Module: Mini Projet Intelligence Artificielle

2ème Année Filière Informatique

Références

- 2AInfo - Cours Programmation Fonctionnelle et Logique
- 2AInfo - Cours Intelligence Artificielle
- Manuel d'utilisation de Allegro Common Lisp
- Manuel d'utilisation de Prolog
- Manuel d'utilisation de Jess® The Rule Engine for the Java™ Platform -Version 7.1 - July 8, 2008
- Manuel « Jess in Action Rule-Based Systems in Java »

SOMMAIRE

1. Introduction-Organisation du travail
2. Partie Prolog: Aperçu sur Visual Prolog
3. Partie Lisp: Aperçu sur Allegro Common Lisp
4. Partie Jess: Aperçu sur Jess

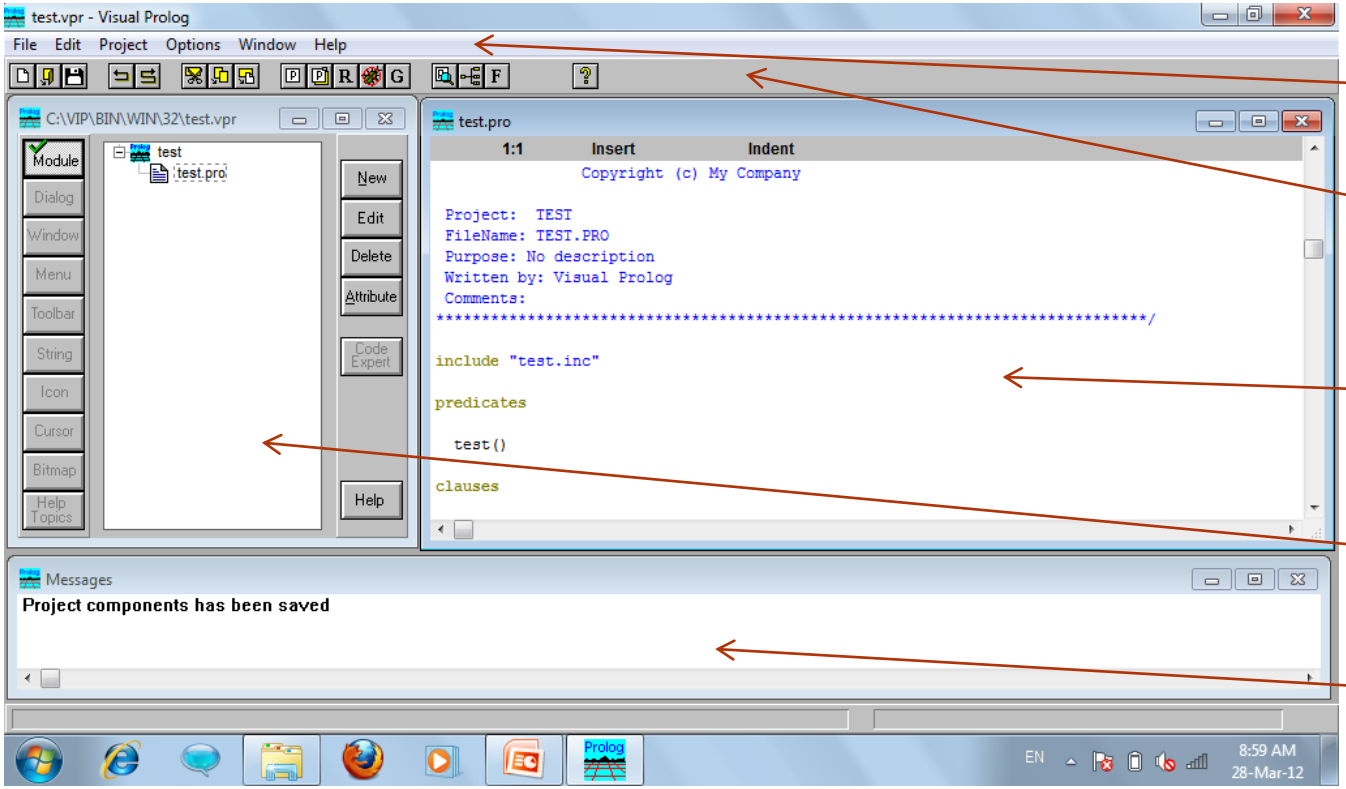
Introduction: organisation du travail

- Travail individuel,
- Tous vos fichiers dans dossier MP-IA sur Bureau,
- Compte rendu **à la fin de chaque séance (sauf semaine 6)**: programme + captures d'écrans d'exécution, fichier au format **PDF** avec pour nom: 2ainfo-**x-nometudiant**.pdf ,

Séance	Objectif	Poids dans la note du module
S1	Réalisation d'exercices en Prolog	15%
S2	Réalisation d'un mini projet en Prolog	20%
S3	Réalisation d'exercices en Lisp	15%
S4	Réalisation d'un mini projet en Lisp	20%
S5	Prise en main de Jess	10%
S6 – S7	Réalisation d'un mini projet en Jess	20%

Partie Prolog: Aperçu sur Visual Prolog

- V-Prolog est un langage de programmation multi paradigme basé sur la programmation logique (paradigmes logique, fonctionnel et objet),
- On va utiliser uniquement le paradigme de la programmation logique avec la version 5.2



