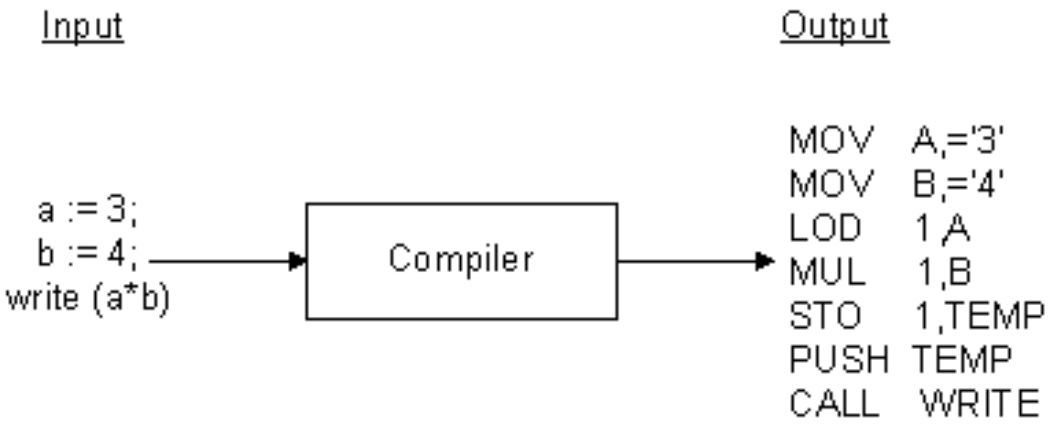
Développement d'un Interpréteur LISP

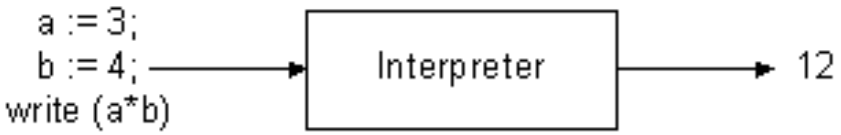
Nous allons tout au long du TP réaliser étape par étape notre **interpréteur du langage Lisp**.

Introduction

L'interpréteur d'un langage est un programme qui prend en entrée un texte écrit dans ce langage et réalise les actions décrites par ce texte et exprimées dans ce langage.
On définit habituellement la compilation comme le processus qui consiste à transformer un programme écrit dans un langage évolué pour le réécrire dans un langage exécutable par une machine.
D'une manière générale un compilateur traduit un code écrit dans un langage dans un autre langage sans en changer la signification.

Un interpréteur peut être vu comme un compilateur qui à la place de produire seulement un langage en sortie effectue les actions définies dans le langage fourni en entrée.



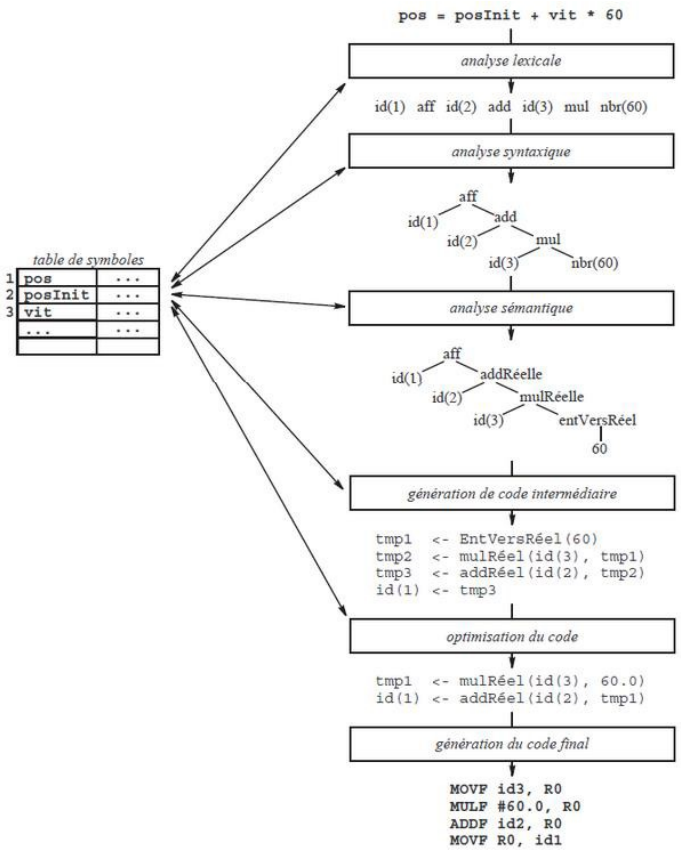


Les Phases d'une Compilation

En entrée d'un compilateur on trouve toujours une suite de caractères appelé **« texte source »**

