MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université SAAD DAHLAB de BLIDA

Faculté des Sciences

Département d'Informatique

Mémoire de fin d'études

Pour l'obtention du diplôme de master en informatique

Thème

Interpréteur d'un langage logique

Présenté par :

Mr. BOUCHAREB MOURAD

Mr. ABEDELKADER BOUNEBBAB

Promoteur :

Mr. HAMOUDA MOHAMED

Encadreur:

Mr. ZAGOUR DJEMAL ELDIN

Promotion: 2012/2013

DEDICACES

Je dédie ce mémoire :

A mes très chers parents qui ont toujours été là pour moi, et qui m'ont donné un magnifique modèle de labeur et de persévérance. J'espère qu'ils trouveront dans ce travail toute ma reconnaissance et tout mon amour.

A mes frères : hamza, Farid, Abd el basset.

A mes sœurs

A mes nièces et neveux

A ma famille bouchareb.

A tout mes ami(e) s.

MOURAD BOUCHAREB

DEDICACES

Je dédie ce mémoire de fin d'études :

A mon très cher père et ma très chère mère en témoignage de ma reconnaissance envers le soutien, les sacrifies et tous les Efforts qu'ils ont faits pour mon éducation ainsi que ma formation.

A mes chers frères, et mes chères sœurs Pour leur affection, compréhension et patience.

A mes ami(e)s.

A tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail.

ABEDELKADER BOUNEBBAB

remerciments

remerciments

On dit souvent que le trajet est aussi important que la destination. Les 5 ans du cursus de master en informatique m'ont permis de bien comprendre la signification de cette phrase toute simple. Ce parcours, en effet, ne s'est pas réalisé sans défis et sans soulever de nombreuses questions pour lesquelles les réponses nécessitent de longues heures de travail.

Je tiens à la fin de ce travail à remercier ALLAH le tout puissant de m'avoir donné la foi et de m'avoir permis d'en arriver là.

Je remercie infiniment MR HAMOUDA MOUHAMED. Enseignant à l'université de Blida.

Je remercie également MR DJEMAL ELDIN ZAGOUR. Enseignant à l'ESI.

Je remercie aussi tous ceux qui m'ont aidé à réaliser ce travail.

Enfin je remercie tous les enseignants de la faculté des sciences de l'université de SAAD DAHLAB de Blida, ainsi que tous les étudiants.

TABLE DES FIGURES

TABLE DES FIGURES

SOMMAIRE

Résumé

Certaines classes de problèmes difficiles ne pouvant être traitées par des moyens informatiques conventionnels, trouvent aujourd'hui des solutions élégantes et efficaces grâce aux langages de la programmation logique.

La principale différence entre les langages de programmation logique, tel que "Prolog", et les langages 'classiques', tel que "Pascal" est que : ces derniers sont de nature impérative, il faut décrire le problème à résoudre selon un algorithme, alors que les langages logiques, sont de nature déclarative, cela veut dire qu'il suffit d'indiquer au système les données du problème à traiter.

Ainsi avec cette nouvelle vision, on peut résoudre des problèmes complexes sans recourir à des techniques algorithmiques.

L'objectif de ce projet est la réalisation d'un interpréteur d'un langage logique en français avec une interface graphique.

Pour cela, nous utilisons le langage c# pour le développent d'un analyseur lexical, syntaxique, sémantique et pour l'interface.

Mots clés : Prolog, interpréteur, langages logiques, analyseur lexical, analyseur syntaxique, analyseur sémantique.

ملخص

فئات معينة من المشاكل الصعبة التي لا يمكن علاجها عن طريق موارد تكنولوجيا المعلومات التقليدية يوجد اليوم حلول فعالة وبسيطة وهذا بفضل لغة البرمجة المنطقية

الفرق الرئيسي بين لغات برمجة المنطق، مثل "برولوج"، ولغات "الكلاسيكي" مثل "باسكال" هو:أن هذا الأخير هو من طبيعة إلزامية ، ولهذا فانه يتعين كتابة المشكلة المراد حلها بواسطة خوارزمية، ثم أن لغة المنطق هي ذات طبيعة تعريفية، فهذا يعني أنه يكفي الإشارة إلى النظام بمعطيات المشكل المراد حله.

مع هذه الرؤية الجديدة، يمكننا حل المشاكل المعقدة دون اللجوء إلى تقنيات الخوارزميات.

والهدف من هذا المشروع هو تحقيق مترجم للغة منطقية بالفرنسية وهذا مع ترجمة الكلمات الدالة للغة البر ولوج الإنجليزية إلى الفرنسية.

لهذا، فإننا نستخدم لغة السي شارب لتطوير محلل المعجمية، محلل لغوي محلل معاني الكلمات إضافة إلى واجهة رسمية.

الكلمات الرئيسية: برولوج ، مترجم ، لغة المنطق، محلل المعجمية ، محلل لغوي، محلل معاني الكلمات.

Abstract

Some classes of difficult problems that can't be treated by conventional means informatique today are stylish and effective solutions thanks to the language of logic programming.

The main difference between the logic programming languages, such as "Prolog", and languages 'classic' as "Pascal" is: these are mandatory in nature; describe the problem to be solved by an algorithm, then that logic languages are declarative nature, it means that the system is sufficient to indicate the nature of the problem being treated.

So with this new vision, we can solve complex problems without resorting to algorithmic techniques.

The objective of this project is the realization of an interpreter of a logical language with a French graphic and the translation of key words prolog English Language to French interface.

For this, we use the C # language to develop a lexical analyzer, syntax, and semantics and interface graphic.

Key words: Prolog, interpreter, logical language, lexical analyzer, syntax analyzer, semantics analyzer.

INTRODUCTION GENERALE

1- Problématique :

Le Langage de programmation logique est une langage très importante pour les programmeurs. Pour ça le département de l'informatique à l'université de Blida fait un programme de l'enseignement de cette Langage à leurs élèves de la troisième année de licence et de la première année de master.

Cette Langage est enseignée en langue française entrecoupées de beaucoup de termes anglais, par exemple:

«Factorielle(X, Y):-X>0, X1 is X-1,".

En outre, les étudiants étudient dans les travaux pratiques sur les logicielles SWI-prolog, GNU-prolog, Easy-prolog, et tous ces logiciels sont en anglais, un problème idéal pour les étudiants. Beaucoup d'entre eux ne comprennent pas l'anglais ce qui les a touchés à comprendre les messages que livraient les interpréteurs de ces logiciels pendant l'exécution des requêtes, c'est ce que l'impact de la concentration de les étudiants et les dissipent leurs idées et doivent diminuer de réussite scolaire.

2- Solutions proposées :

Nous proposons les solutions suivantes:

- réaliser un interpréteur d'un langage logique en français.
- Réaliser une interface graphique de cet interpréteur.

3- Organisation du mémoire :

Pour atteindre l'objectif cité ci-dessus, ce mémoire est organisé comme suite :

- ✓ Chapitre 1 : Etat de l'art.
- ✓ Chapitre 2 : Contribution.
- ✓ Chapitre 3 : Mise en œuvre.
- ✓ conclusion générale.
- ✓ GLOSSAIRE.
- ✓ BIBLIOGRAPHIE.
- ✓ ANNEXE.

CHAPITRE I

Etat de l'art

I.1- Introduction :

Dans ce chapitre nous allons définir les différents types de programmation (impérative, fonctionnelle, orientée objet, logique) ainsi que les caractéristiques de type programmation logique.

I.2- Les modes de programmation:

I.2.1- Langages fonctionnels :

I.2.1.1- historique:

En 1978 Backus présentait un langage fonctionnel (FP) proposant un petit nombre de combinateurs offrant une puissance d'expression suffisante pour la plupart des applications.

Même si le langage FP n'a pas considérablement influencé l'histoire de la programmation fonctionnelle, l'article de Backus a eu un impact énorme sur le monde de la recherche.

Dans ce texte, Backus exposait en effet pourquoi la programmation impérative était "mauvaise", et pourquoi la programmation fonctionnelle était "bonne", l'une des difficultés de la programmation traditionnelle étant le recours constant aux effets de bord : un programme impératif ne peut se comprendre que par une analyse séquentielle de son déroulement, la présence d'affectations et de branchements interdisant pratiquement toute analyse formelle du texte d'un programme.

L'un des concepts importants mis en valeur par Backus est celui de transparence révérencielle. Dans un langage purement fonctionnel, il y a équivalence formelle entre expressions du type :

x + x where x = f a # (f a) + (f a)

Ce n'est naturellement pas le cas dans un langage impératif, ou les deux expressions peuvent avoir des valeurs différentes si la fonction f a des effets de bord. C'est cette transparence révérencielle qui va permettre le raisonnement équationnel sur les programmes, la preuve de ceux-ci, leur optimisation, leur fonctionnement sur des machines parallèles [2].

I.2.1.2- Paradigme fonctionnel :

La couche fonctionnelle peut être considérée comme la plus profonde de toutes les couches sémantiques dans un langage évolué. La machine qui exécute un programme

fonctionnel « focalise son attention » sur le concept d'expression : objet qui engendre une valeur.

Nous avons donc les constantes littérales : nombres, chaînes de caractères (considérées comme atomiques), etc. Les fonctions seront elles aussi des valeurs.

Ils existent également des variables qui donnent des noms aux valeurs, et son usage implique l'existence de l'environnement permettant d'établir le rapport entre un nom et la valeur correspondante.

Dans le style fonctionnel les notions d'un objet et de sa valeur se confondent.

Il n'y a pas de modifications de valeurs (affectation : " :="), et dans le même environnement la variable x signifie toujours la même chose.

Il est évident, que pour pouvoir construire des nouvelles valeurs il faut savoir appliquer des opérations aux arguments, mais toute affectation, toute opération de genre x:=x+1 est strictement interdite.

Absence d'affectations n'empêche l'usage de variables locales qui s'identifient avec les expressions qu'elles représentent. Voici la définition du sinus hyperbolique en Haskell:

```
sh x = (y-1.0/y)/2.0
where y=exp x
```

Où where est un mot réservé.

Il faudra s'habituer à une particularité syntaxique de Haskell (existant dans quelques autres langages, comme Clean ou Python : l'indentation remplace le parenthésage – si la ligne suivante est plus indentée que la précédente, ceci signifie la continuation.

La même indentation dénote une définition collatérale, au même niveau, et une indentation plus petite termine la structure syntaxique précédente, sans besoin de parenthèses.

La même définition en Scheme serait:

```
(define (sh x)
(let ((y (exp x)))
(/ (- y (recip y)) 2)))
```

Dans les deux cas conceptuellement important est le fait que ces définitions de fonctions sont des abréviations des opérations plus primitives :

– Création d'un objet fonctionnel, d'une valeur qui représente la fonction, et assignation de cet objet fonctionnel à une variable.

Des fonctions anonymes existent aussi, en Scheme nous aurons (lambda (x) (let ((y (exp x))) (/ (- y (recip y)) 2))) [3].

I.2.1.3- Exemples de langages fonctionnels :

Le père de tous les langages fonctionnels est le Lisp et ses descendants, comme Scheme.

Ce sont des langages typés dynamiquement, c'est à dire que les variables n'ont pas d'attribut de type. Les valeurs sont toujours typées.

Parfois dans les descriptions populaires en sème la confusion en faisant l'amalgame entre l'absence de typage statique et la sémantique fonctionnelle du Lisp. Ceci n'est pas justifié, mais légèrement pardonnable : les langages fonctionnels sont beaucoup mieux adaptés à l'implantation des routines polymorphes, les fonctions écrites en Lisp dans le but pédagogique sont très souvent génériques, et l'absence de typage statique facilite la construction syntaxique de telles fonctions. Pourquoi ? Car on peut aisément assurer l'homogénéité des structures comme des listes avec des éléments de type quelconque, si la seule donnée qui y est stockée est un pointeur.

Mais il y a d'autres langages fonctionnels, notamment de la famille ML :

SML et CAML d'INRIA (avec son héritier Objective CAML) qui sont strictement typés. Ce typage est statique, mais l'utilisateur n'a pas besoin de déclarations. Si la définition d'une fonction a la forme : f(x) = x + 1.0/x, le compilateur reconnaît immédiatement que x doit être flottant, sinon la division ne serait pas définie. Alors, même avec la surcharge de l'opérateur + (l'addition) est flottante, et le résultat retourné par la fonction également. Cette inférence automatique des types porte le nom de système de Hindley-Milner. L'inférence automatique des types sera discutée plus tard.

La famille ML est stricte, mais des langages paresseux existent aussi. Le plus populaire est actuellement notre favori Haskell, un langage fonctionnel pur, sans affectations et sans effets de bord. Son trait caractéristique est un système de types polymorphes beaucoup plus puissant que le système de Hindley-Milner implanté dans les langages ML. Le système de Haskell définit des classes de types, qui est une version fonctionnelle très puissante de la technique de programmation par objets.

Le concurrent le plus "dangereux" de Haskell est Clean, un langage presque jumeau de Haskell, mais avec un autre système de typage hiérarchique. Il est difficile de dire si les deux langages vont converger ou pas. Les auteurs de Clean ont l'ambition de faire de ce langage un outil performant dans le domaine du calcul parallèl. Le parallélisme fonctionnel est très vivant, et le langage Sisal a été conçu spécifiquement pour les ingénieurs physiciens dans le but de faciliter le passage aux modèles de calcul plus modernes que ceux offerts par Fortran.

On peut mentionner encore un petit langage paresseux Hope, utile pour des petits exercices, et un langage paresseux jadis populaire Miranda qui a été commercialisé par son auteur David Turner (un excellent spécialiste des langages et très bon pédagogue), ce qui a finalement écarté le langage du marché, car tous les autres possèdent des implantations gratuites.

Notons finalement que la famille Lisp est stricte, mais la sémantique du langage est suffisamment générale pour pouvoir implanter "manuellement" les constructions paresseuses, et grâce aux macros ces constructions sont faciles à utiliser [4].

I.2.2- Langages impératifs :

I.2.2.1- Historique:

Langages machine: Les langages impératifs les plus anciens sont les langages machine des premiers ordinateurs. Dans ces langages, le jeu d'instructions est minimal, ce qui rend la mise en œuvre matérielle plus simple.

A-0: Le premier compilateur – un programme destiné à vérifier un programme au préalable et à le traduire en langage machine – dénommé A-0, fut écrit en 1951 par Grace Murray Hopper.

FORTRAN: FORTRAN, développé par John Backus chez IBM à partir de 1954, fut le premier langage de programmation capable de réduire les obstacles présentés par le langage machine dans la création de programmes complexes. FORTRAN était un langage compilé, qui autorisait entre autres l'utilisation de variables nommées, d'expressions complexes, et de sous-programmes. Après plusieurs révisions du langage, en 1978 et en 1990, FORTRAN est toujours utilisé dans le milieu scientifique pour la qualité de ses bibliothèques numériques et sa grande rapidité, ce qui en fait le langage informatique ayant eu la plus grande longévité.

Algol: Les deux décennies suivantes virent l'apparition de plusieurs autres langages de haut niveau importants. ALGOL, développé en 1958 par un consortium américano-européen

pour concurrencer FORTRAN, qui était un langage propriétaire, fut l'ancêtre de nombreux langages de programmation d'aujourd'hui.

COBOL et BASIC: COBOL (1960) et BASIC (1963) étaient deux tentatives pour rendre la syntaxe plus proche de l'anglais, et donc plus accessible. COBOL était spécifiquement destiné aux applications de gestion, tandis que BASIC avait essentiellement un but éducatif. Sa simplicité et le fait qu'il soit interprété facilitaient grandement la mise au point des programmes, ce qui lui conféra rapidement une grande popularité, malgré la pauvreté de ses constructions. Malheureusement, cette pauvreté même devait mener à une quantité de programmes non structurés et donc difficilement maintenables. Après un article de Edsger Dijkstra dénonçant les ravages de BASIC, la réputation de BASIC comme langage pour l'enseignement de la programmation déclina.

Pascal: Dans les années 1970, le Pascal fut développé par Niklaus Wirth, dans le but d'enseigner la programmation structurée et modulaire. Pascal combinait les meilleures constructions des langages COBOL, FORTRAN et ALGOL dans un ensemble élégant (servi par un grand nombre de types prédéfinis: ensemble, enum, intervalle), qui lui assura un succès durable comme langage d'initiation (en remplacement de BASIC). Par la suite, Niklaus Wirth fut à l'origine de Modula-2, Modula-3, et d'Oberon, les successeurs de Pascal.

En opposition à C, Pascal effectue un contrôle rigoureux des types, des index de tableaux et les chaines de caractères bénéficient d'une définition bien moins simpliste. Les pointeurs sont également typés (sauf décision explicite du programmeur). Cette rigueur ne pèse en réalité que sur le compilateur, le développeur peut se reposer sur les contrôles effectués à la compilation pour détecter le quasi totalité des fautes d'inattention ou les confusions dues à un algorithme complexe.

Les rares erreurs passant la compilation sont liées à l'algorithme lui-même, à des pointeurs mal maitrisés (mais leur typage évite les bugs les plus pernicieux) ; il est habituel pour un programmeur Pascal de voir s'exécuter au premier lancement, un programme de complexité supérieure à la moyenne.

Le vieux débat de la vitesse d'exécution entre le Pascal et le C n'a pas lieu d'être, il suffisait de lever les sécurités du Pascal par les directives de compilation (Turbo Pascal de Borland) pour avoir une vitesse d'exécution comparable à C. Mais le développeur Pascal ne passait que peu de temps en débogage et le code était par nature beaucoup plus maintenable grâce à la rigueur du compilateur.

Ceci dit, les légères différences de tolérance à la structuration, ou au typage, entre autres, ne remplaceront en aucun cas la qualité du programme. Le temps de débogage sera par définition tout aussi important, dès lors que les conseils de base et la structuration seront imposés par Pascal, et considérés comme implicite ou acquis par d'autres.

Pascal s'interface simplement avec un autre langage en déclarant des procédures externes. On peut le lier par exemple avec du C ou de l'assembleur ce qui permet un accès total au fonctionnement interne du système en profitant d'un langage très structuré, richement typé, facilement maintenable.

C: À la même époque, Dennis Ritchie créa le fameux langage C aux laboratoires Bell, pour le développement du système Unix. La puissance du C, permettant grâce aux pointeurs de travailler à un niveau proche de la machine, ainsi qu'un accès complet aux primitives du système, lui assura un succès qui ne s'est jamais démenti depuis.

La cause principale du succès du langage C par rapport aux autres langages procéduraux de la même génération vient de son mode de distribution : les universités américaines pouvaient acheter une licence au prix de 300 dollars pour toute l'université et tous ses étudiants. Ce système a formé toute une génération de développeurs — dont Bill Joy grand contributeur du Berkeley Unix (distribution BSD) et pilier de Sun.

