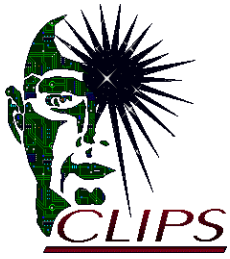
CLIPS



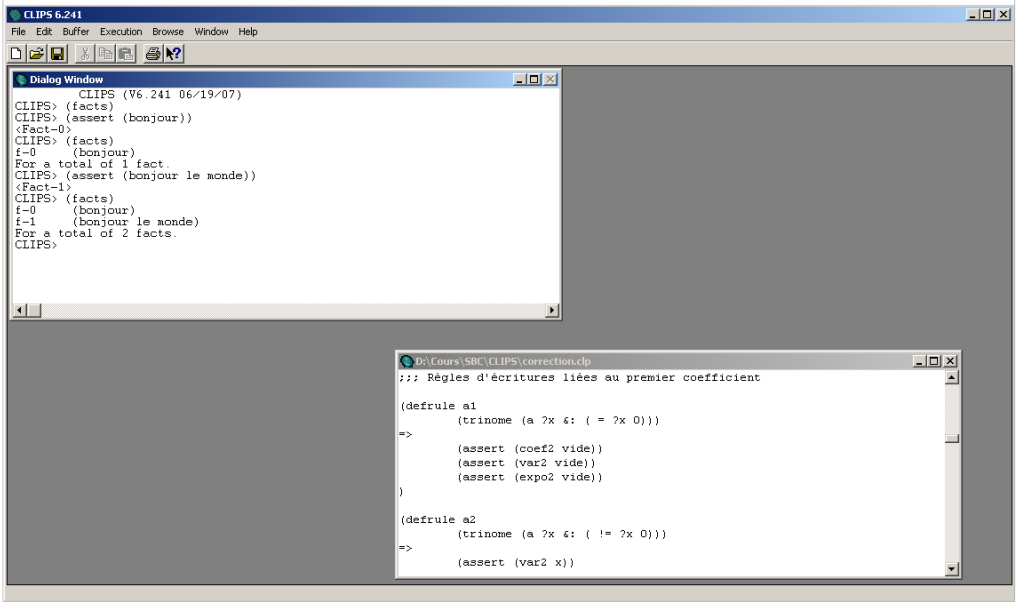
- C Language Integrated Production System
- Historique : OPS5 (1978), ART (1980), CLIPS (1985), CLIPS 6.0 (1994)
- Originellement développé à la NASA pour
  - Le contrôle de processus
  - Le diagnostic de panne
  - La planification des missions spatiales
- CLIPS est devenu le noyau de milliers de SE

Quelques aspects techniques

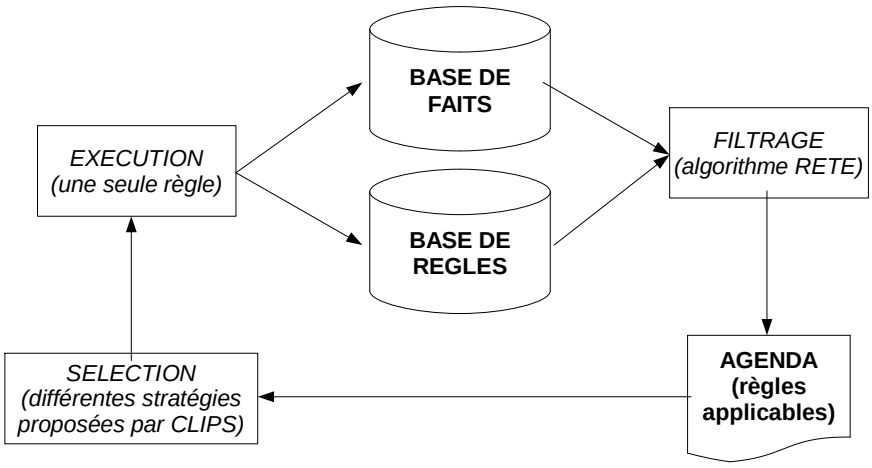
- CLIPS est en open source et intégré à de nombreuses distributions LINUX
- La syntaxe de CLIPS est proche de LISP (beaucoup de parenthèses!)
- Interface avec interpréteur de commandes
- Possibilité d'intégrer des procédures écrites en C
- Interfaçage Java Native Interface (il existe aussi un clône de CLIPS écrit en Java, Jess)

