LISP

Introduction

- Lisp est le deuxième langage de programmation de haut niveau le plus ancien après Fortran
- Ila beaucoup changé depuis ses débuts, et un certain nombre de dialectes ont existé au cours de son histoire.
- Aujourd'hui, les dialectes Lisp à usage général les plus connus sont Common Lisp et Scheme.
- Lisp a été inventé par John McCarthy en 1958 alors qu'il était au (MIT).

Introduction

Objectif:

- Préparer les étudiants afin de les aider à comprendre les concepts de base liés au langage de programmation LISP.

Conditions préalables

- Notions de logique
- Programme informatique

Aperçu

- John McCarthy a inventé LISP en 1958, peu de temps après le développement du FORTRAN.
- Il a d'abord été implémenté par Steve Russell sur un ordinateur IBM 704.
- Il est particulièrement adapté aux programmes d'intelligence artificielle, car il traite efficacement les informations symboliques.
- Common Lisp est né, au cours des années 1980 et 1990, dans une tentative d'unifier le travail de plusieurs groupes d'implémentation qui ont succédé à Maclisp, comme ZetaLisp et NIL (New Implémentation de Lisp)
- Il sert de langage commun, qui peut être facilement étendu pour une implémentation spécifique.
- Les programmes écrits en Common LISP ne dépendent pas des caractéristiques spécifiques à la machine, telles que la longueur des mots, etc.

Caractéristiques du Common LISP

- Il est indépendant de la machine
- Il utilise une méthodologie de conception itérative et une extensibilité facile.
- Il permet de mettre à jour les programmes de manière dynamique.
- Il fournit un débogage de haut niveau.
- Il fournit une programmation orientée objet avancée.
- Il fournit un système de macros pratique.
- Il fournit de nombreux types de données tels que des objets, des structures, des listes, des vecteurs, des tableaux ajustables, des tables de hachage et des symboles.
- Il est basé sur l'expression.
- Il fournit un système de conditions orienté objet.
- Il fournit une bibliothèque d'E/S complète.
- Il fournit des structures de contrôle étendues.

Quelques applications réussies en Lisp.

- Emacs
- G2
- AutoCad
- Igor Engraver
- Yahoo Store

Structure du Programme LISP

- Les expressions LISP sont appelées expressions symboliques ou s-expressions.
- Les s-expressions sont composées de trois objets valides, des atomes, des listes et des chaînes.
- Toute s-expression est un programme valide.
- Les programmes LISP s'exécutent soit sur un interpréteur, soit sous forme de code compilé.
- L'interpréteur vérifie le code source dans une boucle répétée, également appelée boucle lecture-évaluation-impression (REPL).
- Il lit le code du programme, l'évalue et imprime les valeurs renvoyées par le programme.
- Interpréteur online https://onecompiler.com/commonlisp

Exemple d’un programme simple

- Écrivons une s-expression pour trouver la somme de trois nombres 6, 8 et 15.
- Pour ce faire, nous pouvons taper à l'invite de l'interpréteur

➔(+ 6 8 15)

- Si vous souhaitez exécuter le même programme qu'un code compilé, créez un fichier de code source LISP nommé test.lisp et tapez le code suivant dedans.

➔ (write (+ 7 9 11))

Évaluation des programmes LISP

- L'évaluation des programmes LISP comporte deux parties :
  - Traduction du texte du programme en objets Lisp par un programme de lecture
  - Implémentation de la sémantique du langage vis-à-vis de ces objets par un programme évaluateur
- Le processus d'évaluation suit les étapes suivantes :
  - lecteur traduit les chaînes de caractères en objets LISP ou en s-expressions.
  - L'évaluateur définit la syntaxe des formulaires Lisp qui sont construits à partir des s-expressions.
  - Ce deuxième niveau d'évaluation définit une syntaxe qui détermine quelles s-expressions sont des formes LISP.
  - L'évaluateur fonctionne comme une fonction qui prend une forme LISP valide comme argument et renvoie une valeur.
  - C'est la raison pour laquelle nous mettons l'expression LISP entre parenthèses, car nous envoyons l'intégralité de l'expression/du formulaire à l'évaluateur en tant qu'arguments.

