Introduction à CLIPS : 1- Faits et règles

(C Language Integrated Production System)

Bernard ESPINASSE
Professeur à l'Université d'Aix-Marseille

Plan

- **Introduction : historique, applications, accés et documentation**
- **Représentation des connaissances dans CLIPS**
- **Les constructs de Clips**
- **Les faits ordonnés (ordered facts)**
- **Les faits structurés ou non-ordonnés (template facts/non-ordered facts)**
- **Les Règles dans Clips**
- **Variables globales et variables locales**
- **Déclenchement d'une règle**
- **Les règles et leurs propriétés : salience et auto-focus**
- **Cycle de base de l'interpréteur CLIPS**
- **Autres commandes de CLIPS**
- **Traces d'exécution avec CLIPS: la commande watch**

Historique de CLIPS

- **Système général** permettant la réalisation de **systèmes experts**
- **Acronyme : C Language Integrated Production System**

- Développé en **1984** au centre **Johnson Space Center de la NASA**
- **Objectifs :**
  - beaucoup de systèmes experts développés en **LISP**: mauvaise portabilité, coût élevé, faible intégration avec d'autres environnements, ...

  -> développement d'un **moteur d'inférence en langage C : CLIPS**
- **Evolutions de CLIPS :**
  - 1984 : version 1.0 : règles de production et algorithme RETE

  .....
  - 1991 : version 5.0: + programmations procédurales et orientée objet
  - 1995 : version 6.0: + pattern-maching d'instance de classes
    + modularité, intégrité
    + deftemplates avec slots multifield
    + "not" dans les conditions
    + environnement convivial (Windows, Mac, Unix)

Applications développées avec CLIPS

- **plus de 4000 utilisateurs de CLIPS dans le monde**
- **nombreuses applications** (cf. document "CLIPS Applications Abstract") **:**
  - **domaine concernés :**
    - ingénierie : aéronautique, nucléaire, chimique ...
    - éducation
    - médical
    - financier
    - ...

  - **tâches :**
    - diagnostic
    - analyse
    - planification
    - optimisation
    - aide à la conception
    - ...

Accés et documentation de CLIPS

- **Acquerir CLIPS (gratuit) : http://www.ghgcorp.com/clips/**
- **Documentation CLIPS : http://www.ghgcorp.com/clips/**
  - manuel de référence
  - tutorial complet
- **Livre sur CLIPS :**

  "**Expert Systems: Principles and Programming**," 3ième édition, Giarratano and Riley (ISBN 0-534-95053-1 - $75.95) vient avec un CD-ROM contenant les exécutables de CLIPS 6.05, le code code, et la documentation. La première moitié du livre est consacrée à la théorie et la seconde à la programmation par règles en clips.

  Contact : International Thompson Publishing :
  7625 Empire Dr.
  Florence, KY 41042
  Phone: (800) 354-9706
  Phone: (606) 525-2230
  WWW: http://www.thomson.com/
- **Tutoriaux sur CLIPS :**
  - concis : http://www.enc.hull.ac.uk/EDM/material/57353/tutor01.htm
  - complet : http://kiri.carleton.ca/Courses/Grad/1994-95/82.562/clipsdoc/usrguide-Contents.html

Travailler en CLIPS

- **Pour lancer CLIPS** double-cliquez sur l'icône CLIPS
  - vous obtenez alors la commande prompt **`CLIPS>`**
  - vous pouvez alors lancer vos commandes et vos programmes
- **Pour sortir de CLIPS**, tapez **`(exit)`** ou sortez de CLIPS

Commandes de base de CLIPS

- toujours encadrées par des parenthèses : `(assert (couleur rouge))`
- commandes principales :

`(exit)`	Sortir de CLIPS
`(clear)`	Efface tout programme et données de la mémoire. Equivalent à sortir et relancer CLIPS.
`(reset)`	Efface l'information dynamique de la mémoire et initialise l'agenda.
`(run)`	Lance l'exécution d'un programme CLIPS.

- ces commandes peuvent aussi être exécutées à partir de la barre de menu de CLIPS

Représentation des connaissances dans CLIPS

Les connaissances factuelles:

connaissances que le système a acquis sur le problème qu'il est en train de résoudre :

- **faits**,
- **instances d'objets**
- **variables globales** dans Clips

Les connaissances opératoires:

expertise, savoir-faire exprimé de façon déclarative sur le domaine abordé (quelles actions accomplir dans telle situation, ...) : **règles** dans Clips

Représentation des connaissances dans CLIPS:

logique des prédicats du premier ordre

l'emploi de variables :

variables = entités quantifiées universellement et existentiellement et substituables par des faits de la base de faits.

Ajout et suppression de faits dans CLIPS

Fait = donnée constituant une unité de connaissance utilisée dans les règles :

`(couleur verte)` ou **`(pere_de Pierre Paul)`**

- un fait est spécifié par :
  - son **fact-index** remis à 0 avec la commande **reset** et **clear**
  - son **fact-address** accessible par des commandes spécifiques
- les faits sont **ajoutés** par la commande **`assert`** :

```
CLIPS>(assert (couleur verte))
  <Fact-0>
```

  `<Fact-0>` est la réponse de CLIPS disant qu'il y a un nouveau fait (fact number 0) qui a été placé dans la base de faits.