Ada: En 1974, le Département de la Défense des États-Unis cherchait un langage dont le cahier des charges mettait l'accent sur la sûreté d'exécution, pour tous ses besoins futurs. Le choix se porta sur Ada, langage créé par Jean Ichbiah à Honeywell, dont la spécification ne fut complétée qu'en 1983.

Smalltalk: Dans les années 1980, devant les problèmes que posaient la complexité grandissante des programmes, il y eut un rapide gain d'intérêt pour la programmation orientée objet. Smalltalk-80, conçu à l'origine par Alan Kay en 1969, fut présenté en 1980 par le Palo Alto Research Center de la compagnie Xerox (États-Unis).

C++ et Objective C: À partir des concepts objet, Bjarne Stroustrup, chercheur aux Bell Labs, conçut en 1985 une extension orientée objet de C nommée C++, qui gardait la vitesse de C. Parallèlement, une extension à C moins ambitieuse, mais inspirée de Smalltalk avait vu le jour, Objective C. Le succès d'Objective C, notamment utilisé pour le développement sur les stations NeXT et Mac OS X, est resté faible par rapport à C++.

Common Lisp: Le langage Common Lisp fut le premier langage à objets (CLOS) standardisé par l'ANSI, en 1995.

Perl, Tcl, Python, PHP et Java: Dans les décennies 1980 et 1990, de nouveaux langages impératifs interprétés ou semi-interprétés doivent leur succès au développement de scripts pour des pages web dynamiques et les applications client-serveur. On peut citer dans ces catégories Perl (Larry Wall, 1987), Tcl (John Ousterhout, 1988), Python (Guido van Rossum, 1990), PHP (Rasmus Lerdorf, 1994) et Java (Sun Microsystems, 1996)[5].

I.2.2.2- Paradigme impératif:

Le concept fondamental ici est celui de l'action, ou d'instruction. La machine (réelle ou virtuelle) exécute quelque chose, et change son état : les valeurs de variables.

On peut apparier le concept d'environnement dans les langages fonctionnels avec l'état ici, mais rappelons que l'environnement fonctionnel pur est immuable, et dans la définition de l'état la notion de changement est inhérente. Nous aurons donc l'instruction d'affectation :

a:=2*x+f(x); en Pascal, et l'équivalent dans d'autres langages de type impératif ou procédural:

"C", Fortran, Basic, Cobol, Modula, ainsi que presque tous les assembleurs. Des langages fonctionnels comme Lisp, ou même ML possèdent des couches impératives, la possibilité de modifier les valeurs de variables, et de générer les effets de bord, mais ceci facilite la production des programmes hybrides, difficiles à déboguer.

Une machine impérative correspond à une couche "basse" de programmation

– elle ressemble à la machine réelle, matérielle, qui est une réalisation du modèle de Von Neumann. Le processeur physique adresse la mémoire (les registres, ou les variables, si on leur assigne des noms symboliques), récupère les valeurs stockées et éventuellement les remplace par d'autres valeurs.

Dans le modèle de Von Neumann le programme stocké dans la mémoire est une liste linéaire d'instructions. La machine exécute une boucle : le registre principal, le compteur d'adresses permet de récupérer l'instruction à exécuter, ensuite on incrémente ce compteur et on continue. Il y a toujours une instruction à exécuter.

La machine ne s'arrête jamais. La structure de contrôle fondamentale dans ce modèle est le branchement inconditionnel ou conditionnel : selon la valeur "Booléenne" d'un des registres, on effectue ou pas un branchement, le goto, une instruction dont l'argument est

l'adresse d'une autre instruction. Ainsi, si l'adresse passée à goto précède l'adresse actuelle, on peut fermer une boucle classique.

Un programmeur lambda ne doit jamais traiter ce programme en "C"

```
while(x>2)
{faire(x,g(x)); x=h(x);}
```

Comme l'abréviation de boucle :

```
z=x-2 ;
if(z<=0) goto bfin ;
faire(x,g(x)) ;
x=h(x) ;
bfin : ...
```

Car psychologiquement c'est totalement inutile. Il faut, bien sûr, comprendre la sémantique de la boucle, et non pas sa forme décortiquée de bas niveau.

Mais les concepteurs et réalisateurs de compilateurs ne sont pas des programmeurs lambda. Ils doivent savoir traduire les constructions de haut niveau en concepts appartenant au modèle d'exécution. Ceci est vital, indépendamment des différences syntaxiques entre langages.

Alors, un petit récapitulatif. La machine virtuelle de plus bas niveau, un interprète "plat" doit permettre:

– L'adressage des zones mémoire contenant les instructions (compteur-programme).

– Auto-incrémentation (exécution séquentielle des instructions) et modification dynamique du compteur des instructions, le branchement (goto).

– Au moins un mécanisme décisionnel (if!goto).

– L'adressage des emplacements des données (registres, variables).

– Récupération des données, leur transfert.

– Modification des données (primitives) par le processeur.

Ceci n'épuise pas les notions de base appartenant au modèle impératif. Il faut ajouter au moins la possibilité de construire des procédures, c'est à dire d'automatiser le goto avec retour. Il faut donc pouvoir stocker quelque part l'adresse d'une instruction traitée comme

donnée. Ceci est une variante très brutale de ce qu'en modèle précédent était un objet fonctionnel.

En Fortran antédiluvien les procédures (subroutines) n'étaient pas récursives, et chaque procédure prévoyait un emplacement statique dans son segment de données pour y stocker l'adresse de retour de cette procédure au module appelant. Le code de l'instruction call f par g se compilait comme :

– Récupérer l'adresse de la procédure appelée g.

– Récupérer l'emplacement du segment de données de g correspondant à l'adresse de retour, disons, le registre $ret.

– Stocker dans ce registre la valeur du compteur-programme (l'instruction suivante à éxécuter, appartenant à la procédure f).

– Exécuter goto g.

Tandis que le retour de la procédure se réduisait à :

– Récupérer la valeur stockée dans $ret, et effectuer le goto.

Dans ce modèle toutes les données étaient statiques, et chaque procédure travaillait dans son "monde privé", ou, éventuellement dans une zone accessible globalement (COMMON BLOCKS).

La possibilité d'opérer avec des procédures récursives implique la protection de l'adresse de retour. Le protocole standard, utilisé par pratiquement toutes les implantations des langages admettant la récursivité est basée sur la pile de retour.

Au lieu de stocker l'adresse de retour dans une zone statique appartenant au module appelé, chaque appel réserve un segment du tableau géré par le système et structuré comme une pile. L'adresse de retour est stockée sur ce segment. Chaque retour détruit le dernier segment alloué.

Bien sur, les langages traditionnels, disposant des procédures paramétrées prévoient également l'allocation d'un segment de données Où on stocke les arguments et les données locales. Cette zone de mémoire est aussi structurée comme une pile.

Cependant conceptuellement ce segment est indépendant du flot de contrôle, bien que la pile des retours et la pile de données souvent fonctionnent en synchronie. Nous verrons

toutefois, qu'il est beaucoup plus facile de construire des machines virtuelles si on garde l'indépendance des deux objets.

Notons que les branchements et les boucles ajoutées aux appels procéduraux n'épuisent pas toutes les structures de contrôle disponibles dans quelques langages.

Un mécanisme particulièrement intéressant pour la simulation est la co-procédure qui permet la réalisation de collaboration symétrique entre deux modules. Avec l'appel standard si module A appelle module B, il empile l'adresse de retour et continue l'exécution après le retour. Si B appelle de nouveau A, un nouvel empilement à lieu. Comment alors organiser un jeu binaire, Où les deux modules représentent deux joueurs : chacun fait son tour, change l'état global du système (l'échiquier par exemple), et passe la main à l'adversaire.

La co-procéduralisation consiste à respecter les règles suivantes :

– Si le module A n'a jamais été appelé, son lancement se déroule de manière standard.

– Si maintenant A veut passer le contrôle à B, il exécute le branchement, mais avant stocke l'adresse de retour dans sa propre zone de données statiques.

– Quand B veut "retourner" à A, il fait la même chose – branche à A après avoir stocké l'adresse de retour.

– Le lancement du module partenaire (l'exécution de l'instruction résume) est un branchement normal, mais le code qui récupère le contrôle vérifie d'abord si une adresse de retour coprocédural n'a pas été stockée au préalable, et si c'est le cas, un branchement secondaire a lieu.

La technique de co-procéduralisation est très importante dans la simulation, et constitue une alternative aux flux (pipes) permettant d'établir une collaboration symétrique entre les modules du compilateur [4].

Exemples:

La majorité des langages sur le marché est impérative : C, C++, Java, Ada,

Oberon (une évolution de Modula), etc. Le modèle impératif se combine avec la programmation par objet de plusieurs façons différentes : C++, Eiffel et Smalltalk sont tous des langages impératifs à objets, mais assez différents. (Smalltalk est même très différent).

La popularité des langages impératifs est le résultat du fait que les assembleurs soient impératifs, et d'une illusion bizarre qu'un langage impératif sera toujours plus efficace qu'un langage fonctionnel ou logique.

Ainsi, même les langages interprétés où l'efficacité brute du code est un facteur secondaire, comme les langages de programmation scientifique : Matlab, Tela, IDL, etc., ou les langages de calcul formel : Maple, Axiom, Reduce, Macsyma, etc., sont des langages impératifs dont la syntaxe ressemble à la famille Pascal, bien que plusieurs parmi eux sont basés sur des machines virtuelles fonctionnelles, notamment sur l'interprète Lisp, ou pareil. La psychologie conservatrice a gagné sur la logique. . .

Les langages interprétés conçus pour écrire des scripts (ont partiellement remplacé les langages de commandes du système d'exploitation) comme Perl sont aussi impératifs.

Sa structure syntaxique est laide, et son débogage pénible. Mais sa gestion des chaînes de caractères et expressions régulières est très riche, et la coopération avec le système d'exploitation (gestion des fichiers et des processus) est mise au point.

Perl reste le langage numéro 1 pour écrire des scripts CGI, mais probablement pas pour longtemps, car d'autres langages (par exemple Python) possèdent déjà presque toutes les fonctionnalités du Perl [6].

I.2.2.3- Continuations du fonctionnel à l'impératif:

On peut avoir l'impression, que si on élimine des langages impératifs l'évaluation des expressions, ce qui constitue la couche fonctionnelle, et si on reste avec le flux de contrôle, les branchements, les boucles, etc., ceci n'a plus rien à voir avec le monde fonctionnel et que les techniques de compilation deviennent très asymétriques : un programme impératif est "réel", correspond au code assembleur exécuté par le processeur, tandis que le code fonctionnel, la réduction et l'évaluation d'un graphe qui représente une expression, force sa translation en code linéaire, impératif.

Cependant le progrès récent dans le domaine de compilation et l'arrivée de nouveaux compilateurs est partiellement le résultat du progrès dans la compilation fonctionnelle qui est beaucoup plus statique, mathématiquement précise, et facile à comprendre.

La sérialisation du code, la notion de séquence, possède sa forme fonctionnelle aussi, et s'appelle la continuation. Brièvement, la continuation d'une expression (rappelons que dans le monde fonctionnel c'est le seul objet intéressant) est le "futur" du calcul, ou

l'opération qui sera exécutée immédiatement après. Dans l'expression f(g(x), h(y, x)) qui en Haskell sera écrite comme f (g x) (h y x).La continuation de l'expression g(x) est l'évaluation de h(y, x) car il faut évaluer le second argument de f, et finalement la continuation de h est f. Ainsi le code sera linéarisé.

La réalisation concrète et détaillée des continuations par le compilateur sera éventuellement faite ultérieurement. Ici il suffit de préciser qu'une telle opération peut être, et est automatique, et suggérer l'approche suivante : toute fonction est modifiée, et prend un paramètre supplémentaire. Ce paramètre est justement la continuation, une fonction d'un argument. Si la fonction originale retournait simplement une valeur, la fonction "continuée" passe cette valeur à sa continuation.

Plus concrètement, au lieu de discuter l'évaluation de (f x), nous allons transmuter f en une "fonction continuée" façon, telle, que façon f x cnt = cnt (f x)

Où la fonction cnt représente la continuation, le futur d'évaluation de (f x).

Nous affirmons que :

– le processus de construction de fonctions "continuées" peut être automatisé dans la plupart de cas.

– le code résultant ressemble plutôt à l'assembleur qu'à l'arborescence représentant une expression composite, hiérarchique.

– Son optimisation est beaucoup plus facile.

– Grâce aux continuations on pourra de manière fonctionnelle, et alors simple, lisible et statique définir les branchements, structures itératives, voire même des structures de contrôle non-déterministes.

Prenons comme exemple l'évaluation de l'expression p: x2 + y2.

Une définition fonctionnelle classique d'une fonction qui effectue ce calcul serait fn x y = sqrt (x*x + y*y)

Rien à ajouter. Supposons néanmoins que la fonction sqrt a été transmutée par font en sa forme continuée : sqc = f_cont sqrt, et que nous avons défini les opérateurs binaires "continués" également :

f_cnt2 op x y cnt = cnt (op x y)

add = f_cnt2 (+)

mul = f_cnt2 (*)

La fonction continuée fnc peut être définie comme

fnc x y cnt =mul x x (\a ->mul y y (\b ->add a b (\c ->sqc c cnt)))

Cet exemple doit être compris. La continuation de la première multiplication est le calcul de y*y, et cette opération n'a pas besoin du résultat a de la multiplication précédente, mais on le garde pour l'avenir. Il sera utilisé par add.

Notre tâche est accomplie. Les continuations possèdent une propriété fabuleuse du point de vue d'un créateur des compilateurs : grâce à elles on peut établir l'ordre d'exécution des opérations (évaluations) dans le programme, sans imposer cet ordre au niveau du métalangage. Nous définissons un langage de manière dénotationnelle, précisons sa syntaxe et sa sémantique statiquement, sans jamais aborder le problème de "temps".

Mais la construction des relations "X est la continuation de Y" permet 'enchaîner l'exécution des instructions, et ainsi nous pouvons créer un langage impératif sans quitter le style fonctionnel.

Les continuations peuvent naturellement être utilisées dans un programme quelconque. Tout enchaînement d'applications fonctionnelles dans un module peut exploiter cette stratégie. Mais attention : si une expression normale se transforme en "continuée", c'est-`a-dire en fonction qui attend un argument – la continuation, et si cette continuation est une fonction continuée comme mul, le résultat est encore une fois une expression continuée, un objet fonctionnel. Quand est'ce que le résultat final sera enfin récupéré ?

Il faut briser cette chaîne enchantée de continuations, et à la fin du module ainsi sérialisé, il faut appliquer une continuation terminale, par exemple la fonction id, définie par id x = x, qui récupère le résultat. Naturellement cet appel de la fonction id possède aussi une continuation, mais cette continuation pour le programmeur est implicite : elle peut appartenir à la boucle principale de l'interprète (le dialogue avec l'utilisateur), Où le résultat est affiché. Ou bien, ceci peut être la fin logique du programme qui s'arrête, t sa continuation est une des procédures du système d'exploitation, par exemple le shell qui fait avec le résultat ce qu'il veut. En tout cas la chaîne de continuations explicites doit être "cassée" pour voir le résultat sinon tous les objets crées sont des fonctions [7].

I.2.3- Programmation orientée objet :

I.2.3.1- Historique:

La programmation orientée objet (POO), ou programmation par objet, est un paradigme de programmation informatique élaboré par les norvégiens Ole-Johan Dahl et Kristen Nygaard au début des années 1960 et poursuivi par les travaux d'Alan Kay dans les années 1970. Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait communiquer avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; la communication entre les objets via leurs relations permet de réaliser les fonctionnalités attendues, de résoudre le ou les problèmes.

Orthogonalement à la programmation par objet, afin de faciliter le processus d'élaboration d'un programme, existent des méthodologies de développement logiciel objet dont la plus connue est USDP (Unified Software Development Process).

Il est possible de concevoir par objet une application informatique sans pour autant utiliser des outils dédiés. Il n'en demeure pas moins que ces derniers facilitent de beaucoup la conception, la maintenance, et la productivité. On en distingue plusieurs sortes :

les langages de programmation (Java, C#, VB.NET, Vala, Objective C, Eiffel, Python, Ruby, C++, Ada, PHP, Smalltalk, LOGO, AS3...)

Les outils de modélisation qui permettent de concevoir sous forme de schémas semi-formels la structure d'un programme (Objecteering, UMLDraw, Rhapsody, DBDesigner…)

Les bus distribués (DCOM, CORBA, RMI, Pyro...)

les ateliers de génie logiciel ou AGL (Visual Studio pour des langages Dotnet, NetBeans pour le langage Java)

Il existe actuellement deux catégories de langages à objets : les langages à classes et ceux à prototypes, que ceux-ci soient sous forme fonctionnelle (CLOS), impérative (C++, Java) ou les deux (Python, OCaml)[8].

I.2.3.2- Paradigme orienté objet :

La programmation OO, les objets, les messages et méthodes – tout ceci est devenu malheureusement un super-domaine rempli de slogans et défini de façon incongrue.

Il faut rappeler que la technique est née avec le langage Simula, et a été développée de manière exhaustive dans le cadre du langage Smalltalk. (Simula était d'ailleurs le premier langage populaire avec des co-procédures, très commodes pour l'implantation de la simulation de systèmes dynamiques. Il est possible que son relatif échec commercial soit dû au fait que Simula était un produit Européen. . .).

L'idée de bas niveau est simple : si nous avons la possibilité de construire les records, les données composites, nous pouvons prévoir qu'un ou plusieurs champs de nos données sont des fonctions qui "savent" comment traiter ces données (les autres champs) de manière spécifique, appropriée. La donnée s'appelle désormais objet. L'appel de la fonction qui est attachée à notre objet-donnée s'exprime comme l'envoi du message à l'objet. Ce message déclenche l'exécution d'une méthode. On voit que ceci n'a rien de magique, et qu'un bon langage fonctionnel permet aisément la construction de systèmes à objets. En effet, le nombre de paquetages OO en Lisp dépasse une centaine.

La vraie puissance des langages OO est la généricité – la possibilité de grouper des objets dans des classes qui partagent les mêmes fonctionnalités, et l'héritage.

La compilation des langages OO peut être assez facile, si toute décision est laissée à la machine virtuelle, mais elle peut contenir des optimisations extrêmement importantes, et partiellement à cause de cela les compilateurs de C++ sont très grands [9].

I.2.3.2- Exemples

Les langages à objets sont si nombreux, qu'une litanie de noms ne servira à rien.