Barre de menu

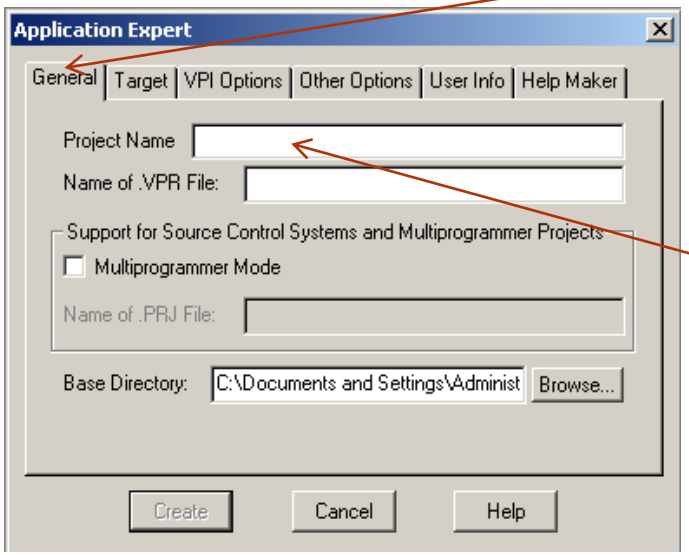
Barre de contrôle

Fenêtre d'édition

Fenêtre du projet

Fenêtre de Messages

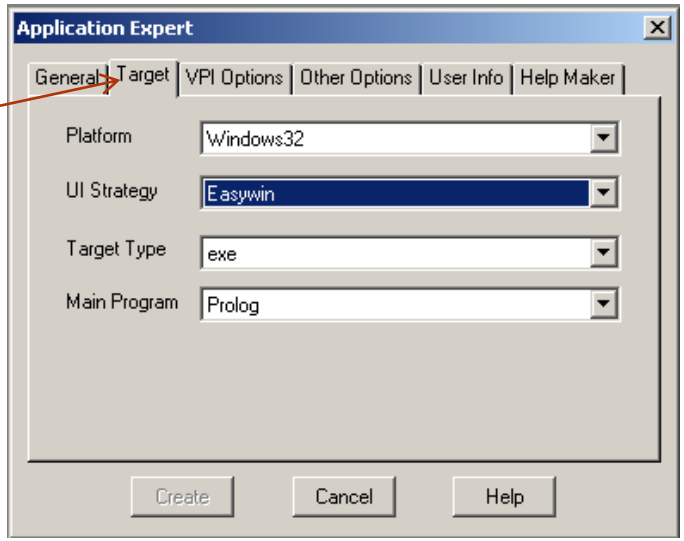
Partie Prolog: Aperçu sur Visual Prolog
création d'un nouveau projet



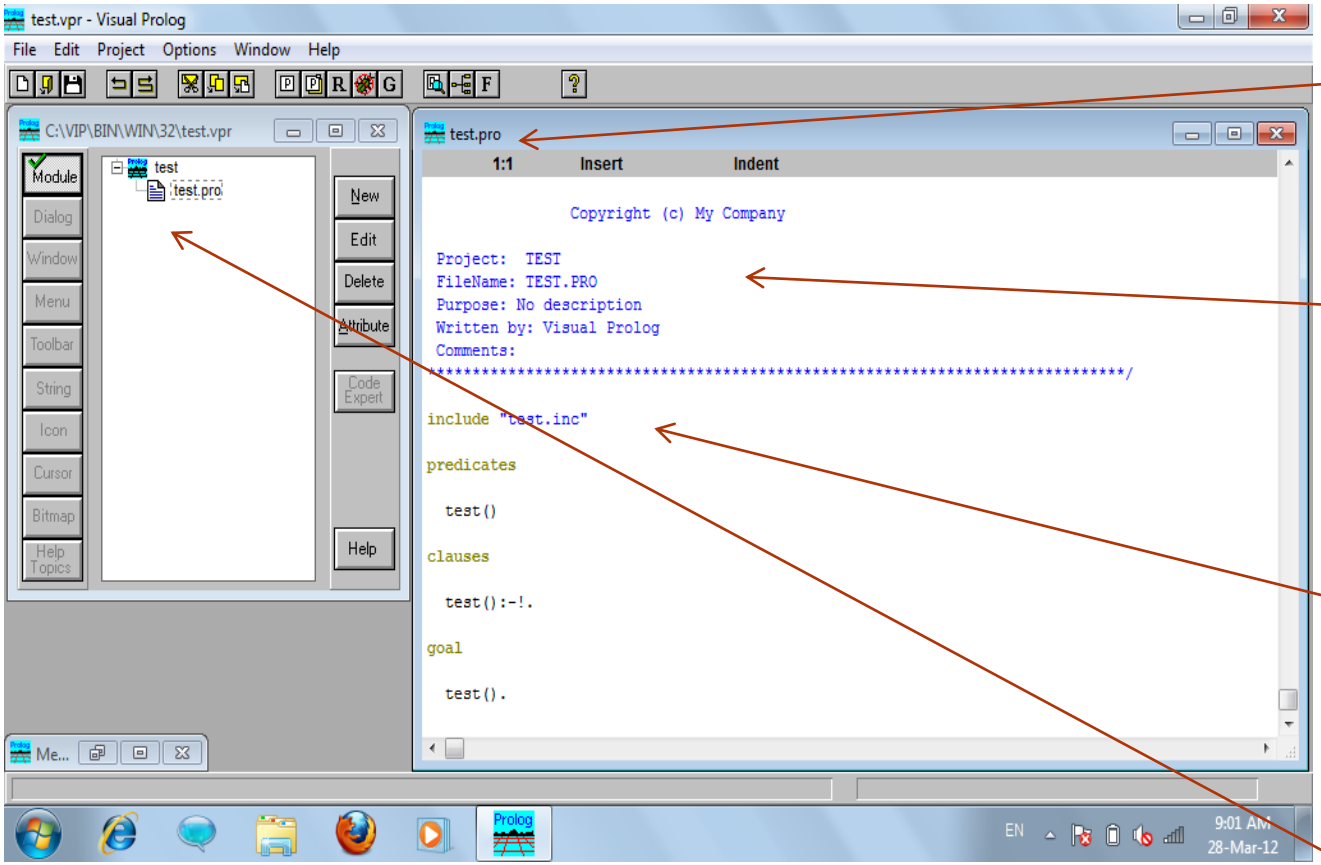
Menu General

Menu Target

Nom du projet



Partie Prolog: Aperçu sur Visual Prolog



Nom du fichier prolog en cours

Partie commentaire sur le projet en cours

Fenêtre d'édition:

zone declarations include, domains, predicates, clauses, goal,…

Fenêtre du projet

Partie Prolog: Aperçu sur Visual Prolog

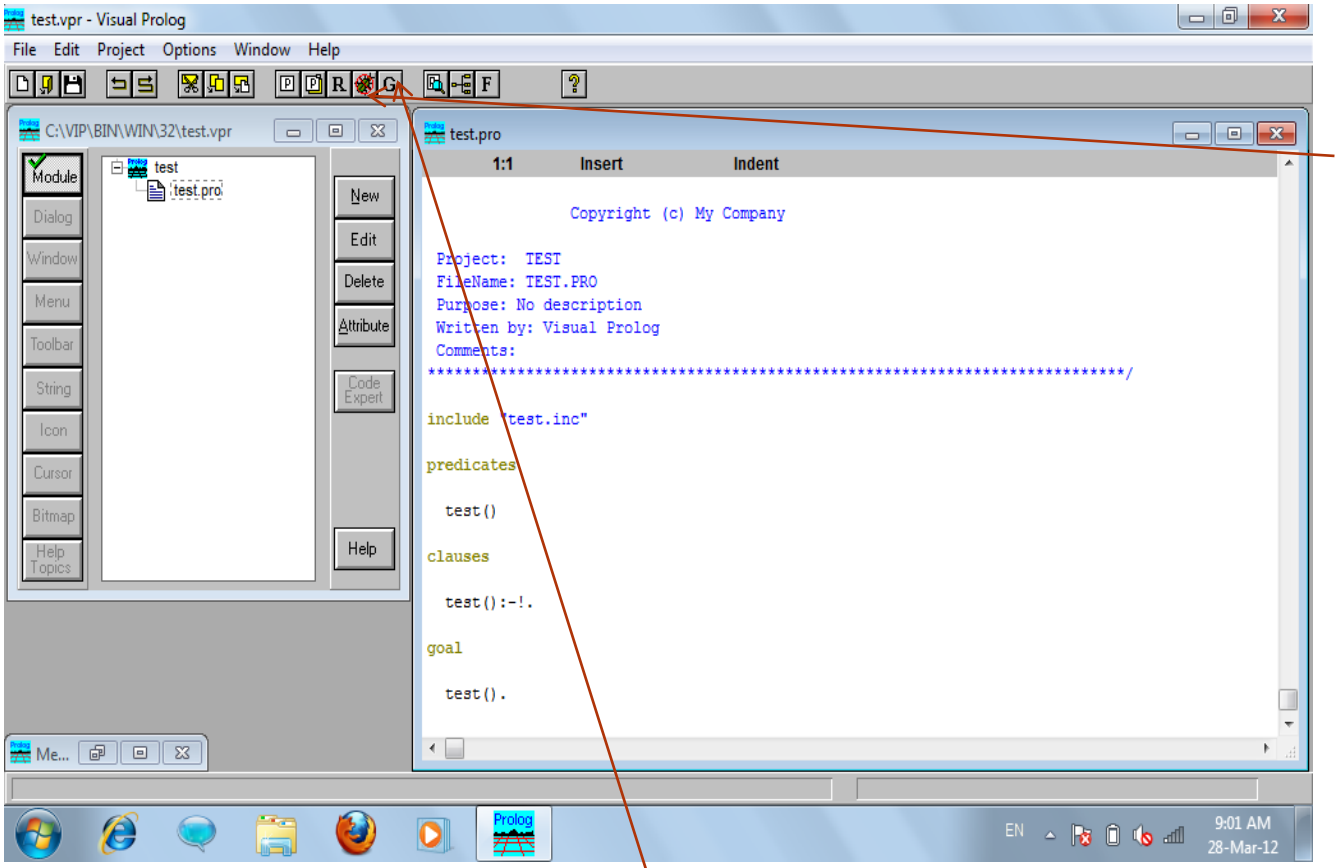
Exemple

```
DOMAINS
  personne, voiture, couleur = symbol

PREDICATES
  nondeterm voitureCouleur(voiture, couleur)
  nondeterm aimeCouleur(personne, couleur)
  nondeterm aprecitVoiture(personne, voiture)
  nondeterm peutAcheter(personne, voiture)
  nondeterm aVendre(voiture)
  nondeterm possedeArgent(personne)

CLAUSES
  aprecitVoiture(X, Y) :-
          aimeCouleur(X, C),
          voitureCouleur(Y, C).

  peutAcheter(X, Y) :-
          aprecitVoiture(X, Y),
          aVendre(Y),
          possedeArgent(X).
  voitureCouleur(c3, bleu).
  voitureCouleur(yaris, gris).

GOAL
  peutAcheter(ali, Y).
```

- Définition des types d'arguments,
- Définition des prédicats, de leurs arités et des arguments acceptables,
- Définition des clauses: par groupe de prédicats
- Définition du but