- la commande **`facts`** **liste** tous les faits de la base de faits :

```
CLIPS>(facts)
  f-0  (couleur verte)
  For a total of 1 fact.
```

- les faits sont **supprimés** par la commande `retract` :

```
CLIPS>(retract (couleur verte))
```

Les constructs de Clips

constructs = une déclaration permettant la construction d'entités :

nom du construct	fonction du construct
defmodule	partitionnement de la base de connaissance
defrule	déclaration d'une règle
deffacts	déclaration de faits initiaux
deftemplate	déclaration d'un fait structuré
defglobal	déclaration de variables globales
deffunction	déclaration d'une fonction utilisateur
defclass	déclaration d'une classe d'objet
definstances	déclaration d'une instance d'objet
defmessage-handler	déclaration d'un type de message
defgeneric	déclaration d'une fonction générique
defmethod	déclaration d'une méthode de classe

Les types de données primitifs de Clips

• CLIPS propose **8 types de données primitifs** pour représenter de l'information :

- **information numérique :**
  - float,
  - integer,
- **information symbolique :**
  - symbol,
  - string,
  - instance-name,
- **information interne :**
  - external-address,
  - fact-address,
  - instance-address

Les fonctions de Clips

fonction = bout de code exécutable identifié par un nom qui **retourne** une **valeur** ou **réalise** un **effet de bord** (affichage)

• **plusieurs types de fonctions**:
  - **fonctions définies dans le système** (system defined functions)
  - **fonctions définies par l'utilisateur** (user defined functions) par le construct **deffunction**
  - **fonctions génériques** permettent l'exécution de divers bouts de code selon les arguments passés : constructs **defgeneric** et **defmethod**

• les **appels de fonctions** adoptent la **notation préfixée** (l'argument apparaît toujours après le nom de la fonction)
• les **arguments** peuvent être des data types primitifs, des variables ou d'autre appels de fonction

• **exemples de fonctions (définies par le système) :**

```
(+ 3 4 5)
( * 5 6.0 2)
(+ 3 (*8 9) 4)
```


Les faits ordonnés (ordered facts)

fait ordonné = symbole :

- suivi d'une **séquence d'aucun** ou **plusieurs champs** :
- les champs sont séparés par des **espaces**
- le **premier champ** spécifie une **relation** ou **prédicat** à laquelle s'appliquent les autres champs
- le fait est encadré par des parenthèses

• le **premier champ** est nécessairement du data-type "**symbol**"

• les autres champs peuvent être de **n'importe quel data-type**

• **mots réservés :** *test, and, or, not, declare, logical, object, exists,...*

• **Exemples :**

```
(pere-de jean paul)
(cueille eric eldelweiss)
(mange alain pomme verte)
(hauteur 10000 metres)
(couleurs-drapeau bleu blanc rouge)
(couleurs drapeau bleu blanc rouge)
```

Les **faits structurés** ou non-ordonnés (template facts or non-ordered facts)

fait structuré = ens. d'informations regroupées selon une **structure** (**template**) et créé par le construct **`deftemplate`**

- permet à l'utilisateur de s'abstraire la **structure** (template) d'un fait en assignant des **noms aux champs**
- encodent de l'information avec une **position** : pour accéder à une information on doit **connaître** le **fait** et le **champ** qui la contient
- les **champs** nommés = **slots (named field) :**
  - ils peuvent être **contraints** par des contraintes de **typage**, de **valeur** ou de **plage numérique**
  - des **valeurs par défaut** peuvent être definies
  - le **premier champ** est nécessairement du data-type "**symbol**"
  - les autres champs peuvent être de **n'importe quel data-type**

Exemples :

- `(client (name "Joe Brown") (id X9345A))`
- `(point-mass (x-velocity 100) (y-velocity -200))`
- `(class (teacher "Martha Jones") (#-students 30) (Room "37A"))`
- `(grocery-list (#-of-items 3) (items bread milk eggs))`

Le construct **deftemplate**

- **Syntaxe :**

```
(deftemplate <deftemplate-name> [<comment>]<slot-definition>*)

<slot-definition>          ::= <single-slot-definition>|<multislot-definition>
<single-slot-definition>   ::= (slot <slot-name> <template-attribute>*)
<multislot-definition>     ::= (multislot <slot-name> <template-attribute>*)
<template-attribute>       ::= <default-attribute> | <constraint-attribute>
<default-attribute>        ::= (default ?DERIVE | ?NONE | <expression>*) |
                               (default-dynamic <expression>*)
```

- **Exemple :**

```
(deftemplate personne
      (slot nom)
      (slot premon)
      (multislot date-naissance))
(deftemplate date-naissance
      (slot jour)
      (slot mois)
      (slot année))
```


Faits **structurés** (non-ordered facts - templates)

- l'**ordre des slots** dans un fait deftemplate n'est **pas important**, ainsi ces 3 faits sont les mêmes :

```
(classe (prof "Bernard Espinasse") (nb-étudiants 20) (Salle "I"))
(classe (nb-étudiants 20) (prof "Bernard Espinasse") (Salle "I"))
(classe (Salle "I") (nb-étudiants 20) (prof "Bernard Espinasse"))
```

- attention ces faits templates sont **différents** :

```
(classe "Bernard Espinasse" 20 "I")
(classe 20 "Bernard Espinasse" "I")
(classe "I" 20 "Bernard Espinasse")
```