Le langage dominant pour les grands programmes est C++ qui doit sa popularité principalement au fait que son ancêtre : le langage C est si populaire. C'est un langage riche et difficile à maîtriser. Pour construire un compilateur de C++ réaliste il faut une équipe de personnes très compétentes (cependant le design original est l'œuvre d'une personne : Bjarne Stroustrup).

Actuellement une bonne partie du "marché" C++ diverge dans la direction de Java, qui est interprété, alors plus lent, et plus pauvre au niveau de syntaxe (pas de surcharge des opérateurs, pas d'héritage multiple, etc.), mais qui est bien adapté à la construction des programmes sécurisés.

Comme il a été dit, les langages fonctionnels constituent une bonne plate-forme pour implanter les langages à objets. Parmi eux, la position privilégiée par le nombre d'utilisateurs est occupée par CLOS (Common Lisp Object System). La syntaxe reste presque la même qu'en Lisp, ce qui n'est pas très clair, mais CLOS a ses inconditionnels. Lisp, Scheme, et autres langages de cette famille ont donné naissance à des langages à objets innombrables.

Les langages fonctionnels modernes reconstruisent ses systèmes de typage hiérarchique (qui réalise le polymorphisme restreint et l'héritage) de manière différente.

Les langages comme Objective CAML, Haskell ou Clean méritent aussi d'appartenir un peu à la famille OO.

Depuis quelques années un autre langage à objets : Python fait une belle carrière. Le langage est simple et joli, et très transparent. On peut l'apprendre en quelques jours, et écrire des applications graphiques très performantes. Il remplace de plus en plus souvent le langage Perl. Python, grâce à sa transparence est utilisé dans quelques établissements comme le langage-modèle sur lequel les étudiants apprennent l'implantation des langages à objets.

Il faut mentionner ici le langage Eiffel, qui a ses partisans déclarés. Ce langage a été conçu par Bertrand Meyer, un Français (comme le nom du langage le suggère) [10].

I.2.4- Langages logiques :

I.2.4.1- Historique:

Dans les années 1930, Herbrand avait posé les conditions de validité d'une démonstration automatique. En 1953, Quine donnait une règle d'inférence originale ; définie pour l'ordre 0, elle présentait peu d'intérêt si ce n'est pour améliorer le calcul des circuits logiques. En 1965, Robinson donnait sa Méthode de Résolution : il basait une démonstration automatique sur les conditions d'Herbrand, avec un raisonnement par l'absurde utilisant des énoncés logiques mis sous forme clausale, et une Règle de Résolution, extension à l'ordre 1 de la règle de Quine. Les premiers essais montrèrent que l'idée y était, mais qu'il restait à en trouver une expression efficace : ce sera Prolog

En 1958, John McCarthy proposait déjà d'utiliser la logique comme langage declaratif de representation des connaissances, un démonstrateur de théorème devenant un résolveur de problème. La résolution de problèmes est alors répartie entre le cogniticien, responsable de la validité de l'application exprimée logiquement, et le moteur d'inférence, responsable d'une exécution valide et efficace.

En un sens plus étroit et plus commun, la programmation logique joue sur une ambivalence représentation declarative/représentation procédurale : ainsi, un raisonnement régressif associera à l'implication B1&...&Bn → H une procédure « pour établir H, établir B1 puis... puis Bn ». De ce fait, au nom de l'efficacité, le programmeur peut être amené à exploiter les propriétés physiques du démonstrateur, se rapprochant ainsi d'une programmation classique. Cependant, les programmes logiques gardent toujours une interprétation logique pure permettant de garantir leur correction, et, du fait de leur caractère déclaratif, sont plus abstraits que leur contrepartie impérative, tout en restant exécutables.

Les premières applications de la programmation logique (1964-69) concernèrent des systèmes de questions/réponses. Absys (1969) fut probablement le premier langage de programmation à base d'assertions.

La programmation logique au sens étroit remonte aux débats de cette époque concernant la représentation des connaissances en intelligence artificielle. Stanford et Édimbourg, avec J. McCarthy et Kowalski, tenaient pour une représentation déclarative, et le MIT, avec Marvin Minsky et Seymour Papert, pour une représentation procédurale.

Planner (Hewitt 1969), langage basé sur la logique, émergea cependant au MIT. Son sous-ensemble Micro-Planner (Sussman, Charniak, Winograd) fut utilisé par Winograd pour SHRDLU, programme basé sur l'interprétation d'un dialogue en langage naturel. Planner invoquait des plans procéduraux à partir de buts et d'assertions, et utilisait des reprises en arrière pour ménager le peu de mémoire disponible. Dérivèrent de Planner QA-4, Popler, Conniver, QLISP, Ether.

Cependant, Hayes et Kowalski à Édimbourg essayaient de réconcilier approche déclarative et représentation des connaissances avec l'approche procédurale à la Planner. Hayes (1973) developpa un langage équationnel, Golux, qui pouvait invoquer diverses procedures en altérant le fonctionnement du moteur d'inférence. Kowalski montrait par ailleurs que la SL-resolution traitait les implications comme procédures réductrices des buts.

Alain Colmerauer, universitaire français passé de la compilation à la traduction automatique (Montréal, 1967-70), eut d'abord l'idée des Q-systèmes (1969), formés de règles de réécriture d'arbres, invoquées selon les besoins et utilisant l'unification. Ces systèmes furent à la base d'une chaîne de traduction anglais→français, puis de la rédaction du système Météo qui, au Canada, traduit chaque jour les bulletins météorologiques de l'anglais au français.

Après 1970, Colmerauer revenu à Marseille s'intéressa davantage à l'exploitation de textes qu'à leur traduction ; voulant utiliser la logique pour représenter la sémantique aussi bien que pour les raisonnements liés aux questions, il s'intéressa aux travaux de Robinson sur le Principe de Résolution.

Durant l'été 1971, Colmerauer et Kowalski virent que la forme clausale pouvait représenter les grammaires formelles et qu'un moteur d'inférence pouvait être utilisé pour l'analyse de textes, certains moteurs fournissant une analyse ascendante, et la SL-résolution de Kowalski une analyse descendante. L'été suivant, ils développèrent l'interprétation procédurale des implications, et établirent qu'on pouvait restreindre les clauses aux clauses de Horn, correspondant à des implications où antécédents et conséquent sont des énoncés atomiques.

1976 vit un premier portage de Prolog sur micro-ordinateur.

En 1977, D. Warren développa à Édimbourg un compilateur Prolog, qui apporta à Prolog la performance qui lui manquait. Le Prolog d'Édimbourg devint ainsi un standard

En 1982 sortit Prolog II, qui utilisait des systèmes d'équations plutôt que l'unification, et abordait le traitement des arbres infinis.

À partir de 1987, Prolog III intégrait au niveau de l'unification : une représentation des arbres rationnels (éventuellement infinis), avec un traitement spécifique pour les listes ; un traitement complet de l'algèbre de Boole ; un traitement numérique portant sur l'addition, la multiplication par une constante et les relations usuelles.

En 1996, Prolog IV s'attaqua résolument au traitement des contraintes. Programmer par contraintes consiste à formuler un problème en termes d'inconnues soumises à une contrainte, énoncé du premier ordre faisant intervenir des opérations et des relations du domaine de calcul. Résoudre la contrainte, et par là le problème, consiste à trouver les valeurs à attribuer aux variables libres de la formule pour la rendre vraie, ce qui unifie la programmation logique et la programmation mathématique (au sens de la recherche opérationnelle). Au prix d'un moteur dix fois plus gros que pour Prolog II, Prolog IV traite un vaste jeu de contraintes, allant des contraintes sur les listes et les arbres aux contraintes numériques, en passant par les contraintes traitées par réduction des intervalles de valeur, s'appliquant aussi bien aux réels qu'aux entiers voire aux booléens[11].

I.2.4.2- Paradigme logique:

Le représentant classique de cette catégorie est Prolog, mais il a perdu son monopole, de plus en plus souvent on parle des langages hybrides (surtout logico/fonctionnels) avec la couche logique très importante : Life, Oz, Leda ou Opal. Le Prolog reste néanmoins le langage logique numéro 1. (En plus, il existe en plusieurs dialectes.) Nous ne pouvons ici définir ce langage.

– Le langage est statique, comme des langages fonctionnels. On n'a pas le droit de modifier une variable, elle se confond sémantiquement avec sa valeur. Il n'y a pas de boucles autres que les appels récursifs terminaux, et les boucles du backtracking, mentionnées ci-dessous.

– La dépendance fonctionnelle entre données (y = f(x)) s'est généralisée en relation plus universelle, et la représentation linguistique d'une relation est un prédicat qui a la forme d'un appel procédural, par exemple r(x, y, [2, x]). Un prédicat qui réalise une relation peut représenter une fonction, par exemple plus(A,B,C) qui modélise l'énoncé : C=A+B, ou peut dénoter une contrainte ou un attribut : négative(X), etc.

– (Une propriété accessoire, mais très typique pour la famille Prolog : la forme syntaxique des structures de données composites (les termes) est la même que la forme des prédicats. On peut construire une fraction par une expression type frac (Num,Den), Où Num et Den sont des expressions (constantes ou variables) qui représentent les champs d'une fraction, et le mot frac définit le type de cette donnée.)

– Le mécanisme décisionnel fondamental est l'Unification (=) des termes simples et composites qui automatise les constructions et les analyses des données, par exemple l'unification combinée Z=..[F,X,a], Z=g(p(A,Y),Y) instancie automatiquement les variables suivantes :

F=g

Y=a

X=p(A,a).

Ce qui rend la programmation en Prolog très compacte. (Pour suivre cet exemple il faut savoir que :

– Les lettres majuscules représentent les variables, et les minuscules sont des constantes symboliques.

– L'unification est une sorte d'équivalence. p(A)=p(B) ssi A=B.

– l'opération (=..) transforme tout terme en liste linéaire, par exemple F(a,B,c) en [F,a,B,c]. La transformation est créatrice, le terme original reste intact, mais la création d'une nouvelle liste a lieu).

– Le langage est non-déterministe et équipé avec le mécanisme de backtracking :

Quelques relations peuvent être ambigües et admettre plusieurs solutions. Prolog possède les moyens de les récupérer toutes. On construit une solution partielle en sauvegardant le contexte du calcul. Si l'utilisateur (ou l'étape suivante du programme) rejette cette solution, Prolog en cherche automatiquement une autre. Ceci augmente de manière considérable la puissance sémantique du langage, et rend sa compilation très difficile.

Une boucle de type backtracking consiste à rejeter plusieurs fois la solution offerte par le programme. On déclare localement l'échec, et le programme "retourne sur ses pas" au point, Où il avait une décision non-déterministe à prendre. Il marque les chemins parcourus, et choisit un autre. Cette technique est connue au moins depuis les temps de Thésée, Minotaure, et Ariane.

En fait, on a besoin de techniques un peu spéciales pour compiler le non déterminisme et l'unification complète. Trop souvent un cours universitaire classique de compilation ignore totalement ce domaine. Nous n'avons pas le temps de le traiter non plus, il faut cependant remarquer qu'un progrès formidable a été fait – grâce aux continuations et à la construction d'une machine virtuelle très simple, mais puissante : WAM – la machine abstraite de Warren, les implantations de Prolog jadis rares, sont devenues des exercices standard pour les étudiants.

En particulier, la construction d'une machine non déterministe à l'aide d'un langage fonctionnel paresseux, est un vrai plaisir intellectuel.

– Finalement, Prolog dispose d'un concept étranger à presque tous les autres langages de programmation : celui de variable logique. Une variable logique ressemble à une variable quelconque, mais elle n'a pas de valeur [11].

I.2.4.3- La logique des prédicats du premier ordre:

✓ théorèmes (utilisant la logique du premier ordre:

• Un programme en Logique est un ensemble d'énoncés.

• Les énoncés sont des formules du calcul du premier ordre ne contenant pas de variables libres.

• Tout problème calculable peut être formulé dans ce langage.

•Programmer en logique consiste :

- à définir les hypothèses (énoncés definis dans un programme exprimant la connaissance relative au problème à résoudre.

- à introduire la conclusion (poser le problème).

✓ Syntaxe:

o Éléments de base:

• Constantes : a, b, c,...

• Variables : x, y, z,...

• Fonctions : f, g, h, ...

• Relation (prédicat ou fonctions booléennes): P, Q, R

o définition d'un terme:

• Toute constante est un terme.

• Toute variable est un terme.

• Si t1, t2, ..tn sont des termes et si f est une fonction n-aire, alors f (t1, t2, ..,tn) est un terme.

o définition d'un atome:

• Si t1, t2, ..., tn sont des termes et si P est un prédicat n-aires alors

P (t1, t2, .., tn) est un atome.

o Les formules bien formées (fbf):

• Tout atome est une formule.

• Si P et Q sont deux formules alors : Non P, P & Q, P V Q, P ==> Q, P <==> Q, ∀ x P(x), ∃x P(x) sont des formules.

✓ Sémantique:

• Quand on attribue des valeurs à chaque terme et à chaque Symbole de prédicat dans une fbf on dit qu'une interprétation est donnée a la fbf.

• Si les valeurs de vérité de fbf sont les mêmes pour toute interprétation on dit qu'ils sont équivalents.

✓ Propriétés:

• L'évaluation de fbf complexes peut être facilité par une phase de substitution (transformation syntaxique)

• On dispose alors des propriétés suivantes :

- F, G et H sont des fbf ne contenant pas de variable F[x] est une fbf contenant la variable x.

- Double négation NON (NON F) = F

- Commutativité F & G = G & F F V G = G V F

- Associativité F & (G & H) = (F & G) & H

F V (G V H) = (F V G) V H

- Distributive F V(G & H) = (F V G) & (F V H)

F &(G V H) = (F & G) V (F & H)

NON (F V G) = NON F & NON G

- F ==> G = NON F V G

- F <==> G = (NON F V G)&(NON G VF)

- NON (∀x F[x]) = ∃x NON F[x]

- NON (∃x F[x]) = ∀x NON F[x]

- ∀x F[x] & ∀x G[x] = ∀x (F[x] & G[x])

- ∃X F[x] V ∃x G[x] = ∃x (F[x] V G[x])

✓ Inconsistance et validité d'une formule:

• fbf inconsistante (in satisfiable) fausse pour toute interprétation.

• fbf valide : vraie pour toute interprétation.

• Conséquences:

- fbf valide ==> fbf satisfiable

- fbf inconsistante ==> fbf invalide

• Q est une conséquence logique de P1, P2, ...Pn] <==>[Si P1 & p2 ...& Pn est vraie pour toute interprétation, alors Q est aussi vrai].

✓ Variables libres, variables liées:

• Dans l'expression:

∀x (P(x) ==> Q(x, y)) . x est une variable liée et y est une variable libre. Une expression ne peut être évaluée que lorsque toutes les variables sont liées. Aussi, nous exigeons que toutes les fbf contiennent que les variables liées.

✓ Forme normale conjonctive:

• Soient F1, F2, ...des formules :

- Si chaque Fi est uniquement sous forme disjonctive de littéraux(V), alors la forme F1 & F2 &.....Fn est dite une forme normale conjonctive.

✓ Forme normale disjonctive:

• Soient F1, F2, ...des formules :

- Si chaque Fi est uniquement sous forme conjonctive de littéraux(V), alors la forme F1 V F2 V.....Fn est dite une forme normale disjonctive.

✓ Règle d'inférence:

•Les règles d'inférence fournissent un moyen de faire des démonstrations ou des déductions logiques.

•L'utilisation de la table de vérité est un moyen de démonstration (sémantique). D'autres méthodes reposent uniquement sur les transformations syntaxiques. Elles utilisent alors les règles d'inférences suivantes : Modus Ponens, Modus tollens, Chaînage, Substitution, Conjonction, contraposée.

• Conjonction: De P et de Q, on déduit P & Q

• Contraposée: De P --> Q, on déduit Non Q --> Non P

P --> Q

I.3-Interpréteur:

Elle exécute le code source "directement" de façon que les instructions d'une boucle sont scannées et analysées à chaque itération.

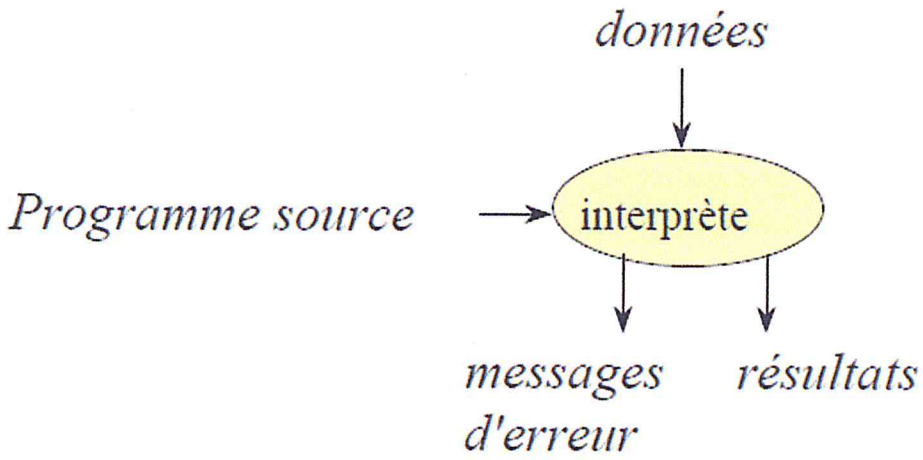

Figure I.1 : fonctionnement d'un interpréteur.

I.3.1-Principe:

L'interprétation repose sur l'exécution dynamique du programme par un autre programme (l'interprète), plutôt que sur sa conversion en un autre langage (par exemple le langage machine), elle évite la séparation du temps de conversion et du temps d'exécution, qui sont simultanés.

On différencie un programme dit script, d'un programme dit compilé :

Un programme script est exécuté à partir du fichier source via un interpréteur de script.

Un programme compilé est exécuté à partir d'un bloc en langage machine issu de la traduction du fichier source.

Le cycle d'un interprète est le suivant :

-Lire et analyser une instruction (ou expression).

-si l'instruction est syntaxiquement correcte, l'exécuter (ou évaluer l'expression).

-passer à l'instruction suivante.

Ainsi, contrairement au compilateur, l'interprète exécute les instructions du programme (ou en évalue les expressions), au fur et à mesure de leur lecture pour interprétation. Du fait de cette phase sans traduction préalable, l'exécution d'un programme interprété est généralement plus lente que le même programme compilé. La plupart des interprètes n'exécutent plus la chaîne de caractères représentant le programme, mais une forme interne, tel qu'un arbre syntaxique.