clipsrules.sourceforge.net

GUI CLIPS



Architecture de CLIPS



Caractéristiques générales

- CLIPS utilise l'hypothèse du monde fermé
- CLIPS intègre un moteur d'ordre 1
- CLIPS est non monotone (faits rétractables)
- CLIPS permet le chainage avant (mais on peut simuler le chainage arrière en utilisant des méta-règles)
- CLIPS est en régime irrévocable (pas de backtrack, mais on peut également simuler le backtrack)

Les types

- Les variables ne sont pas typées mais CLIPS propose les types suivants :
  - **`SYMBOL`** : suite de caractères ASCII hors mots réservés
  - **`STRING`** : chaine de caractères entre doubles guillemets
  - **`INTEGER`** et **`FLOAT`**
  - **`EXTERNAL-ADDRESS`** : pointeurs sur les structures
  - **`FACT-ADDRESS`** : adresses des faits
  - **`INSTANCES-NAME`** et **`INSTANCE-ADDRESS`**

Les faits

- Un fait peut être :
  - une liste de mots
  - une structure nommée composée d'une liste de paires (attribut, valeur)
  - une instance d'un objet

```
(est-homme Socrate)

(habite Hansel maison-en-pain-d'epices)
```

- Affichage des faits (chaque fait est identifié par un indice) : **`(facts)`**

```
CLIPS> (facts)
f-0     (est-homme Socrate)
f-1     (habite Hansel maison-en-pain-d'epices)
f-2     (soeur Gretel Hansel)
f-3     (meurt sorciere dans-le-four)
For a total of 4 facts.
CLIPS>
```


Commandes sur les faits (1/2)

- Ajout des faits (échec si le fait existe déjà)

```
(assert (est-homme Socrate) (habite Hansel maison-en-pain-d'epices))
```

- Définition de faits (ils ne sont pas ajoutés)

```
(deffacts mesfaits (est-homme Socrate) (habite Hansel maison-en-pain-d'epices))
```

- Retrait de faits (par leur indice)

```
(retract 0 2)
```

- Effacement de la base de faits

```
(clear)
```

Commandes sur les faits (2/2)

- Remise à zéro : **`(reset)`**
  - effacement des faits de la base
  - ajout du fait **`(initial-fact)`** qui peut être utilisé pour initier des inférences
  - ajout des faits définis par **`deffacts`**
- Modification des faits (par leur indice)

```
(assert (personne (nom "lili") (age 2)))

(modify 0 (age 34))
```


Structures (1/2)

- Les structures (template) sont des « proto-objets » possédant
  - Un *nom*
  - Des *attributs* (slots et multislots) valués et éventuellement typés
- Des valeurs par défaut existent :
  - "" pour les chaines
  - 0 et 0.0 pour les nombres
  - **`?DERIVE`** affecte la valeur par défaut du type

Structures (2/2)

- Cliqu

```
(deftemplate personne
      "représente une personne" ;commentaires optionnels
      (slot nom
          (type STRING)
          (default ""))
      (slot age
          (type INTEGER)
          (default ?DERIVE))
      (slot taille
          (type FLOAT)
          (default 1.50))
      (multislot adresses
          (type STRING)))
```

```
(deffacts mespersonnes
      (personne
          (nom "Tutu")
          (taille 1.89))
      (personne
          (nom "Titi")
          (age 27)
          (adresses "3 rue Machin, Amiens" "7 avenue Chose, Marseille")))
(reset)
```