- les faits **deftemplate** peuvent être aussi **ajoutés, détruits, modifiés et dupliqués** :
  - `(assert (personne (nom "Durant") (prenom "Joe") ((jour 10) (mois 09) (annee 1960)))`
  - `(detract (personne (nom "Durant") (prenom "Joe") ((jour 10) (mois 09) (annee 1960)))`
  - `(modify (personne (nom "Durant") (prenom "Joe") ((jour 10) (mois 09) (annee 1960)))`
  - `(duplicate (personne (nom "Durant") (prenom "Joe") ((jour 10) (mois 09) (annee 1960)))`

Fonctions de mise à jour de faits dans CLIPS

- mises en oeuvre à partir du prompt **CLIPS>** ou dans partie droite de **règles** :

Faits ordonnés et faits structurés :

Commandes	Objet	Exemple
`assert`	**Ajoute** un fait de la base de faits courante	• `CLIPS>(assert (couleur verte))` • `CLIPS> (assert (status (temp high) (pressure low)))`
`retract`	**Supprime** un fait de la base de faits courante	• `CLIPS>(retract (couleur verte))` • `CLIPS>(retract *)` • `(defrule change-valve-status ?f1 <- (valve ?v open) ?f2 <- (set ?v close) => (retract ?f1 ?f2) (assert (valve ?v close)))`

Faits structurés :

`modify`	**Modifie** un fait de la base de faits courante	• `?f1<-(status (valve open)) ?f2<-(close-valve) => (retract ?f2) (modify ?f1 (valve closed)))`
`duplicate`	**Duplique** un fait de la base de faits courante	• `(defrule duplicate-part ?f1 <- (duplicate-part ?name) ?f2 <- (part (name ?name)) => (retract ?f1) (duplicate ?f2 (id (gensym*))))`

Les faits initiaux : construct deffacts

- le construct **deffacts** permet de définir un ensemble de **faits initiaux**
- qui sont **automatiquement rejoutés** à la base de faits **après une initialisation** (commande **reset**)

```
(deffacts <deffacts-name> [<comment>]<RHS-pattern>*)
```

- **exemple :**

```
(deffacts initial
          (nature alain garcon)
          (noncueille alain arnica)
          (imcueille alain gentiane)
          (cardimcueille alain 1)
          (noncueille alain rhododendron)
          (noncueille alain edelweiss)
          (noncueille alain chardonbleu)
)
```

- aprés un **reset** :

  les 7 faits sont automatiquement rajoutés à la base de faits.

Les Règles dans Clips

- déclarée par le construct **defrule** et composée :
  - d'une **entête** (Header)
  - d'une partie **antécédent** (Left-Hand Side) : ensembles de **conditions (patterns)** à satisfaire portant sur **l'existance** ou la **non-existance** :
    - de **faits** de la liste de faits
    - **d'instance d'objets** (de classe utilisateur) de la liste d'objets
  - d'une partie **conséquent** (Right-Hand Side) : ensemble d'actions à exécuter lorsque la règle est exécutée

Le construct defrule

- **Sytaxe générale :**

```
(defrule <rule-name> [<comment>]
     [<declaration>]
     <conditional-element>*
     =>
     <action>*)
```

- **comment :** commentaires (soit entre quotes "xxxx" soit précédé d'un point virgule ; xxxx)
- **declaration :**

```
<declaration>     ::= (declare <rule-property>+)
<rule-property>       ::= (salience <integer-expression>) |
                          (auto-focus <boolean-symbol>)
```

(salience = priorité)

Exemples de règles en CLIPS

- **Règle sans variable :**

```
(defrule canard "coin"        ; Rule header
  (animal-est canard)           ; Left-Hand Side (un pattern)
=>                            ; THEN arrow
  (assert (cris-est coin)))   ; Left-Hand Side (une action)
```

- **Règle avec variables :**

```
(defrule regle1; Mise a jour de l'ensemble imcueille
(noncueille ?I ?J)
?f1 <- (imcueille   ?I ?J)
?f2 <- (cardimcueille ?I ?V)
=>
(retract ?f1 ?f2)
(assert (cardimcueille ?I (- ?V 1))))
```

  - **?f1** et **?f2** sont des **variables de faits** ou **fact index** (utiles pour la manipulation de faits dans les règles)

Variables globales et variables locales

- **Variables globales :**
  - le construct **defglobal** permet la définition de **variables globales** à l'environnement Clips
  - une variable **globale** peut être **accedée dans tout l'environnement** Clips et garde sa valeurs indépendantes des autres constructs
- **Variables locales :**
  - les constructs comme **defrule** (définition de règles) ou **deffunction** (définition de fonctions) permettent de définir des **variables locales** au construct
  - les variables **locales** ne peuvent être **accédées qu'au sein du construct**

Les variables **locales** d'une règle

```
(defrule regle1
; Mise a jour de l'ensemble imcueille
      (noncueille ?I ?J)
      ?f1 <- (imcueille   ?I ?J)
      ?f2 <- (cardimcueille ?I ?V)
=>
(retract ?f1 ?f2)
(assert (cardimcueille ?I (- ?V 1)))
)
```