Partie Prolog: Aperçu sur Visual Prolog

Exécution d'un programme

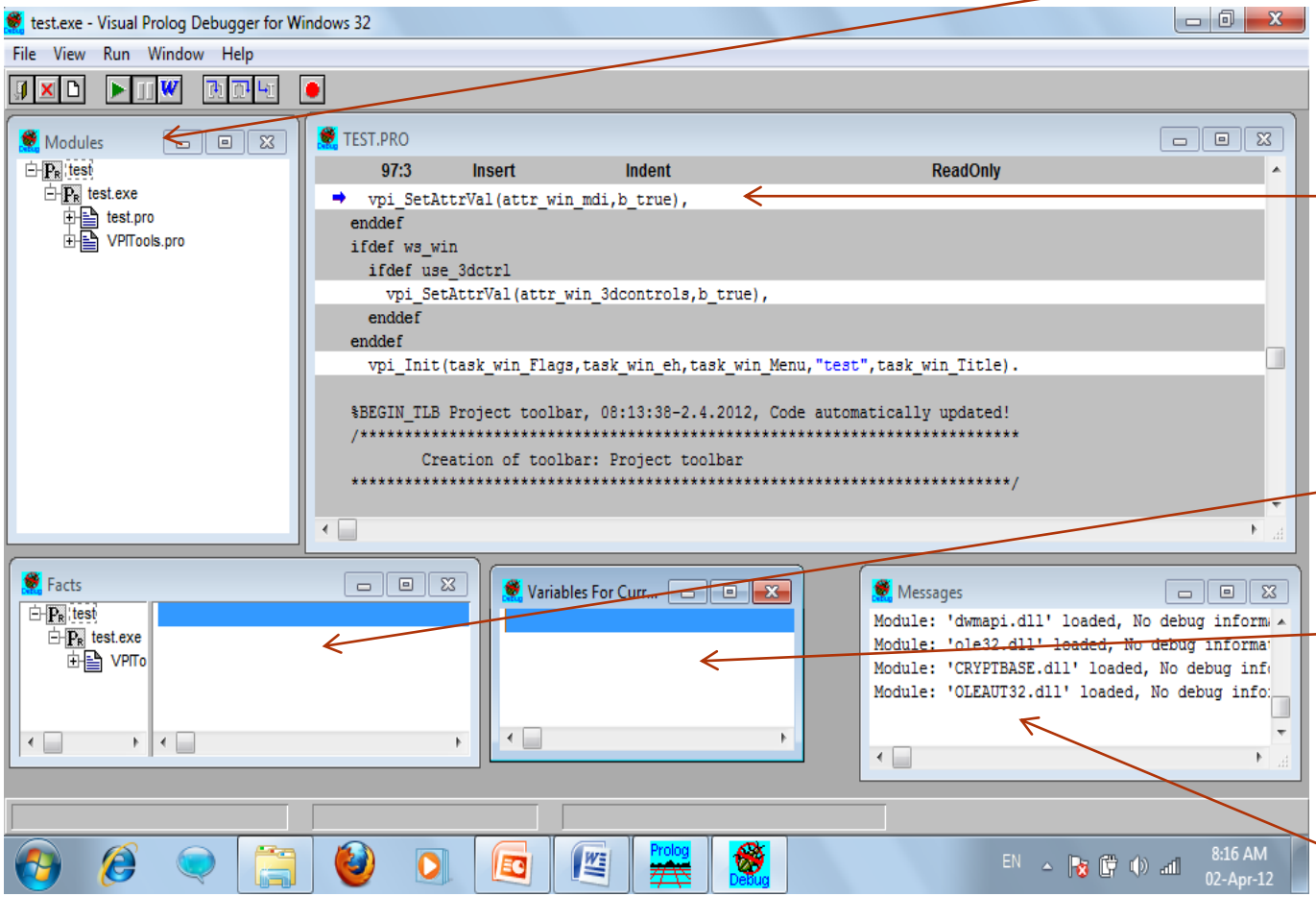


Lancement de la fonction debugage

Lancement de l'exécution du Goal

Partie Prolog: Aperçu sur Visual Prolog

Fonction Debugage



Fenêtre arborescence du projet

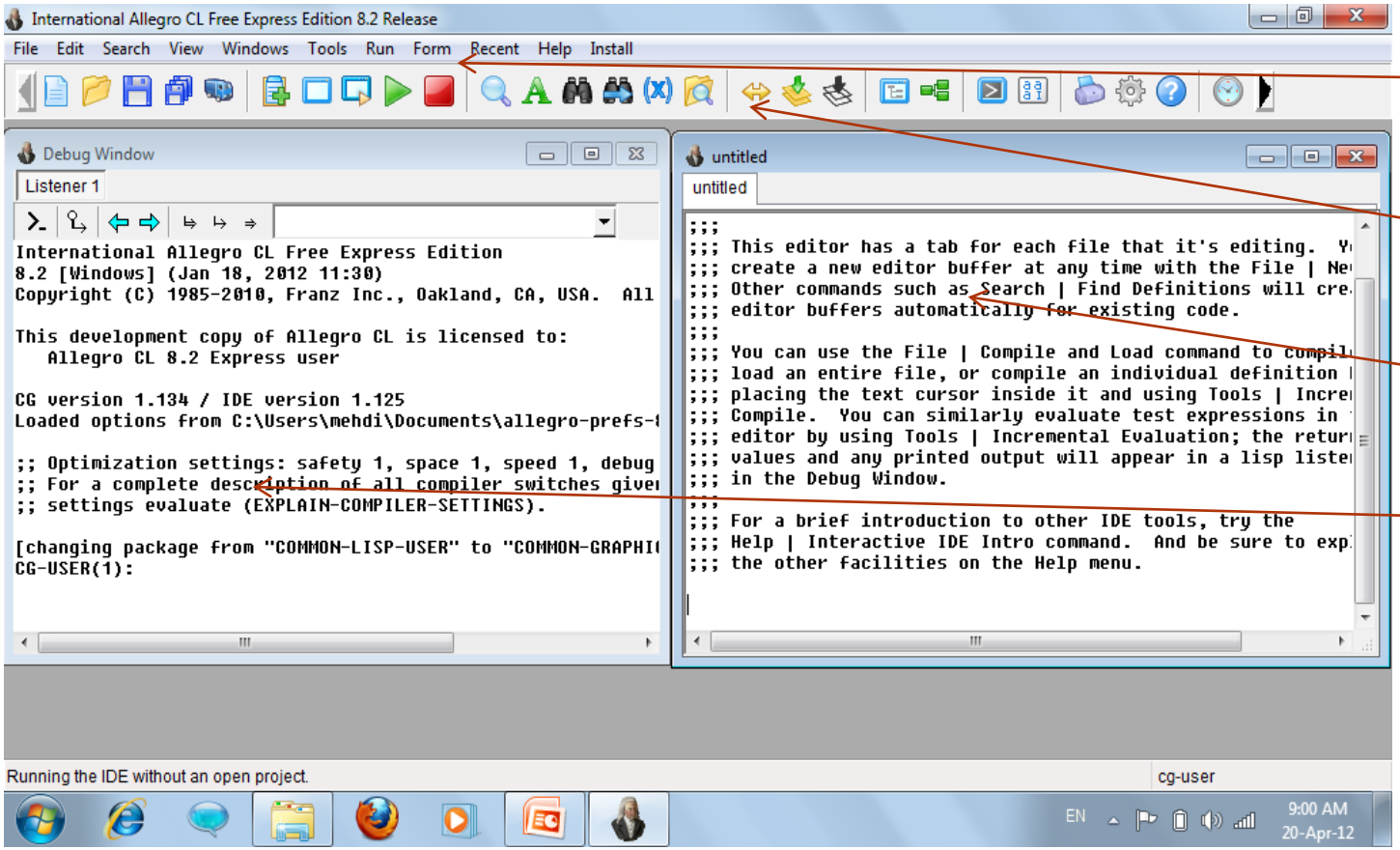
Fenêtre d'édition du fichier en cours avec position sur clause courante