En sortie d'un compilateur on peut obtenir :

- **Du code machine pour un processeur**
- **Du code pour une machine virtuelle**
- **Du code abstrait** destiné à être traité pour générer du code exécutable
- **Le** codage **d'un arbre représentatif de la structure logique** d'un programme.



Analyse Lexicale :

Cette phase consiste à consommer les caractères du texte source pour découvrir les « Unités Lexicales » ou « Lexèmes » qui sont les mots du langage.

L'analyse lexicale est réalisée par un analyseur lexical ou Lexeur qui découpe le texte source en mot du langage.

L'analyseur Lexical est initialisé avec un texte ou un lecteur de texte (la source). Il lit ce texte en le découpant en mots du langage.

L'analyseur lexical est exploité par l'analyseur syntaxique à qui il fournit les mots un après l'autre.

Analyse syntaxique :

Cette phase consiste à consommer les unités lexicales (mots) fournies par l'analyseur syntaxique pour constituer les phrases du langage.

Cette phase vérifie que les mots fournis constituent bien des phrases correctes du langage.

Analyse sémantique :

L'analyse syntaxique a garanti que le texte est correct syntaxiquement. Les phrases sont correctes mais sont-elles cohérentes ? C'est ce que doit contrôler cette phase

- Les identificateurs ont-ils été déclaré ?

- Les type des opérandes sont-ils du type requis par les opérateurs ?

- Les type des arguments des fonctions ont-ils le nombre et le type requis ?

Cette phase vérifie que les mots fournis constituent bien des phrases correctes du langage.

(Utilisation de l'Inférence de types et d'une table de symboles)

Génération de code intermédiaire :

Production d'un pseudocode abstrait destiné à servir de base à la génération de codes exécutable pour différentes cibles.

Notion de « Séparation » en Faces avant (dépendantes du langage source) et Faces arrière (dépendantes uniquement du langage cible)

Optimisation structurelle du code intermédiaire :

- Suppression des identificateurs inutilisés

- Détection et sortie des boucles des invariants

Génération du code final :

- Transformation du code intermédiaire en un code spécifique à une cible particulière (machine réelle, virtuelle ou abstraite)

- Dépendant des possibilités de la machine : Registres, Pile, etc.

Pour une interprétation quelles adaptations sont à prévoir ?

Phases inutiles ?

Phases supplémentaires ?

TP - Conception de notre interpréteur Lisp :

Les étapes de l'analyse de notre interpréteur :

- Spécification fonctionnelle de l'interpréteur à concevoir

- Conception Orienté objet (1 ère itération) :
  - Diagramme de classes (sommaire)
  - Diagrammes de séquences pour une interprétation

Spécification Fonctionnelle:

Nous devons concevoir et développer un interpréteur basé sur le langage Lisp. Il devra être en mesure de fournir le résultat d'évaluation à effectuer à partir de code source écrit dans le langage LISP fourni sous forme de texte.

L'interprétation s'effectuera en plusieurs phases (cf. compilation) :

- Analyse Lexicale
- Analyse Syntaxique
- Analyse Sémantique
- Interprétation du code analysé
- Restitution du résultat

Analyseur Lexical:

Analyseur Syntaxique

Analyseur Sémantique:

Evaluateur:

Analyse UML (base)

Diagramme de classes :

Diagramme de séquence d'une simple évaluation :

L'analyse Lexicale

Dans le texte source, qui se présente comme un flot de caractères, l'analyseur lexical reconnaît des unités lexicales, qui sont les mots (lexèmes) avec lesquels les phrases du langage sont formées, et les fournit à la phase suivante : l'analyse syntaxique.