- ?I, ?J sont des **variables locales** à la règle "regle1"
- les variables seront **instanciés** par des faits de la base de fait (liste des faits)
- obligatoirement **précédé d'un "?"** (non confondu avec une valeur alphanumérique)
- **2 variables différentes** dans la partie prémisse d'une même règle entraînera l'instanciation avec **deux faits différents**
- pour avoir **2 variables identiques**, utiliser le **même identificateur**
- **variable de faits : ?f1** et **? f2**

Le construct **defrule**

- **Syntaxe générale :**

```
(defrule <rule-name> [<comment>]
    [<declaration>]
    <conditional-element>*
    =>
    <action>*)
```

- **conditional elements :**

```
<conditional-element>   ::= <pattern-CE> | <assigned-pattern-CE> |
                            <not-CE> | <and-CE> | <or-CE> |
                            <logical-CE> | <test-CE> |
                            <exists-CE> | <forall-CE>

<test-CE>               ::= (test <function-call>)
<not-CE>                ::= (not <conditional-element>)
<and-CE>                ::= (and <conditional-element>+)
<or-CE>                 ::= (or <conditional-element>+)

<exists-CE>             ::= (exists <conditional-element>+)
<forall-CE>             ::= (forall <conditional-element>
                                    <conditional-element>+)
<logical-CE>            ::= (logical <conditional-element>+)
```


Le construct **defrule** (suite)

- **pattern conditional element :**

```
<assigned-pattern-CE>   ::= ?<variable-symbol> <- <pattern-CE>

<pattern-CE> ::= <ordered-pattern-CE> | <template-pattern-CE> |<object-pattern-CE>
<ordered-pattern-CE>   ::= (<symbol> <constraint>*)
<template-pattern-CE>  ::= (<deftemplate-name <LHS-slot>*)
<object-pattern-CE>    ::= (object <attribute-constraint>*)
<attribute-constraint>      ::= (is-a  <constraint>)  |  (name  <constraint>)  |  (<slot-name>
  <constraint>*)
<LHS-slot>             ::= <single-field-LHS-slot> | <multifield-LHS-slot>
<LHS-slot>              ::= <single-field-LHS-slot> |
<single-field-LHS-slot> ::= (<slot-name> <constraint>)
<multifield-LHS-slot>   ::= (<slot-name> <constraint>*)
```

- **pattern constraints :**

```
<constraint>   ::= ? | $? | <connected-constraint>

<connected-constraint> ::= <single-constraint> | <single-constraint> & <connected-constraint>|
                           <single-constraint> | <connected-constraint>
<single-constraint>     ::= <term> | ~<term>
<term>                  ::= <constant> | <single-field-variable> | <multifield-variable> |
                            :<function-call> | =<function-call>
```


Exemple de représentation de connaissances en Clips

Un puzzle logique inspiré de Lewis Carroll :

"Alain, Eric, Patrick, Daniel et Jean-Marc vont cueillir des fleurs de montagne. Ils choisissent la gentiane bavaroise, l'arnica, le rhododendron, l'edelweiss et le chardon bleu et ils décident que chacun ne rapportera qu'une espèce de fleur.

Celui qui cueille l'arnica et celui qui cueille le rhododendron conseillent à Alain de mettre ses fleurs dans un sac en plastique, car elles fanent très vite.

Jean-Marc veut garder ses fleurs fraîches, tandis que le garçon qui cherche l'edelweiss et Eric ont l'intention de faire sécher les leurs.

Jean-Marc et celui qui cueille le rhododendron craignent d'avoir des difficultés, car les plantes qu'ils cherchent commencent à défleurir. Au contraire, le chardon bleu commence tout juste à se dorer.

Alain et Daniel recommandent à celui qui cherche l'edelweiss et à celui qui cherche le chardon bleu de ne pas trop cueillir de fleurs car ce sont des espèces rares.

Quelle fleur a choisie chacun de ces enfants ?"

Connaissances contenues dans l'énoncé

• **6 relations (prédicats) :**
  - **cueille**: (cueille I J) indique que le garçon i cueille la fleur j.
  - **noncueille**: (noncueille I) donne l'ensemble des fleurs que i ne peut cueillir.
  - **imcueille**: (imcueille I) donne l'ensemble des fleurs que i peut encore cueillir.
  - **cardimcueille**: (cardimcueille I) donne le nombre de fleurs que i peut encore cueillir (cardinal de l'ensemble imcueille)
  - **nature**: (nature I) indique la nature de I soit : garçon. Cette propriété est utile afin d'éviter des instanciations qui n'auraient aucun sens comme par exemple, qu'une fleur puisse cueillir une autre fleur

Faits ordonnés

• la phrase de l'énoncé suivante:

"Eric est un garçon, il peut cueillir toutes les fleurs (soit 4 fleurs) excepté l'Edelweiss,"

• sera représentée par les faits ordonnés suivants :

```
(nature eric garcon)
(imcueille eric gentiane)
(imcueille eric arnica)
(imcueille eric rhododendron)
(imcueille eric chardonbleu)
(cardimcueille eric 4)
(noncueille eric edelweiss)
```