Commandes sur les structures

- Affichage des structures

```
(list-deftemplates)
```

- Affichage d'une structure

```
(ppdeftemplate <nom de structure>)
```

- Suppression d'une structure

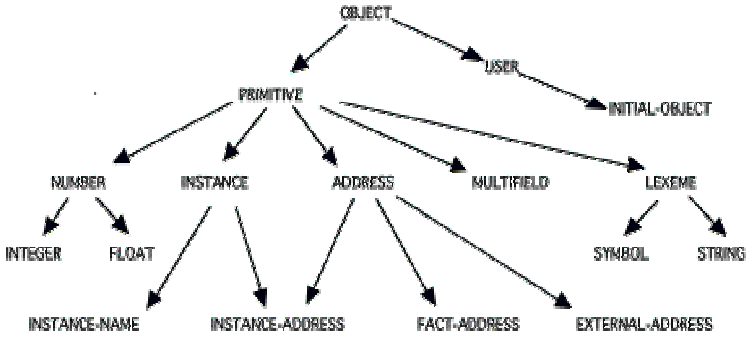
```
(undeftemplate <nom de structure>)
```

CLIPS OO

- Possibilité de créer des classes avec
  - Héritage
  - Aspect concret ou abstrait
  - Des attributs (slots)
  - Des méthodes

```
(defclass Personne
      "représente une personne" ;commentaires optionnels
      (is-a OBJECT)
      (role concrete)
      (slot nom
          (type STRING)
          (default ""))
      (slot age
          (type INTEGER)
          (default ?DERIVE))
      (slot taille
          (type FLOAT)
          (default 1.50))
      (multislot adresses
          (type STRING)))
```


Les classes de base



Attention :

- le nom d'une instance n'est pas un symbole
- Le nom d'une instance est donné entre crochets

```
(symbol-to-instance-name Toto)
(instance-name-to-symbol [Toto])
```

Commandes POO

- Création d'instances

```
(make-instance Tutu of Personne (nom "Tutu"))

(definstances instancesDePersonnes
       (Toto of Personne (nom "Toto") (age 27))
       (Titi of Personne (nom "Titi") (taille 1.78)))
(reset)

(unmake-instance Tutu)
```

- Autres commandes sur classes et instances

```
(list-defclasses) ;affiche les definitions de classes
(browse-classes) ;affiche l'arborescence des classes
(ppdefclass <class>) ;affiche la definition de la classe
(describe-class <class>) ;affiche une description complète de la classe
(undefclass <class>) ;elimine la definition de la classe
(instances) ;affiche les instances
(modify-instance <instance> <slot-override>*) ;modification des champs
```

```
(super-classp <class1> <class2>) ;retourne vrai si class1 est super-classe de class2
(subclassp <class1> <class2>) ;retourne vrai si class1 est sous-classe de class2
```

Messages

- Cliquez pour ajo

```
(send [Toto] put-age 28)

(send [Titi] delete)
```

```
CLIPS> (describe-class Personne)
==========================================================================
**************************************************************************
Concrete: direct instances of this class can be created.
Reactive: direct instances of this class can match defrule patterns.

Direct Superclasses: USER
Inheritance Precedence: Personne USER OBJECT
Direct Subclasses:
--------------------------------------------------------------------------
SLOTS     : FLD DEF PRP ACC STO MCH SRC VIS CRT OVRD-MSG     SOURCE(S)
nom       : SGL STC INH RW  LCL RCT EXC PRV RW  put-nom      Personne
age       : SGL STC INH RW  LCL RCT EXC PRV RW  put-age      Personne
taille    : SGL STC INH RW  LCL RCT EXC PRV RW  put-taille   Personne
adresses  : MLT STC INH RW  LCL RCT EXC PRV RW  put-adresses Personne

Constraint information for slots:

SLOTS     : SYM STR INN INA EXA FTA INT FLT
nom       :      +
age       :                           +      RNG:[-oo..+oo]
taille    :                              +   RNG:[-oo..+oo]
adresses  :      +                           CRD:[0..+oo]
--------------------------------------------------------------------------
Recognized message-handlers:
init primary in class USER
delete primary in class USER
create primary in class USER
print primary in class USER
direct-modify primary in class USER
message-modify primary in class USER
direct-duplicate primary in class USER
message-duplicate primary in class USER
get-nom primary in class Personne
put-nom primary in class Personne
get-age primary in class Personne
put-age primary in class Personne
get-taille primary in class Personne
put-taille primary in class Personne
get-adresses primary in class Personne
put-adresses primary in class Personne
**************************************************************************
==========================================================================
CLIPS>
```