En pratique, il existe une continuité entre interprètes et compilateurs. Même dans les langages compilés, il subsiste souvent une partie interprétée (souvent les formats d'impressions 'FORMAT' de Fortran, 'printf' de C…, restent interprétés). Réciproquement, la plupart des interprètes utilisent des représentations internes intermédiaires (arbres syntaxiques abstraits, ou même code octet) et des traitements (analyses lexicale et syntaxique) ressemblant à ceux des compilateurs.

Enfin, certaines implémentations de certains langages (par exemple SBCL pour Common Lisp) sont interactifs comme un interprète, mais traduisent dès que possible le texte d'un bout de programme en du code machine directement exécutable par le processeur.

Le caractère interprétatif ou copulatoire est donc propre à l'implémentation d'un langage de programmation, et pas au langage lui-même.

L'intérêt des langages interprétés réside principalement dans la facilité de programmation et dans la portabilité.

Les langages interprétés facilitent énormément la mise au point des programmes car ils évitent la phase de compilation, souvent longue, et limitent les possibilités de bogues.

Il est en général possible d'exécuter des programmes incomplets, ce qui facilite le développement rapide d'applications ou de prototypes d'applications.

Ainsi, le langage BASIC fut le premier langage interprété à permettre au grand public d'accéder à la programmation, tandis que le premier langage de programmation moderne interprété est Lisp.

La portabilité permet d'écrire un programme unique, pouvant être exécuté sur diverses plates-formes sans changements, pourvu qu'il existe un interprète spécifique à chacune de ces plates-formes matérielles.

Un certain nombre de langages informatiques sont aujourd'hui mis en œuvre au moyen d'une machine virtuelle applicative.

Cette technique est à mi-chemin entre les interprètes tels que décrits ici et les compilateurs.

Elle offre la portabilité des interprètes avec une bonne efficacité. Par exemple, des portages de Java, Lisp, Scheme, Ocaml, Perl (Parrot), Python, Ruby, Lua, C, etc. sont faits via une machine virtuelle.

L'interprétation abstraite (inventée par Patrick et Radhia Cousot) est une technique et un modèle d'analyse statique de programmes qui parcourt, un peu à la manière d'un interprète, le programme analysé en y remplaçant les valeurs par des abstractions.

Par exemple, les valeurs des variables entières sont abstraites par des intervalles d'entiers, ou des relations algébriques entre variables [12].

I.3.2-Historique:

Avec l'apparition du langage compilé Pascal et de compilateurs commerciaux rapides comme Turbo Pascal, les langages interprétés connurent à partir du milieu des années 1980 un fort déclin. Trois éléments changèrent la donne dans les années 1990 .

avec la nécessité d'automatiser rapidement certaines tâches complexes, des langages de programmation interprétés (en fait, semi-interprétés) de haut niveau comme Tcl, Ruby, Perl ou Python se révélèrent rentables.

la puissance des machines, qui doublait tous les dix-huit mois en moyenne (selon la loi de Moore), rendait les programmes interprétés des années 1990 d'une rapidité comparable à celle des programmes compilés des années 1980 [12].

I.3.3-Utilisations des langages interprétés:

Les langages interprétés trouvent de très nombreuses utilisations :

Dans le domaine éducatif, les langages interprétés permettent de se concentrer sur les algorithmes et les structures de données, et non sur les particularités de tel ou tel langage. Les calculs scientifiques ne demandant pas de calcul intensif (itérations sur de très grandes matrices, par exemple) peuvent s'écrire avec profit dans un langage interprété. Ils permettent d'appeler des algorithmes de calcul performants précompilés. Les systèmes de calcul symbolique utilisent aussi cette possibilité. Les interprètes de ligne de commande (désignés par le nom shell dans la terminologie Unix).

Ces interprètes sont capables de comprendre des commandes frappées sur un clavier ou en provenance d'une autre source. Ils disposent d'une syntaxe spécifique à chaque système

d'exploitation, et permettent de gérer les ressources matérielles d'une machine (disques, mémoire centrale, entrées/sorties, etc.) ainsi que la communication entre les programmes.

Il existe des extensions permettant la programmation rapide d'interfaces graphiques à l'aide de langages interprétés. Le plus répandu est Tcl/Tk, mais il existe également Python/Tk, Python/wxWidgets, Perl/wxWidgets Python/Qt ou encore Gambas .

Dans le monde industriel, de plus en plus de machines sont pilotables par un langage interprété : les robots industriels, les machines-outils (APT, langage ISO (ou blocs)), les traceurs de plan, souvent pilotés en PostScript [12].

I. 4-C'est quoi une grammaire?

La conception de la grammaire d'un langage de programmation est la partie principale de la conception de cette langue.

Les grammaires de la langue ordinateur peut être décrite en utilisant une notation couramment utilisée s'appelle la BNF (Notation Backus-Naur) [13].

I.5- Analyse lexicale:

L'analyse lexicale est la première phase de la compilation. Dans le texte source, qui se présente comme un flot de caractères, l'analyse lexicale reconnait des unités lexicales, qui sont les " mots ".

Les principales sortes d'unités lexicales qu'on trouve dans les langages de programmation courants sont :

- les caractères spéciaux simples : +, =,... etc.

- les caractères spéciaux doubles : <=, ++,... etc.

-les mots-clés : if, while, ...etc.

- les constantes littérales : 123, -5,... etc.

- et les identificateurs.

A propos d'une unîtes lexicale reconnue dans le texte source on doit distinguer quatre notions importantes :

- l'unîtes lexicale, représentée généralement par un code conventionnel.

- le lexème, qui est la chaîne de caractères correspondante.

-un attribut, qui dépend de l'unité lexicale en question, et qui la complète.

Outre la reconnaissance des unîtes lexicales, les analyseurs lexicaux assurent certaines tâches mineures comme la suppression des caractères de décoration (blancs, tabulations, etc.) et celle des commentaires (généralement considères comme ayant la même valeur qu'un blanc), l'interface avec les fonctions de lecture de caractères, a travers lesquelles le texte source est acquis, la gestion des fichiers,... etc. [14].

I.6-Analyse syntaxique:

L'analyse syntaxique contrôle la bonne forme d'une séquence de terminaux pour déterminer si elle constitue une phrase du langage

Un analyseur syntaxique (parser) est basé sur un accepteur, fonction booléenne indiquant si le texte source est conforme aux règles de grammaire définissant le langage.

Un texte de plusieurs milliers de lignes dans un langage donné est une phrase au vu de la grammaire de ce Langage

Lors de l'analyse syntaxique d'une séquence de terminaux:

• il faut exhiber un arbre de dérivation pour cette séquence, comme sous-produit de son acceptation

• seule sa structure compte, peu importe dans quel ordre il est obtenu [14].

I.6.1-Relation entre analyseurs lexical et syntaxique:

Cette relation est du type producteur-consommateur et peut être implantée de plusieurs manières:

• L'analyseur syntaxique appelle la procédure «analyse lexicale» lorsqu'il a besoin du terminal suivant.

• en Pascal-S, cette procédure s'appelle insymbol ().

• dans le cas de l'analyse prédictive de Lista, c'est AccepterUnTerminal ().

• l'analyseur syntaxique réactive la coroutine «analyse lexicale» lorsqu'il a besoin du terminal suivant.

• une première passe «analyse lexicale» produit une séquence de terminaux en mémoire vive, qui est ensuite relue par une seconde passe «analyse syntaxique». Ce stockage explicite n'apporte rien.

• une première passe «analyse lexicale» produit un fichier de terminaux qui est ensuite relu par une seconde passe «analyse syntaxique».

• «Analyse lexicale» et «analyse syntaxique», processus concurrents, se synchronisent lorsque ce dernier a besoin du prochain terminal [15].

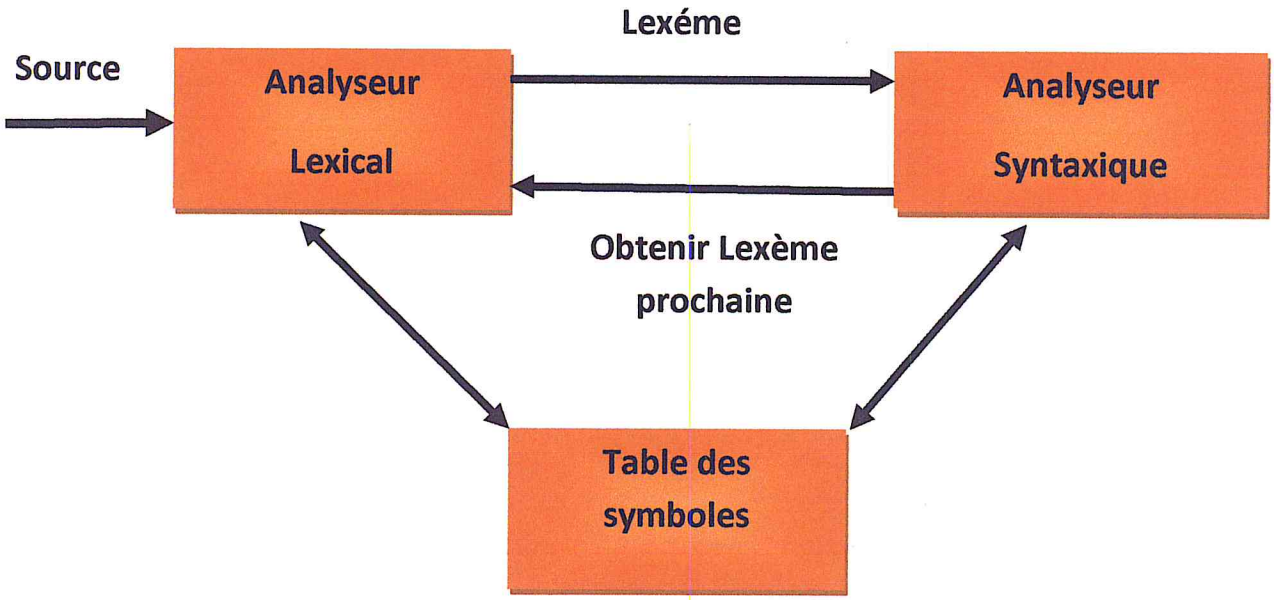

Figure I.2 : La relation entre l'analyseur syntaxique et l'analyseur lexical.

I.7-Analyse sémantique:

Les aspects lexicaux et syntaxiques d'un langage ne sont qu'un support pour l'essentiel, à savoir la sémantique véhiculée par les phrases du langage

Par exemple, le fragment de code C++:

int i = 2.567. est correct lexicalement et syntaxiquement, mais il contient une erreur sémantique.

On ne peut en effet par affecter une valeur flottante à une variable entière dans ce langage.

L'analyse sémantique effectue les vérifications de sémantique, c'est-à-dire de Signification, sur le code source en cours de compilation [15].

I.7.1-Bases de l'analyse sémantique:

Pour cette analyse:

• on se base sur la définition du langage, qui précise le sens des phrases bien formées syntaxiquement

• on s'appuie sur des structures de données représentant la source en cours de compilation

Il n'est pas strictement nécessaire de s'appuyer sur une forme source textuelle: il est tout à fait possible de faire l'analyse sémantique d'informations stockées directement dans des structures de données [14].

I.7.2-Structure de la table des symboles:

La table des symboles (symbol table) est formée de l'ensemble des dictionnaires contenant les identificateurs déclarés.

Dans certains langages, comme Fortran IV, on compilait indépendamment un programme (program), une procédure (subroutine) ou une fonction (function).

Il n'y a donc à chaque instant qu'un niveau de déclarations propre au source en cours de compilation en plus des identificateurs prédéfinis et des zones de communs (common), visibles dans tous les cas selon la sémantique du langage considéré, il peut être nécessaire de construire la table de tous les dictionnaires comme structure de données dans le langage d'implantation, pour faire les contrôles sémantiques[14].

I.8- Conclusion :

Dans ce chapitre nous avons étudié les différents langages de programmation, ainsi que les caractéristiques de chaque mode programmation.

Dans le chapitre suivant, nous allons étudier le langage logique que nous avons réalisé.

CHAPITRE II

Contribution

II.1- Introduction :

Nous avons choisi USDBPROLOG comme un nom de notre langage logique, USDB est Université Saad Dahleb Blida.

Nous allons définir la constitution du programme USDBPROLOG, les termes, les opérateurs, comment faire la substitution, l'unification et l'implémentation de notre prolog.

II.2- Conception d'un langage logique

II.2.1-Constitution d'un programme USDBPROLOG:

II.2.1.1- Les faits :

Les faits sont des données élémentaires qu'on considère vraies. Ce sont des formules atomiques constituées du nom d'un prédicat (c'est à dire d'une relation (au sens mathématique du terme)) suivi entre parenthèse d'une liste ordonnée d'arguments qui sont les objets du problème principal ou d'un sous-problème.

Un programme USDBPROLOG est au moins constitué d'un ou plusieurs fait(s) car c'est à partir de lui (d'eux) qu'USDBPROLOG va pouvoir rechercher des preuves pour répondre aux requêtes de l'utilisateur.

Exemple :

```
pere(mohamed,ali).
mere(aicha,leila).
ancetre(Adam,X).
plusfort(superman,X).
factorielle(0,1).
```

II.2.1.2- Les règle:

Un programme USDBPROLOG contient presque toujours des règles, cependant ce n'est pas une obligation.

Si les faits sont les hypothèses de travail d'USDBPROLOG, les règles sont des relations qui permettent à partir de ces hypothèses d'établir de nouveaux faits par déduction (si on a démontré F1 et F1□F2 alors on a démontré F2).

Exemple :

plusfort(X,Y):-invincible(X).

grandPere(X,Y) :-père(X,Z),père(Z,Y).

II.2.2- les Termes:

Un programme USDBPROLOG est constitué de trois types d'éléments différents :

• Des symboles de constantes : 0, 1, 2002… omar, superman…

• Des symboles de fonctions : s,g(x) , sqrt…

• Des variables (tout ce qui commence par une majuscule): X, FacExac, Qui…

USDBPROLOG traite ces éléments en les regroupant en « termes ».

Définition :

-toute variable est un terme

- toute constante est un terme

-si f est un symbole de fonction à n arguments et si t1, t2, …tn sont des termes alors f(t1, …tn) est un terme.

Exemple :

X, Qui… - 2, pi, ali… - 2+pi, sqrt(2*pi*N), f(X1,2)…

En fait, il est possible de représenter n'importe quel terme par un arbre.

Et inversement tout arbre ou sous-arbre est un terme.

$\sqrt{2\pi n} \times (\frac{n}{e})^n \pi 2 \equiv$ sqrt(2*2pi*N)*(N/e)**N est un terme représentable par :

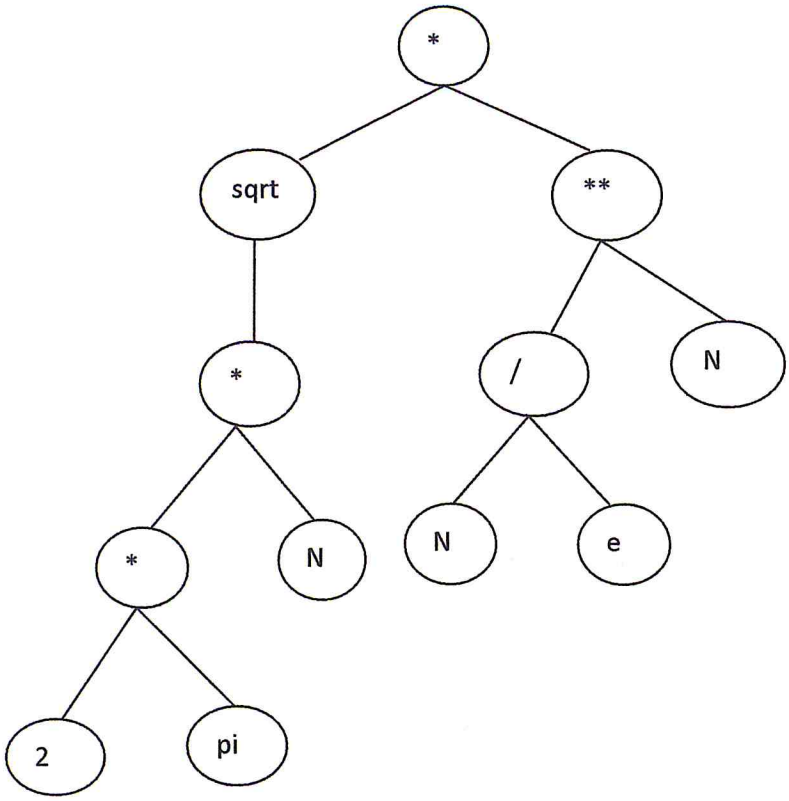

Figure II.3 : Représentation d'un terme par un arbre.

II.2.3- Les variables:

Une variable (ou inconnue) peut remplacer n'importe quel terme USDBPROLOG, elle se commence par les lettres majuscules.

Exemple :

Père (mohamed, X).

X : est une variable.

Mohamed : est une constante.

II.2.4- Constantes :

USDBPROLOG possède des constantes qui se subdivisent en atomes et nombres.

Les atomes comprennent les noms (roi, mourad, aicha, nil), les symboles (comme « :- » ou « + ») et les chaînes de caractères (délimités par « ' »).

Les nombres peuvent êtres des entiers (34, -45, +12) ou des réels (-34.14,0.987, -0.4e+2).

II.2.5- Fonctions :

USDBPROLOG manipule les fonctions.

Exemple :

La fonction factorielle est définie en USDBPROLOG comme suit :

fact(0,1).

fact(X,Y):-X>0 , X1 est X-1 , fact(X1,Z) , Y est Z*X.

II.3- L'implémentation :

II.3.1- Interpréteur USDBPROLOG:

o Analyse Lexicale

Cette analyse consiste à segmenter un texte source en un ensemble de mots que l'on appelle traditionnellement « tokens» (le terme exact est « lexème», ce qui signifie unité lexicale). Il s'agit d'une part de déterminer la suite des caractères comprenant le token, et d'autre part d'identifier le type de token (identificateur, nombre, opérateur, etc.).

Le but de cette analyse est de vérifier la validité du vocabulaire employé par le programmeur. L'analyseur lexical est basé sur un automate fini déterministe capable de reconnaître chacun des mots autorisé par le langage.

Principe : l'analyse lexicale peut être réalisée par deux méthodes :

- méthode des procédures
- méthode des automates.

Nous avons utilisé la méthode des procédures car d'une part elle est simple a implémenter et d'autre part elle est facilite l'extensibilité du code.

✓ Symboles non-alphanumériques :

Nom de l'unité lexical	Symbole de l'unité lexical
DebutComment	/*
FinComment	*/
UniqueComment	%
Quote	"
Arobase	@
Point	.