Base de **faits** de l'exemple

```
(deffacts startup
        (nature alain garcon)
        (noncueille alain arnica)
        (imcueille alain gentiane)
        (cardimcueille alain 1)
        (noncueille alain rhododendron)
        (noncueille alain edelweiss)
        (noncueille alain chardonbleu)

        (nature eric garcon)
        (imcueille eric gentiane)
        (imcueille eric arnica)
        (imcueille eric rhododendron)
        (imcueille eric chardonbleu)
        (cardimcueille eric 4)
        (noncueille eric edelweiss)

        (nature patrick  garcon)
        (imcueille patrick  gentiane)
        (imcueille patrick  arnica)
        (imcueille patrick  rhododendron)
        (imcueille patrick  edelweiss)
        (imcueille patrick  chardonbleu)
        (cardimcueille patrick 5)

        (nature daniel garcon)
        (imcueille daniel gentiane)
        (imcueille daniel arnica)
        (imcueille daniel rhododendron)
        (cardimcueille daniel 3)
        (noncueille daniel edelweiss)
        (noncueille daniel chardonbleu)

        (nature jeanmarc garcon)
        (imcueille jeanmarc gentiane)
        (imcueille jeanmarc arnica)
        (cardimcueille jeanmarc 2)
        (noncueille jeanmarc rhododendron)
        (noncueille jeanmarc edelweiss)
        (noncueille jeanmarc chardonbleu)
)
```


Base de **règles** de l'exemple

```
(defrule regle1
; Mise a jour de l'ensemble imcueille
          (noncueille ?I ?J)
?f1 <- (imcueille   ?I ?J)
?f2 <- (cardimcueille ?I ?V)
=>
(retract ?f1 ?f2)
(assert (cardimcueille ?I (- ?V 1)))
)
```

• met à jour l'ensemble des fleurs que I peut encore cueillir en tenant compte des informations lui indiquant que la fleur J ne peut être cueillie par I (noncueille ?I ?J),
• met à jour (cardimcueille ?I) qui est le cardinal de imcueille

```
(defrule regle2
; surjectivite de la fonction cueille
           (cardimcueille ?I 1)
           (imcueille ?I ?K)
=>
(assert (cueille ?I ?K))
)
```

• lorsque l'on sait qu'il ne reste plus qu'une fleur, que I peut alors cueillir (cardimcueille ?I 1)
• permet d'assigner la fleur que I cueille à l'aide de (assert (cueille ?I ?k))

```
(defrule regle3
; injectivite de la fonction cueille
            (cueille ?I ?J)
            (nature ?K garcon)
=>
(assert (noncueille ?K ?J))
)
```

• si l'on sait qu'une fleur a déjà été cueillie par un enfant, permet de l'ajouter parmi celles que ne peuvent cueillir les autres

Déclenchement d'une règle

```
(defrule regle1
; Mise a jour de l'ensemble imcueille
          (noncueille ?I ?J)
?f1 <- (imcueille   ?I ?J)
?f2 <- (cardimcueille ?I ?V)
=>
(retract ?f1 ?f2)
(assert (cardimcueille ?I (- ?V 1)))
)
```

Si l'ensemble des relations de la partie gauche (patterns) d'une règle peut être mis en correspondance avec les faits de la base de faits:
  - **instanciation des prémisses** : substitution des variables des prémisses par des faits de la base de fait

ALORS la règle sera déclenchée:
  - **instanciation de la partie droite (actions)** : substitution des variables des actions par des faits de la base de faits
  - **actions de modification ou création de faits dans la base de faits**
  - **actions sur la base de règles**

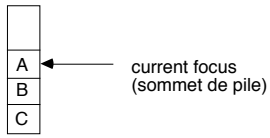
Construct defmodule

• **syntaxe** :

```
(defmodule <module-name> [<comment>] <port-spec>*)

<port-specification> ::= (export <port-item>) |
                         (import <module-name> <port-item>)
```

• permet le **partitionnement** de la base de connaissance en rassemblant un **ensemble de constructs**
• permet un **contrôle explicite d'accés à ces constructs par d'autres modules**
• utilisé pour **controler l'éxécution de règles** (commande "**focus**")

• **pile des focus** (focus stack) :
  • soit 3 modules : A, B, C
    focus (A B C)
• construit la pile des focus suivants :



Exemple de defmodule :

```
(defmodule FOO
   (import BAR ?ALL)
   (import YAK deftemplate ?ALL)
   (import GOZ defglobal x y z)
   (export defgeneric +)
   (export defclass ?ALL))
```


Les règles et leurs propriétés : salience et auto-focus

• Sytaxe générale du construct **defrule :**

```
(defrule <rule-name> [<comment>]
    [<declaration>]
    <conditional-element>*
    =>
    <action>*)

<declaration>    ::= (declare <rule-property>+)

<rule-property>      ::= (salience <integer-expression>) |
                         (auto-focus <boolean-symbol>)
```

• une règle peut appartient à un **module** (module "main" par défaut)
• **salience = priorité** (entier compris entre -10000 et +10000 - valeur par défaut = 0)
  • soit explicité par **l'utilisateur**
  • soit assigné par **CLIPS**
• **auto-focus** : true/false (valeur par défaut = false)
  si True, alors une commande de focus sur le module est automatiquement exécutée quand la règle est exécutée