Les règles

SI <conjonction de CONDITIONS>
ALORS <conjonctions d'ACTIONS>

CONDITION = teste l'appartenance de certains faits à la base de faits (s'il n'y a pas de condition, (initial-fact) est pris comme condition)

ACTION = modification de la base de faits (les actions sont exécutées séquentiellement)

```
(defrule maRegle
       "ceci est une règle exemple"
       (j'ecoute en cours)
       (je travaille en TD)
=>
       (assert (j'aurais le module))
       (printout t "cool" crlf))
```


Lancement du MI

- **`(run [<integer>])`** : si un entier (positif) est précisé, l'exécution s'arrête après le nombre d'application de règles correspondant (ou lorsque l'agenda ne contient plus d'activations de règles)

```
CLIPS> (facts)
f-0     (initial-fact)
f-1     (j'ecoute en cours)
f-2     (je travaille en TD)
For a total of 3 facts.
CLIPS> (run)
cool
CLIPS> (facts)
f-0     (initial-fact)
f-1     (j'ecoute en cours)
f-2     (je travaille en TD)
f-3     (j'aurais le module)
For a total of 4 facts.
CLIPS>
```

- **`(watch <item>)`** : mode de traçage (**`(watch all)`** pour tout voir)

Commandes sur les règles

- Affichage des règles

```
(list-defrules)
```

- Affichage d'une regle

```
(ppdefrule <nom de regle>)
```

- Suppression d'une regle

```
(undefrule <nom de regle)
```

- Affichage des faits qui permettent l'activation d'une règle

```
(matches <nom de regle)
```


Variables (1/3)

- Les variables peuvent servir à contenir la valeur du champ d'un ou plusieurs faits (variable de champ)
  - pour désigner un ensemble de faits
  - pour la mise en correspondance des faits

  - **?** : variable sans nom (valeur non récupérée)

  - **?<nom>** : variable nommée

```
(deffacts etudes
       (j'ecoute cours SBC)
       (j'ecoute cours Decidabilite)
       (je_travaille en_TD SBC))

(defrule ObtentionModule
       (j'ecoute cours ?intitule)
       (je_travaille en_TD ?intitule)
=>
       (assert (j'aurais module ?intitule))
       (printout t "module obtenu" crlf))

(defrule bonus
       (j'ecoute cours ?)
=>
       (j'aurais jury bien_dispose))
```

Variables (2/3)

- Les variables peuvent servir à contenir l'indice d'un fait de la base de faits (variable d'indice)
  - permet d'agir sur les faits

```
(defrule TricheModule
       (j'ai_triche module ?intitule)
       ?f <- (j'aurais module ?intitule)
=>
       (retract ?f))

(defrule UnAnDePlus
       ?f <- (personne (age ?a))
=>
       (modify ?f (age (+ ?a 1))))
```


Variables (3/3)

- Les variables peuvent contenir des listes

- $?<nom> : variable de liste nommée

- $? : variable de liste sans nom (valeur non récupérée)

```
(deffacts etudiants
       (Mr Robert Duchmol)
       (Mr Otto Rhino Laryngo Logiste)
       (Mme Augusta Ada Byron comtesse de Lovelace))

(defrule afficheMR
       (Mr $?param)
=>
       (printout t "Mr " $?param " est etudiant chez
   nous" crlf))

(defrule afficheMME
       (Mme $?param)
=>
       (printout t "Mme " $?param " est etudiante
   chez nous" crlf))
```