Exemple d'unités lexicales rencontrées dans les langages courants :
Les **caractères spéciau**x qui sont les opérateurs et séparateurs : **+ - * / = ; « » ++ -- <=** etc.
Les **mots-clés** du langage: if , for, while, do etc.
Les **constantes littérales** comme les nombres et les chaînes : **12 -5 «Bonjour»**
Les **identificateurs** (nom de variables) : **a monNombre age_du_capitaine**

A propos des unités lexicales :

L'analyseur lexical associe un nom ou code conventionnel à chaque unité lexicale :
+ → PLUS, = → EQUAL, nnn -> NOMBRE etc.

Les mots correspondants à une unité lexicale sont appelés les « Lexèmes ».
« + » → PLUS, « = » → EQUAL, « 12 » → NOMBRE (constantes littérales nombre)

Certaines unités lexicales nécessitent des **attributs pour ajouter des informations complémentaires**. Par exemple, les unités lexicales « constantes littérales » nécessitent un attribut pour stocker leur valeur.

La reconnaissance des unités lexicales par l'analyseur se base sur des **modèles descriptifs** : Pour les caractères spéciaux et les mots-clés le lexème et le modèle de l'unité lexicale sont identiques, pour les nombres et les littéraux ils divergent.

Exemples de modèles :

Unité lexicale	Modèle
NOMBRE	Éventuellement un signe suivi de chiffres
IDENTIFICATEUR	Une lettre suivie de lettres, chiffres ou de caractères « _ »
PLUS	Le caractère '+'

La spécification des modèles d'unités lexicales

La spécification des modèles doit se faire de manière formelle, les **expressions rationnelles** fournissent une notation permettant de décrire un modèle. Vocabulaire des Expression rationnelle :

Alphabet : ensemble de symboles.
L'ensemble des chiffres est un alphabet.
L'ensemble de toutes les lettres est un alphabet.
{A, C, G, T} est un alphabet avec 4 symboles (alphabet de l'ADN)

Chaîne ou mot sur un alphabet : séquence finie de symboles d'un alphabet.
ACCAGTTGAAGTG est un mot basé sur l'alphabet {A, C, G, T}

Langage sur un alphabet : ensemble de tous les mots créés à partir d'un alphabet.

Opération sur les langages : application d'un opérateur sur un ou plusieurs langages.
Soit les alphabets :

- L = {A, B… Z, a, b… z} (alphabet de toutes les lettres)

- C = {0, 1,2…9} (alphabet de tous les chiffres) Opérateurs :

L'**union** L **U** C : ensemble de tous les chiffres et toutes les lettres *(L | C)*
La **concaténation** LC : ensemble des mots composés d'une lettre et d'un chiffre *LC*
L^4 **:** ensemble des mots de 4 lettres *LLLL*
La **fermeture de Kleene** L^* **:** ensemble des mots d'un nombre quelconque de lettres (de 0 à une infinité) *L**
La **fermeture positive** C^+ **:** ensemble des mots comportant un chiffre ou plus (de 1 à *) *C+*

Une **expression rationnelle** est une formule qui définit un langage sur un ou plusieurs alphabets. Les expressions rationelles se construisent à partir d'autres expressions rationnelles

Exemples de définition de l'unité lexicale « identificateur » :

lettre → A | B | … | a | b | … | z
chiffre → 0 | 1 | … | 9
identificateur → lettre (lettre | chiffre)*

Les Expressions régulières en java :

Le moteur d'expressions régulières en java est dans le package : `java.util.regex.*`

Les classes :

- `Pattern` : représentation compilée d'un motif
- `Matcher` : moteur de recherche d'un motif dans une chaîne

Exemple :

```java
/**
 * Recherche de 'toto'
 * @param txt
 */
public static void demo1(String txt) {
    System.out.println("demo1 --> " + txt);
    // création du motif
    Pattern pattern = Pattern.compile("toto");
    // recherche des correspondances du motif pour la chaine
    Matcher matcher = pattern.matcher(txt);
    // tant qu'il existe des correspondances
    while(matcher.find()) {
        System.out.println("Trouvé:" + matcher.group() + " ==> pos:" + matcher.start() + " longueur:" + matcher.group().length());
    }
}
```

```
demo1 --> le chien de toto est comme toto
Trouvé:toto ==> pos:12 longueur:4
Trouvé:toto ==> pos:27 longueur:4
```

Syntaxe des motifs :