Les actions des règles dans CLIPS

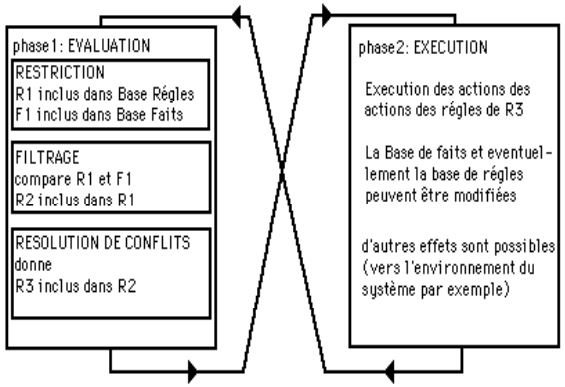
• non restreint aux clauses de HORN, CLIPS accepte **plusieurs actions** pour une **même règle**
• peuvent être la plupart des **fonctions** et **commandes** de CLIPS
• l'**ordre d'écriture** des actions est pris en considération lors de leurs exécutions

Natures des actions

• **Actions sur les connaissances factuelles :**
  • ajout ou modification de connaissance
  • suppression de connaissance
  ...
• **Actions sur les connaissances opératoires :**
  • modification des priorités
  • arrêt de l'interpréteur
  ...
• **Actions sur l'environnement :**
  • Écriture à l'écran....
  ...
• **Remarque** : CLIPS fournit aussi quelques **structures de contrôle procédural** qui peuvent être utilisées dans la partie droite de la règle : **while**, **if then else**.

Cycle de base de l'exécution de règle dans CLIPS

cycle d'EVALUATION-EXECUTION (H.FARRENY):



Cycle de base de l'exécution de règle dans CLIPS

Phase d'évaluation :

- **Étape de SÉLECTION**
  - dans CLIPS cette phase est assurée grâce aux **modules** et à la pile de **focus**
- **Étape de FILTRAGE**
  - CLIPS sélectionne l'ensemble des règles pouvant être activées et les range dans l'**agenda**
- **Étape de RÉSOLUTION DE CONFLITS**
  - CLIPS fournit 7 stratégies de résolution de conflits : **depth, breadth, simplicity, complexity, lex, mea,** et **random**

Phase d'exécution :

- **Étape de xxxxx**
  - xxx

Exemple d'exécution de règle

- ajoutons un **fait** :

```
CLIPS> (assert (animal-est canard))
<Fact-0>
CLIPS> (facts)
f-0     (animal-est canard)
For a total of 1 facts.
```

- soit la **règle** :

```
(defrule canard "coin"                ; Rule header
   (animal-est canard)                ; Pattern
=>                                    ; THEN arrow
   (assert (cris-est coin)))          ; Action
```

- elle ne comporte qu'un seul pattern dans sa partie gauche, il est sans variable, totalement instancié, c'est une contrainte litteral (**litteral constraint**)
- la commande "**run**" permet d'exécuter la règle "canard" et on a :

```
CLIPS> (run)
CLIPS> (facts)
f-0     (animal-est canard)
f-1     (cris-est coin)
For a total of 2 facts.
```

- il y a ajout d'un nouveau fait dans la base soit `(cris-est coin)` (total 2 faits)

Exemple d'exécution de règle

Exemple d'exécution en utilisant la commande **watch** pour observer faits et activations:

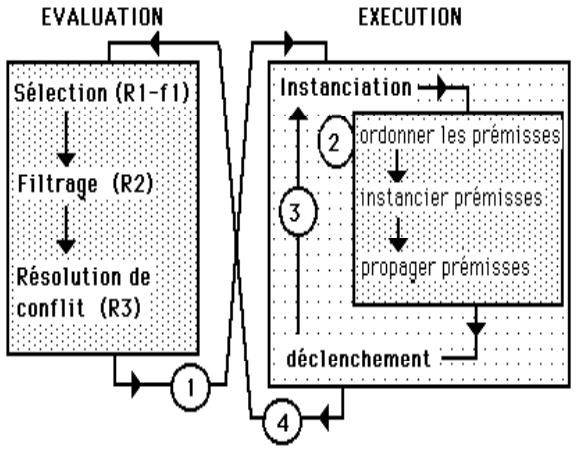
```
CLIPS> (clear)
CLIPS> (defrule canard
   (animal-est canard)
=>
   (printout t "coin" crlf))
CLIPS> (watch facts)
CLIPS> (watch activations)
CLIPS> (assert (animal-est canard))
==> f-0     (animal-est canard)
==> Activation 0       canard: f-0 ; La priorité d'activation est 0 pour <Fact-0> ;
  defaut, then rule name:pattern entity index

CLIPS> (assert (animal-est canard))   ; Notons que cela duplique le fait
FALSE                         ; ne peut être rentré

CLIPS> (agenda)
0      canard: f-0
For a total of 1 activation.

CLIPS> (run)
coin

CLIPS> (agenda)    ; Rien dans l'agenda après l'exécution de la règle
CLIPS> (facts)         ; Even though fact matches rule, refraction
f-0     (animal-est canard); Ne permet pas cette activation parce que
For a total of 1 fact.; La règle a déjà ete exécuté sur ce fait
CLIPS> (run)
CLIPS>
```

- pour **re-exécuter** la règle : **supprimer** le fait `(animal-is canard)` et **l'ajouter** à nouveau.

