Formulaire syntaxe CLIPS

1. Les faits ordonnés :

Un fait ordonné est une liste plate de termes commençant par une relation (symbole quelconque). Les faits ordonnés ne nécessitent pas de déclaration préalable. Voilà quelques exemples :

```
(pere-de Jerome Pierre)             ; Jérôme est le père de Pierre
(est-dans-sa-baignoire Paul)        ; Paul est dans sa baignoire
(refrigerator light on)             ; La lumière du réfrigérateur est allumée
(phrase le chat mange la souris)    ; Le chat mange la souris
```

L'opération qui permet de reconnaître un fait ordonné est le pattern matching. Il s'agit de comparer terme à terme un fait ordonné avec un filtre (pattern) et d'associer, le cas échéant, des variables à des valeurs. Un pattern est composé de constantes littérales (symboles, nombres, chaines), de variables et de jockers. Les variables et les jockers sont monovalués ou multivalués :

```
(pere-de ?pere ?)                   ; Rechercher tous les pères
(pere-de ?pere ?enfant)             ; Rechercher des couples (père, enfant)
(pere-de Jerome ?)                  ; Savoir si Jérôme a des enfants
(pere-de Jerome ?enfant)            ; Rechercher les enfants de Jérôme
(pere-de ~Jerome ?enfant)           ; Rechercher les enfants qui ne sont pas de Jérôme
(phrase $?liste1 mange $?liste2)    ; Savoir qui mange qui
(phrase ?premier $?)                ; Extraire les premiers mots des phrases
```

2. Les faits structurés :

Un fait structuré (template) comporte des attributs nommés qui contiennent chacun une ou plusieurs données. Un attribut monovalué (slot) contient une donnée, un attribut multivalué (multislot) contient une liste de données (éventuellement vide). L'ordre des attributs n'a pas d'importance puisqu'ils sont identifiés par leur nom. Les faits structurés nécessitent une déclaration préalable :

```
(deftemplate personne "Le modèle des personnes"
       (slot ident (type INTEGER SYMBOL) (default ?NONE))       ; Identificateur
       (slot nom (type STRING) (default ?NONE))                 ; Nom de la personne
       (slot prenom (type STRING) (default ?DERIVE))            ; Prénom de la personne
       (slot sexe (allowed-symbols H F) (default ?NONE))        ; Sexe de la personne
       (slot alive (allowed-symbols TRUE FALSE) (default TRUE)) ; État de la personne
       (multislot aime (type INTEGER SYMBOL)))                  ; Affinités de la personne
```

Les attributs de contrainte (type, default, etc.) ne sont pas détaillés dans ce document. Ajoutons que l'on peut spécifier un type, une liste de valeurs, un intervalle, un cardinal, etc. Une fois le modèle personne déclaré, on peut déclarer autant de faits « personne » que l'on veut :

```
(deffacts faits-initiaux
       (personne (ident 007) (nom "Bond") (prenom "James") (sexe M))
       (personne (ident MP) (nom "Moneypenny") (sexe F) (aime 007)))
```

3. Syntaxe des règles :

Un programme CLIPS est un ensemble de règles composées d'un nom et de 2 parties : les préconditions (membre gauche ou LHS) et les actions (membre droit ou RHS). Déclencher une règle signifie exécuter le RHS en réutilisant les éventuelles associations variable=valeur issues du LHS. Les actions peuvent être des ajouts, suppressions ou modifications de faits, des affichages, ou encore des appels à des fonctions ayant des effets de bord.

Pour supprimer ou modifier un fait, il faut mémoriser son « fact-adress » dans une variable :

```
(defrule meurtre "Une personne vient de mourir"

        ?f1 <- (action ? tue ?ident)                    ; Fait ordonné adressé par ?f1
        ?f2 <- (personne (ident ?ident))                ; Fait structuré adressé par ?f2
        =>
        (retract ?f1)                                   ; Suppression du fait ordonné
        (modify ?f2 (alive FALSE))                      ; Modification du fait structuré
        (format t "%s vient de mourir %n" ?ident))      ; Affichage vers la console
```