Contraintes (1/4)

- Il est possible de contraindre les valeurs d'une variable en partie gauche d'une règle
  - **~<val>** : vrai si le champ est différent de **val**
  - **<val1>|<val2>** : vrai si le champ est **val1** ou **val2**
  - **?x&<contrainte>** : vrai si la valeur de **x** vérifie la contrainte

```
(deffacts objets
       (objet chaise bleu)
       (objet table rouge)
       (objet bureau vert))

(defrule trouveObjetVertOuBleu
       (objet ?ob vert|bleu)
=>
       (printout t "l'objet " ?ob " est vert ou bleu" crlf))

(defrule trouveObjetNiRougeNiBleu
       (objet ?ob ?coul&~rouge&~bleu)
=>
       (printout t "l'objet " ?ob " est de couleur " ?coul crlf))
```

```
CLIPS> (run)
l'objet bureau est vert ou bleu
l'objet bureau est de couleur vert
l'objet chaise est vert ou bleu
CLIPS>
```


Contraintes (2/4)

- Les valeurs peuvent être contraintes par des prédicats
  - Tests de type : **(numberp <arg>)**, **(stringp <arg>)**, **(symbolp <arg>)**, ...
  - Egalité et inégalité : **(eq <arg1> <arg2>+)**, **(neq <arg1> <arg2>+)**
  - Comparaisons numériques : **(<predicat> <nb1> <nb2>+)** avec les prédicats **=**, **<>**, **>**, **<**, **<=**, **>=**
  - Fonctions booléennes : **(and <arg>+)**, **(or <arg>+)**, **(not <arg>)**
  - ...

Contraintes (3/4)

```
(deffacts gens
       (personne "lili" 2)
       (personne "toto" 23)
       (personne "titi" 54)
       (personne "tutu" "vieux")
       (personne "tata" "jeune")
       (personne "Mathusalem" "tres tres vieux"))

(defrule trouveVieux1
       (personne ?nom ?age&:(numberp ?age)&:(> ?age 50))
=>
       (printout t ?nom " est vieux" crlf))

(defrule trouveVieux2
       (personne ?nom ?age&:(stringp ?age)&:(neq FALSE (str-index "vieux" ?age)))
=>
       (printout t ?nom " est vieux" crlf))
```

```
CLIPS> (run)
Mathusalem est vieux
tutu est vieux
titi est vieux
CLIPS>
```


Contraintes (4/4)

- Contraintes existentielles et universelles
  - **(exists <conditional-element>+)**
  - **(forall <conditional-element> <conditional-element>+)**

```
(deffacts gens
       (personne "lili" 2)
       (personne "toto" 86)
       (personne "titi" 54)
       (personne "tutu" 67)
       (personne "tata" 23)
       (personne "Mathusalem" 969))

(defrule lesRetraitesPeuventEtreSauvees
       (exists (personne ? ?age&:(numberp ?age)&:(< ?age 50)))
=>
       (printout t "il y a au moins un jeune" crlf))

(defrule LesRetraitesSontFichues
       (forall (personne ?nom ?age) (personne ?nom ?age&:(numberp ?age)&:(> ?age 50)))
=>
       (printout t "tout le monde est vieux" crlf))
```

Priorité des règles (1/2)

- La priorité des règles (salience) peut être modifiée
  - par défaut la salience vaut 0
  - les valeurs possibles vont de -10000 à 10000
  - plus la salience est grande, plus la règle est prioritaire

```
(defrule trouveVieux1
       (declare (salience 100)) ;la règle utilisant les nombres est prioritaire
       (personne ?nom ?age&:(numberp ?age)&:(> ?age 50))
=>
       (printout t ?nom " est vieux" crlf))

(defrule trouveVieux2
       (declare (salience 10)) ;la règle utilisant les symboles n'est pas prioritaire
       (personne ?nom ?age&:(stringp ?age)&:(neq FALSE (str-index "vieux" ?age)))
=>
       (printout t ?nom " est vieux" crlf))
```