Cycle de base de l'exécution de règle dans CLIPS

- **étape 1:**
  - si limite d'exécution de règle atteinte ou pas de focus courant => l'exécution stopée
  - sinon la règle du sommet de l'agenda du module du focus courant est sélectionnée
  - si pas de règles dans l'agenda => focus courant supprimé de la pile des focus et le focus courant devient le module suivant
  - si pile des focus vide => exécution est stopée, sinon l'étape 1 est relancée
- **étape 2:**
  - actions de la partie droite de la règle sont exécutées (la fonction return sur la partie droite de la règle peut supprimer le focus courant de la pile des focus)
  - le nombre de règles exécutées est incrementé.
- **étape 3:**
  - en résultat de l'étape 2, des règles peuvent être activées ou désactivées :
    - Règles activées (conditions satisfaites) placées dans l'agenda de leur module. Règle placée dans agenda selon sa **priorité** (salience) et la **stratégie** de résolution retenue.
    - Règles désactivées sont supprimées de l'agenda.
- **étape 4:**
  - si une **priorité dynamique** (dynamic salience) est utilisée, la valeur des priorités de toutes les règles est réévaluée. Le cycle est repeté à l'étape 1.

Cycle de base de l'exécution de règle dans CLIPS ??



Agenda

- liste de toutes les **règles dont conditions satisfaites** - **règle activées** - (pas encore exécutées)
- chaque **module** a son **propre agenda**
- agit comme une **pile** : la **règle du sommet est la première à être exécutée**
- le **placement de la règle dans l'agenda** respecte, dans l'ordre, les facteurs suivants :
  a) elle est placée **au-dessus** toutes les règles de priorité plus **faible** et **au-dessous** toutes les règles de priorité plus **élevées**
  b) les **règles d'égale priorités** sont ordonnées selon la **stratégie de résolution de conflits courante**
  c) si une règle est activée avec d'autres règles par la même assertion ou retraction d'un fait, et que les étapes a) et b) ne peuvent spécifier un ordre, alors la règle est arbitrairement (not random) ordonnée avec les autres règles aussi activées.

- Il est possible de **contrôler** ce qu'il y a dans l'agenda avec la commande "**agenda**" :

```
CLIPS> (agenda)
0      canard: f-0
For a total of 1 activation.
CLIPS>
```

  - la priorité de l'activation "canard" = 0;
  - "f-0" = identifiant du fait de la base avec lequel le pattern de la règle s'unifie

Les 7 Stratégies de Résolution de Conflits de CLIPS

- CLIPS fournit 7 stratégies de résolution de conflits : **depth, breadth, simplicity, complexity, lex, mea,** et **random**.
- la stratégie par défaut est "**depth**"
- une stratégie peut être fixée par la commande **set-strategy** qui réordonne alors l'agenda.

1 - Depth Strategy (stratégie en profondeur)

- Les règles nouvellement activées sont placées avant toutes les règles de même priorité.
- **Exemple** : soit un fait fact-a activant les règles rule-1 et rule-2 et fact-b activant rule-3 et rule-4, alors si fact-a est ajouté avant fact-b, rule-3 et rule-4 seront **au-dessus** rule-1 et rule-2 dans l'agenda. Cependant, la position relative de rule-1 par rapport à rule-2 et de rule-3 par rapport à rule-4 sera arbitraire, soit :

```
rule-3, rule-4, rule-1, rule-2
```

2 - Breadth Strategy (stratégie en largeur)

- Les règles nouvellement activées sont placées sous toutes les règles de même priorité.
- **Exemple** : soit fact-a activant rule-1 et rule-2 et fact-b activant rule-3 et rule-4, alors si fact-a est ajouté avant fact-b, rule-1 et rule-2 seront **au-dessus** rule-3 et rule-4 dans l'agenda. Cependant la position relative de rule-1 par rapport à rule-2 et de rule-3 par rapport à rule-4 sera arbitraire, soit : rule-1, rule-2, rule-3, rule-4

Les 7 Stratégies de Résolution de Conflits de CLIPS

3 - Simplicity Strategy

- Parmis les règles de même priorité, les règles nouvellement activées sont placées au-dessus de toutes les règles avec une **specificité** (specificity) egale ou supérieure.
- **spécificité d'une règle** : déterminée par le nombre de comparaisons qui doivent être faites sur la partie gauche (LHS) de la règle. Chaque comparaison à une constante ou une **previously bound variable** ajoute un à la spécificité. Chaque appel de fonction fait par la LHS de la règle considérée, comme une partie de ":", "=" ou un test conditionel d'élément ajoute un à la spécificité. Les fonctions booléenne and, or et not n'ajoutent pas de point à la spécificité de la règle, mais leurs arguments le font. Les appels de fonctions faits au sein d'un appel de fonction n'ajoutent pas de point à la spécificité de la règle.
- **Exemple** :

```
(defrule example
   (item ?x ?y ?x)
   (test (and (numberp ?x) (> ?x (+ 10 ?y)) (< ?x 100)))
   =>)
```

la règle a une spécificité de 5 :

+1 : La comparaison à un item contant

+1 : la comparaison de ?x a son ?x précedent (second pattern)

+3 : l'appel aux 3 fonctions : numberp puis < , >

Notons que l'appel des fonctions and et + n'ajoutent pas à la spécificité de la règle.

Les 7 Stratégies de Résolution de Conflits de CLIPS

4 - Complexity Strategy (stratégie complexité)

- Parmi les règles de même complexité, les règles nouvellement activées sont placées au-dessus des toutes les règles ayant une spécificité égale ou inférieure.