Syntaxe

- Les programmes LISP sont constitués de trois blocs de construction de base
  - Atome
  - Liste
  - chaîne de caractères
- Un atome est un nombre ou une chaîne de caractères contigus.
- Il comprend des chiffres et des caractères spéciaux.
- Une liste est une séquence d'atomes et/ou d'autres listes entre parenthèses.
- Une chaîne est un groupe de caractères entre doubles guillemets(‘’LISP’’).
- L’ajout d’un commentaire se fait par le biais du caractère (;)

Syntaxe

- Les opérations numériques de base dans LISP sont +, -, * et /
- LISP représente un appel de fonction f(x) comme (f x), par exemple cos(45) est écrit comme (cos 45)
- Les expressions LISP sont insensibles à la casse, cos 45 ou COS 45 sont identiques.
- LISP essaie de tout évaluer, y compris les arguments d'une fonction.
- Seuls trois types d'éléments sont des constantes et renvoient toujours leur propre valeur:
  - Nombres
  - La lettre t, qui signifie vrai.
  - La valeur nil, qui signifie faux ou une liste vide.

Syntaxe

- Le nom ou les symboles peuvent être constitués d'un nombre quelconque de caractères alphanumériques autres que
  - des espaces,
  - des parenthèses ouvrantes et fermantes,
  - des guillemets doubles et simples,
  - une barre oblique inverse,
  - une virgule,
  - deux points,
  - un point-virgule et
  - une barre verticale.
- Pour utiliser ces caractères dans un nom, vous devez utiliser le caractère d'échappement (\).
- Un nom peut avoir des chiffres mais pas entièrement composé de chiffres, car il serait alors lu comme un nombre.
- De même, un nom peut avoir des points, mais ne peut pas être entièrement composé de points.

Les Types

- Dans LISP, les variables ne sont pas typées, mais les objets de données le sont.
- Les types de données LISP peuvent être classés comme:
  - Types scalaires (types de nombres, caractères, symboles, etc.)
  - Structures de données (listes, vecteurs, vecteurs binaires et chaînes)
- N'importe quelle variable peut prendre n'importe quel objet LISP comme valeur, à moins que vous ne l'ayez déclaré explicitement.
- Bien qu'il ne soit pas nécessaire de spécifier un type de données pour une variable LISP, cela aide cependant dans certaines extensions de boucle, dans les déclarations de méthode
- Les types de données sont organisés dans une hiérarchie.
- Un type de données est un ensemble d'objets LISP et de nombreux objets peuvent appartenir à un tel ensemble.
- Le prédicat « typep » est utilisé pour déterminer si un objet appartient à un type spécifique.
- La fonction type-of renvoie le type de données d'un objet donné.

Les Types

Quelques Types spécifiques dans LISP

array	fixnum	package	simple-string
atom	float	pathname	simple-vector
bignum	function	random-state	single-float
bit	hash-table	ratio	standard-char
bit-vector	integer	rational	stream
character	keyword	readtable	string
[common]	list	sequence	[string-char]
compiled-function	long-float	short-float	symbol
complex	nill	signed-byte	t
cons	null	simple-array	unsigned-byte
double-float	number	simple-bit-vector	vector

- Outre ces types définis par le système, vous pouvez créer vos propres types de données.
- Lorsqu'un type de structure est défini à l'aide de la fonction « defstruct », le nom du type de structure devient un symbole de type valide.

Les Macro

- Les macros vous permettent d'étendre la syntaxe du LISP standard.
- Techniquement, une macro est une fonction qui prend une s-expression comme arguments et renvoie une forme LISP, qui est ensuite évaluée.
- Une macro nommée est définie à l'aide d'une autre macro nommée defmacro.
- La syntaxe pour définir une macro est:

  (nom de macro defmacro (liste de paramètres))

  "Chaîne de documentation facultative "

  forme du corps
- La définition de la macro se compose du nom de la macro, d'une liste de paramètres, d'une chaîne de documentation facultative et d'un corps d'expressions Lisp qui définit le travail à effectuer par la macro.

Les Variables

Variables globales

- En LISP, chaque variable est représentée par un symbole.
- Le nom de la variable est le nom du symbole et il est stocké dans la cellule de stockage du symbole.
- Les variables globales ont des valeurs permanentes dans tout le système LISP et restent en vigueur jusqu'à ce qu'une nouvelle valeur soit spécifiée.
- Les variables globales sont généralement déclarées à l'aide de la construction defvar.

Par exemple

```
(defvar x 234)
(write x)
```

- Lorsque vous cliquez sur le bouton Exécuter ou tapez Ctrl+E, LISP l'exécute immédiatement et le résultat renvoyé est

```
234
```

- Comme il n'y a pas de déclaration de type pour les variables dans LISP, vous spécifiez directement une valeur pour un symbole avec la construction setq.

