Programmer avec CLIPS

Support de cours de l'EP "Intelligence Artificielle" (n°365)

Erwan TRANVOUEZ

Polytech'Marseille, Département GII

Sources :

Expert Systems : principles and programming, Giarratano and Riley, 3ème édition.
CLIPS Reference Manual, Volume I, Basic Programming Guide [1].
CLIPS User's Guide, Joseph C. Giarratano, nov 1997. [2]
[1] et [2] disponible `http://www.ghgcorp.com/clips/download/documentation/`

Sommaire

1 Introduction

CLIPS est l'acronyme de **C** **L**anguage **I**ntegrated **P**roduction **S**ystem. Ce langage a été élaboré par la NASA pour développer rapidement et à moindre coût des programmes portable sur différentes plates-formes et aisément intégrable avec d'autres applications. Le choix du langage C s'explique ainsi au vu de ces objectifs. A cet effet, le code source est disponible pour pouvoir utiliser CLIPS sur toute plate-forme disposant d'un compilateur C ANSI.

CLIPS est ce qu'on appelle un générateur de Systèmes Experts (*Shell* en anglais) à **base de règles** de production. Il propose :

- Un Moteur d'Inférence en chaînage avant
- Un langage orienté objet (COOL)
- Une programmation procédurale (deffunction)

Nous utiliserons plus précisément CLIPS V.6.10 sous Windows (voir Figure 1.). Cette version propose une aide intégrée sur le langage CLIPS (fonctions usuelles, fonctionnalités de l'outil), une interface graphique et surtout des outils de débuggages intégrés. Ce logiciel est disponible gratuitement sur le site `http://www.ghgcorp.com/clips/`.

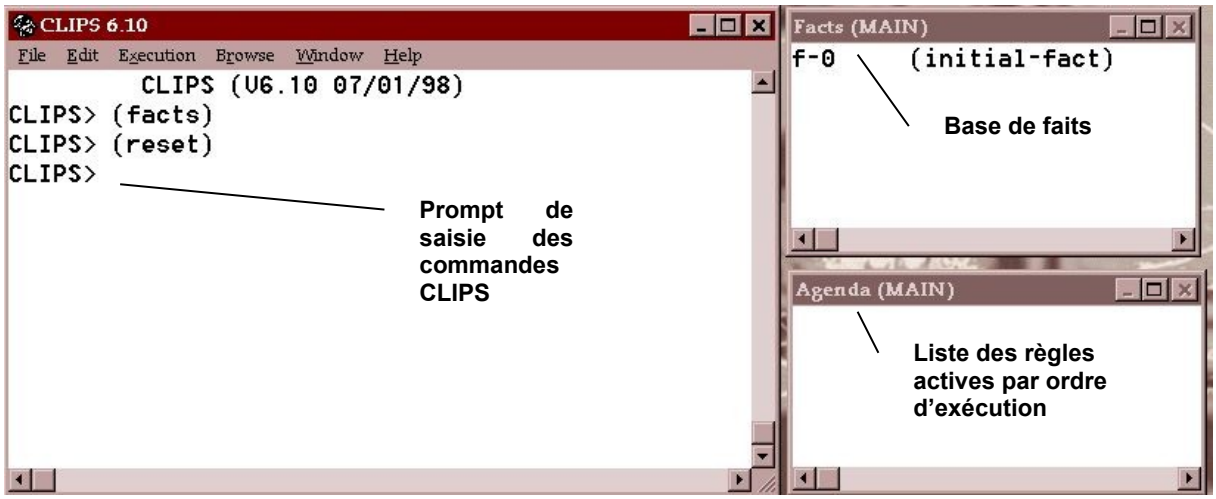

Figure 1 – L'environnement CLIPS

Nous reviendrons sur la spécificité de cet environnement après avoir introduit le langage CLIPS. Notons que la partie Objet (langage COOL) ne sera pas abordée, celle-ci ne participant pas directement à la compréhension du fonctionnement et du développement d'un Système Expert.

2 Le langage CLIPS

CLIPS est un Système Expert permettant de manipuler des faits et règles à l'aide d'un moteur d'inférence en chaînage avant. Nous précisons tout d'abord la syntaxe du langage, avant d'aborder la problématique de la représentation de connaissance en CLIPS.

2.1 Syntaxe

CLIPS utilise une syntaxe similaire à LISP. Ceci se traduit par l'omniprésence de parenthèses ouvrantes "(" et fermantes ")". Elles sont notamment nécessaires pour appeler des fonctions systèmes ou définies par le programmeur (*programmation procédurale*).

Par exemple, la commande suivante quitte l'environnement CLIPS :

```
CLIPS>(exit)
```

Par ailleurs, les opérations mathématiques utilisent une notation préfixée. Ainsi les calculs 3+2 et 2*5-8 s'écrivent en CLIPS comme suivant :

```
CLIPS>(+ 3 2)
5
CLIPS>(- (* 2 5) 8)
2
CLIPS>
```

Ceci nous conduit à préciser les types de données reconnus par CLIPS dans le tableau ci-dessous. Nous utilisons à cet effet la notation BNF (voir annexe 1). Seuls les types de la partie grisée vous seront nécessaires pour la réalisation de votre EP.

Type de donnée	Définition	Exemple
Integer	`[+|-] nombre`	`1  +3  -1  65`
Float	**`Integer`** `[.[nombre] [e [+|-] nombre]`	`1.5  1.0  9e-1  3.5e1`
Symbol	`<lettre|chiffre|!|#|^|*><caractères>*` **Attention :** CLIPS distingue minuscules et majuscules	`Bonjour, bonjour, hello-world, 345B, 127-0-0-1`
String	`<"><caractère>*<">`	`"hello world"  ""\"hello\" "  "10 francs "`
External address	Adresse mémoire d'une structure de donnée externe i.e. retournée par une **fonction utilisateur**[1] `<Pointer-XXXXXX>` où `XXXXXX` est l'adresse mémoire externe	`<Pointer-00CF61AB>`
Instance name	`<[>` **`Symbol`** `<]>` : nom d'une instance de classe dans COOL	`[pump-1]`
Instance address	Adresse mémoire d'un objet COOL. `<Instance-XXX>` où `XXX` est le nom de l'instance.	`[Instance-pump-1]`

Tableau 1 – Types de données dans CLIPS

A ces types de données primitifs (appelé aussi **champ**) s'ajoute celui de liste ou **multifield**. Une liste consiste en une succession de valeurs entre parenthèses.

```
(bleu 12 rouge 15.5)
```

Une liste vide est alors notée :

```
()
```

Enfin, le caractère "**;**" est utilisé pour commenter des lignes. Tout caractère situé après un point virgule sur la même ligne est ignoré par le *parser* CLIPS. Dans l'exemple ci-dessous, les caractères en gras ne seront pas lus par CLIPS :

```
(printout t "Hello World" ) ; affiche hello world sur le terminal
; t précise le flux de sortie (ici l'écran)
```

Par ailleurs, il se peut que dans la documentation ou autre source d'information vous tombiez sur le mot **construct**. Ce dernier recouvre les différents concepts CLIPS utilisés pour écrire un programme. Vous le retrouverez notamment lors de la définition ou manipulation de faits (deftemplate, deffacts), ou de règles (defrule).

[1] Programmée en C par exemple. Ceci traduit la propriété d'intégration de CLIPS avec d'autres programmes.

2.2 Représentation des faits en CLIPS

Un fait CLIPS est défini comme suivant :

```
(nom-de-la-relation valeurs)
```

ex :

```
(pere albert gerard)
```

Le nom de la relation identifie le sens, la portée ou l'objectif du fait. Les valeurs précisent l'information véhiculée par le fait. Ces valeurs peuvent être exprimées sous la forme d'une liste de valeurs ou de champs : on parlera dans le premier cas de faits ordonnés ou non structurés et de faits structurés ou non ordonnés dans le second cas.

Les faits ordonnés

C'est la forme la plus simple utilisée. L'ordre des valeurs est important. Ainsi les deux faits suivants sont considérés comme différents alors qu'ils utilisent les mêmes valeurs.

```
(etudiant gerard lambert 22 2)
(etudiant lambert gerard 22 2)
```

Ainsi, l'utilisation des faits ordonnés suppose le respect de l'ordre des valeurs notamment si ces valeurs sont utilisées dans une règle. Cela implique donc de définir une fois pour toute comment se renseigne le fait `etudiant` : i.e. `nom-prenom-age-classe` ou `prenom-nom-age-classe`.

Ces faits sont souvent utilisés pour des informations simples ou si l'ordre est important. Exemple :

```
(porte 1 ouverte)
(porte 2 fermee)
```

Les faits structurés

L'utilisation de faits structurés répond au même besoin que les `record` du langage Pascal ou `struct` du langage C. Ils sont utilisés pour enregistrer des informations sur un même objet. Ainsi, le fait `etudiant` ci-dessus aurait profité d'une représentation par un fait structuré.

```
(etudiant (prenom gerard) (age 22) (nom lambert) (classe 2) )
```

Qu'importe l'ordre des champs, ce fait est unique.