5 - LEX Strategy (stratégie lex)

- Parmi les règles de même complexité, les règles nouvellement activées sont placées en utilisant la stratégie utilisée par OPS5, stratégie lex.

Autres commandes de CLIPS

Commandes	Objet
DECLARE SALIENCE	permet un contrôle explicite sur quelles règles peuvent être mises à l'agenda (être prudent dans son usage afin de ne pas trop contraindre le programme)
SET-INCREMENTAL-RESET	interdit aux règles de voir les faits qui ont été entrés avant que celles-ci soit entrées dans l'agenda
GET-INCREMENTAL-RESET	pour obtenir la valeur courante de incremental reset.
REFRESH RULE	une façon de faire ré-exécuter une règle est de forcer la règle à être réactivée
SAVE / LOAD	permet de sauver et de charger un fichier de règles sous le nom "xxx.clp" sur disque
BSAVE / LOAD BINARY	plus rapide que les commandes précedentes, celles-ci permettent de sauver et de charger un fichier de règles en binaire sur disque
SAVE-FACTS ET LOAD-FACTS	permet de sauver et de charger un fichier de faits sur disque.
BATCH	permet d'exécuter des commandes à partir d'un fichier comme si elles étaient tapées
SYSTEM	permet l'exécution de commandes du système d'exploitation ou d'exécutables au sein de CLIPS

Commandes et Fonctions d'agenda

• **Commandes :**

AGENDA	Displays all activations on the agenda of the specified module.	`(agenda [<module-name>])`
RUN	Starts execution of rules. Rules fire until agenda is empty or the number of rule firings limit specified by the first argument is reached (infinity if unspecified).	`(run [<integer-expression>])`
FOCUS	Pushes one or more modules onto the focus stack.	`(focus <module-name>+)`
HALT	Stops rule execution.	`(halt)`
SET-STRATEGY	Sets the current conflict resolution strategy.	`(set-strategy <strategy>)` `<strategy> ::= depth | breadth | simplicity | complexity | lex | mea | random`
GET-STRATEGY	Returns the current conflict resolution strategy.	`(get-strategy)`
LIST-FOCUS-STACK	Lists all module names on the focus stack.	`(list-focus-stack)`

Commandes (suite) et Fonctions d'agenda

CLEAR-FOCUS-STACK	Removes all modules from the focus stack.	`(clear-focus-stack)`
SET-SALIENCE-EVALUATION	Sets the salience evaluation behavior.	`(set-salience-evaluation <behavior>)` `<behavior> ::= when-defined | when-activated | every-cycle`
GET-SALIENCE-EVALUATION	Returns the salience evaluation behavior.	`(get-salience-evaluation)`
REFRESH-AGENDA	Forces reevaluation of salience of rules on the agenda of the specified module.	`(refresh-agenda [<module-name>])`

• **Fonctions :**

GET-FOCUS	Returns the module name of the current focus	`(get-focus)`
GET-FOCUS-STACK	Returns all of the module names in the focus stack as a multifield value	`(get-focus-stack)`
POP-FOCUS	Removes the current focus from the focus stack and returns the module name of the current focus	`(pop-focus)`

Traces d'exécution avec CLIPS: la commande watch

Syntax : `(watch<watch-item>)` **exemple :** `CLIPS> (watch rules)`
annulation : `CLIPS> (unwatch rules)`

watch-item :	sont observés :
COMPILATIONS	the progress of construct definitions will be displayed.
FACTS	all fact assertions and retractions. Optionally, facts associated with individual deftemplates can be watched by specifying one or more deftemplate names.
RULES	all rule firings will be displayed.
ACTIVATIONS	all rule activations and deactivations will be displayed. Optionally, rule firings and activations associated with individual defrules can be watched by specifying one or more defrule names.
STATISTICS	timing information along with other information (average number of facts, average nb of activations, etc.) will be displayed after a run.
FOCUS	is watched, then changes to the current focus will be displayed.
GLOBALS	variable assignments to globals variables will be displayed. Optionally, variable assignments associated with individual defglobals can be watched by specifying one or more defglobal names.
DEFFUNCTIONS	the start and finish of deffunctions will be displayed. Optionally, the start and end display associated with individual deffunctions can be watched by specifying one or more deffunction names.

Traces d'exécution avec CLIPS: la commande watch (suite)

watch-item :	sont observés :
GENERIC-FUNCTIONS	the start and finish of generic functions will be displayed. Optionally, the start and end display associated with individual defgenerics can be watched by specifying one or more defgeneric names.
METHODS	the start and finish of individual methods within a generic function will be displayed. Optionally, individual methods can be watched by specifying one or more methods using a defgeneric name and a method index.
INSTANCES	creation and deletion of instances will be displayed.
SLOTS	changes to any instance slot values will be displayed. Optionally, instances and slots associated with individual concrete defclasses can be watched by specifying one or more concrete defclass names.
MESSAGE-HANDLERS	the start and finish of individual message-handlers within a message will be displayed. Optionally, individual message-handlers can be watched by specifying one or more message-handlers using a defclass name, a message-handler name, and a message-handler type.
MESSAGES	the start and finish of messages will be displayed. For the watch items that allow individual constructs to be watched, if no constructs are specified, then all constructs of that type will be watched.
ALL	all other watch items will be watched. By default, only compilations are watched. The watch function has no return value.