Par exemple

```
(setq x 10)
```

L'expression ci-dessus affecte la valeur 10 à la variable x.

Les Variables

- Vous pouvez faire référence à la variable en utilisant le symbole lui-même comme expression.
- La fonction symbole-valeur vous permet d'extraire la valeur stockée à l'emplacement de stockage du symbole.

Par exemple

```
(setq x 100)
(setq y 200)
(format t "x = ~2d y = ~2d" x y)
```

- Lorsque vous cliquez sur le bouton Exécuter ou tapez Ctrl+E, LISP l'exécute immédiatement et le résultat renvoyé est

```
x = 100 y = 200
```

Les Variables

Variables locales

- Les variables locales sont définies au sein d'une procédure donnée.
- Les paramètres nommés comme arguments dans une définition de fonction sont également des variables locales.
- Les variables locales ne sont accessibles que dans la fonction concernée.
- Comme les variables globales, les variables locales peuvent également être créées à l'aide de la construction setq.
- Il existe deux autres constructions - let et prog pour créer des variables locales.
- La construction let a la syntaxe suivante

  (let ((var1 val1) (var2 val2).. (varn valn))<s-expressions>)
- Où var1, var2, ..varn sont des noms de variables et val1, val2, .. valn sont les valeurs initiales attribuées aux variables.

Les Variables

Variables locales

- Lorsque let est exécuté, chaque variable se voit attribuer sa valeur respective et enfin la s-expression est évaluée.
- La valeur de la dernière expression évaluée est renvoyée.
- Si vous n'incluez pas de valeur initiale pour une variable, elle est affectée à nil.

Exemple

```
(let ((x 'a) (y 'b)(z 'c))
(format t "x = ~a y = ~a z = ~a" x y z))
```

- Lorsque vous cliquez sur le bouton Exécuter ou tapez Ctrl+E, LISP l'exécute immédiatement et le résultat renvoyé est

```
x = A y = B z = C
```

Les Variables

Variables locales

- La construction prog a également la liste des variables locales comme premier argument, qui est suivi du corps du prog et d'un nombre quelconque de s-expressions.
- La fonction prog exécute la liste des s-expressions dans l'ordre et renvoie nil à moins qu'elle ne rencontre un appel de fonction nommé return.
- Ensuite, l'argument de la fonction de retour est évalué et renvoyé.

Exemple

```
(prog ((x '(a b c))(y '(1 2 3))(z '(p q 10)))
  (format t "x = ~a y = ~a z = ~a" x y z))
```

Résultat

```
x = (A B C) y = (1 2 3) z = (P Q 10)
```

Les Constantes

- Dans LISP, les constantes sont des variables qui ne changent jamais leurs valeurs pendant l'exécution du programme.
- Les constantes sont déclarées à l'aide de la construction defconstant.

Exemple

```
(defconstant PI 3.141592)
(defun area-circle(rad)
(terpri)
(format t "Radius: ~5f" rad)
(format t "~%Area: ~10f" (* PI rad rad)))
(area-circle 10)
```

Résultat

```
Radius:  10.0
Area:   314.1592
```

Les Opérateurs

- Un opérateur est un symbole qui indique au compilateur d'effectuer des manipulations mathématiques ou logiques spécifiques.
- LISP permet de nombreuses opérations sur les données, prises en charge par diverses fonctions, macros et autres constructions.
- Les opérations autorisées sur les données pourraient être classées comme
  - Opérations arithmétiques
  - Opérations de comparaison
  - Opérations logiques
  - Opérations au niveau du bit

Les Opérateurs

Opérations arithmétiques

- Le tableau suivant présente tous les opérateurs arithmétiques pris en charge par LISP.

Opérateur	Description
+	Addition
-	Soustraction
*	Multiplication
/	Division
mod,rem	Le mode et le reste de la division
incf	Incrementation
decf	Decrementation

Les Opérateurs

Opérations de comparaison

Opérateur	Description
=	Vérifie si les valeurs des opérandes sont toutes égales ou non, si oui, la condition devient vraie.
/=	Vérifie si les valeurs des opérandes sont toutes différentes ou non, si les valeurs ne sont pas égales, la condition devient vraie.
>	Vérifie si les valeurs des opérandes diminuent de manière monotone.
<	Vérifie si les valeurs des opérandes augmentent de manière monotone.
>=	Vérifie si la valeur d'un opérande gauche est supérieure ou égale à la valeur de l'opérande droit suivant, si oui, la condition devient vraie.
<=	Vérifie si la valeur d'un opérande gauche est inférieure ou égale à la valeur de son opérande droit, si oui, la condition devient vraie.
max	Il compare deux arguments ou plus et renvoie la valeur maximale.
min	Il compare deux arguments ou plus et renvoie la valeur minimale.