L'utilisation d'une telle structure impose de la définir au préalable notamment afin de vérifier la bonne syntaxe des faits. Cela est effectué à l'aide du mot clé `deftemplate` dont l'utilisation est la suivante[2] :

```
(deftemplate nom-du-fait-structuré  "commentaires explicatifs"
       (slot champ1)
       (slot champ2)
       (slot champ3)
)
```

Notons que le mot clé `slot` indique que le champ ne peut avoir qu'une seule valeur. Pour spécifier qu'il peut y avoir plusieurs valeurs (i.e. une **liste** de valeurs), il faut utiliser le mot clé **`multislot`**. Ainsi, pour indiquer qu'un étudiant d'avoir plusieurs prénoms, le `deftemplate` correspondant est le suivant.

```
(deftemplate etudiant  "renseignement sur un étudiant"
       (slot nom)
       (multislot prenoms)
       (slot age)
       (slot classe)
)
```

[2] Cf Annexe pour description complète

Exemple de faits :

```
(etudiant
       (nom lambert)
       (prenoms gerard lucien)
       (age 22)
       (classe 2)
)
```

Attention : il n'est pas possible de **redéfinir** ce `deftemplate` **tant qu'**il existe dans la base de fait un fait `etudiant`. Ainsi, si l'on voulait ajouter un champ `ecole` à ce `deftemplate` il faudrait tout d'abord supprimer le fait ci-dessus.

Ajout de faits

L'insertion de faits dans CLIPS est effectuée par la commande `(assert <fait>*)`.

```
CLIPS> (assert (voiture renault 19) (voiture renault R5))
<fact-1>
```

En retour, la commande `assert` donne l'identifiant du dernier fait inséré. Cet identifiant (ou `fact-id`) est formé du mot `fact-` et d'un numéro incrémenté à chaque insertion. La commande `(facts)`[3] permet de lister le contenu de la base de faits. Sur l'exemple précédent on obtient :

```
CLIPS> (facts)
f-0    (voiture renault 19)
f-1    (voiture renault R5)
For a total of 2 facts
```

Attention : l'insertion d'un fait structuré nécessite la définition au préalable du deftemplate associé. Dans le cas contraire, CLIPS interprète le champ comme un appel de fonction ayant de grande chance de ne pas exister. Ceci conduit alors au message d'erreur suivant :

```
CLIPS> (assert (voiture (marque renault) ) )
[EXPRNPSR3] Missing function declaration for marque
```

Suppression de faits

La suppression d'un fait est effectuée à l'aide de la commande `(retract)` qui prend comme paramètre un ou plusieurs fact-id. Ainsi si le véhicule Renault R5 doit être retiré de la base de fait (pour marquer sa destruction suite à un accident par exemple), il est nécessaire de connaître sa position dans la base de faits. Cette information peut être obtenue grâce à la commande `(facts)`.

```
CLIPS> (retract 1)
CLIPS> (facts)
f-0     (voiture renault 19)
For a total of 1 fact
```

Cette information, nous le verrons ci-après, peut également être récupérée dans la partie condition d'une règle.

Modification de faits

Les **faits structurés** présentent également l'avantage de pouvoir être partiellement modifiés à l'aide de la commande `(modify)`. L'équivalent avec les faits ordonnés reviendrait à effectuer un `retract` suivi d'un `assert` avec les nouvelles valeurs. La syntaxe est la suivante :

```
(modify <fact-id> <champ-a-modifier nouvelle-valeur>* )
```

[3] Se référer à l'aide en ligne pour les paramètres supplémentaires.

Exemple : si l'âge de l'étudiant gerard lucien lambert est 23 au lieu de 22 et qu'il soit en 3ème année et non pas en deuxième ; le fact-id de ce fait étant f-0 il est possible de corriger ces erreurs comme suivant :

```
CLIPS>(modify 0 (age 23) (classe 3) )
<Fact-1>
CLIPS>(facts)
f-1    (etudiant (nom lambert) (prenom gerard lucien) (age 23)
       (classe 3) )
For a total of 1 fact
```

Ceci conduit à affecter un nouvel fact-id a ce fait. Nous verrons plus loin les conséquences de cette fonctionnalité.

Base de fait initiale

Il peut être nécessaire de disposer, au démarrage du SE, d'un ensemble de faits. Le mot clé `deffacts` permet de déclarer un ensemble de faits comme devant être inséré après un appel à la commande `(reset)`. Cette dernière supprime tous les faits existants dans la base de faits, insère le fait `(initial-fact)` et insère ensuite tous les faits contenus dans des groupes `deffacts`. La syntaxe est la suivante :

```
(deffacts nom-descriptif
      <fait>+
)
```

Exemple :

```
(deffacts base-de-faits-initial-enseignements
      (ecole
            (nom politech-marseille)
            (departement "Génie Industriel et Informatique")
      )
      (cours
            (intitule "Aide à la décision par Systèmes Experts")
            (enseignant tranvouez)
            (type EP)
            (id 2263)
      )
)
```

2.3 Représentation des règles en CLIPS

CLIPS fonctionne en chaînage avant, les règles se présentent donc sous la forme d'expression `SI…ALORS`. . . . La structure générale d'une déclaration de règle est la suivante :

```
(defrule nom-de-la-regle  "commentaire explicatif"
      (declare           ; propriétés particulières (facultatif)
            (salience <nombre-entier>)
      )
      ; partie condition
      ; faits, Patterns cad patron d'appariement
=>
      ; partie conclusion
      ; insertion de faits, calculs, …
)
```

Le partie `declare` permet de préciser les propriétés particulières, et notamment l'ordre de priorité de la règle défini par la valeur entière inscrite dans le champ `salience`. Cette valeur doit être comprise entre –10 000 et +10 000. L'absence de cette définition, revient à donner une valeur moyenne : 0.

La partie condition d'une règle précise les conditions requises pour l'activation de la règle. Le terme activation signifie que la règle est prête à être exécutée ou plus précisément les instructions situées dans sa partie conclusion. Pour plus de clarté, différents cas sont abordés à l'aide d'exemples.

Exemple : la condition spécifie l'existence d'un fait.

Enoncé :

Si la porte est ouverte **Alors** demander à l'utilisateur de la fermer.

Cette affirmation distingue facilement la condition de la conclusion. Notons que la conclusion implique un envoi d'information à un utilisateur. Nous utiliserons pour ceci la fonction `printout` qui permet d'afficher sur un flux une chaîne de caractères (habituellement le terminal i.e. l'écran avec le paramètre `t`). Une fois l'affichage effectué nous pouvons considérer la porte comme étant fermée.

Ceci implique une modification de la base de faits, soit une suppression et une insertion de fait. Cette opération requiert de disposer du fact-id du fait à supprimer (cf. § 2.2 p. 5) à l'aide de l'opérateur "`<-`". Ainsi l'opération "`?f <- (toto)`" copie le fact-id du fait `(toto)` dans la variable f. Précisons qu'une variable se note par le caractère "`?`" suivi d'un symbole (tel que défini § 2.1).

Exemple de noms de variable valides : `?a, ?a1, ?etat, ?ex-de-variable`.

Attention ! Ce type de variable ne peut stocker des listes (voir ci-après).

L'énoncé est traduit en Jess par la règle suivante :

```
(defrule fermeture-de-la-porte-1  "Informe qu'une porte est ouverte"
       ; partie condition
       ?f <- (porte ouverte)
=>
       ; partie conclusion
       (printout t "Fermer la porte SVP" crlf) ; crlf marque un retour à la
                                               ; ligne (equivalent de \n en C)
       (retract ?f)
       (assert (porte fermee) )
)
```

La condition ici est simple. Supposons que la base de faits contienne le fait `(porte ouverte)`, les conditions d'activation de la règle `fermeture-de-la-porte-1` sont donc vérifiées. Ceci se traduit par son ajout dans l'**agenda**[4], c'est à dire une liste des règles à exécuter organisée en pile. La position exacte de cette règle dans cette pile dépendra de la stratégie employée ainsi que de sa priorité (ou *salience*) (cf. détails § 3.1). L'exécution des règles dans l'agenda est effectuée par la commande `(run)`. Le SE s'exécute alors tant que des règles actives sont présentes dans l'agenda. Il est également possible d'effectuer une exécution par étape en utilisant `(run n)` qui exécute **n** règles puis redonne accès au prompt CLIPS. Le résultat de cette commande dans le cas présent est le suivant :

```
CLIPS>(facts)
f-0    (porte ouverte)
CLIPS>(run 1)
Fermer la porte SVP
CLIPS>(facts)
f-1    (porte fermee)
For a total of 1 fact
CLIPS>
```

La règle informe donc l'utilisateur de la nécessité de fermer la porte et met à jour en conséquence la base de fait.