Implique	:-
Antislash	\
Plus	+
Produit	*
Moin	-
Div	/
Vergule	,
Antislash	\
DAntiSlash	\\
DArobase	@@
ParenthOuvr	(
ParenthFerm	)

✓ **Mots-clés :**

Ecrire, consulter, est.

✓ **Ne sont pas des unités lexicales et doivent être ignorés :**

espace, retour à la ligne, saut de ligne et commentaire (entre /* et */).

○ Analyse syntaxique :

La syntaxe du langage régit la forme des phrases: les phrases acceptables au vu de la définition syntaxique appartiennent au langage, les autres n'y appartiennent pas

La définition syntaxique d'un langage s'appuie sur:

• les terminaux, définis au niveau syntaxique

• des règles de bonne forme pour des séquences de terminaux appelées notions non-terminales

Les terminaux sont écrits en majuscule et les non-terminales en minuscule

Grammaire du prolog :

Prolog ⟶ Regles | Faits

Faits ⟶ Fait . Faits | ξ

Regles ⟶ Regle . Regles | ξ

Fait ⟶ Terme

Regle ⟶ Terme :- Termes

Terme ⟶ Nombres | Atomes | Variables | Atomes (Termes)

Termes ⟶ Terme | Terme , Termes | Terme ; Termes

Nombres ⟶ Nombre Nombres | ξ

Nombre ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Atomes ⟶ Atome Atomes | ξ

Atome ⟶ a | b | c | d | ... | z | + | - | / | *

Variables ⟶ Variable Variables

Variable ⟶ A | B | C | ... | Z

II.3.2-Substitutions :

Définition :

Soient Y1, …, Yk k variables toutes différentes τ1, …, τk k termes différents ou non Une substitution σ est l'association qui fait correspondre : τ1 à Y1 … τk à Yk.

Remarque :

Si on reprend la représentation d'un terme par un arbre, une substitution correspond à la greffe d'un arbre τi à chaque occurrence de Yi.

Exemple :

Soit la substitution σ : n ⟶ $(2x + 5y)^2$

A partir des arbres :

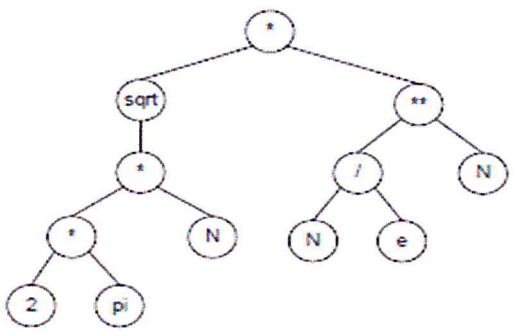

Figure II.4 : Représentation de substitution σ par un arbre.

ET

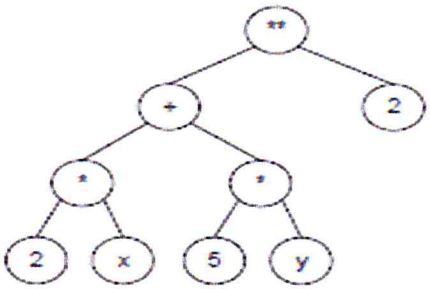

Figure II.5 : Représentation de substitution σ par un arbre.

On obtient l'arbre :

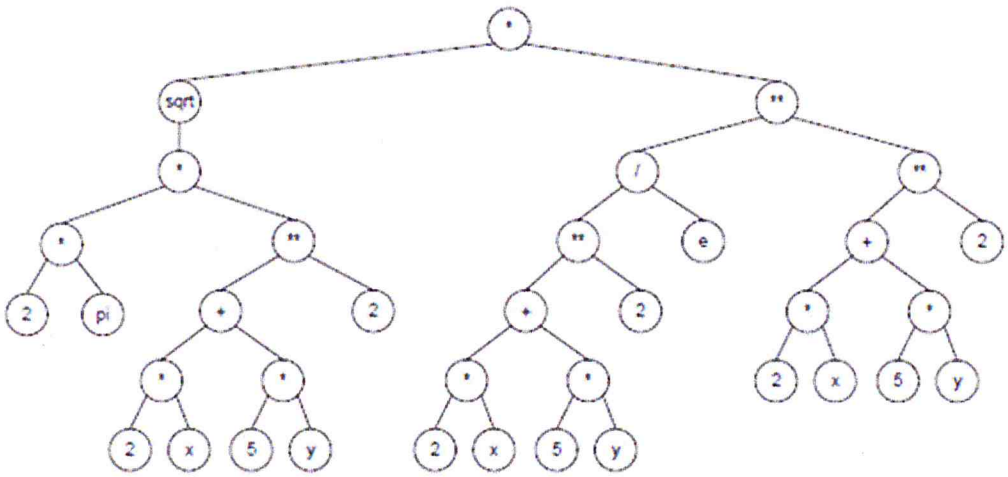

Figure II.6 : Représentation du terme après la substitution.

Cet arbre représente l'expression :

$$\sqrt{2\pi(2x+5y)^2}\ \text{x}\ \left(\frac{2x+5y)^2}{e}\right)^{(2x+5y)^2}$$

Lorsque la substitution agit sur plusieurs variables, tous les remplacements sont faits en parallèle.

II.3.3-Unification :

Définition :

Soient deux termes t1 et t2 Unifier t1 et t2 c'est trouver la substitution la plus générale qui permet de faire de t1 et t2 un seul et même terme.

II.3.4-Résolution:

Pour trouver une preuve et afficher la solution associée, USDBPROLOG utilise deux piles de piles :

• La pile des buts dans laquelle il stocke tous les niveaux de buts qu'il a cherché à atteindre. Un niveau de buts est lui-même représenté par une pile de buts qui contient toujours au moins un but sauf lorsqu'on a atteint la fin de la démonstration (on peut alors retourner le résultat en cas de succès ou non sinon). Une preuve est donc l'ensemble des buts par lesquels on est passé avant d'obtenir une pile de niveau de buts vide.

• La pile d'environnement est elle-aussi une pile de piles car à chaque niveau de buts est associée une pile d'environnement de niveau de buts. USDBPROLOG stocke dans chacune de ces sous-piles les variables intervenues dans l'unification qui l'a amené à ce niveau. Dans la pile d'environnement, il conserve donc toutes les variables qui ont été utilisées pendant la démonstration. Elle lui sert ainsi à retrouver le résultat de la preuve en cas de succès.

Le changement de niveau de but correspond à un pas d'itération dans la démonstration il est validé par une unification. En effet, pour prouver un but USDBPROLOG :

• Soit trouve un fait pour lequel il existe une unification avec le but. Dans ce cas le but courant est démontrable et itérativement la requête est démontrable.

• Soit essaie d'appliquer la règle dont le membre gauche (la conclusion) est unifiable avec le but. S'il en existe une, alors il se fixe comme nouveau(x) but(s) la (ou les) formule(s) atomique(s) du membre droit de cette règle. Il passe ainsi à un autre niveau de but. Si plusieurs règles ont un membre gauche unifiable, USDBPROLOG choisit la première dans le code du programme. Si celle-ci mène à une démonstration (succès), l'utilisateur peut demander une autre solution. Alors USDBPROLOG remonte au dernier choix qu'il a du faire et applique la règle suivante s'il y en a encore une, sinon il remonte au choix précédent et s'il

n'y a plus aucun choix possible il retourne non. Le schéma suivant montre comment se fait la démonstration.

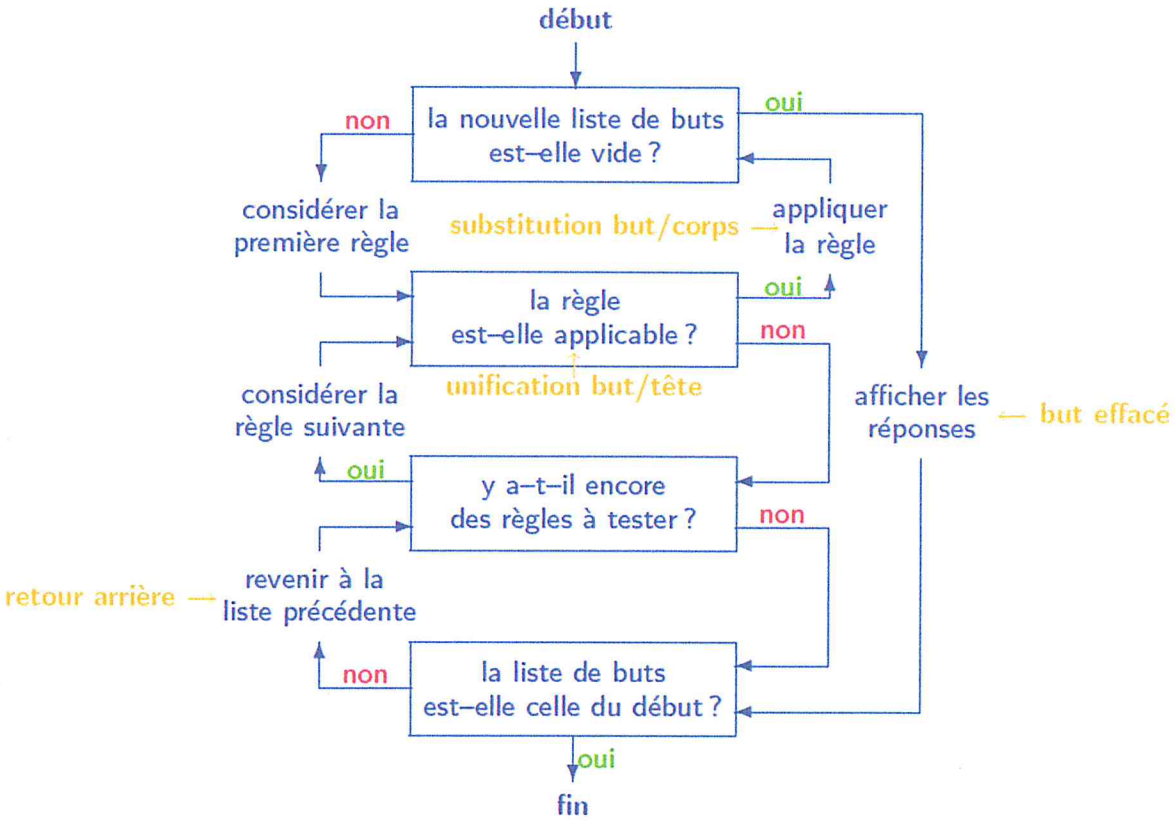

Figure II.7 : résolution d'une requête.

Exemple :

Soit la base faits de suivant :

parent (ahmed,ali). % (1)

parent (ali,omar). % (2)

ancetre1(X, Y) :- parent(X, Y). % R1

ancetre1(X, Y) :- parent(X, Z), ancetre1 (Z, Y). % R2

ancetre2(X, Y) :-parent(X, Z), ancetre2 (Z, Y). % R2

ancetre2(X, Y) :-parent(X, Y). % R1

Si on pose la question ancetre(X,omar).USDBPROLOG va prouver la question comme suit :

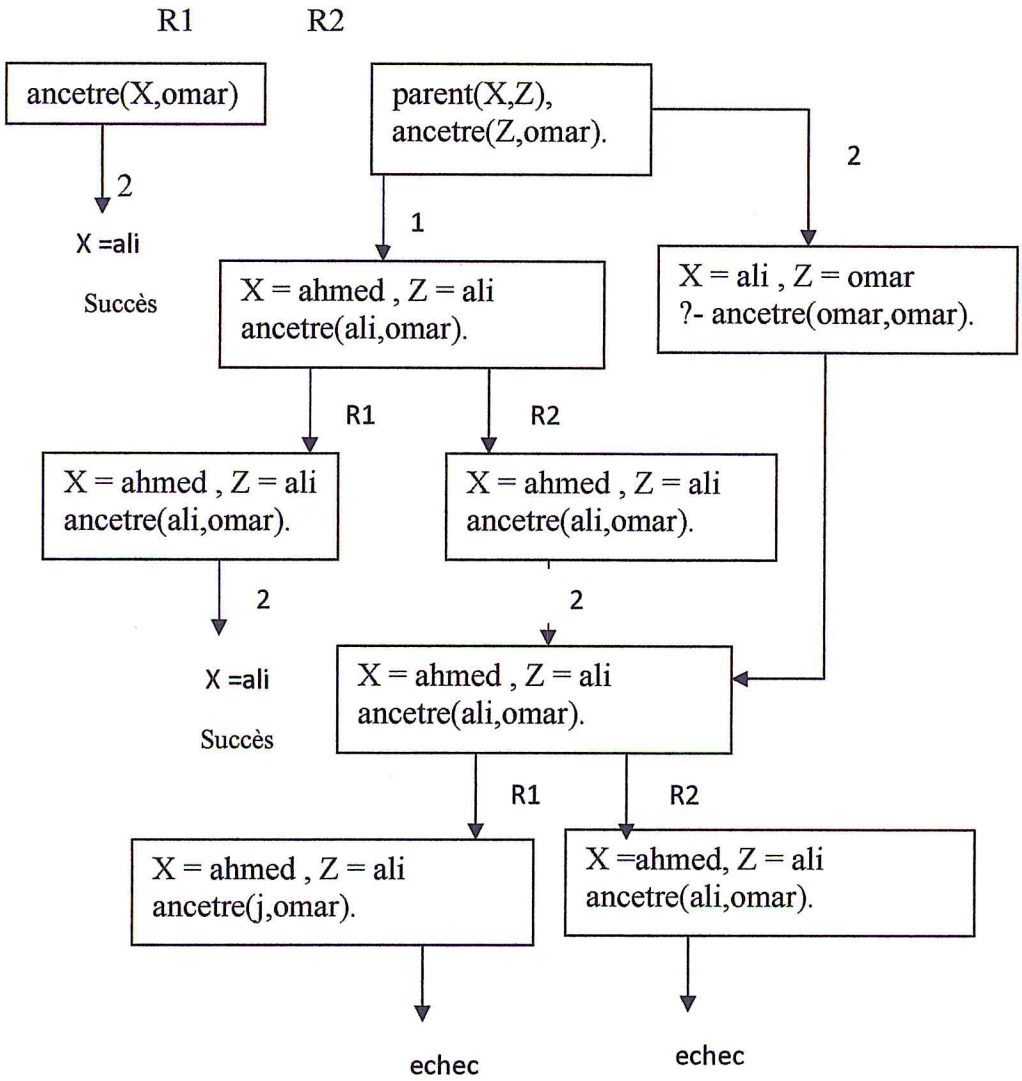

Figure II.8: arbre de recherche.

II.4- Conclusion:

Dans ce chapitre nous avons défini la constitution du programme USDBPROLOG, analyse lexical et analyse syntaxique du USDBPROLOG, les termes, comment faire la substitution, l'unification et l'implémentation de notre prolog.

CHAPITRE III

Mise en œuvre

III.1- Introduction :

Dans ce chapitre nous allons voire l'environnement de programmation utilisé et des captures d'écran de l'application usdbprolog .

III.2- environnement de programmation utilisé:

Nous avons utilisé c# comme un logiciel de développement.

III.2.1- Définition de C#:

Le C# est un langage de programmation orienté objet à typage fort, créé par la société Microsoft, et notamment un de ses employés, Anders Hejlsberg, le créateur du langage Delphi.

Il a été créé afin que la plate-forme Microsoft .NET soit dotée d'un langage permettant d'utiliser toutes ses capacités. Il est très proche du Java dont il reprend la syntaxe générale ainsi que les concepts (la syntaxe reste cependant relativement semblable à celle de langages tels que le C++ et le C). Un ajout notable à Java est la possibilité de surcharge des opérateurs, inspirée du C++. Toutefois, l'implémentation de la redéfinition est plus proche de celle du Pascal Objet [15].

III.2.2- Capacités du langage:

Le C# est, d'une certaine manière, le langage de programmation qui reflète le mieux l'architecture Microsoft .NET qui fait fonctionner toutes les applications .NET, et en est par conséquent extrêmement dépendant. Les types natifs correspondent à ceux de .NET, les objets sont automatiquement nettoyés par un ramasse-miettes (garbage collector en anglais), et beaucoup de mécanismes comme les classes, interfaces, délégués, exceptions, ne sont que des moyens explicites d'exploiter les fonctionnalités de la bibliothèque .NET. Pour achever de marquer cette dépendance, le CLR (Common Language Runtime) est obligatoire pour exécuter des applications écrites en C#, comme l'est la JVM (Java Virtual Machine ou Machine virtuelle Java) pour des applications Java.

Le langage compte un certain nombre de changements par rapport au C/C++ ; On notera particulièrement les points suivants :

La manipulation directe de pointeurs ne peut se faire qu'au sein d'un code marqué unsafe, et seuls les programmes avec les permissions appropriées peuvent exécuter des blocs de code unsafe.

La plupart des manipulations de pointeurs se font via des références sécurisées, dont l'adresse ne peut être directement modifiée, et la plupart des opérations de pointeurs et d'allocations sont contrôlées contre les dépassements de mémoire.

Les pointeurs ne peuvent pointer que sur des types de valeurs, les types objets, manipulés par le ramasse-miettes, ne pouvant qu'être référencés.

Les objets ne peuvent pas être explicitement détruits. Le ramasse-miettes s'occupe de libérer la mémoire lorsqu'il n'existe plus aucune référence pointant sur un objet.

Toutefois, pour les objets gérant des types non managés, il est possible d'implémenter l'interface IDisposable pour spécifier des traitements à effectuer au moment de la libération de la ressource.

L'héritage multiple de classes est interdit, mais une classe peut implémenter un nombre illimité d'interfaces, et une interface peut hériter de plusieurs interfaces.

Le C# ne gère pas les templates, mais cette fonctionnalité a été remplacée par les types génériques apparus avec C# 2.0.

Les propriétés ont été introduites, et proposent une syntaxe spécifique pour l'accès aux données membres [15].

III.2.3- Différences entre Java et C#:

Bien que le C# soit similaire à Java, il existe des différences notables, par exemple :

-Java n'autorise pas la surcharge des opérateurs,

-Java n'a pas de mode unsafe permettant l'arithmétique de pointeurs,

-Java a des exceptions vérifiées, alors que les exceptions du C# ne sont pas vérifiées, comme en C++,

-Java permet la génération automatique de la documentation HTML à partir des fichiers sources à l'aide des descriptions Javadoc-syntax, tandis que le C# utilise des descriptions basées sur le XML,

-C# supporte indexers (indexeurs), delegates (délégué ou liste de pointeurs sur fonctions) et events (événements),

-C# supporte les structures en plus des classes (les structures sont des types valeur : on stocke le contenu et non l'adresse),

-C# utilise une syntaxe intégrée au langage (DllImport) et portable pour appeler une bibliothèque native, tandis que Java utilise Java Native Interface [16].

Pour plus d'information voir ANNEXE.