Fenêtre de la situation des faits

Fenêtre de la situation des variables

Fenêtre de messages

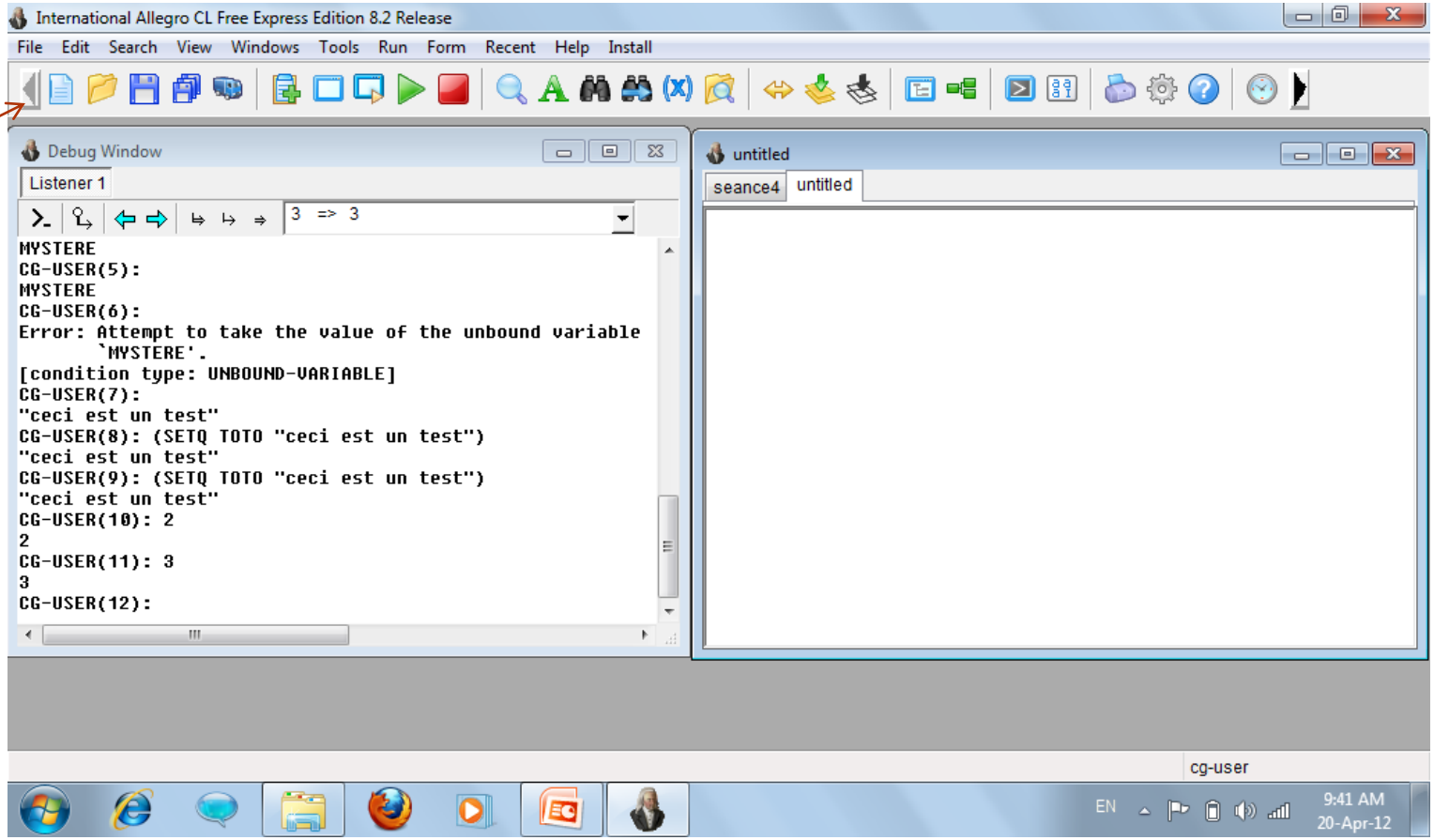
Partie Lisp: Aperçu sur Allegro Common Lisp: http://franz.com/support/documentation/10.0/doc/contents.htm