Questions :

Que se passerait-il si la base de faits contenait `(porte Ouverte)` ?

Combien de fois cette règle se serait elle exécutée si l'on n'avait pas supprimé le fait (porte ouverte).

Supposons qu'un établissement contienne plusieurs portes, il serait alors nécessaire de connaître l'identifiant de la porte à fermer. Il y aurait alors au moins deux informations : l'identifiant de la porte et son état. Ces informations sont représentées à l'aide du `deftemplate` suivant :

```
(deftemplate porte
```

[4] Cf. § 3.1.1 p.10.

```
        (slot id)
        (slot etat (default fermee) )
)
```

Pour éviter la création d'autant de règles que de portes, l'utilisation de variables s'impose. La nouvelle règle peut alors préciser la porte en cause et marquer son changement d'état d'*ouverte* à *fermée*. La nouvelle règle est la suivante :

```
(defrule fermeture-de-la-porte-2  "Informe qu'une porte x est ouverte"
        ?f <- (porte (etat ouverte) (id ?p) )
=>
        ; partie conclusion
        (printout t "Fermer la porte " ?p " SVP ." crlf)
        (modify ?f (etat fermee) )
)
```

Imaginons que le nombre important de porte nécessite de contacter un gardien précis. Il est alors nécessaire d'avoir des faits informant du nom d'un gardien et des portes dont il a la responsabilité. La base de faits pourrait par exemple contenir les informations suivantes :

```
(gardien paul p0)
(gardien paul p1)
(gardien paul p3)
(gardien pierre p2)
(gardien pierre p4)
(gardien pierre p5)
(porte (etat ouverte) (id p1) )
```

La nouvelle règle s'écrit alors :

```
(defrule fermeture-de-la-porte-3  "Informe qu'une porte est ouverte"
        ; partie condition
        ?f <- (porte (etat ouverte) (id ?p) )
        (gardien ?nom ?p)
=>
        ; partie conclusion
        (printout t "M. " ?nom ". Veuillez fermer la porte " ?p " SVP ." crlf)
        (modify ?f (etat fermee) )
)
```

La première condition donnera la valeur **p1** à la variable **p** (cf. base de fait ci-dessus). Une fois instanciée la variable p garde sa valeur et ne pourra en changer qu'au niveau de la partie conclusion de la règle. Ainsi la deuxième conclusion de la règle spécifie qu'il doit exister un fait respectant le pattern suivant :

```
(gardien <un-nom> p1)
```

Ainsi la variable **nom** aura pour valeur **paul** . Les deux conditions ayant été vérifiées la partie conclusion de la règle sera exécutée. Attention : les variables définies dans une règle ont leur portée limitée à cette règle. Ainsi, la transmission d'information d'une règle à l'autre ne peut se faire que par l'entremise de faits CLIPS[5].

L'exécution du système donne alors le résultat suivant :

```
CLIPS>(run)
M. paul. Veuillez fermer la porte p1 SVP
CLIPS>
```

Question :

Que se passerait-il si le fait suivant existait dans la base de faits : `(gardien gerard p1)`

Supposons que l'on sache que la porte p3 est condamnée et qu'il soit nécessaire d'ignorer les signaux la concernant. Il est possible de spécifier dans la partie condition les valeurs à ignorer à l'aide de l'opérateur logique not ' **~**'. La première condition s'écrirait alors :

[5] Bien qu'il soit en fait possible d'utiliser des variables globales (symbole `*?*nom-var`), vous vous limiterez à l'utilisation de faits.

```
?f <- (porte (etat ouverte) (id ?p&~p3) )
```

Ceci ce lit : trouver un fait se nommant `porte` dont le champ `etat` a la valeur ouverte, dont le champs `id` a une valeur quelconque **autre** que `p3`.

L'inconvénient de cette représentation est qu'il doit exister autant de faits `(gardien <nom> <porte>)` que de portes qu'un gardien a la charge. Une autre solution consisterait à utiliser des listes et des variables adaptées. Les faits se réécriront alors comme suivant :

```
(gardien paul p0 p1 p3)
(gardien pierre p2 p4 p5)
(porte (etat ouverte) (id p1) )
```

La base de fait s'en trouve simplifiée.

Question : Quel avantage pourrait en tirer le SE ?

Cependant, il devient nécessaire de modifier la première condition de la règle. Il ne s'agit plus de déterminer si un fait existe dans la base de fait, mais si la valeur recherchée (ici p1) est dans la liste de valeur d'un fait. Cela implique premièrement de disposer d'une variable adaptée c'est à dire des *multi-variables* dont la notation est la suivante. Les caractères "$?" suivis d'un symbole. Ex : `$?a`, `$?liste`, `$?nom-long1`. Deuxièmement, il est nécessaire de pouvoir effectuer une recherche de valeur dans une liste[6]. Nous utiliserons pour cela la fonction `member$` dont l'usage est le suivant :

```
(member$ <valeur-recherchée> <liste-de-valeurs> )
```

Cette fonction retourne la position de la valeur dans la liste (i.e. une valeur de 1 à n considérée comme TRUE) ou FALSE si la valeur n'est pas trouvée. L'appel d'une fonction dans la partie condition d'une règle n'est possible que dans un fait ou dans un bloc test qui doit retourner TRUE ou FALSE.

La nouvelle règle s'écrie alors :

```
(defrule fermeture-de-la-porte-4  "Informe qu'une porte est ouverte"
       ; partie condition
       ?f <- (porte (etat ouverte) (id ?p) )
       (gardien ?nom $?liste-portes)
       (test (member$ ?p $?liste-portes) )
=>
       ; partie conclusion
       (printout t "M. " ?nom ". Veuillez fermer la porte " ?p " SVP ." crlf)
       (modify ?f (etat fermee) )
)
```

Cette règle contient trois conditions : les deux premières n'ont pour but que de donner des valeurs aux variables `p`, `nom` et `liste-portes` et la troisième de comparer les valeurs de `p` et de `liste-portes`. La commande `test` permet de placer un test logique dans la partie condition d'une règle et ainsi de rajouter d'autres types de condition notamment l'appel de fonctions. Selon la valeur des paramètres de test, la règle est activée ou non[7]. Ainsi est-il possible d'effectuer les tests suivants :

```
(test (> ?x ?y) )   ; teste si ?x est plus grand que ?y
(test (and
              (> ?x ?y)     ;teste si ?x est plus grand que ?y
              (> ?x ?z)     ;et si ?x est plus grand que ?z
       )
)
(test (numberp ?x) )       ; test si ?x est un nombre
…
```

Notons que les deux dernières conditions de la règle `fermeture-de-la-porte-4` peuvent être contractée comme suivant :

[6] Cf. Aide CLIPS > Function Summary > MULTIFIELD_FUNCTIONS.
[7] **Attention :** si la partie fait n'est pas modifiée les conditions de la règle ne seront évaluée qu'une seule fois !

```
(gardien ?nom $?liste-porte &:(member$ ?p $?liste-portes))
```

Ceci se lit : la variable $?list-porte a une valeur **et** (symbole &) l'appel de la fonctions qui suit (symbole **:**) retourne TRUE (ie l'instruction `(member$ ?p $?liste-porte)` est vraie).

Le résultat de l'exécution de la règle précédente, compte tenu des faits présents dans la base de faits, est le suivant :

```
CLIPS>(run)
M. Paul. Veuillez fermer la porte p1 SVP
CLIPS>
```

3 Utilisation avancée de CLIPS

Cette section peut ne pas vous intéresser lors de votre EP. Cependant, il peut vous éclairer sur la raison de bugs dans votre programme ou tout simplement améliorer votre compréhension de CLIPS en particulier, et des systèmes experts en général.

3.1 Fonctionnement du moteur d'inférence

Afin de mieux comprendre le fonctionnement du moteur d'inférence de CLIPS, cette section présente d'abord le fonctionnement du moteur d'inférence puis décrit les stratégies de résolution disponibles dans CLIPS.

3.1.1 Cycle d'exécution d'une règle

L'agenda fonctionne comme un gestionnaire de pile dans lequel les jetons sont les règles actives. Le remplissage de cette pile nécessite une phase d'analyse de la base de faits et de la base de règles. Sans entrer dans les détails, précisons que CLIPS étant basé sur l'algorithme RETE [Forgy 85], cette analyse est dirigée par les faits. Une fois l'analyse effectuée les règles actives sont placées dans l'agenda selon le processus décrit dans la Figure 2.