III.3- USDBPROLOG et Les systèmes d'exploitation:

Les systèmes d'exploitation pris en charge par le logiciel USDBPROLOG sont:

-Windows NT 3.51.

-Windows NT 4.0.

-Windows 2000.

-Windows Server 2003.

-Windows XP P1.

-Windows XP P2.

-Windows XP P3.

III.5- Tests et résultats:

Nous allons montrer maintenant l'interface de l'application USDBPROLOG.

Figure III.17 : interface principale de l'application.

L'application contient :

Une barre des tâches qui contient trois sous menus (fermer, fichier, aides).

Un bouton " effacer "pour effacer le code source.

Un champ de texte pour saisir le code source de l'utilisateur.

Quatre boutons:

-" Exécuter" : pour l'exécution de code.

-"plusieurs" : pour afficher d'outre réponse.

-"stop" : pour stopper l'exécution de code.

-" effacer ": pour effacer les réponses dans le champ de réponse.

Un champ de réponses pour voir les réponses.

Un bouton "fermer" pour fermer l'application.

Le menu fichier contient trois sous menus :

Nouveau: pour faire un nouveau projet.

Ouvrir: pour ouvrir un projet

Enregistrer: pour enregistrer un projet.

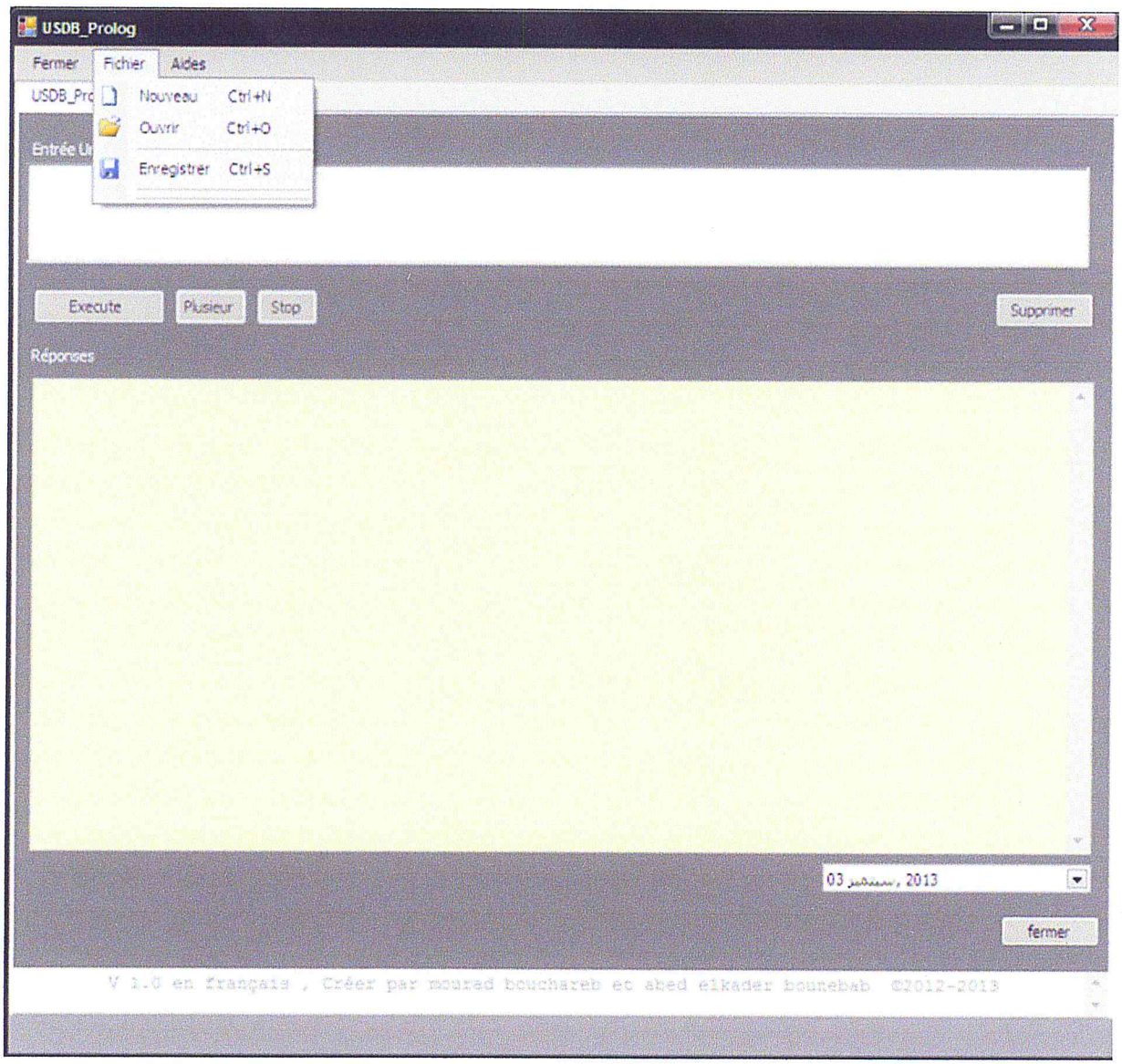

Figure III.18 : interface principale de l'application1.

Exemples:

Apprêt la explication de l'interface de l'application USDBPROLOG on prend quelques exemples:

Exemple1 (famille):

Voici le code dans le fichier "famille.usdb" :

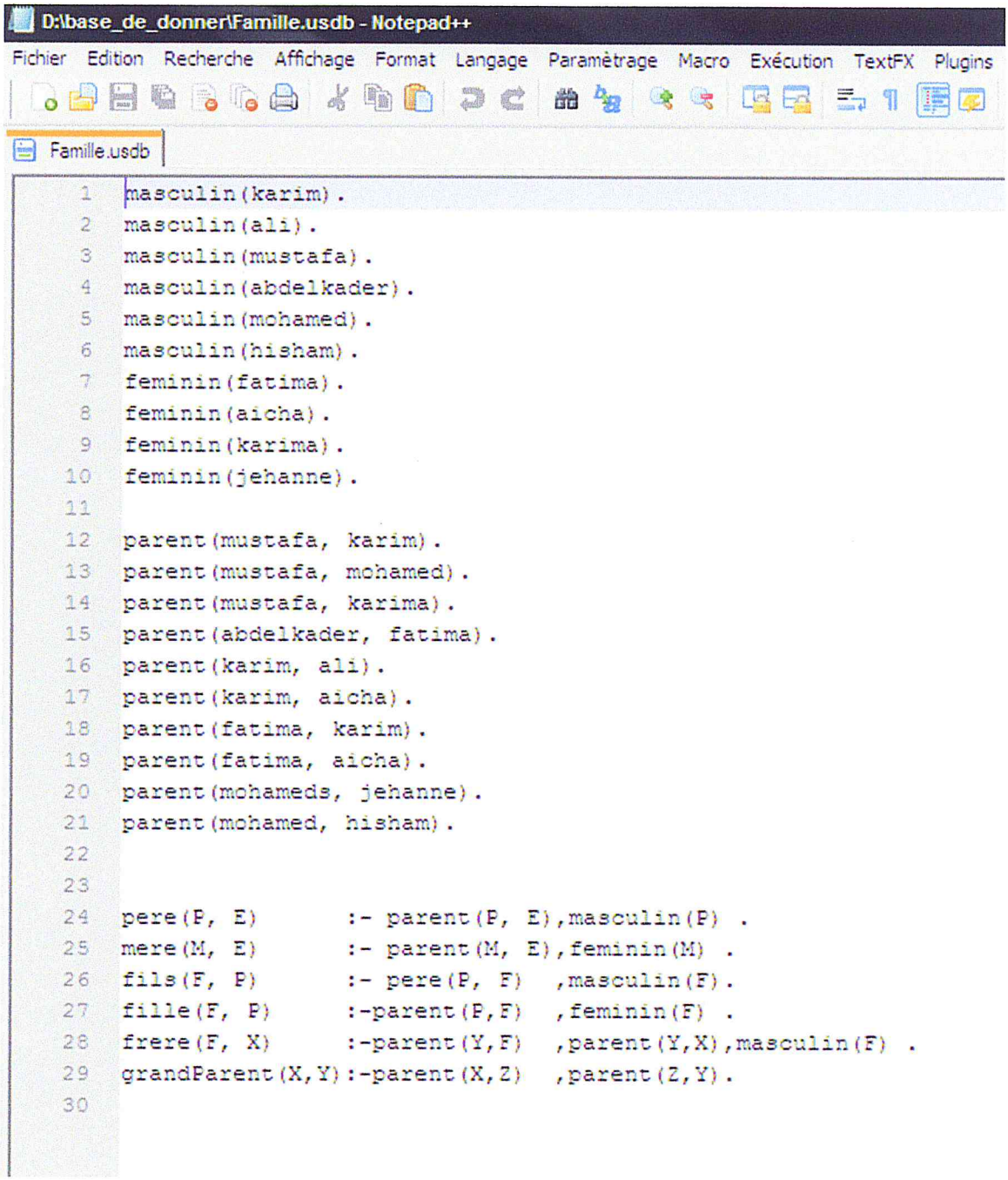

Figure III.19 : le fichier "famille.usdb".

Pour l'exécution de fichier "famille.usdb" on saisit la commande

"Consulter ("d:\\base_de_donner\\famille.usdb")"

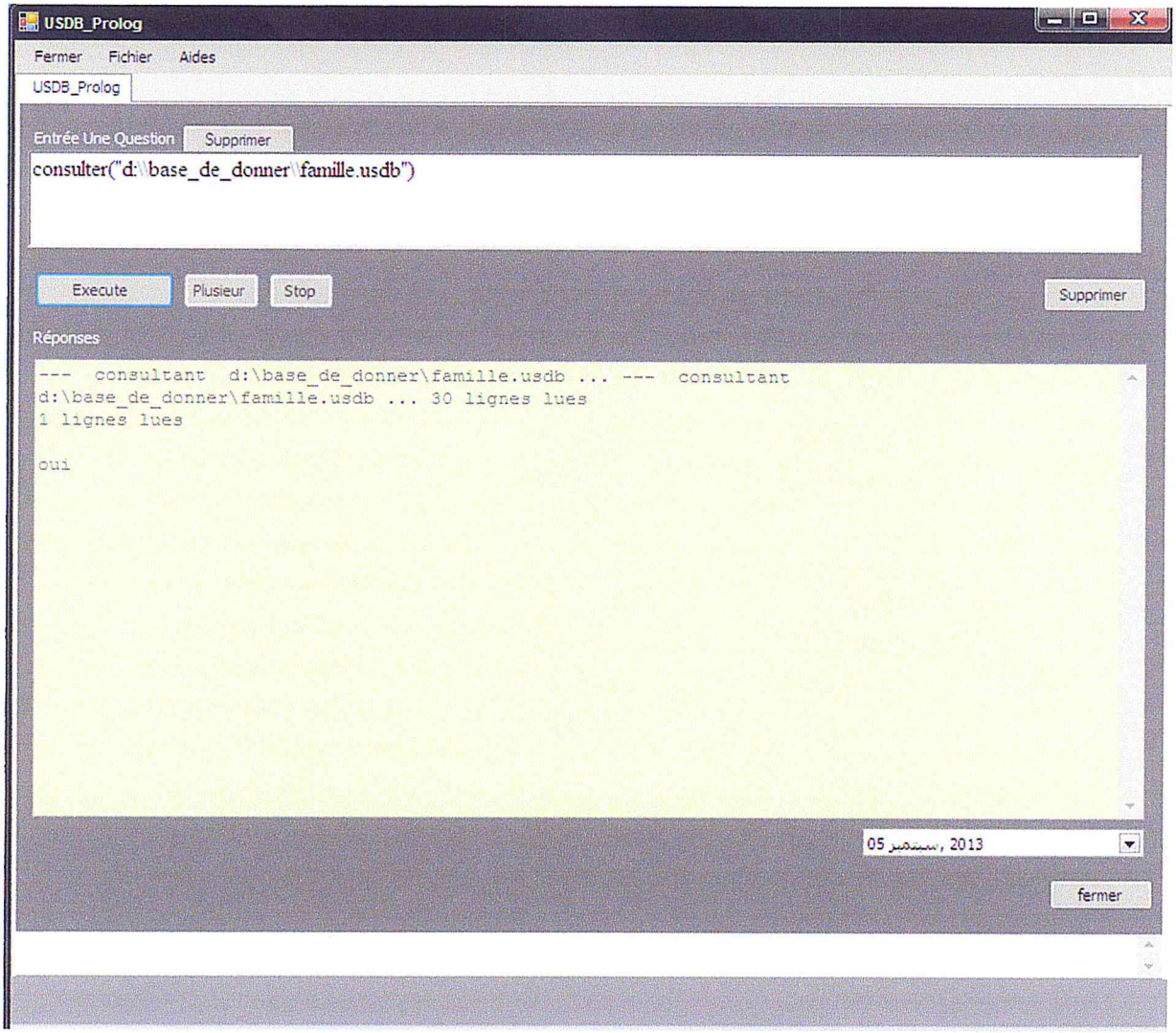

Figure III.20 : l'exécution de programme "famille.usdb".

Si nous appuyions sur le bouton "Execute" un message va afficher.

"--- consultant d:\base_de_donner\famille.usdb ... ": signifie que le fichier famille.usdb est en cours de consultation.

"30 lignes lues": le compilateur donne le nombre de lignes dans le programme.

"oui" signifie que le fichier famille.usdb est consulté correctement.

Nous essayons la question suivante: parent(X,Y) .

L'interpreteur va afficher "X=mustafa Y =karim": signifie que mustafa est le parent de karim.

Si on clique sur le bouton "plusieurs" l'interpréteur de l'usdbprolog va afficher

" X=mustafa Y =mohamed " veut dire que mustafa est aussi le parent de karim et ainsi de suite.

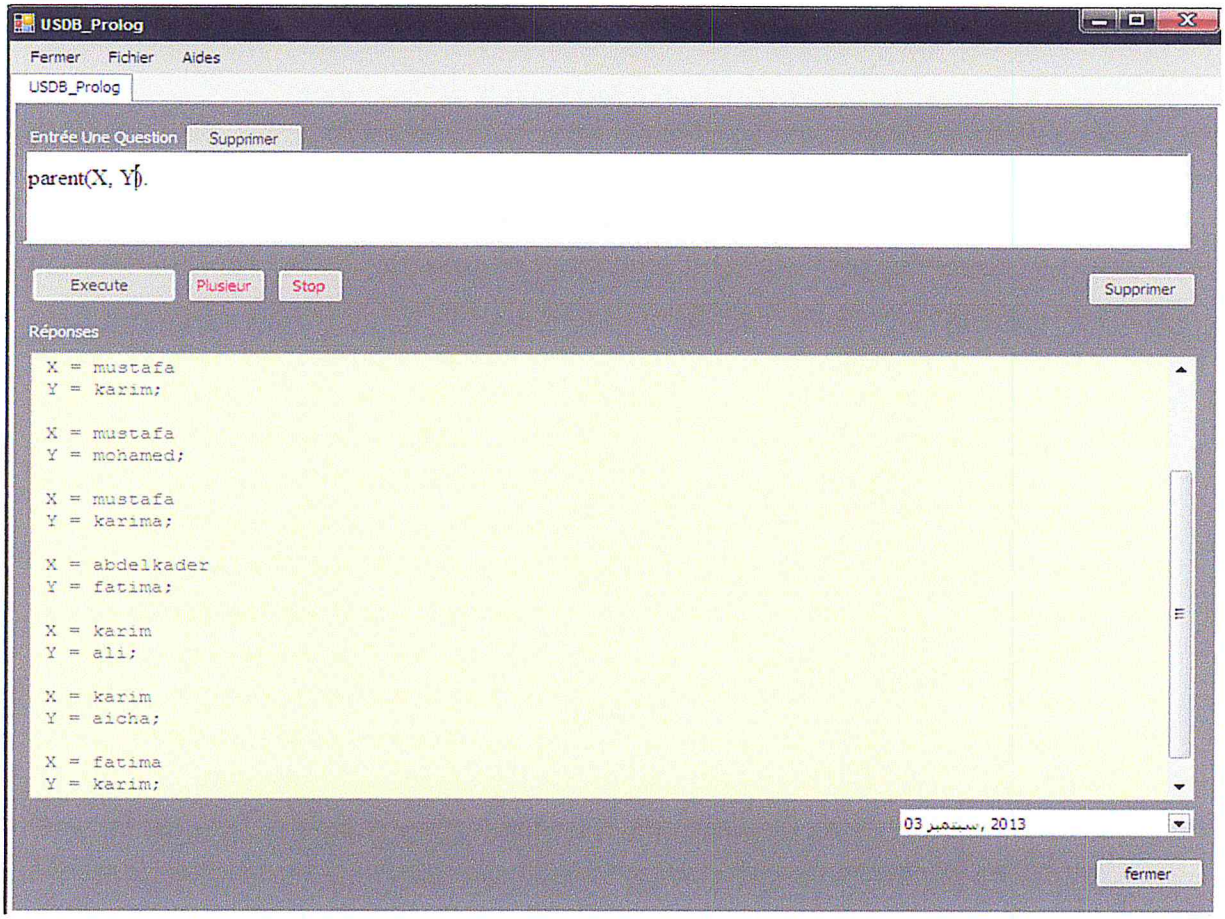

Figure III.21 : l'exécution de prédicat parent(X,Y).

Exemple2 (factoriel):

Voila le code du programme fact.usdb:

fact(0,1).

fact(X,Y):-X>0 , X1 est X-1 , fact(X1,Z) , Y est Z*X.

On calcule le factorielle de 12

On écrire la question fact(12,X): c.a.d factorielle de 12 est X et X=?.

L'exécution de la question fact(12,X) donne X=479001600: c.a.d factorielle de 12 est 479001600.