Les Opérateurs

Opérations logiques

Opérateur	Description
and	Il faut un certain nombre d'arguments. Les arguments sont évalués de gauche à droite. Si tous les arguments ont une valeur non nulle, la valeur du dernier argument est renvoyée. Sinon, rien n'est retourné.
or	Il faut un certain nombre d'arguments. Les arguments sont évalués de gauche à droite jusqu'à ce que l'on évalue à non nul, dans ce cas la valeur de l'argument est renvoyée, sinon elle renvoie nil.
not	Il prend un argument et renvoie t si l'argument est évalué à nil.

Les Opérateurs

Opérations au niveau du bit

p	q	p and q	p or q	p xor q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Si on suppose que p = 60; et q = 13; maintenant au format binaire, ils seront comme suit:
p = 0011 1100
q = 0000 1101
---
p and q = 0000 1100
p or q = 0011 1101
p xor q = 0011 0001
not p = 1100 0011

Les Opérateurs

Opérations au niveau du bit

Opérateur	Description
logand	Cela renvoie le ET logique binaire de ses arguments. Si aucun argument n'est donné, le résultat est -1, qui est une identité pour cette opération.
logior	Cela renvoie le BIT logique INCLUSIVE OR de ses arguments. Si aucun argument n'est donné, le résultat est nul, ce qui est une identité pour cette opération.
logxor	Cela renvoie l'EXCLUSIF OU logique binaire de ses arguments. Si aucun argument n'est donné, le résultat est nul, ce qui est une identité pour cette opération.
lognor	Cela renvoie le BIT NOT de ses arguments. Si aucun argument n'est donné, le résultat est -1, qui est une identité pour cette opération.
logeqv	Cela renvoie l'ÉQUIVALENCE logique en bits (également appelée exclusive nor) de ses arguments. Si aucun argument n'est donné, le résultat est -1, qui est une identité pour cette opération.

La prise de décision

- Les structures de prise de décision exigent que le programmeur spécifie une ou plusieurs conditions à évaluer ou à tester par le programme, ainsi qu'une instruction ou des instructions à exécuter si la condition est déterminée comme étant vraie, et facultativement, d'autres instructions à exécuter si la condition est déterminé comme étant faux.

La prise de décision cond

- La construction cond dans LISP est le plus souvent utilisée pour permettre le branchement.
- La syntaxe pour cond est

```
(cond   (test1    action1)
        (test2    action2)
                  ...
        (testn    actionn))
```

- Chaque clause de l'instruction cond se compose d'un test conditionnel et d'une action à effectuer.
- Si le premier test suivant cond, test1, est évalué comme étant vrai, alors la partie d'action associée, action1, est exécutée, sa valeur est renvoyée et le reste des clauses est ignoré.
- Si test1 est évalué à nil, alors le contrôle passe à la deuxième clause sans exécuter action1, et le même processus est suivi.
- Si aucune des conditions de test n'est évaluée comme vraie, l'instruction cond renvoie nil.

La prise de décision cond

- Exemple

```
(setq a 10)
(cond ((> a 20)
(format t "~% a is greater than 20"))
(t (format t "~% value of a is ~d " a)))
```

Résultat

```
value of a is 10
```

- Noter que le t dans la deuxième clause garantit que la dernière action est effectuée si aucune autre action ne le ferait.