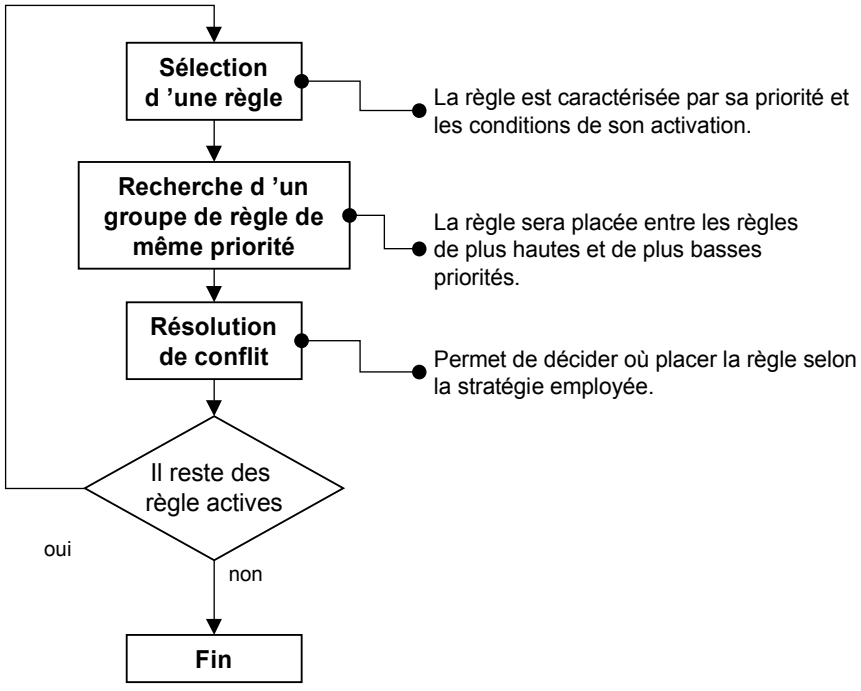

Figure 2 –Schéma simplifié du fonctionnement de l'agenda

Ce schéma révèle ainsi l'importance de l'utilisation des priorités lors du placement des règles dans l'agenda et par conséquence leur exécution ultérieure[8]. La résolution de conflit permet de décider comment la règle sera

[8] Par ailleurs cela souligne que l'existence de règles de priorités différentes rallonge le cycle. De plus, cette propriété n'est bien souvent qu'une astuce de programmation qui si elle rend service au programmeur, obscurci le code et rend plus difficile son débuggage. Son usage

placée vis à vis des autres règles de même priorité. La section 3.1.2 décrit plus précisément comment ce placement est effectué. Une fois rangées, les règles placées dans l'agenda sont exécutées après l'appel à la commande `(run)`. Le cycle d'exécution de ces règles est décrit dans la Figure 3 ci-dessous.

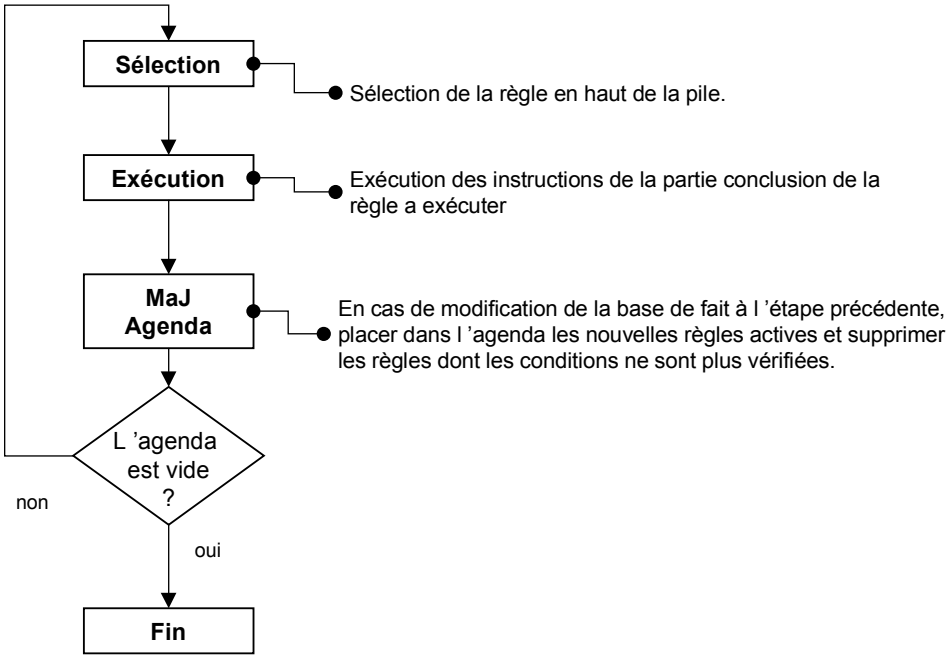

Figure 3 – Cycle d'exécution d'une règle

Notons que l'étape de mise à jour de l'agenda (MaJ) suit le même processus que celui décrit dans Figure 2 lors de l'ajout de règles dans l'agenda.

3.1.2 Stratégies existantes

Clips propose 7 stratégies de résolution. Vous n'aurez cependant pas, a priori, à changer de stratégie. Rappelons qu'une stratégie permet de décider comment placer une règle venant d'être activée dans l'agenda parmi les règles de même priorité (salience). Nous appliquerons ces stratégies sur les règles suivantes :

```
(defrule R1 (nbre 2)                    => (printout t " 2 !!") crlf) )
(defrule R2 (nbre 5)                    => (printout t " 5 !" crlf) )
(defrule R3 (nbre 8)                    => (printout t " 8 !" crlf) )
(defrule R4 (nbre 2) (nbre 5)           => (printout t " 2 et 5 !!" crlf) )
(defrule R5 (nbre 2) (nbre 5) (nbre 8)  => (printout t " 2, 5 et 8!!" crlf) )
```

Stratégie en profondeur (depth)

Principe : *Les règles récemment activées sont placées **avant** les autres règles de même priorité.*

Cette notion de récence est déterminée en fonction de la position des faits constituant la condition des règles en concurrences comme le montre l'exemple suivant :

Base de fait	*Agenda*
f-1 (nbre 2) f-2 (nbre 5) f-3 (nbre 8)	0 R3: f-3 0 R5: f-1,f-2,f-3 0 R2: f-2

doit donc être restreint au minimum.

	0 R4: f-1,f-2 0 R1: f-1

L'ordre des faits va favoriser les règles utilisant le fait `(nbre 8)` sont ainsi favorisées par rapport à celles utilisant `(nbre 5)` et encore `(nbre 2)`. Par contre l'ordre de règles utilisant les mêmes faits est arbitraire (ex : règles R3 et R5 ou encore R2 et R4.

Ceci conduit CLIPS à fonctionner comme s'il explorait une idée à la fois. Ainsi, il explorera jusqu'au bout les conséquence d'une règle. On se trouve dans une gestion de pile en LIFO (Last In First Out).

Stratégie en largeur (Breadth) :

Principe : *Les règles récemment activées sont placées* ***après*** *les autres règles de même priorité.*

Cette stratégie est symétrique par rapport à la précédente : les règles récentes sont placées après les autres soit une gestion de pile en LILO (Last In Last Out).

Base de fait	*Agenda*
f-1 (nbre 2) f-2 (nbre 5) f-3 (nbre 8)	0 R1: f-1 0 R4: f-1,f-2 0 R2: f-2 0 R5: f-1,f-2,f-3 0 R3: f-3

Les règles ayant été activée grâce à `(toto)` sont défavorisées. Ainsi la règle `toto` passe **après** `tata` et il en est de même pour la règle `toto-et-tata`. La remarque portant sur la position relative de `toto-et-tata` et cependant toujours vrai.

Ceci conduit CLIPS à fonctionner comme s'il explorait plusieurs idées à la fois. Ainsi, il explorera en parallèle les conséquences de plusieurs règles.

Stratégie par simplicité (Simplicity) :

Principe : *Les règles activées sont placées par ordre* ***croissant*** *de* ***spécificité.***
La notion de spécificité permet de mesurer la complexité d'une règle. Celle ci utilise le nombre de comparaison effectuée dans la règle (ex comparaison d'une valeur dans un fait avec celle des faits existants, appel à un fonction avec test, etc…).

Base de fait	*Agenda*
f-1 (nbre 2) f-2 (nbre 5) f-3 (nbre 8)	0 R1: f-1 0 R2: f-2 0 R3: f-3 0 R4: f-1,f-2 0 R5: f-1,f-2,f-3

Stratégie par complexité (Complexity) :

Principe : *Les règles activées sont placées par ordre* ***décroissant*** *de* ***spécificité.***

Cette stratégie fonctionne de manière symétrique par rapport à la stratégie précédente.

Base de fait	*Agenda*
f-1 (nbre 2) f-2 (nbre 5) f-3 (nbre 8)	0 R5: f-1,f-2,f-3 0 R4: f-1,f-2 0 R1: f-1

	0 R2: f-2 0 R3: f-3

Cependant, ne vous fiez pas à l'ordre relatif entre les règles de même complexité (i.e. les règles f-1, f-2, f-3).