Il est également possible de dupliquer un fait structuré tout en modifiant (ou non) ses slots :

```
(defrule clonage "Une personne vient d'être clonée"

        ?f1 <- (action ? clone ?ident)                  ; Fait ordonné adressé par ?f1
        ?f2 <- (personne (ident ?ident))                ; Fait structuré adressé par ?f2
        =>
        (retract ?f1)                                   ; Suppression du fait ordonné
        (duplicate ?f2 (ident (gensym*)))               ; Duplication du fait structuré
        (format t "%s vient d'être cloné %n" ?ident))   ; Affichage vers la console
```

4. Syntaxe des préconditions :

Les LHS des règles sont constitués d'une série de préconditions devant toutes être vérifiées pour que la règle soit déclenchable. Les préconditions les plus courantes sont :

- Les filtres `(<relation> ...)` pour trouver un ou plusieurs faits => plusieurs activations possibles
- Les conditions `(exists (<relation> ...))` pour trouver un groupe de faits => une activation
- Les conditions `(not (<relation> ...))` pour vérifier l'absence de faits => une activation
- Les combinaisons `(or (...) (...))` et `(and (...) (...))` sur tout types de préconditions
- Les tests booléens `(test ...)` portant sur des variables déjà liées

L'ordre des préconditions influe sur l'efficacité du programme mais il est difficile de définir des règles d'écriture qui améliorent systématiquement l'efficacité des programmes. En règle générale, les conditions les plus contraignantes seront placées en tête. Le reste est souvent une affaire de bon sens.

La syntaxe des préconditions permet également de faire des tests au moment où une variable va être liée. Chaque test doit être précédé du connecteur & (et) ou du connecteur | (ou) :

```
?x&~20                      ; La valeur de ?x doit être différente de 20
?x&oui|non                  ; La variable ?x ne peut prendre que les valeurs oui et non
?x&=(* ?y 2)                ; La valeur de ?x doit être le double de celle de ?y
?x&:(numberp ?x)            ; La valeur de ?x ne peut être que du type nombre
?x&:(>= ?x 0)&:(<= ?x 10)   ; La valeur de ?x doit être comprise dans l'intervalle [0,10]
```

5. Syntaxe des actions et fonctions :

La syntaxe de CLIPS a été construite pour ne pas dérouter les programmeurs LISP. Par exemple, l'expression `(assert (resultat (nth$ 2 ?var)))` ajoute le fait `(resultat B)` si `?var` a la valeur `(A B C)`. Il existe beaucoup de fonctions prédéfinies, et il est facile d'en définir de nouvelles, récursives ou non :

```
(deffunction intersection (?liste1 ?liste2) "Intersection ensembliste non récursive"

        (bind ?resultat (create$))
        (progn$ (?element ?liste1)
                (if (member$ ?element ?liste2)
                        then (bind ?resultat (create$ ?resultat ?element))))
        (return ?resultat))
```

Quelques fonctions prédéfinies :

Égalité et différence :	`(eq <expr> <expr>+) (neq <expr> <expr>+)`
Opérateurs booléens :	`(and <expr>+) (or <expr>+) (not <expr>)`
Prédicats de type :	`(integerp <expr>) (stringp <expr>) (multifieldp <expr>) ...`
Prédicats sur les nombres :	`(evenp <expr>) (oddp <expr>) ...`
Comparaisons sur les nombres :	`(= <expr> <expr>+) (!= <expr> <expr>+) (> <expr> <expr>+) ...`
Opérateurs arithmétiques :	`(+ <expr> <expr>+) (/ <expr> <expr>+) (div <expr> <expr>+) ...`
Fonctions mathématiques :	`(abs <expr>) (min <expr>+) (cos <expr>) (sqrt <expr>) ...`
Fonctions sur les chaines :	`(upcase <str>) (lowcase <str>) (str-index <str> <str>)`
	`(str-cat <str>+) (sub-string <int> <int> <str>)`
	`(explode$ <str>) (str-length <str>) ...`
Fonctions sur les listes :	`(implode$ <liste>) (member$ <expr> <liste>) (create$ <expr>*)`
	`(subseq$ <liste> <int> <int>) (delete$ <liste> <int> <int>)`
	`(nth$ <int> <liste>) (length$ <liste>) ...`
Fonctions d'entrée/sortie :	`(readline) (printout t <expr>*) (format t <str> <expr>*) ...`
Commandes particulières :	`(bind <variable> <expr>)` associe une valeur à une variable
	`(gensym*)` génère un symbole du type `genX` où `X` est un entier
	`(random <int> <int>)` génère un nombre entier aléatoire
	`(sort <fonction-name> <expr>*)` trie une liste de valeurs
Fonctions sur les faits :	`(fact-index <fact-adress>)` retourne le numéro d'un fait
	`(fact-existp <fact-adress>)` permet de savoir si un fait existe
	`(fact-relation <fact-adress>)` retourne la relation d'un fait
	`(fact-slot-value <fact-asdress> <slot>)` retourne la valeur d'un slot
	Pour obtenir les données d'un fait ordonné, utilisez le slot « implied »
Structures algorithmiques :	`(if <expr> then <action>* [else <action>*])`
	`(while <expression> <action>*)`
	`(loop-for-count (<variable> <int> <int>) <action>*)`
	`(progn$ (<variable> <liste>) <expr>*)`
	`(switch <expr> (case <expr> then <action>*)`
	`...`
	`[(default <action>*)])`