Partie Lisp: Aperçu sur ACL

Ouverture d'un nouveau fichier

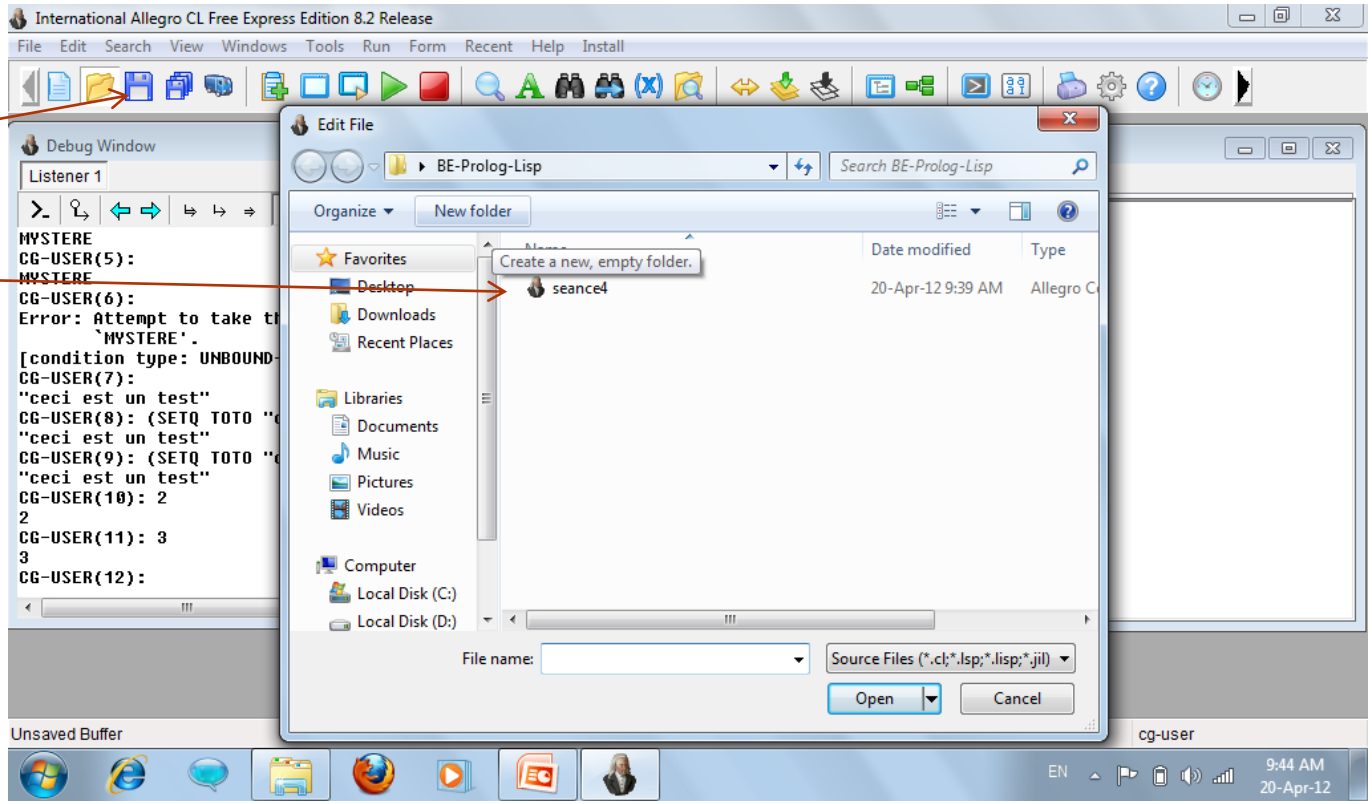
Ouverture d'un nouveau fichier



Partie Lisp: Aperçu sur ACL

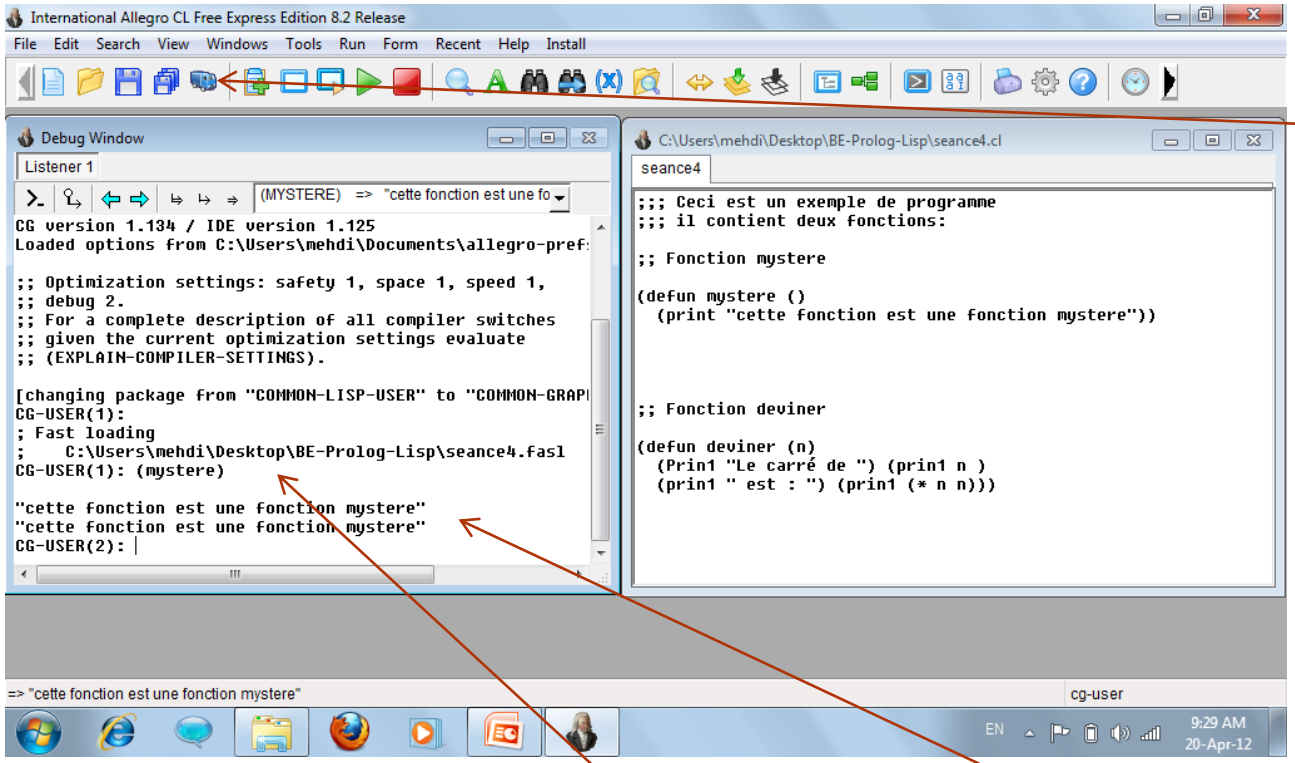
Ouverture d'un fichier existant

Ouverture d'un fichier existant



Partie Lisp: Aperçu sur ACL

Exécution d'un programme

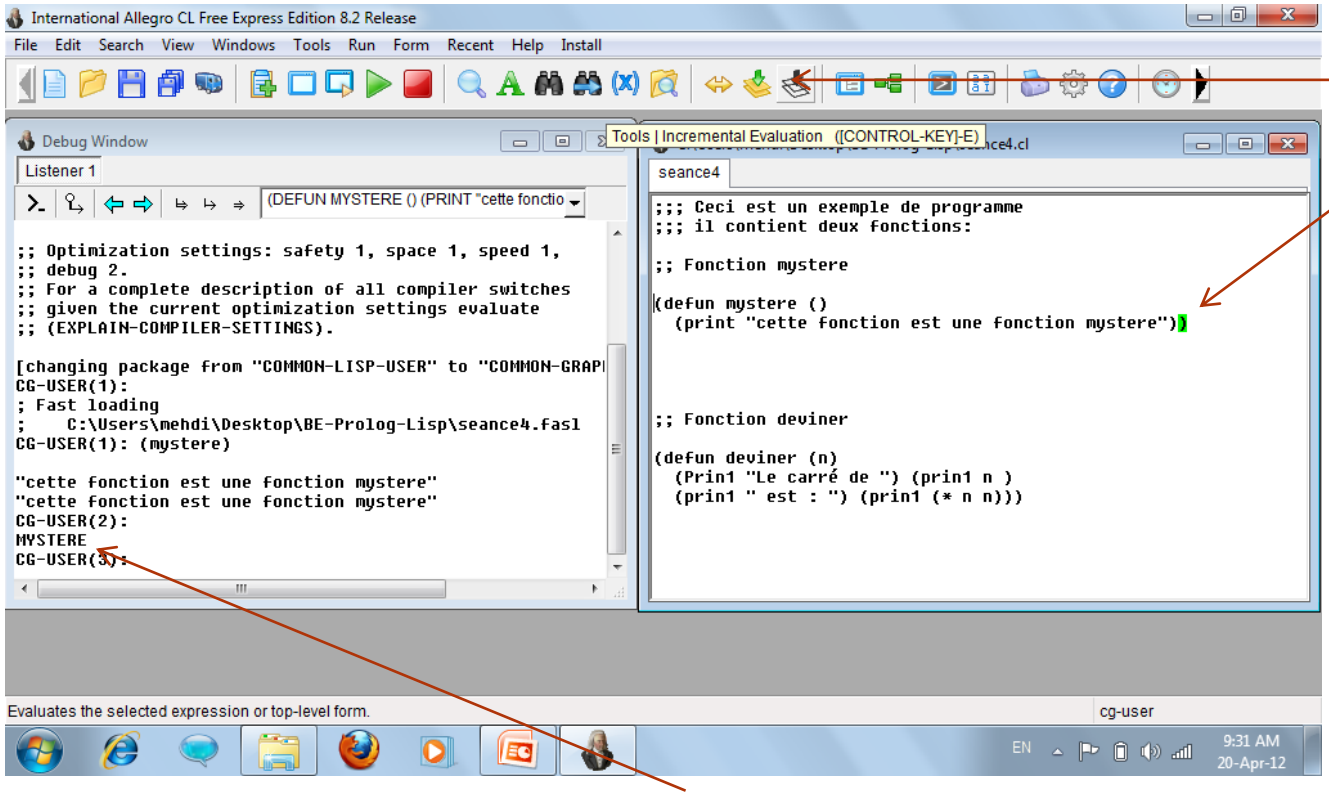


Chargement et compilation d'un programme

Message mentionnant le bon chargement

Exécution d'une fonction

Partie Lisp: Aperçu sur ACL
Évaluation incrémentale



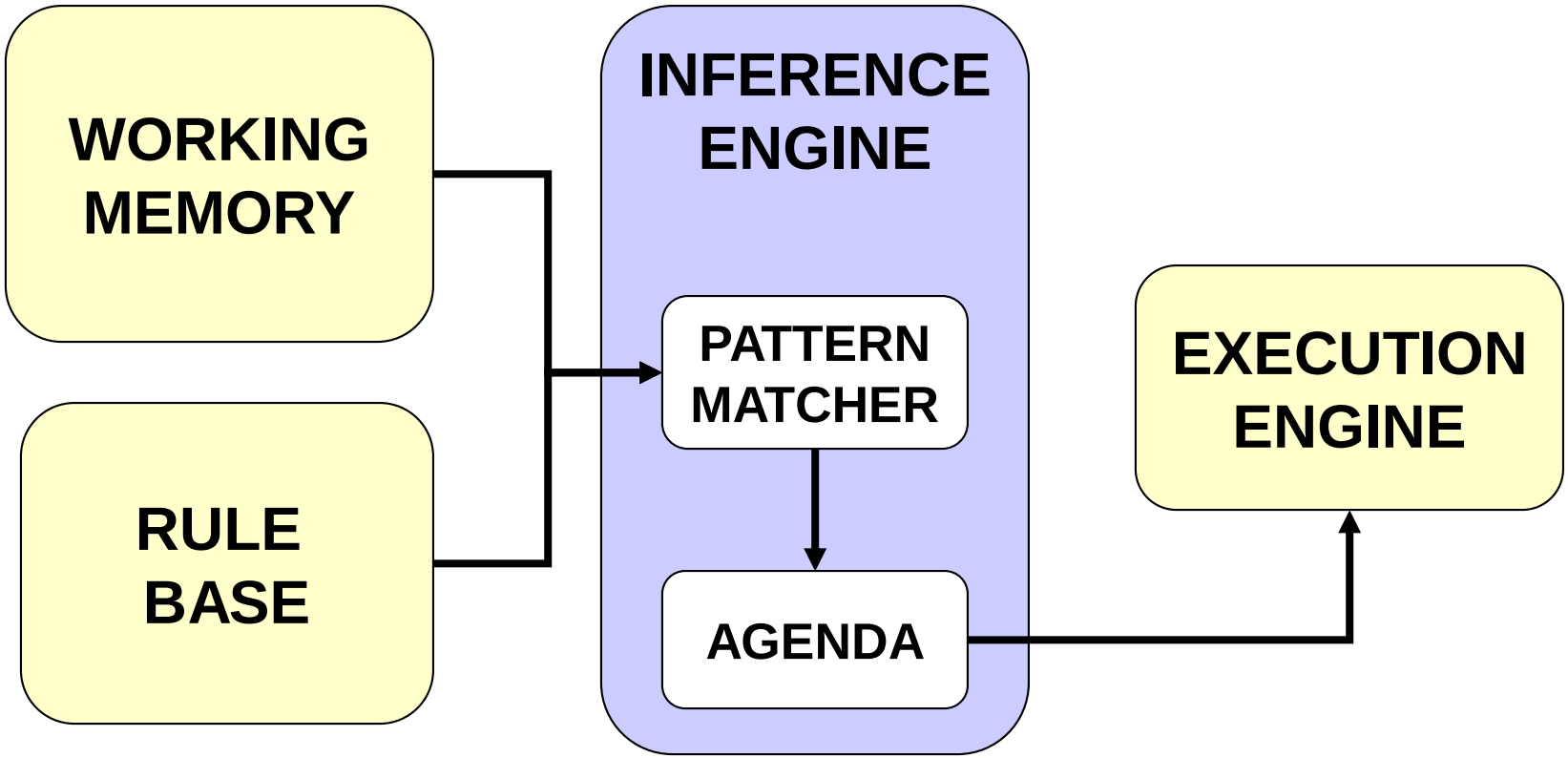
Chargement de la fonction courante pointée par le curseur dans l'éditeur

Message mentionnant le bon chargement de la fonction

Partie Jess: Introduction

- Développé à Sandia National Laboratories - USA durant les années 1990s.
- Créé par Dr. Ernest J. Friedman-Hill.
- Inspiré d'un outil bien connu CLIPS "C Language Integrated Production System" développé par les équipes de la NASA durant les années 1985s.
- Entièrement développé en Java API pour la création de Systèmes Experts à base de règles de production

Partie Jess: Architecture



Partie Jess: Architecture

- **Working Memory:** collection de faits qui définie la situation de la connaissance à tout instant,
- **Rule base:** ensemble des règles du système contenant ce qu'il faut faire sur la mémoire de travail 'working memory'
