_table;
```

```
/* Debut table de hachage */
```

```
/* Debut union */
```

```
owner_union:
```

```
owner_union_longeurs=owner_nombre;
```

```
owner_union_caracteres=owner_texte;
```

```
owner_union_methodes=owner_noeud;
```

```
owner_union;
```

```
/* Fin union */
```

```
/* Debut importation des expressions */
```

```
owner_expressions:
```

```
owner_expressions_valeur="MonIf"
```

```
owner_expressions_valeur="MonThen"
```

```
owner_expressions_valeur="MonElse"
```

```
owner_expressions_valeur="MonEndIf"
```

```
owner_expressions_valeur="MesCaractes"
```

```
owner_expressions;
```

```
/* Fin importation des expressions */
```

```
owner_ss:
```

```
/* Debut syntaxes */
```

```
owner_ss_code="MonIf" {
```

```
"mavariab1=true;"
```

```
owner_message(owner_fichier_sorti "MonFichier", valeur "if(");
```

```
}
```

```
owner_ss_code="MonThen" {
```

```
owner_message(owner_fichier_sorti "MonFichier", valeur "){");
```

```
"mavariab2=true;"
```

```
}  
  
owner_ss_code="MonElse" {owner_message(owner_fichier_sorti  
"MonFichier", valeur "}")else{}}  
  
owner_ss_code="MonEndIf" {owner_message(owner_fichier_sorti  
"MonFichier", valeur "}")};}  
  
/* Fin syntaxes */
```

```
/* Debut sémantiques */  
  
owner_ss_code="MonIf"  
+="MesCaractes"  
+="MonThen"  
{  
si: "mavariante1==true || mavariante2==true" alors:  
owner_erreur_semantique=desactiver;  
autres:  
owner_erreur_semantique=activer;  
finsi;  
}  
  
/* Fin sémantiques */  
  
owner_ss;
```

```
/* Debut définition de(s) fichier(s) et extension(s) de sortie */
```

```

owner_sortie:
owner_variable="MonFichierEntree";
"MonFichierEntree", valeur ("strdup(argv[1])");
owner_variable="MonFichierSorti";
"MonFichierSorti", valeur ("C:/Resultat.c");
"MonFichier", valeur owner_ouvrir ("MonFichierSorti");
owner_fermer("MonFichier");
owner_fichier();
owner_sortie;
/* Fin définition de(s) fichier(s) et extension(s) de sortie */

```

Copier les code d'exemple complet puis crée un fichier avec n'importe quel nom et enregistrez-le avec l'extension **.own** ensuite collé tous les codes dedans.

Pour tester et avoir un nouveau langage, il suffit de double cliqué votre fichier. Et sur le même répertoire vous verrez votre nouveau langage executable. Vous pouvez le tester jalousement !

Teste et Utilisation :

Une fois vous avez sur votre répertoire un nouveau langage crée, il vous faudra donc de le tester. Pour se faire, crée un nouveau fichier et enregistrez-le avec n'importe quel extension. Dans cet exemple nous utiliserons l'extension **.ae** et rajoutez-le les codes ci-après :

Essaie.ae :

```
< //
```

```
If MaVariable=1 Then
```

```
Message "Valide !"
```

```
Else
```

Message "Non valide !"

End If

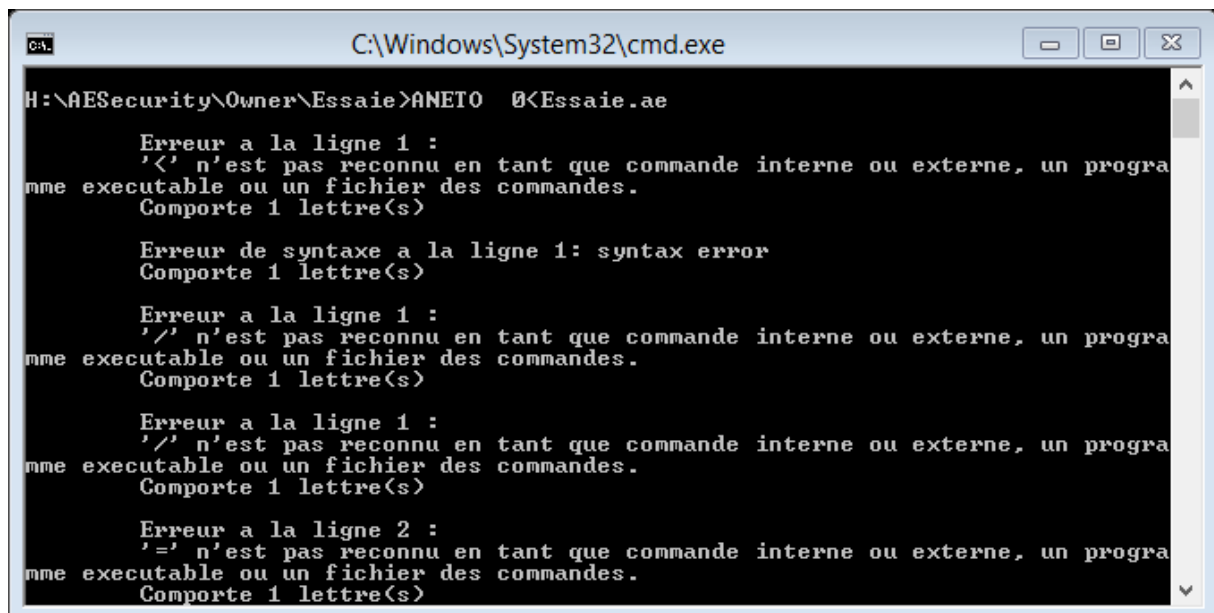
< //

Puis crée encore un nouveau fichier (Teste.bat) avec l'extension **.bat** et saisissez les commandes ci-dessous :

Teste.bat :

C:/MonLangage.exe < C:/Essaie.ae

Il vous faudra ensuite de double cliqué sur le fichier **Teste.bat** et vous verrez l'aperçus de l'écran suivant :



```
C:\Windows\System32\cmd.exe

H:\AE\Security\Owner\Essaie>ANETO 0<Essaie.ae

    Erreur a la ligne 1 :
    '<' n'est pas reconnu en tant que commande interne ou externe, un programme
    exécutable ou un fichier des commandes.
    Comporte 1 lettre(s)

    Erreur de syntaxe a la ligne 1: syntax error
    Comporte 1 lettre(s)

    Erreur a la ligne 1 :
    '/' n'est pas reconnu en tant que commande interne ou externe, un programme
    exécutable ou un fichier des commandes.
    Comporte 1 lettre(s)

    Erreur a la ligne 1 :
    '/' n'est pas reconnu en tant que commande interne ou externe, un programme
    exécutable ou un fichier des commandes.
    Comporte 1 lettre(s)

    Erreur a la ligne 2 :
    '=' n'est pas reconnu en tant que commande interne ou externe, un programme
    exécutable ou un fichier des commandes.
    Comporte 1 lettre(s)
```

On voit que tant qu'on n'a pas défini les signes inférieur, supérieur, slash et égal. Notre analyseur composeur affiche les messages des erreurs de chaque signe à chaque ligne.

A vous maintenant de créer votre propre environnement pour votre langage !