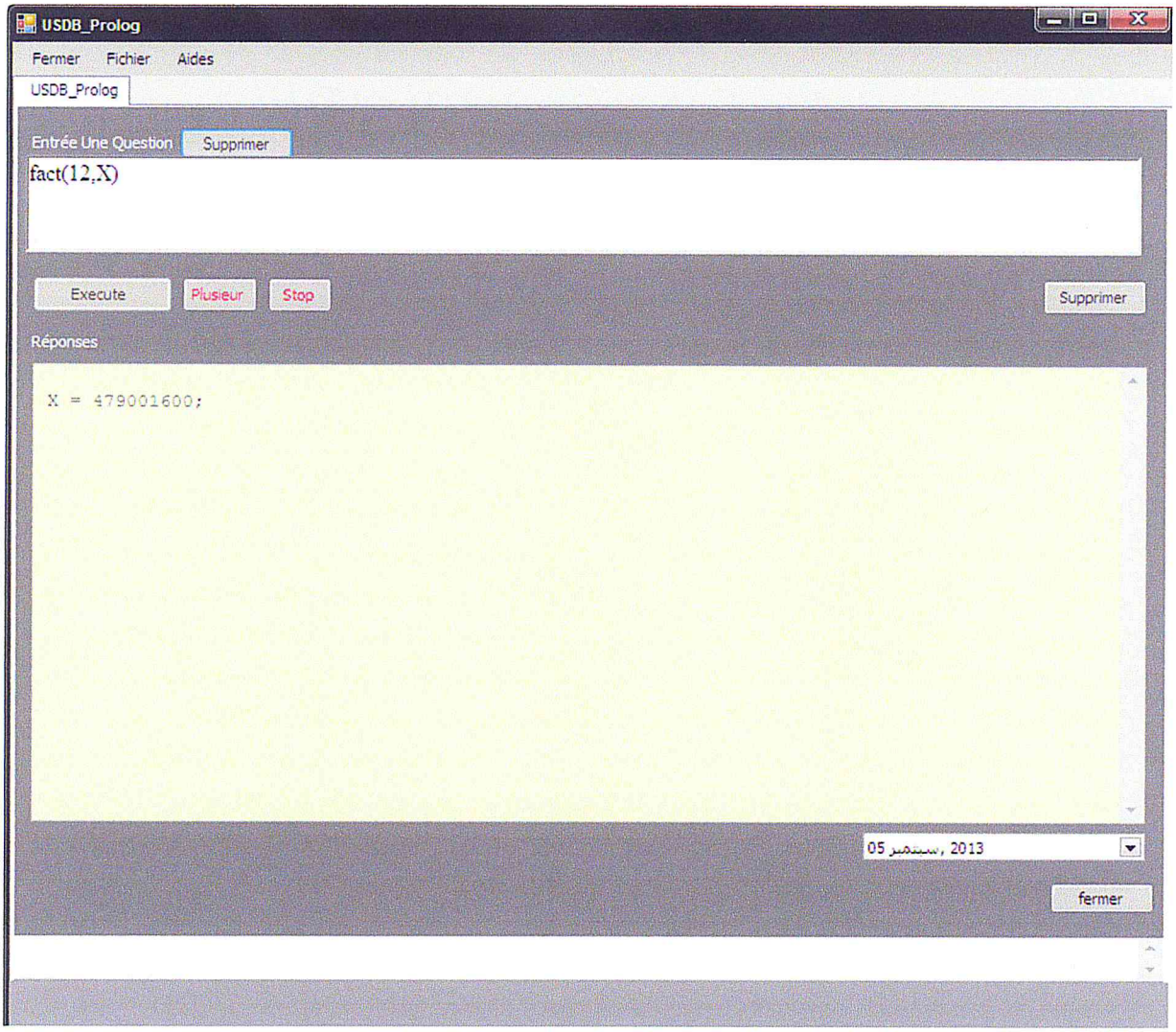

Figure III.22 : l'exécution de prédicat fact(12,X).

Si nous mettons un prédicat n'est pas défini dans le code de "fact.usdb".le message d'erreur va afficher voir la figure suivante:

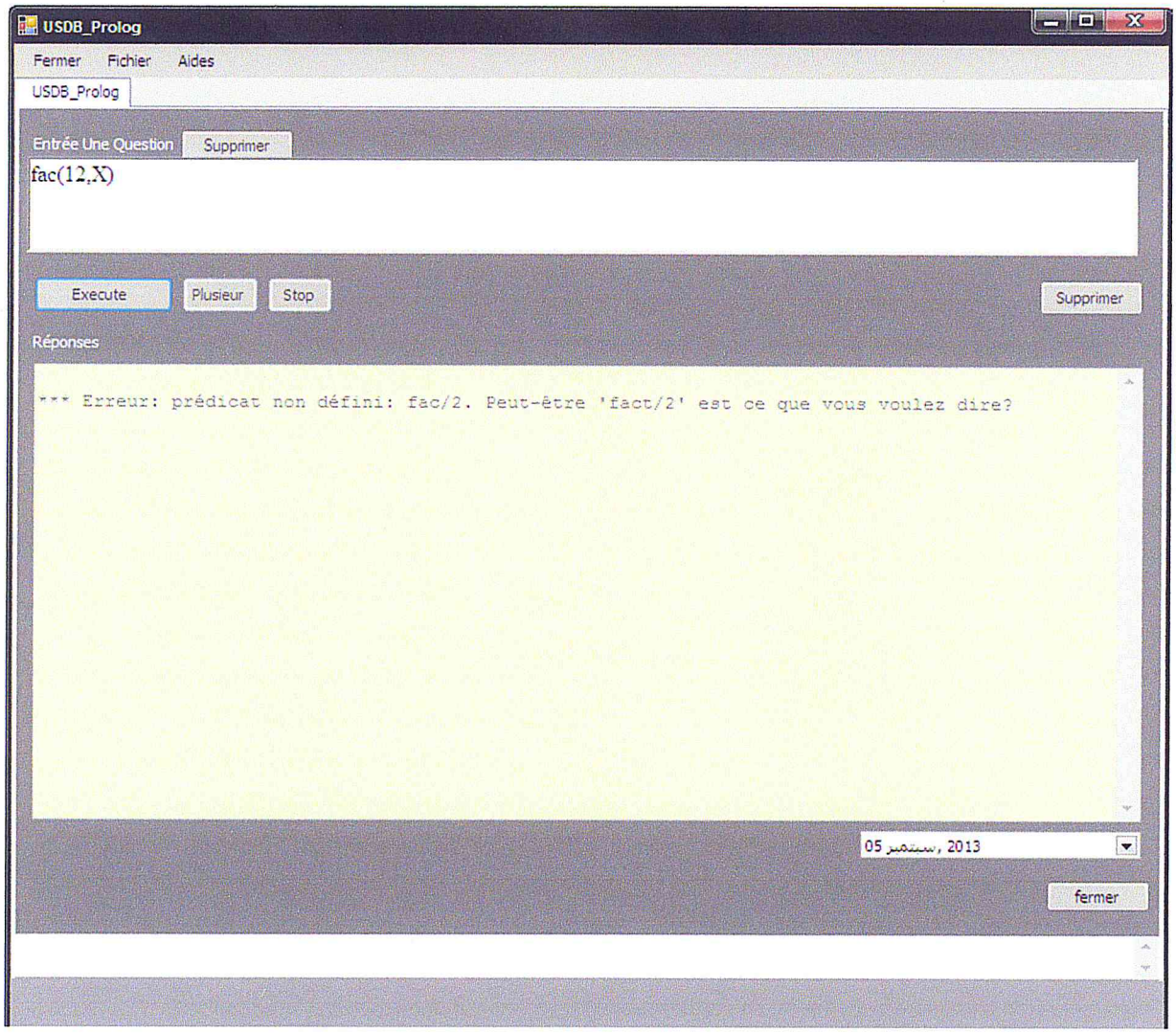

Figure III.23 : test sur les messages d'erreur.

Exemple3 (pgcd):

Avec un peu d'arithmétique, on peut donner les règles pour trouver le plus grand diviseur commun (soit D) de deux nombres (soit X et Y), on peut utiliser l'algorithme d'Euclide qui correspond aux règles suivantes :

1. si X = Y, alors D = X ;
2. si X < Y, alors D est le pgcd de X et de la différence Y – X ;
3. si Y < X, alors utiliser la règle 2 en changeant X et Y.

Il s'écrit de la manière suivante (voir la figure suivante):

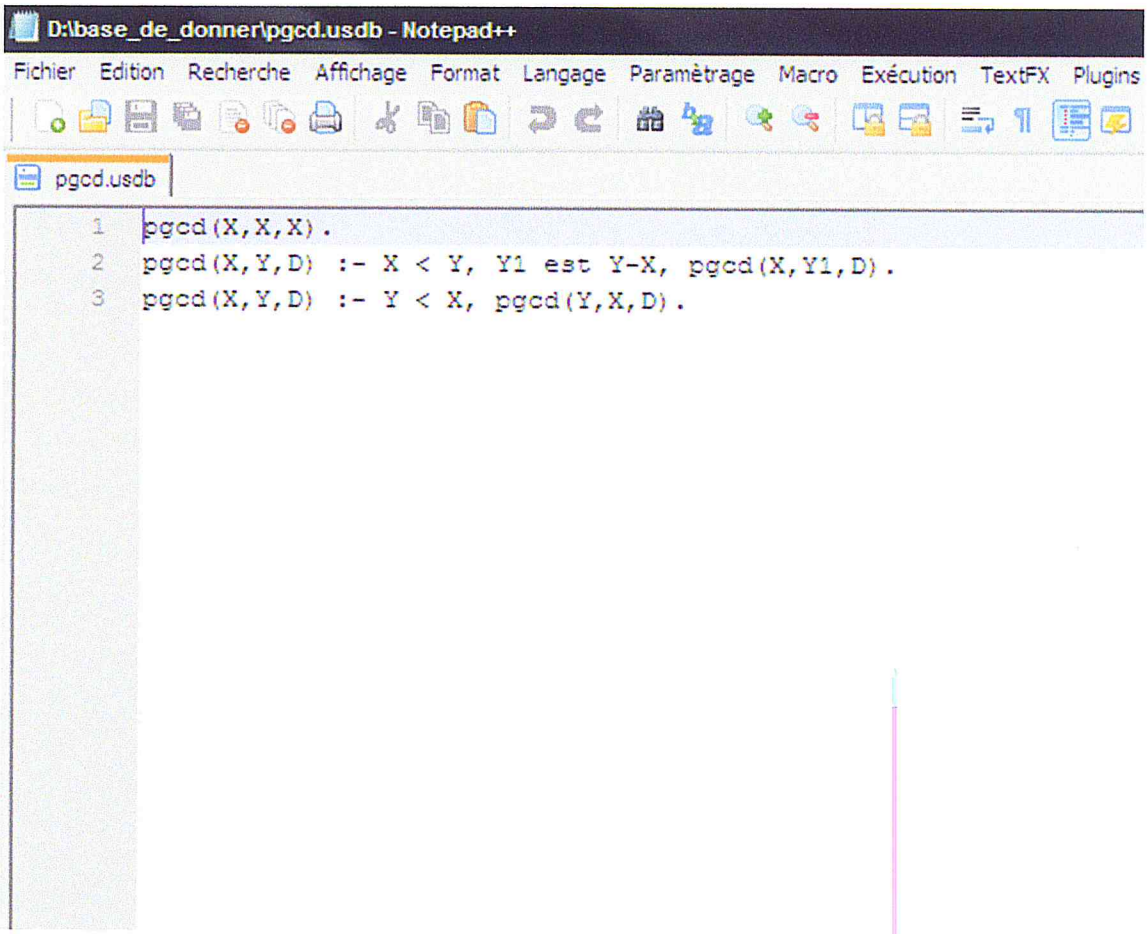

Figure III.24 : fichier pgcd.usdb.

Nous testons maintenant sur USDBPROLOG .

Pgcd(2,4,D): veut dire que le plu grand commun diviseur de 2 et 4 est D .l'interpréteur va répondre D=2.

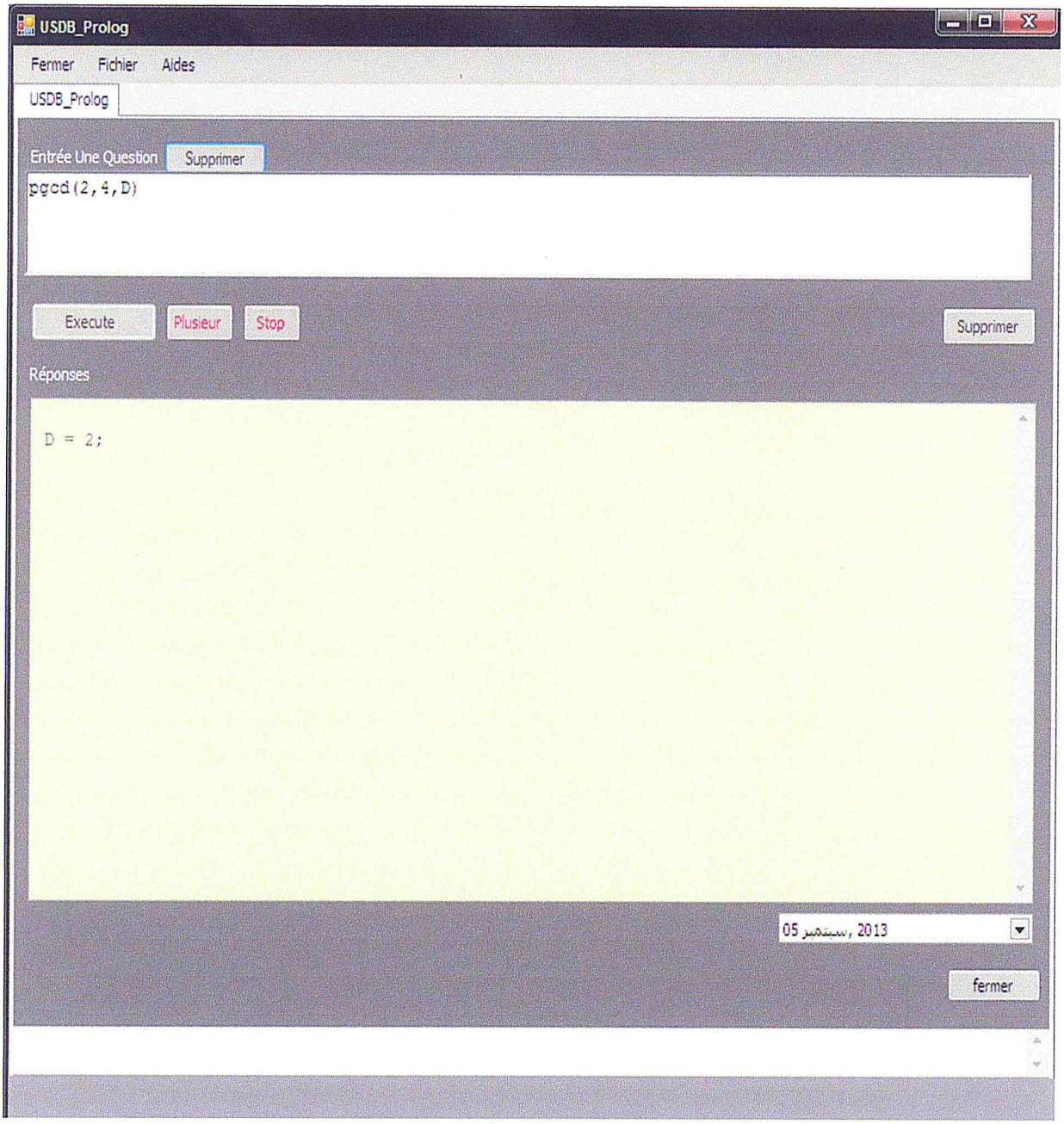

Figure III.25: calcul pgcd.

Exemple4 (le nombre de Fibonacci):

Autre exemple, le nombre de Fibonacci :

Le code écrit de la manière suivante (voir la figure suivante):

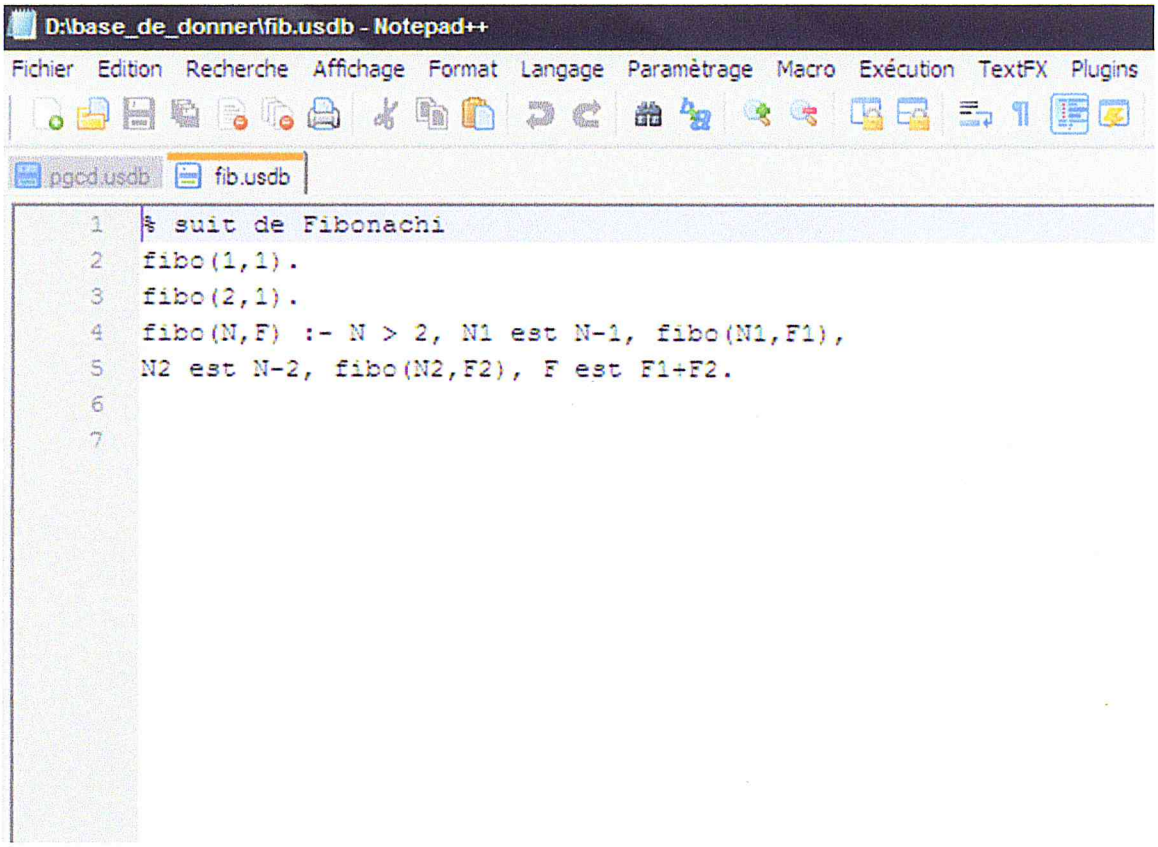

Figure III.26 : fichier fib.usdb.

Nous testons maintenant sur USDBPROLOG .

Fibo(2,X) : veut dire que le fibonacci de 2 est X. l'interpréteur va répondre X=1.