6. Programmation modulaire :

Il est possible de déclarer des « paquets de règles » comme autant de programmes distincts (modules). Chaque module possède sa propre base de connaissances (règles, faits, etc.) et son propre agenda. Un seul module est actif (focus) et le passage d'un module à l'autre est géré à l'aide d'une pile de focus :

```
(reset)
(get-focus-stack)       > (MAIN)          ; Le module par défaut est MAIN
(defmodule A)                             ; Déclaration d'un nouveau module A
(defmodule B)                             ; Déclaration d'un nouveau module B
(focus A B)             > TRUE            ; Empiler les modules A et B (A au dessus)
(get-focus-stack)       > (A B MAIN)      ; État de la pile de focus
(pop-focus)             > A               ; Dépiler le module courant
(get-focus-stack)       > (B MAIN)        ; État de la pile de focus
```

Le module courant est défini par le sommet de la pile de focus. Lorsqu'il n'y plus rien à faire dans le module courant (agenda vide) alors celui-ci est automatiquement dépilé. Si la pile de focus n'est pas vide, un nouveau module prend le relais, sinon le programme s'arrête naturellement :

```
; --------------------------------------
; Définitions du module MAIN
; --------------------------------------

(defrule MAIN::initialisation "Règle qui construit la pile de focus"
       (declare (salience 999))
       =>
       (printout t "Firing first rule in module MAIN" crlf)
       (focus A B))

(defrule MAIN::terminaison "Règle qui va terminer le programme"
       =>
       (printout t "Firing last rule in module MAIN" crlf))

; --------------------------------------
; Définitions du module A
; --------------------------------------

(defmodule A)
(deffacts A::faits-initiaux (foo))
(defrule A::exemple (foo) => (printout t "Firing rule in module A" crlf))

; --------------------------------------
; Définitions du module B
; --------------------------------------

(defmodule B)
(deffacts B::faits-initiaux (bar))
(defrule B::exemple (bar) => (printout t "Firing rule in module B" crlf))

; --------------------------------------

CLIPS> (reset)                      ; Pile de focus = (MAIN)
CLIPS> (run)
Firing first rule in module MAIN    ; La règle empile deux focus => pile = (A B MAIN)
Firing rule in module A             ; Plus rien à faire dans A => pile = (B MAIN)
Firing rule in module B             ; Plus rien à faire dans B => pile = (MAIN)
Firing last rule in module MAIN     ; La dernière règle de MAIN se déclenche
```

Il est également possible de partager certains faits pas un mécanisme de visibilité entre modules :

```
(defmodule A (export deftemplate foo bar))
(deftemplate A::foo (slot x))
(deftemplate A::bar (slot y))
(deffacts A::init (foo (x 3)) (bar (y 4)))
(defmodule B (import A deftemplate foo))

CLIPS> (reset)
CLIPS> (facts A)                    ; Afficher la base de fait du module A
f-1    (foo (x 3))
f-2    (bar (y 4))
For a total of 2 facts.
CLIPS> (facts B)                    ; Afficher la base de fait du module B
f-1    (foo (x 3))
For a total of 1 fact.              ; Seul le fait foo est visible du module B
```