Stratégie LEX (LEX) :

La stratégie LEX procède à un classement lexicographique des règles actives en utilisant, à la place de lettre, un mot formé des dates des conditions formant les règles. Ainsi, dans l'exemple ci-dessous, en considérant la position du fait comme une date, le mot correspondant à la règle R5 est 3-2-1 et 2-1 pour la règle R4. La comparaison est faite date par date et classe par ordre décroissant les règles. En cas d'égalité des conditions comparées, la règle présentant le plus de conditions est placée avant l'autre (ainsi R5 est placée avant R3, et R4 avant R2). Si les deux règles ont le même nombre de conditions, leur placement est départagé par leur complexité.

On obtient alors le résultat suivant :

Base de fait	*Agenda*
f-1 (nbre 2) f-2 (nbre 5) f-3 (nbre 8)	0 R5: f-1,f-2,f-3 0 R3: f-3 0 R4: f-1,f-2 0 R2: f-2 0 R1: f-1

On peut vérifier le comportement de l'algorithme en considérant le fact-id comme une date. On obtient alors les mots suivants :

```
321  3  21  2  1
```

Stratégie MEA (MEA) :

La stratégie MEA est également une combinaison de stratégies. Elle combine la stratégie profondeur avec la stratégie LEX. Les règles sont évaluées tout d'abord en comparant leur date d'activation, en cas d'égalité la stratégie LEX est appliquée. Ainsi cette stratégie est plus complète que la stratégie profondeur ou aucun placement est arbitraire.

On obtient alors le résultat suivant :

Base de fait	*Agenda*
f-1 (nbre 2) f-2 (nbre 5) f-3 (nbre 8)	0 R3: f-3 0 R2: f-2 0 R5: f-1,f-2,f-3 0 R4: f-1,f-2 0 R1: f-1

Stratégie aléatoire (random) :

Lors de leur activation, chaque règle se voit attribuée aléatoirement un nombre utilisé ensuite pour leur placement dans l'agenda.

3.2 Piège de programmation en CLIPS

La programmation en CLIPS, comme avec tout autre langage, n'est pas exempt de pièges. Vous trouverez ici quelques exemples de ces pièges. N'hésitez pas à contribuer à cette section, en proposant des pièges que vous avez vous-même rencontrés.

Piège des stratégies :

La stratégie d'exploration en profondeur `depth`, consiste à favoriser les faits les plus récents. Ceci peut conduire, dans certains cas, à une boucle infinie, c'est à dire l'exécution d'une même règle indéfiniment. La règle suivante en est un exemple :

```
(defrule boucle-infinie "la règle passe devant les autres"
       ?f <- (cycle ?n)
=>
       (retract ?f)
       (assert (cycle (+ ?n 1) ) ) ; réactive la règle au prochain cycle
)
```

Ici, l'insertion du nouveau fait effectuée dans la partie conclusion de la règle réactive la règle. Or, la stratégie `depth` favorisant les faits récents, la règle `boucle-infinie` sera placée en tête de l'agenda comme étant la prochaine règle à exécuter. Le processus se reproduit ainsi indéfiniment. Un tel bug est détectable en utilisant la fenêtre Agenda de CLIPS et en observant l'évolution de l'agenda, notamment en notant que certaines règles ne sont jamais exécutées.

Piège des faits structurés :

La modification d'un fait structuré conduit à changer son fact-id et le marque donc comme un nouveau fait. Ceci traduit la modification des informations contenues dans ce fait et active éventuellement les règles utilisant ce fait. Cette propriété a un effet pervers comme dans la règle ci-dessous :

```
(defrule perpetuelle
       ?f <- (liste (nom etudiant) (valeur $?liste-etudiants) )
       (etudiant ?nom)
=>
       (modify ?f (valeur (create$ ?nom $?liste-etudiants)) )
)
```

A moins qu'une règle ne supprime le fait `liste` ou tous les faits `etudiant`, cette règle sera toujours active (voire la seule exécutée avec la stratégie `depth`). Pour éviter ce piège, il faut supprimer le fait `etudiant` chaque fois qu'il est enregistré. Cette solution étant quelque peu drastique, il est également possible d'éviter qu'un étudiant soit enregistré plus d'une fois. Ceci peut être effectué en rajoutant une troisième condition :

```
(test (not (member$ ?nom $ ?liste-etudiants) ) )
```

La règle ne sera alors pas activée deux fois pour un même étudiant. Pour détecter un tel "bug" il suffit d'étudier l'évolution des valeurs enregistrées ainsi que de la l'exécution récurrente anormale d'une règle. Pour information voici le résultat après 7 exécutions de CLIPS, selon les deux stratégies `depth` et `breadth`, avec la règle `perpetuelle` et les faits suivants : `(etudiant albert) (etudiant gerard) (etudiant marcel).`

Avec la stratégie `depth` :

```
(liste (nom etudiant) (valeur marcel marcel marcel marcel marcel marcel marcel) )
```

Avec la stratégie `breadth` :

```
(liste (nom etudiant) (valeur albert albert albert albert albert albert albert) )
```

4 Manuel d'utilisation de CLIPS V6.10

Avant de décrire en détail l'environnement CLIPS, l'éditeur intégré de CLIPS est présenté. En effet, l'interface de saisie de CLIPS étant rudimentaire, l'utilisation de cet éditeur vous sauvera beaucoup de temps et de sueur.

4.1 Utilisation de l'éditeur intégré

L'éditeur est un programme exécutable nommé `clipsedt.exe` accessible dans le répertoire de CLIPS mais également à partir de l'environnement CLIPS dans le menu `Files` commande `Editor` (cf. Figure 4).

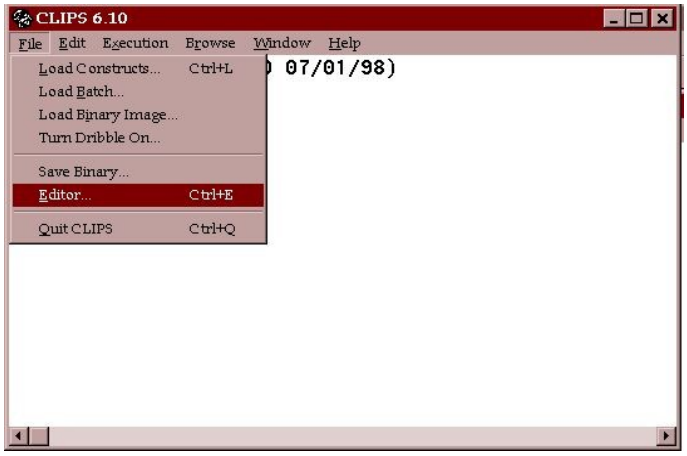

Figure 4 – Lancement de l'éditeur depuis CLIPS

Cet éditeur présente l'avantage, par rapport au Bloc-note de Windows (programme `Notepad`), de proposer deux fonctions très utiles pour la programmation en CLIPS. Tout d'abord une fonction de **complétion** des mots clés CLIPS accessible avec la combinaison des touches **Contrôle et J** (notée `Ctrl+j` ou `^j`). Dans l'exemple de la Figure 5 le mot `defr` peut être complété par deux propositions. Cette fonction peut être utile dans le cas où l'on a oublié l'orthographe d'une fonction CLIPS sans avoir recours a l'aide en ligne. Notez que cette commande est également disponible dans CLIPS.

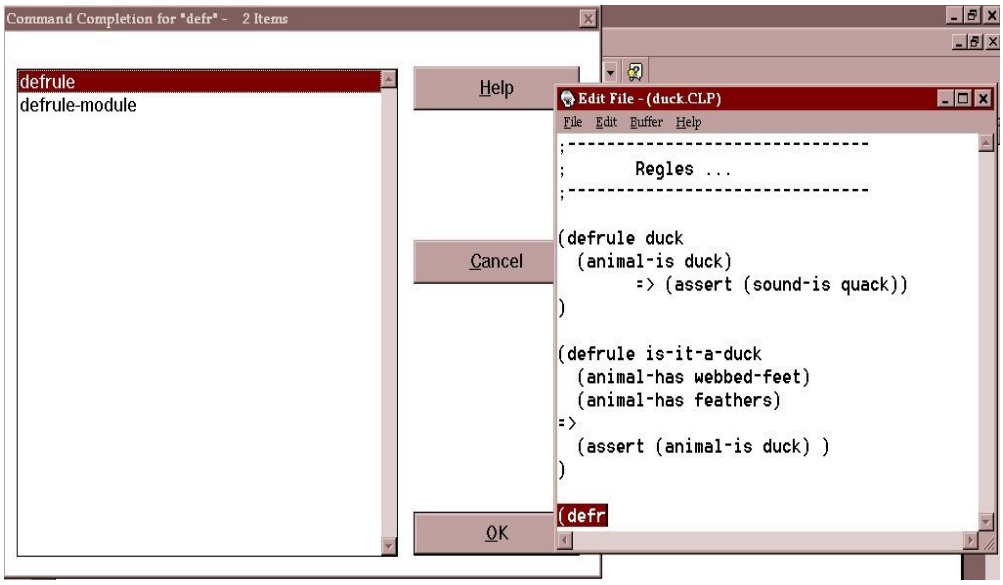

Figure 5 – Utilisation de la commande **Ctrl+j**

Une autre commande forte utile du fait de l'abondance de parenthèses dans CLIPS est la commande **Balance**[9] accessible par le raccourci `Ctrl+b`. Cette commande permet de vérifier que chaque parenthèse ouverte est fermée. Elle permet ainsi d'éviter les erreurs habituelles dans CLIPS voire d'autres langages, provenant de parenthèses manquantes. L'appel de cette commande sélectionne la zone de texte comprise entre deux parenthèses ouvrante et fermante. Un nouvel appel remonte d'un niveau de parenthèse et permet ainsi de vérifier, par exemple que les parenthèses ouvertes dans la définition d'une règle sont toutes fermées. La Figure 6, ci-dessous, donne un exemple obtenu après deux appels de la commande Balance.