- **Pattern Matcher** : unité qui décide quelles règles déclencher et quand.
- **Agenda** : planifie l'ordre de déclenchement des règles activées.
- **Execution Engine** : unité responsable du déclenchement des règles et de l'exécution d'autres codes

Partie Jess: Architecture

Processus d'Inférence

- **Match**: met en correspondance les faits avec les règles,
- **Choose** : choisit les règles à déclencher.
- **Execute** : exécute les actions associées aux règles déclenchées

Partie Jess: Fonctionnement

- JESS met en correspondance les faits '**facts**' de la base des faits avec les règles '**rules'** de la base des règles.
- Les règles contiennent des appels de fonctions qui manipulent la base des faits et/ou d'autres codes Java
- JESS utilise l'algorithme **Rete** (ree-tee) **algorithm** pour la mise en correspondance '**matching**'.
- **Rete network** : une collection de nœuds interconnectés qui constitue la mémoire de travail 'working memory'

Partie Jess: Langage Jess

- Architecture inspirée de CLIPS
- Syntaxe à la LISP
- Structure de données de base: List
- Utilisable pour Java API Script
- Utilisable pour l'accès à JavaBeans
- « Facile » à apprendre et à utiliser

Partie Jess: Langage Jess

Votre premier programme JESS!

```
Jess>(printout t "Salut 2èmeInfo
  ENIT!" crlf)

Salut 2èmeInfo ENIT!
```