Priorité des règles (2/2)

- Dans un SE correctement développé, la priorité des règles sert à distinguer (par ordre de priorité décroissante)
  - Les contraintes : règles qui permettent de réduire le champ des hypothèses
  - Les règles du domaine : règles représentant l'expertise sur le domaine de connaissances
  - Les questions : règles permettant l'interrogation de l'utilisateur
  - Les règles de contrôles : règles servant à changer de phase de raisonnement
- En dehors de ces cas, **il ne faut pas abuser des modifications de priorité**

Modules (1/2)

- Il est possible de définir des modules de raisonnement en regroupant des faits et des règles
- Par défaut, la commande **`clear`** crée un module **`MAIN`** où se trouvent tous les éléments prédéfinis de CLIPS
- On donne la main à un module par **`(set-current-module <nom_module>)`** ou **`(focus <nom_module>)`**
- Un module peut exporter ou non son contenu (ou une partie de son contenu) et importer des éléments d'autres modules
- Pour l'exécution des règles, il est nécessaire d'importer initial-fact du module **`MAIN`**

Modules (2/2)

```
(defmodule MAIN (export ?ALL))

(defmodule testRetraites (import MAIN deftemplate initial-fact))

(defrule testRetraites::lesRetraitesPeuventEtreSauvees
       (declare (salience 1000))
       (exists (personne ? ?age&:(numberp ?age)&:(< ?age 50)))
=>
       (printout t "il y a au moins un jeune" crlf))

(deffacts testRetraites::gens
       (personne "lili" 2) (personne "toto" 86) (personne "titi" 54)
       (personne "tutu" 67) (personne "tata" 23) (personne "Mathusalem" 969))
```

```
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
For a total of 1 fact.
CLIPS> (focus testRetraites)
TRUE
CLIPS> (facts)
f-0     (initial-fact)
f-1     (personne "lili" 2)
f-2     (personne "toto" 86)
f-3     (personne "titi" 54)
f-4     (personne "tutu" 67)
f-5     (personne "tata" 23)
f-6     (personne "Mathusalem" 969)
For a total of 7 facts.
CLIPS> (run)
il y a au moins un jeune
CLIPS>
```

Fonctions (1/2)

- CLIPS définit en plus des fonctions de prédicat des dizaines de fonctions
  - Fonctions sur les listes
  - Fonctions sur les chaines de caractères
  - Fonctions mathématiques
  - Fonctions d'entrée/sortie
  - Fonctions structures de contrôle (conditionnelles, boucles, ...)
  - Fonctions sur les faits, sur les structures, sur les classes et objets, sur les règles
  - ...

Fonctions (2/2)

- Création de fonction : **`deffunction`**
  - La valeur retournée est l'évaluation de la dernière action (**FALSE** si aucune action n'est spécifiée ou si l'exécution ne réussit pas)

```
(deffunction carre (?x)
      (* ?x ?x))

(deffunction boucleCarre ()
      (bind ?i 0) ;affectation
      (while (neq ?i q) do
          (printout t "Donnez un nombre (q pour quitter) ")
          (bind ?i (read t))
          (printout t crlf)
          (if (numberp ?i) then (printout t "le carre est " (carre ?i) crlf))))
```

Filtrage des règles

- CLIPS construit un agenda des règles applicables
- L'agenda contient des associations entre un ensemble de faits et la règle qu'ils peuvent déclencher
- La construction de l'agenda se fait par filtrage des règles à l'aide de l'algorithme RETE
- RETE (réseau en latin) a pour principe la construction d'un réseau des prémisses des règles au travers duquel on filtre les faits