[9] Balance signifie équilibre en anglais.

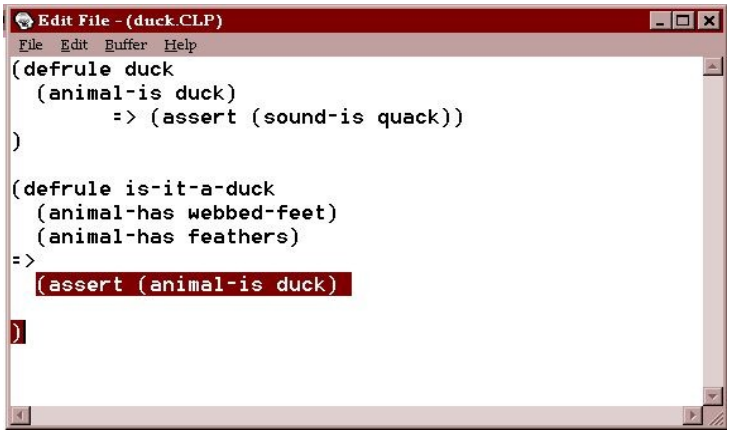

Figure 6 – Utilisation de la commande Balance

Dans cet exemple, le second appel de la commande montre qu'il manque une parenthèse pour fermer la règle `is-it-a-duck`. En regardant mieux, il apparaît que la parenthèse fermante de la commande `assert` manque.

4.2 Chargement et sauvegarde de fichier avec CLIPS.

Une fois l'écriture du code terminée vous pouvez sauvegarder votre fichier (l'extension par défaut étant CLP) et l'ouvrir par la suite dans l'environnement CLIPS avec la commande `Files>Load Constructs`. La commande `Files>Load Batch` est utilisée pour charger un fichier batch contenant une liste de commande CLIPS à exécuter. Ce type de fichier peut être utile pour centraliser le chargement de différents fichiers CLIPS. La commande `File>Load Binary` est symétrique de la commande `File>Save Binary`. Cette dernière sauvegarde **les règles** CLIPS dans un format binaire accélérant ainsi leur chargement[10]. La sauvegarde des **faits** est effectuée par la commande CLIPS `(save-facts` *`nom-fichier`*`)` et le chargement de ces faits par `(load-facts` *`nom-fichier`*`)`. Ceci illustre bien la distinction faite par CLIPS entre le raisonnement (i.e. les règles) et les données (i.e. les faits) qui soutiennent ce raisonnement.

En phase de développement, où le code est souvent modifié vous aurez intérêt à travailler en mode texte (i.e. la commande `load-constructs`). En phase de déploiement, vous privilégierez les commandes binaires.

4.3 Fonctions de debuggage

CLIPS propose un ensemble d'outils aidant au développement d'un programme. Ces outils seront présentés par ordre de menu dans lesquels ils ont accessibles.

4.3.1 Menu File

L'item `Turn Dribble On` du menu `File` permet de garder une trace des affichages à l'écran et des saisies au clavier. Un click sur cet item déclenche l'ouverture d'une boîte de dialogue permettant de choisir un nom de fichier texte dans lequel toutes les informations seront recopiées.

Une fois activé, cet item est renommé `Turn Dribble Off`. Pour arrêter la trace (mais pas l'exécution des règles) cliquez à nouveau sur cet item.

Ces fonctions sont accessible depuis le prompt avec les commandes `(dribble-on <nom-fichier>)` et `(dribble-off)`.

4.3.2 Menu Execution

Le menu `Execution` permet de gérer l'exécution des règles actives. Il permet de réinitialiser le système (commande `reset`, raccourci `Ctrl+U`) et d'exécuter toutes les règles présentes dans l'agenda (commande `Run`, raccourci `Ctrl+R`). Les autres commandes sont présentées avec plus de détail.

[10] La commande associée en CLIPS est `(bsave` *`nom-fichier`*`)`.

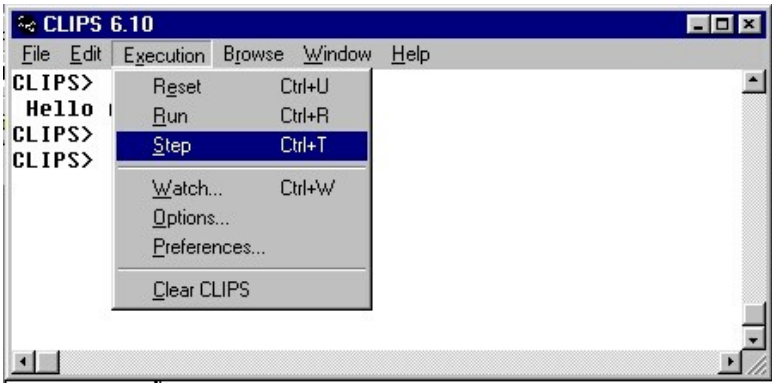

Figure 7 – Menu Execution

Item Step : Exécution pas à pas

La commande `step` accessible par le raccourci `Ctrl+T` est équivalente à la commande `(run 1)`. Elle déclenche la première règle active de l'agenda. Vous pouvez également indiquer le nombre de règles à déclencher comme suivant : `(run 4)` qui déclenche 4 règles et redonne accès au prompt CLIPS pour par exemple examiner le nouvel état de la base de faits ou de l'agenda.

Item Watch

Cette commande est accessible par le menu `Execution>Watch` et permet de surveiller des changements dans l'état du système expert en développement. Différents paramètres peuvent être surveillés :

`Compilation :` Affiche le progrès des définitions des constructs (ie règles, faits structurés, etc…). En pratique, cette option ne semble pas fonctionner.

`Facts :` Surveille l'insertion (symbole ==>) et la suppression de faits (symbole <==).

`Rules :` Surveille le déclenchement des règles (i.e. exécution de la partie conclusion de la règle et suppression de la règle de l'agenda) en affichant : `CLIPS> FIRE nombre`[11] `nom-de-la-règle : fait(s) déclenchant(s).`

`Statistics :` Après un appel à `(run)` indique une liste d'informations détaillées sur l'état du système (nombre de règles déclenchées, temps d'exécution, nombre de règles moyenne exécutée par seconde, nombre moyen de faits, etc…).

`Activations :` Surveille l'activation des règles i.e. vérification de toutes les conditions de la partie prémisse de la règle (symbole ==>) et la désactivation des règles (symbole <==) i.e. condition de déclenchement invalidée (ex : suppression d'un fait). Affiche : `CLIPS> Activation nombre nom-de-la-règle : fait(s) déclenchant.`

`All :` Surveille tous les paramètres possibles. L'utilisation de ce paramètre peut conduire à un affichage illisible.

[11] Valeur de la priorité de la règle (*salience*). cf. § 2.3

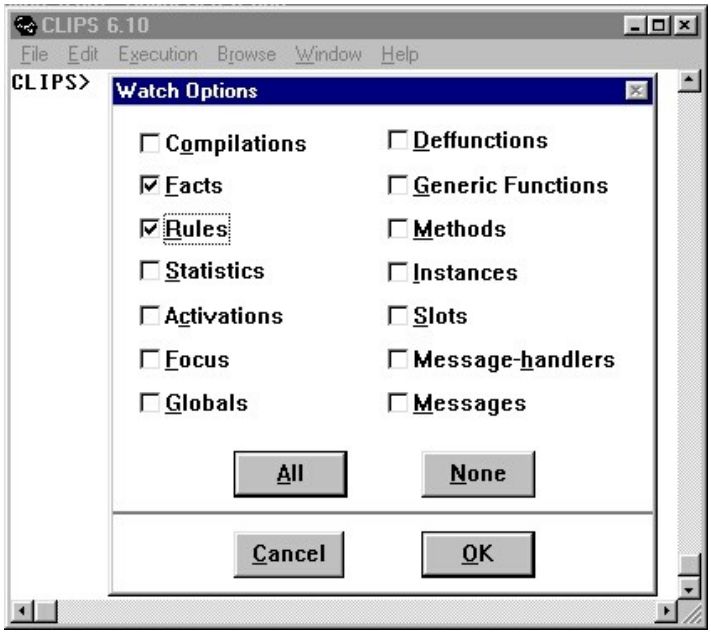

Figure 8 – Sous menu Watch

Les autres paramètres, cités ci-dessous, ne vous seront pas utiles à ce niveau d'expérimentation de CLIPS. Pour information, ces paramètres concernent la gestion :

- de groupes de règles indépendants ou defmodule : `Focus`
- de variables globales : `Globals`
- de fonctions définies par l'utilisateur (programmation procédurale) : `Deffunctions`, `Generic Function`
- de COOL, la partie objet de CLIPS : `Methods`, `Instances`, `Slots`, `Message_handlers`, `Messages`.