Les chaînes littérales :

Le motif comporte simplement la chaîne à rechercher. *Exemple : « toto »*

Les métacaractères et les classes de caractères :

.	Remplace n'importe quel caractère
*	Remplace une chaîne de 0, 1 ou plusieurs caractères
?	Remplace exactement un caractère

()	Groupe capturant pour accéder à des sous-parties extraites par le motif
[ ]	Intervalle de caractères (*Exemple : [a-z])*
{}	Quantificateur (*Exemple : [a-z]{3} 3 lettres)*
\	Déspécialise le caractère spécial qu'il précède (échappement)
^	Négation ou début de ligne
$	Fin de ligne
|	Ou logique entre deux sous-motifs (*Exemple : toto | tutu)*
+	Numérateur (au moins 1)
\d	Un chiffre (comme [0-9])
\D	Pas un chiffre (comme ^[0-9])
\s	Un caractère blanc ou invisible : [\t\n\x0B\f\r]
\S	[^\s]
\w	Le caractère d'un mot [a-zA-Z_0-9]
\W	[^w]

Construction de classes de caractères personnalisées :

[abc]	**Ensemble simple**, tout caractère parmi l'un des caractères suivants : a, b et c
[^abc]	**Négation** de l'ensemble précédent
[a-z]	**Ensemble complexe :** tout caractère parmi ceux de l'alphabet naturel compris entre a et z
[a-zA-Z] [a-z[A-Z]]	**Union** d'ensembles, tout caractère de l'alphabet minuscule ou majuscule
[abc&&[a-z]]	**Intersection** d'ensembles, tout caractère faisant partie de l'ensemble : a, b, c et aussi de l'ensemble de a jusqu'à z (c'est-à-dire uniquement a, b et c)
[a-z&&[^abc]]	**Soustraction** d'ensembles, tout caractère de l'alphabet compris entre a et z, excepté ceux de l'intervalle suivant : a, b et c

Les groupes de captures :

En utilisant les métacaractères parenthèses, on va pouvoir définir des *groupes capturant*. Les groupes représente un sous-motif dont les occurrences trouvées nous intéressent. On va ensuite pouvoir récupérer ces occurrences grâce à la méthode `group`, de la classe `Matcher`

```java
public static void demoGroupe(String txt) {
    System.out.println("demoGroupe --> " + txt);
    // Motif date jj/mm/aaaa
    Pattern pattern = Pattern.compile ("(\\d{2})/(\\d{2})/(\\d{4})");
    Matcher matcher = pattern.matcher(txt);
    while(matcher.find()) {
        System.out.println (matcher.groupCount() + " groupes découverts");
        System.out.println ("Jour : " + matcher.group (1));
        System.out.println ("Mois : " + matcher.group (2));
        System.out.println ("Année : " + matcher.group (3));
    }
}
```

```
demoGroupe --> Il est né le 20/12/2002 et a été trouvé vers le 15/01/2003
3 groupes découverts
Jour : 20
Mois : 12
Année : 2002
3 groupes découverts
Jour : 15
Mois : 01
Année : 2003
```

Groupes capturants imbriqués :

Le motif (a((b)(c))) comporte 4 groupes

Groupe 0 : (a((b)(c))) (le groupe 0 correspond toujours à la totalité du motif)

Groupe 1 : **(**a((b)(c))**)**

Groupe 2 : **(**(b)(c)**)**

Groupe 3 : **(**b**)**

Groupe 4 : **(**c**)**

Les méthodes du `Matcher` pour traiter les chaînes et sous-chaînes capturées :

`int groupCount()`	Nombre de sous-chaînes capturées
`String group()`	Sous-chaîne capturée par la dernière recherche
`String group(int group)`	Sous-chaîne capturée par le groupe group
`boolean find()`	Recherche de la prochaine sous-chaîne satisfaisant la regex
`boolean find(int start)`	Recherche de la prochaine sous-chaîne satisfaisant la regex, en commençant la recherche à l'index start
`int start()`	Index de début de la sous-chaîne capturée
`int start(int group)`	Index de début de la sous-chaîne capturée par le groupe group
`int end()`	Index de fin de la sous-chaîne capturée
`int end(int group)`	Index de fin de la sous-chaîne capturée par le groupe group