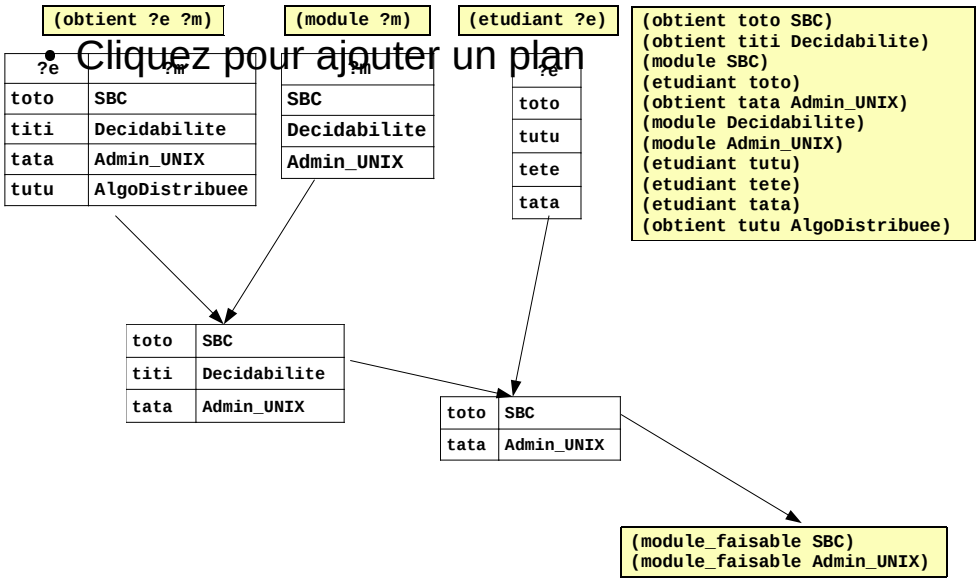
RETE (1/3)

- Pour chaque règle, un arbre de patterns est constitué à partir des premisses.
- Les faits vont « filtrer » à travers ce réseau

```
(deffacts etudes
       (obtient toto SBC)
       (obtient titi Decidabilite)
       (module SBC)
       (etudiant toto)
       (obtient tata Admin_UNIX)
       (module Decidabilite)
       (module Admin_UNIX)
       (etudiant tutu)
       (etudiant tete)
       (etudiant tata)
       (obtient tutu AlgoDistribuee))

(defrule faisable
       (obtient ?e ?m)
       (module ?m)
       (etudiant ?e)
=>
       (assert (module_faisable ?m)))
```

RETE (2/3)



- Cliquez pour ajouter un plan

RETE (3/3)

- RETE permet d'éviter l'itération systématique dans la base de faits
- Complexité
  - R : nombre de règles
  - F : nombre de faits
  - C : nombre max de conditions dans les règles
  - Meilleur cas : *O(logR)*
  - Pire cas : $O(R*F^C)$
  - En moyenne : *O(R*F)*

Agenda

- La commande **`(agenda)`** affiche l'agenda
- Dès qu'une règle a été déclenchée par un ensemble de faits (par **`run`**), l'association correspondante est supprimée de l'agenda (réfraction)
- Les associations sont triées :
  - selon le plus grand indice de leurs faits (recency)
  - selon le nombre de conditions de la règle (spécificité)
  - selon la priorité de la règle (salience)

Stratégies de sélection (1/2)

- **`(set-strategy <strategy>)`** permet de choisir la stratégie de sélection des règles parmi **`depth`**, **`breadth`**, **`simplicity`**, **`complexity`**, **`lex`**, **`mea`** et **`random`**
- **`depth`** - stratégie en profondeur (par défaut) : l'agenda est une pile et les règles y sont triées par priorité
- **`breadth`** - stratégie en largeur : l'agenda est une file et les règles y sont triées par priorité
- **`random`** : l'agenda est trié aléatoirement

Stratégies de sélection (2/2)

- **`simplicity`** : les règles sont triées par nombre de conditions croissant
- **`complexity`** : les règles sont triées par nombre de conditions décroissant
- **`lex`** - stratégie lexicographique : les règles sont triées selon l'indice max des faits associés (pour les règles de même indice, le nombre de conditions prime)
- **`mea`** - stratégie means-ends-analysis : les règles sont triées selon l'indice du premier fait associé (pour les règles de même indice, le nombres de condition prime)