Cette boîte de dialogue transmet les options sélectionnés à la commande CLIPS `(watch [paramètre]*)`. Une fois activée, une option peut être désactivée par la commande `unwatch` qui fonctionne de la même manière.

Item Options

Le menu option permet de configurer le fonctionnement du moteur d'inférence d'une part et de l'agenda d'autre part.

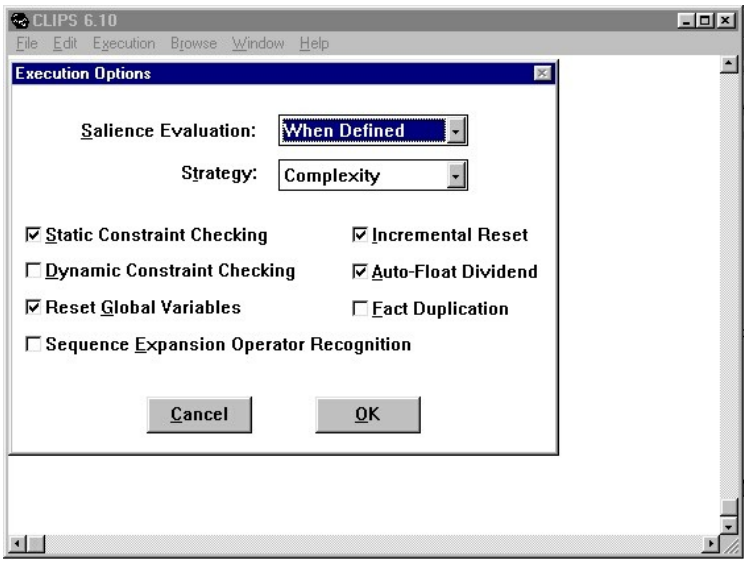

Figure 9 – Menu Option

Concernant le moteur d'inférence

Trois options essentielles peuvent vous intéresser : `Static Constraint Checking`, `Dynamic Constraint Checking` et `Fact Duplication`.

Les deux premières options concernent la vérification, dans les faits structurés, du respect du type de donné requis (cf. § 5.2). L'option `Static Constraint Checking` assure, lors de la compilation des règles (ou parsing), que les données manipulées sont correctes. Dans l'exemple suivant, le champ `id` étant défini comme étant de type `STRING`, l'addition effectuée dans la partie conclusion de la règle `regle-test` viole cette contrainte (il n'est pas possible d'ajouter des String entre eux[12]). Comme le montre la trace, ceci déclenche une erreur.

```
CLIPS>(deftemplate essai (slot id (type STRING) ) )
CLIPS>(defrule regle-test
?f <- (essai (id ?id) )
=>
(modify ?f (id (+ ?id 10) ) ) ; déclenche une erreur
)
[RULECSTR3] Previous variable bindings of ?id caused the type restrictions
for argument #1 of the expression (+ ?id 10)
found in the rule's RHS to be violated.

ERROR:
(defrule MAIN::regle-test
   ?f <- (essai (id ?id))
   =>
   (modify ?f (id (+ ?id 10))))
CLIPS>
```

L'option `Dynamic Constraint Checking` assure la vérification dynamique de ce type d'erreur. Cette vérification s'avère nécessaire lorsque l'on ne connaît pas le type de la variable manipulé à l'avance (ex. saisie au clavier d'une valeur : quatre et 4 sont deux types de données différents !)

Concernant l'agenda

Vous n'aurez éventuellement à modifier que le paramètre `Strategy`. Ce dernier vous donne le choix entre 7 stratégies : `Breadth`, `Complexity`, `Depth`, `LEX`, `MEA`, `Random`, `Simplicity` (voir section 3.1 pour détail).

4.3.3 Menu Browse

A travers le menu `Browse` (cf. Figure 10), il est possible de gérer (affichage, suppression, options, etc) les constructs CLIPS (règle, faits structurés, deffacts, etc…).

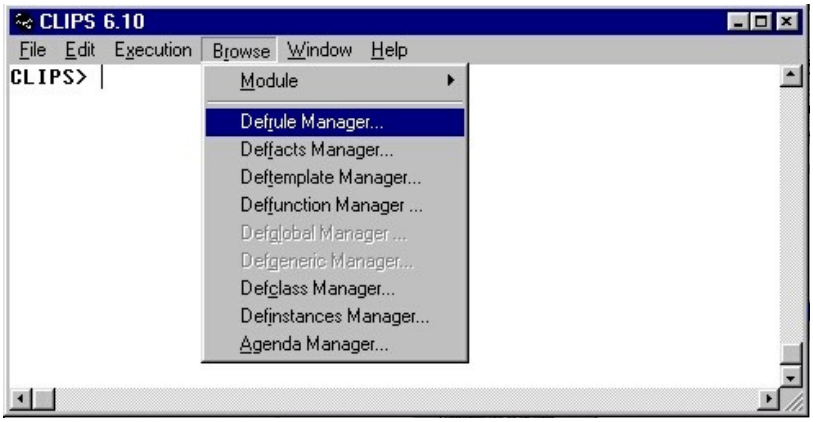

Figure 10 – Menu de gestion des constructs CLIPS

[12] Pour information, la concaténation de chaîne est effectuée avec la fonction `str-cat`

Les fonctionnalités les plus utiles seront : `Defrule Manager`, `Deftemplate Manager` et éventuellement `Agenda Manager` et `Deffunction Manager` (en cas de programmation procédurale).

- item `Defrule Manager :`

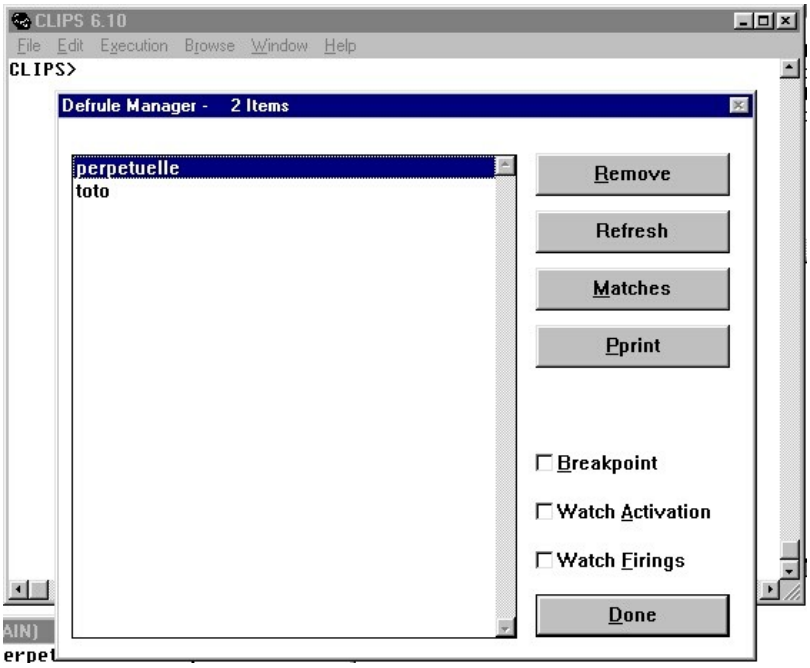

Figure 11 – Menu Defrule Manager

Comme on peut le voir dans la Figure 11, ce menu permet de :

1. supprimer une règle (bouton `Remove`),
2. place toutes les activations d'une règle dans l'agenda (bouton `Refresh`),
3. afficher les faits à l'origine de l'activation d'une règle (bouton `Matches`),
4. afficher "proprement" la définition d'une règle (bouton `Pprint`[13]),
5. Marquer la règle comme un point d'arrêt (case à cocher `Breakpoint`) - détails suivent - ,
6. Surveiller les activations d'une règle (case à cocher `Watch Activations`),
7. Surveiller les exécutions d'une règle (case à cocher `Watch Firings`).

Le 5ème point de cette liste porte sur le débuggage d'un programme par l'insertion de points d'arrêt. A l'instar de ce que l'on trouve dans d'autres logiciels de programmation procédurale, cette option permet de marquer les règles sur lesquels l'exécution doit s'arrêter. Lorsqu'une règle est marquée comme un point d'arrêt, CLIPS exécute toutes les règles de l'agenda jusqu'au moment d'exécuter cette règle et redonne accès au prompt. Ceci permet ainsi de mieux étudier le comportement de cette règle ainsi que l'état de la base de fait **avant** son exécution[14].