La prise de décision if

- La macro if est suivie d'une clause de test qui vaut t ou nil.
- Si la clause de test est évaluée au t, alors l'action suivant la clause de test est exécutée. S'il est nul, la clause suivante est évaluée.
- Syntaxe de if

```
(if (test-clause) (action1) (action2))
```

La prise de décision if

Exemple

```
(setq a 10)
(if (> a 20)
(format t "~% a is less than 20"))
(format t "~% value of a is ~d " a)
```

Résultat

```
value of a is 10
```

La prise de décision When

- La macro when est suivie d'une clause de test qui vaut t ou nil.
- Si la clause test est évaluée à nil, alors aucune forme n'est évaluée et nil est renvoyé
- Si le résultat du test est t, alors l'action suivant la clause test est exécutée.
- Syntaxe pour quand macro

(when (test-clause) (<action1))

La prise de décision When

- Exemple

```
(setq a 100)
(when (> a 20)
(format t "~% a is greater than 20"))
(format t "~% value of a is ~d " a)
```

Résultat

```
a is greater than 20
value of a is 100
```

La prise de décision CASE

- La construction case implémente plusieurs clauses test-action comme la construction cond.
- Cependant, il évalue une forme clé et autorise plusieurs clauses d'action basées sur l'évaluation de cette forme clé.
- La syntaxe de la macro case est

```
(case  (keyform)
((key1)   (action1   action2 ...) )
((key2)   (action1   action2 ...) )
...
((keyn)   (action1   action2 ...) ))
```

La prise de décision CASE

- Exemple

```
(setq day 4)
(case day
(1 (format t "~% Dimanche"))
(2 (format t "~% Lundi"))
(3 (format t "~% Mardi"))
(4 (format t "~% Mercredi"))
(5 (format t "~% Jeudi"))
(6 (format t "~% Vendredi"))
(7 (format t "~% Samedi")))
```

Les boucles

- Il peut arriver que vous deviez exécuter un bloc de code plusieurs fois.
- Une instruction de boucle nous permet d'exécuter une instruction ou un groupe d'instructions plusieurs fois
- LISP fournit les types de constructions suivants pour gérer les exigences de bouclage.

Les boucles loop

- La construction de boucle est la forme d'itération la plus simple fournie par LISP.
- Dans sa forme la plus simple, il vous permet d'exécuter plusieurs instructions à plusieurs reprises jusqu'à ce qu'il trouve une instruction de retour.
- Il a la syntaxe suivante

(loop (s-expressions))

Les boucles loop

- Exemple

```
(setq a 10)
(loop
(setq a (+ a 1))
(write a)
(terpri)
(when (> a 17) (return a))
)
```

Les boucles loop for

- La boucle for construct vous permet d'implémenter une itération de type boucle for comme la plus courante dans d'autres langages.
- Il vous permet de
  - configurer des variables pour l'itération
  - spécifier les expressions qui termineront l'itération de manière conditionnelle
  - spécifier une ou plusieurs expressions pour effectuer un travail à chaque itération
  - spécifier des expressions et des expressions pour effectuer un travail avant de quitter la boucle
  - La boucle for pour la construction suit plusieurs syntaxes -

Les boucles loop for

- La boucle for pour la construction suit plusieurs syntaxes

(loop for loop-variable in <a list>

do (action)

)

(loop for loop-variable from value1 to value2

do (action)

)

Les boucles Do

- La construction do est également utilisée pour effectuer une itération à l'aide de LISP.
- Il fournit une forme structurée d'itération.
- La syntaxe de l'instruction do est

```
(do ((variable1    value1   updated-value1)
     (variable2   value2   updated-value2)
     (variable3   value3   updated-value3)
     ...)
    (test return-value)
    (s-expressions)
    )
```

Les boucles Do

- Les valeurs initiales de chaque variable sont évaluées et liées à la variable respective.
- La valeur mise à jour dans chaque clause correspond à une instruction de mise à jour facultative qui spécifie comment les valeurs des variables seront mises à jour à chaque itération.
- Après chaque itération, le test est évalué et s'il renvoie une valeur non nulle ou vraie, la valeur de retour est évaluée et renvoyée.
- La ou les dernières expressions-s sont facultatives.
- S'ils sont présents, ils sont exécutés après chaque itération, jusqu'à ce que la valeur de test renvoie true.

Les boucles Do

- Exemple

```
(do ((x 0 (+ 2 x))
   (y 20 ( - y 2)))
   ((= x y)(- x y))
(format t "~% x = ~d  y = ~d" x y)
)
```

Les boucles Dotimes

- La construction dotimes permet de boucler pour un nombre fixe d'itérations.
- Exemple

```
(dotimes (n 11)
(print n) (prin1 (* n n))
)
```

Les boucles Dotlist

- La construction dolist permet l'itération à travers chaque élément d'une liste.
- Exemple

```
(dolist (n '(1 2 3 4 5 6 7 8 9))
(format t "~% Number: ~d Square: ~d" n (* n n))
)
```