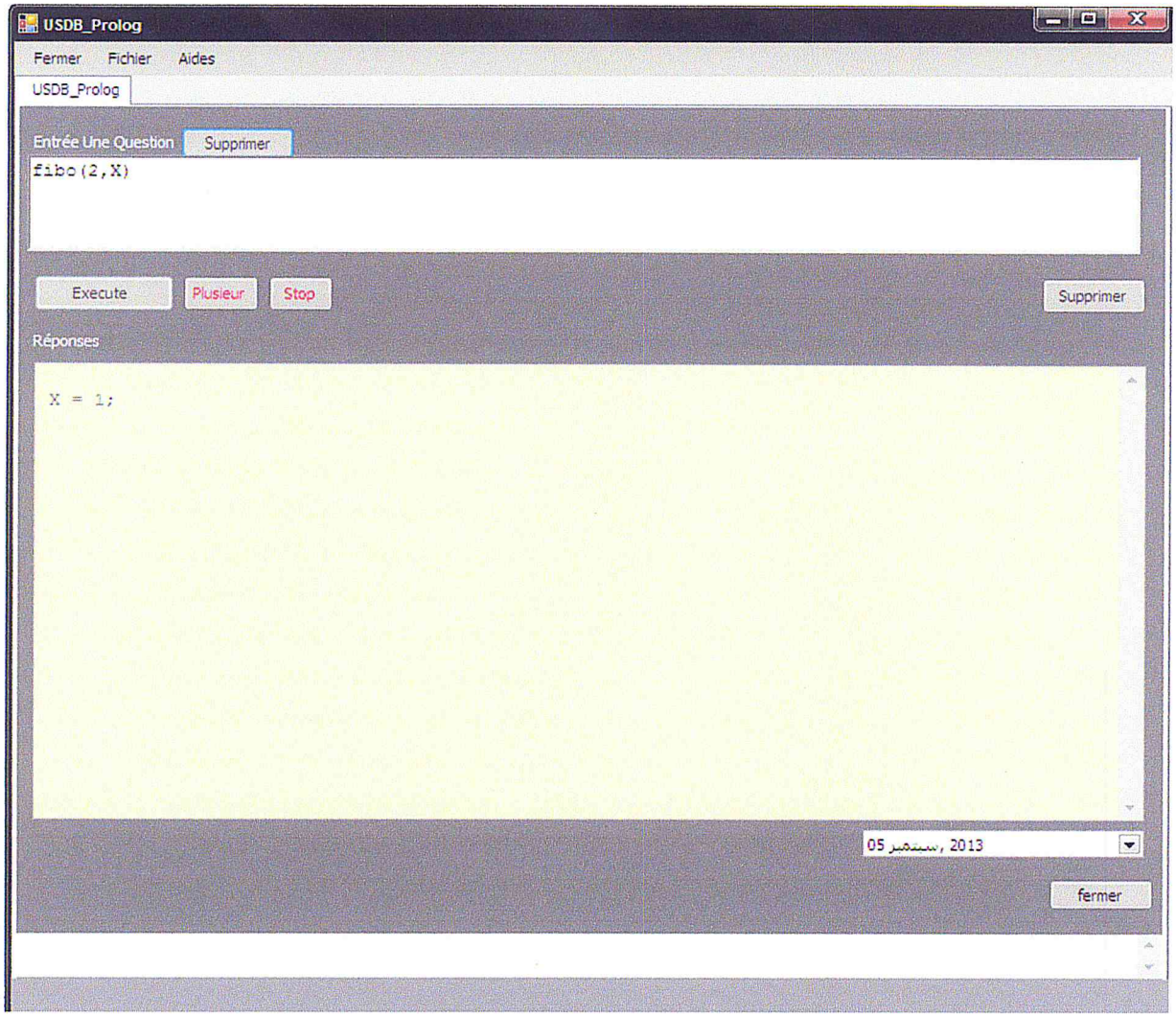

Figure III.27: le nombre de Fibonacci.

III.6- Conclusion:

Nous avons réalisé un interpréteur de programmation logique en français en utilisant le langage c#. Nous avons réalisé également une interface graphique pour cet interpréteur.

CONCLUSION GENERALE

Conclusion générale

L'objectif de notre travail du projet de fin d'études est la réalisation d'un interpréteur d'un langage logique .Pour cela, nous avons réalisé un analyseur lexical, syntaxique, sémantique basé sur la plateforme C#. Nous avons aussi réalisé une interface de cet l'interpréteur.

Au cours de notre travail, nous nous sommes familiariser avec le domaine de la création des compilateurs, des interpréteurs, nous avons appris à manipuler C#.

Ce travail constitue notre premier contact avec le monde du travail, il nous a offert la possibilité d'approfondir nos connaissances et les concrétiser à l'aide d'un travail pratique.

GLOSSAIRE

FP: un langage fonctionnel.

Lisp: un langage fonctionnel typé dynamique.

Scheme: un langage fonctionnel de typé dynamique.

ML: une famille pour les langages fonctionnels.

POO: programmation par objet.

USDP: Unified Software Development Process.

AGL : ateliers de génie logiciel.

CLOS: Common Lisp Object System.

WAM: la machine abstraite de Warren.

USDBPROLOG: universiter saad dahleb blida programmation logique.

prolog: programmation logique.

C#: langauge de developpemment.

CLR: Common Language Runtime.

HTML: HyperText Markup Language.

XML: sigle d'Extensible Markup Language.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[1]. http://www.enib.fr

[2]. www.lsv.ens-cachan.fr/~goubault/lambdaidx.html

[3].A Colmerauer An introduction de Prolog III Communication of the ACM, July 1990.

[4]. M. Condillac.Prolog, fondements et application, Edition Dunod, Paris 1986.

[5]. dictionnaire.phpmyvisites.net/definition-Langage-imperatif--10934.htm.

[6]. J. Jaffar, J. L. Lassez Constraint logic programming. Research report, University of Melbourne1986, Alsoin the proceeding of POPL'87, 1987.

[7]. D. W. Lioyd,Fondation of logic programming ,Pring verlof, Edition Eyrolles, 1984.

[8]. François Fages, Programmation logique par contraintes. Ecole poly technique,Garnes, Janvier 1996.

[9]. Lassaigne R, De Rougemont M, Logique et fondement de l'informatique. Edition Hermes, 1993.

[10]. Philipe Codognet, Programmation logique avec contraintes, 1995

[11]. Jaques Cohen. Constraint logic programming language. Communication of the ACM, 1990.

[12].http://cui.unige.ch/DI/cours/CompInterpretes

[13].grammaire.reverso.net/1_2_04_la_proposition_relative.shtml

[14].www.definitions-webmarketing.com

[15].http://fr.wikipedia.org/wiki/Interpr%C3%A8te_(informatique)

[16]. www.dotnet-france.com/Cours/IntroCSharp3.html

[17]. www.mono-project.com/CSharp_Compiler

ANNEXE

Plate-forme d'exécution:

- La plate-forme Microsoft .NET (possibilité de mixage avec des modules d'autres langages)
- Des implémentations libres de ce langage et de sa plate-forme d'exécution sont en cours de finalisation, comme le projet Mono maintenu par Novell, ou dotGNU maintenu par la *Free Software Foundation*. L'idée fondatrice de ces projets est qu'une application en C# puisse s'exécuter sans modification sur une plate-forme propriétaire comme Windows ou libre comme Linux.

Capacités introduites avec C# 2.0:

Microsoft mit à disposition du public en octobre 2005, après une longue période de beta-tests, la version 2.0 de la bibliothèque .NET, accompagnée d'une nouvelle version de la quasi-totalité des outils associés. C# ne fait pas exception à la règle et sort donc en version 2.0, avec les ajouts suivants :

- Les classes partielles, permettant de répartir l'implémentation d'une classe sur plusieurs fichiers.
- Les types génériques, qui ne sont pas une simple copie des *templates* C++. Par exemple, on trouvera dans les génériques C# la restriction de types (pour spécifier les types utilisables dans une généralisation). Par contre, il est impossible d'utiliser des expressions comme paramètres pour la généralisation.
- Un nouvel itérateur qui permet l'utilisation de coroutines via le mot-clé yield, équivalent du yield que l'on trouve en Python.
- Les méthodes anonymes avec des règles de fermeture configurables.
- Les types « nullables », c'est-à-dire la possibilité de spécifier qu'un type de valeur peut être nul. Ceux-ci sont déclarés avec le caractère point d'interrogation « ? » suivant le nom du type, comme ceci : int? i = null;.
- Le nouvel opérateur double point d'interrogation « ?? » utilise deux opérandes et retourne le premier non nul. Il a été introduit pour spécifier une valeur par défaut pour les types « nullables ».

À titre de référence, les spécifications complètes des nouveautés introduites dans la version 2.0 sont disponibles dans les liens externes.

Anders Hejlsberg, le père de Delphi, s'est exprimé sur l'implémentation des génériques dans C#, Java et C++ dans cette interview (en).

La fonctionnalité des types nullables fut fixée quelques semaines seulement avant la sortie publique de la version 2.0, car il a été mis en lumière que si la valeur de la variable était bien nulle, cette variable n'était pas nulle au sens traditionnel du terme, c'est-à-dire qu'il ne s'agit pas d'une référence vide. Ainsi, la conversion d'un type primitif de valeur nulle en objet donnait une référence non nulle vers une valeur nulle. Il fallut donc, pour corriger ce problème, corriger le noyau du CLR et effectuer de nombreuses vérifications et corrections sur tous les produits de la gamme .NET 2.0 (Visual Studio 2005, SQL Server 2005, C# et VB.NET).

Capacités introduites dans C# 3.0:

Le C# 3.0 fut présenté au salon PDC 2005. La version finale est disponible depuis le 19 novembre 2007 au téléchargement sur le site de Microsoft (en) . Les principales nouveautés sont les suivantes :

- L'ajout des mots-clefs select, from et where pour permettre la formation et l'exécution de requêtes SQL, XML, ou directement sur des collections. Cette fonctionnalité fait partie du programme Language Integrated Query (LINQ) (en).
- Nouvelle possibilité d'initialisation d'un objet : À la place de Client c = new Client(); c.Nom = "Dupont";, on peut utiliser Client c = new Client{ Nom = "Dupont" };
- Expressions lambda : ListeTrucs.Where(delegate(Truc x) { return x.Size > 10; }); devient ListeTrucs.Where(x => x.Size > 10);
- Inférence du type des variables locales : string s = "Dupont" peut être remplacé par var s = "Dupont"
- Introduction des types anonymes : var x = new { Nom = "Dupont" } peut être utilisé à la place de class __anonymous { private string _nom; public string

Nom { get { return _nom; } set { _nom = value; } } } __anonymous x = new __anonymous(); x.Nom = "Dupont";

- Les arbres d'expressions (expression trees) : permettent la compilation du code sous formes d'arbres d'objets facilement analysables et manipulables.
- Méthodes étendues : permet d'ajouter des méthodes à une classe en y ajoutant un premier paramètre this.

Une présentation du C# 3.0 et de LINQ peut être trouvée sur la page du centre de développement de .NET Framework.

Le code compilé en C# 3.0 est entièrement compatible avec celui du 2.0, étant donné que les améliorations apportées ne sont que purement syntaxiques ou ne consistent qu'en des raccourcis compensés au moment de la compilation. Les nouveautés introduites dans les bibliothèques de la version 3.5 (LINQ…) ne sont cependant pas utilisables avec les versions précédentes de C#.

Cette version exige Windows XP ou une version supérieure (Vista ou Windows 7). Elle n'est pas disponible pour Windows 2000.

C# 4.0:

La version 4 du langage apporte plusieurs nouveautés:

- le typage dynamique des variables à l'aide du mot clé dynamic;
- les arguments nommés et facultatifs;
- le support de la covariance et de la contravariance pour les interfaces et les délégués génériques ;

Le framework .NET 4.0 est sorti le 12 avril 2010, accompagné de Visual Studio 2010[3]. Il propose entre autres :

- la nouvelle bibliothèque parallèle : Task Parallel Library ;
- une version optimisée de la plateforme entité d'accès aux Bases de Données (Entity Framework) via l'utilisation de LINQ ;
- la version parallèle de LINQ appelée PLINQ.

C# 4.5:

La version 4.5 du langage permet de programmer plus simplement des programmes asynchrones grâce à l'ajout des mots clés async et await.

Le comportement des closures dans la boucle foreach a été modifié. Il n'est désormais plus nécessaire d'introduire une variable locale dans une boucle foreach pour éviter les problèmes de closure.

À noter également les informations relatives à l'appelant permettent de connaître le nom de la méthode qui a appelé une propriété.

Standardisation:

Le C# a été normalisé par l'ECMA (ECMA-334) en décembre 2001 et par l'ISO/CEI (ISO/CEI 23270) en 2003.

Les modifications survenues dans la Version 2.0 ont été normalisées par l'ECMA (ECMA-334) en juin 2006 et par l'ISO/CEI (ISO/IEC 23270:2006) en septembre 2006.

Microsoft a ouvert le code source de certaines bibliothèques utilisées par le C# en octobre 2007 sous la licence Microsoft Reference License (MS-RL).

Nom du langage:

C# est une note de musique dans le système de notation musical américain (cette note correspond à "Do dièse" dans la notation française). Le symbole # signifie en musique que la note doit être augmentée, l'idée de "C augmenté" rappelle la manière dont le langage C++ a été nommé (++ étant l'opérateur d'incrémentation).

Usuellement, un croisillon (#) est utilisé comme second caractère à la place du dièse (#) car ce dernier est moins accessible sur le clavier et non reconnu dans certaines polices de caractères.

Le langage:

Voici un exemple d'un programme *Hello world* typique, écrit en C# :

```
using System;

class HelloWorld
{
   public static void Main()
   {
      Console.WriteLine("Hello World!");
   }
}
```

Gestion des exceptions:

C# possède les instructions *try* et *catch* permettant de gérer les exceptions (comportement non attendu des instructions du programme), similaires dans la syntaxe à celles du C++.
Exemple de code tentant de créer un fichier "document.txt" sur le serveur "Toto":

```
try
{
   // Tentative de création du fichier 'document.txt' sur le serveur 'Toto'
   File.Create(@"\\toto\document.txt");

}
catch
{
   // Impossible de contacter le server Toto
   MessageBox.Show("L'application n'arrive pas à créer le fichier 'document.txt' sur le serveur 'Toto' !", "Alerte");
```

Notez l'utilisation d'une chaîne de caractères verbatim : le caractère arobase précède le guillemet donc l'anti-slash n'est pas doublé. Ce genre de chaîne de caractères est pratique pour les chemins sous Windows.

Dans cet exemple, la fonction File.Create retourne un flux (FileStream), ou elle peut lancer une exception si une erreur s'est produite (problème de connexion par exemple).

Dans cet exemple, aucune information sur l'exception n'est obtenue : on cherche juste à savoir si le programme ne s'est pas comporté normalement, auquel cas on arrive dans le bloc catch.

À l'instar de C++ qui a un type d'exception de base (class exception dans l'en-tête <exception>) et dont les autres exceptions héritent, toute exception C# est héritée (ou une instance) du type System.Exception. Ainsi, si on cherche à savoir ce qui s'est passé, une solution simple reste d'obtenir une référence vers l'exception de la manière suivante :

```
try
{
   // Tentative de création du fichier 'document.txt' sur le serveur 'Toto'
   File.Create(@"\\toto\document.txt");

}
catch(Exception err)
{
   // Impossible de contacter le server Toto
   MessageBox.Show("L'application n'arrive pas à créer le fichier 'document.txt' sur le serveur 'Toto' ! Erreur:" + err.Message, "Alerte");
}
```

Ainsi, une information complète sera retournée, décrivant la nature de l'exception qui s'est produite.

En fonction des fonctions appelées, le framework .NET fournit la liste des exceptions que l'appel est susceptible de retourner en cas d'erreur. Dans le cas de la fonction 'Create', voici la liste des exceptions possibles:

- System.UnauthorizedAccessException
- System.ArgumentException
- System.ArgumentNullException
- System.IO.PathTooLongException
- System.IO.DirectoryNotFoundException
- System.IO.IOException
- System.NotSupportedException

```
try
{
   // Tentative de création du fichier 'document.txt' sur le serveur 'Toto'
   File.Create(@"\\toto\document.txt");

}
catch(System.ArgumentException ArgumentErr)
{
   // L'argument n'est pas valable. Le nom de fichier 'document.txt' n'est pas valable
   MessageBox.Show("L'argument n'est pas valable. Le nom de fichier 'document.txt'
n'est pas valable ! Erreur:" + ArgumentErr.Message, "Alerte");
}
catch(Exception err)
{
   // Impossible de contacter le server Toto
   MessageBox.Show("L'application n'arrive pas à créer le fichier 'document.txt' sur le
serveur 'Toto' ! Erreur:" + err.Message, "Alerte");
}
```

De la même manière qu'en C++, l'envoi d'une exception se fait avec le mot-clef throw :

```
public uint Divide(uint num, uint div) //Fonction de division dans N
```

```
{
   if(div == 0)
      throw new Exception ("Division par 0 !");
   if(num < div)
      throw new Exception ("num est strictement inférieur à div : le résultat de la
division ne sera pas un entier naturel !");
   return num / div;
}
//Code ailleurs :
try
{
   Divide(12, 6); //Retourne 2, aucune exception
   Divide(10, 20); //Exception, cette division ne donne aucun résultat dans N
   Divide(10, 0); //Exception, division par zéro impossible
}
catch (Exception e)
{
   MessageBox.Show("L'erreur suivante a été retournée :\n" + e.Message, "Alerte");
}
```

L'envoi d'une exception (via throw) ou la levée d'une exception (dans un bloc try/catch) met immédiatement fin au bloc en cours. Ainsi, si div est nul, le code de la fonction Divide s'arrêtera à la ligne 2. De même, dans l'exemple précédent, la troisième fonction Divide ne sera jamais exécutée à cause de la levée d'une exception lors du deuxième appel à Divide. Une exception non attrapée (« *catchée* ») - c'est-à-dire que l'application n'a pas encadré le code lançant l'exception par un bloc catch approprié - met fin à l'application immédiatement sous la forme d'une exception de type UnhandledException (exception non gérée en français).

Comme en Java, il est possible d'ajouter un bloc finally pour exécuter une série d'instructions, quoi qu'il se passe (exception lancée ou non). Cela est utile pour libérer des ressources quel que soit ce qui peut se passer entre la prise de ressource et la libération. Exemple typique : lecture de fichiers

```
FileStream fs = new FileStream(@"C:\Fichier.txt", FileMode.OpenOrCreate,
FileAccess.ReadWrite);
// arrivé ici, le fichier est ouvert
try
{
   // ... opérations de lecture / écriture pouvant lancer des exceptions ...
}
finally
{
   // fermer le fichier quoi qu'il se passe :
   fs.Close();
}
```