La commande CLIPS pour effectuer cette opération est : `(set-break <nom-de-la-règle>)`. Vous pouvez alors lister les points d'arrêts avec la commande `(show-break)`. Enfin, vous pouvez supprimer un point d'arrêt avec la commande `(remove-break <nom-de-la-regle>)`. L'appel à cet commande sans paramètre conduit à supprimer tous les points d'arrêt.

- item `Deftemplate Manager :`

Ce menu permet d'afficher les définitions des faits structurés avec bouton `Pprint` (les faits ordonnés sont également listé mais non affichable). Il permet également de marquer la surveillance de l'insertion et de la suppression de faits (case à cocher `Watch`).

[13] Pprint est le nom de la commande. Elle est le raccourci de "`pretty-print`"
[14] Voire après son exécution avec la commande `(run 1)` ou le raccourci `Ctrl+T`.

- item `Agenda Manager :`

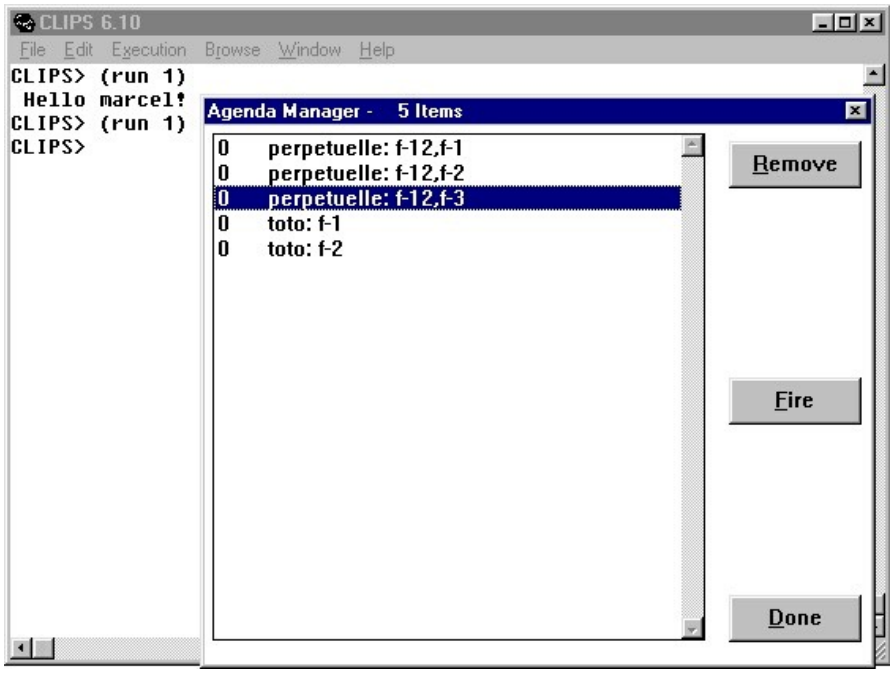

Figure 12 – Menu Agenda Manager

Ce menu permet de gérer manuellement l'agenda. Vous pouvez ainsi supprimer ou exécuter vous-même les activations placées dans l'agenda.

- item `Deffunction Manager :`

Ce menu permet d'afficher la définition des fonctions définies par l'utilisateur (`deffunction`) ou de les supprimer individuellement.

5 Annexes

5.1 Notation BNF

La grammaire d'un langage de programmation est très souvent présentée sous la forme dite grammaire BNF (acronyme de Backus-Naur Form). La légende des notations utilisées dans ce document est indiquée ci-dessous.

< >	les chevrons entourent les métavariables ;
::=	signifie "doit être obtenu par" ;
|	signifie "ou" (réunion au sens des ensembles) ;
[ ]	les crochets encadrent les parties optionnelles ;
( )	les parenthèses entourent les blocs ;
*	signifie "à répéter une ou plusieurs fois" ;
+	signifie "à répéter zéro ou plusieurs fois" ;
…	signifie "etc".

Figure 13 – Symboles d'une grammaire BNF

5.2 Définition d'un fait structuré

La définition complète d'un fait structuré est la suivante :

```
(deftemplate nom-du-fait-structuré  "commentaires explicatifs"
       (<slot|multislot> nom-champ1
              [(default <valeur-par-défaut>)]
              [(type <SYMBOL|STRING|LEXEME|NUMBER|INTEGER|FLOAT>)]
              [(allowed-symbol <valeur-autorisée>*)]
              [(range <min> <max>)]
       …
)
```

Il est ainsi possible de préciser :

La valeur par défaut avec **`default`** : ceci assure le remplissage des champs non remplis. Par défaut, la valeur utilisée est `nil`.

Le type de donné requis avec **`type`** : ceci peut s'avérer nécessaire selon l'utilisation des champs. Par exemple un champ `age` requiert de saisir un nombre. Encore faut-il préciser qu'il ne doit être saisis que sous forme numérique (4 plutôt que quatre). Ce typage suppose que la vérification des contraintes est activée (cf. § 4.3.2, sous-menu Options).

Les types autorisés sont :

`SYMBOL` : un mot tel que défini en §2.1.
`STRING` : une chaîne de caractère.
`LEXEME` : i.e. un `SYMBOL` ou un `STRING`.
`INTEGER` : un nombre entier.
`FLOAT` : un nombre à virgule.
`NUMBER` : un `INTEGER` ou un `FLOAT`.

En cas de violation de type le parser rejette la règle en cause et affiche à l'écran l'origine de l'erreur.

Exemple :

```
(deftemplate personne
       (slot nom (type SYMBOL) )
       (slot age (type INTEGER)
       (slot adresse (type STRING) )
)
```

Les valeurs autorisées avec **`allowed-symbols`** : lorsque les valeurs requises sont restreintes à un petit ensemble de valeur. Ex : `(slot sexe (allowed-values masculin feminin) )`

Des bornes de valeurs avec **`range`** : concernant des valeurs numériques, il est possible de définir des bornes de valeurs limites autorisées avec les valeurs `<min>` `<max>`. Ces deux valeurs peuvent être des entiers ou le mot **`?VARIABLE`**. Placé à la place de `<min>`, `?VARIABLE` signifie $-\infty$ et $+\infty$ s'il est placé à la place de `<max>`.

Exemple :

```
(deftemplate personne
       (slot age (range 0 ?VARIABLE) )  ; age superieur ou egal a 0
       (slot taille (range 0 300) )     ; taille entre 0 et 300 (cm)
       (slot bidon (range?VARIABLE 10) ) ; bidon inferieur ou egal a 10
)
```

Ces contraintes peuvent être combinées.

5.3 Saisie d'informations au clavier avec CLIPS

La saisie d'informations en CLIPS en dehors de l'insertion de faits est effectuée à l'aide des commandes `read` et `readline`.

La commande read :

La commande `read` lit un mot au clavier et le retourne. Cette valeur de retour peut être soit récupérée dans une variable avec la commande `bind`, soit directement utilisée pour insertion dans un fait.

Exemple 1 :

```
CLIPS>(assert (etudiant (read) (read) ) )
   Albert
Dupont caracteres-ignores
<Fact-0>
CLIPS>(facts)
f-0    (etudiant Albert Dupont)
CLIPS>
```

Le texte saisi au clavier est marqué en gras. Remarquez que les espaces précédents Albert ont été ignorés par `(read)` de même que le mot suivant Dupont.

Exemple 2 :

```
CLIPS>(defrule test-read
(initial-fact)
=>
(printout t "Quel est ton nom ? ")
(bind ?nom (read) )
(printout t "Quel est ton age ? ")
(bind ?age (read) )
(assert (personne ?nom ?age) )
)
CLIPS>(reset)
CLIPS>(run)
Quel est ton nom ? Albert
Quel est ton age ? 12
CLIPS>(facts)
f-0    (initial-fact)
f-1    (personne Albert 12)
CLIPS>
```

Notez que dans cet exemple, il n'y a pas de vérification sur ce qui est saisi. L'utilisateur aurait pu répondre à la deuxième question aussi bien `douze` que `bicyclette`.

Les valeurs saisies par l'utilisateur avec (read) sont converties dans les types CLIPS. Un chiffre sera converti en INTEGER ou FLOAT, les mots en atomes, les chaînes en string (ex : l'utilisateur a taper : **"hello world"**).
Ces informations sont donc directement utilisables comme toute valeur inscrite dans un fait.

La commande readline

Cette commande fonctionne de la même manière que la commande `(read)` à ceci prés qu'elle lit une chaîne de caractère. Le résultat est donc toujours récupérable dans un variable simple. Pour information, sachez qu'il est cependant possible de récupérer individuellement les mots contenus dans le chaîne à l'aide de la commande `explode$`. La combinaison de ces deux commandes permet alors d'éviter à recourir à plusieurs appels à `(read)`.

Vous n'aurez, a priori, pas besoin d'utiliser cette commande.

5.4 Table des figures