Partie Jess: Langage Jess

Quelques listes valides en Jess

- `(a b c) ; liste de symboles`
- `(1 2 3) ; liste d'entiers`
- `(+ 2 3) ; une expression`
- `("Salut 3èmeInfo ENIT!") ; une chaîne de caractères 'string'`
- `(foo ?x ?y) ; un appel de fonction`

Partie Jess: Eléments de base du Langage Jess

- Symboles :
  - Concept central dans tout langage de type LISP == identifiant
  - Contient lettres, chiffres et ponctuations ($,*,=,+,/,<,>,_,?,#, …..) **mais ne commence jamais par un chiffre**
  - Symboles spéciaux: nil, TRUE, FALSE
  - Exemple: truc, Truc, ma-valeur, test#1, _abc ,
- Nombres :
  - Entiers: 3, -1
  - et flottants: 4., 5.643
- Chaînes : "Hello" "Bonjour" crlf
- Listes : (+ 3 2) (a b c)
- Commentaires : ; exemple de commentaire

Partie Jess: Appels de fonctions

- Comme en Lisp tous les programmes en JESS se présentent comme des appels de fonctions qui sont aussi des listes,
- Un appel de fonction utilise une notation préfixée de type liste où la tête est un symbole identifiant une fonction:

```
Jess>(+ 2 3)
5
Jess> (+ (+ 4 5) (* 3 9))
36
```

```
Jess>(batch "c:\\program
   files\\jess71p2\\examples\\jess\\hello.clp")
Hello, World!
Jess>
```

Partie Jess: Les variables en Jess

- Les variables de programmation sont des symboles non typés qui commencent par « **?** »
- Le nom peut contenir toute combinaison de lettres, de nombres, et autres caractères sauf **(** et **)**
- Le type d'un symbole non typé peut changer durant sa durée de vie,
- Les affectations de valeurs se font par l'appel de la fonction **bind**

```
Jess>(bind ?x 2011)
2011
Jess> ?x
2011
```

Partie Jess: Les variables en Jess

- Les variables peuvent être locales ou globales,
- Les variables locales sont '*effacées*' de la mémoire après l'appel de la fonction **`reset`**
- Les variables globales doivent être déclarées en tant que tel,
- La fonction **`reset`** n'agit sur les variables globales que si on le souhaite, à travers la fonction **`set-reset-globals`**
- Autres types de variables intéressantes:
  - Les variables multi-champs « miltifield »

Partie Jess: Les variables en Jess

Les variables locales:

```
Jess>(bind ?x 2)
2
Jess>(bind ?y 3)
3
Jess>(bind ?result (+ ?x ?y))
5
Jess> ?result
5
Jess> (reset)
TRUE
Jess> ?result
…
```

Partie Jess: Les variables en Jess

Les variables globales:

```
Jess> (set-reset-globals nil)
FALSE
Jess>(defglobal ?*x* = 2)
TRUE
Jess> ?*x*
2
Jess>(bind ?*x* (+ 2 3))
5
Jess> ?*x*
5
Jess> (reset)
TRUE
Jess> ?*x*
5
```

Partie Jess: Les variables en Jess

Les variables globales:

```
Jess> (set-reset-globals TRUE)
TRUE
Jess>(defglobal ?*x* = 2)
TRUE
Jess> ?*x*
2
Jess>(bind ?*x* (+ 2 3))
5
Jess> ?*x*
5
Jess> (reset)
TRUE
Jess> ?*x*
2
```

Partie Jess: Les variables en Jess

Les variables « Multifield »:

- Ce type de variable est utilisé pour la définition de fonction avec un nombre arbitraire d'arguments. Il doit avoir un nom de la forme: $<nom>
- La valeur de ce type de variable est la liste composée des éléments qui sont mis en correspondance avec l'argument de la fonction,

```
Jess> (deffunction min ($?args)
        "Retourne la plus petite valeur parmi plusieurs"
        (bind ?minval (nth$ 1 ?args))
        (foreach ?n ?args
          (if (<  ?n ?minval) then (bind ?minval ?n))
        )
        (return ?minval))
TRUE
•Jess> (min 10 100 77 6 43)
6
```

Partie Jess: Le contrôle en Jess

- **Structures standards:**
  - **while**
  - **foreach**
  - **if/then/else**

```
Jess>(bind ?x 2011)
2011
Jess> (while (< ?x 2017)
          (printout t "Année " ?x crlf)
          (++ ?x))
Année 2011
Année 2012
Année 2013
Année 2014
Année 2015
Année 2016
FALSE
```

Partie Jess: Les fonctions en Jess

- Les fonctions peuvent être utilisées: en **partie action** d'une règle, ou en **ligne de commande**, ou dans **la définition d'une autre fonction**
- Structure de définition d'une fonction spécifique:

(deffunction <function-name>

(<parametres>) [<commentaires>] <expr> [<return-specifique>])

- Un seul argument de fonction peut être une variable multifield; le dernier;

```
Jess> (deffunction get-input ()
                "Attend une entrée de l'utilisateur  depuis la console."
                (bind ?s (read))
                (return ?s))
TRUE
Jess> (get-input)
.....
```

Partie Jess: Les fonctions en Jess

Exemple d'utilisation

```
Jess>  (deffunction aire-sphere (?rayon)
                    "Calcule l'aire d'une sphere"
                    (bind ?aire (* (* (pi) 4)(* ?rayon ?rayon)))
                    (return ?aire))
TRUE
Jess>  (printout t
            "La superficie d'une sphère de rayon = 2 mètres est "
            (aire-sphere  2)  " m^2")
.....
```

L'interface graphique en Jess

- Jess est un ensemble de classes Java,
- Jess est facile à ajouter comme bibliothèque dans les applications graphiques écrites en Java,

- Exemple de la section 13 « Creating Graphical User Interfaces in the Jess Language » du document "Jess® The Rule Engine for the Java™ Platform: quand un bouton "Hello" est pressé , la chaîne « Hello, word! » s'afficher sur la sortie standard,

Partie Jess: Les faits en Jess

- Les Faits '**facts**' sont les entités qui constituent la mémoire de travail d'un Jess SE ;
- La mémoire de travail contient par défaut au moins un fait:

  f-0 (MAIN::initial-fact)

- Chaque fait a un **template** similaire à la notion de:
  - classe dans Java
  - table dans une base de données
- Chaque template dispose d'une entête '**head**' et un ou plusieurs champs '**slots**' dont le fait associé en dispose;
- Les Slots contiennent les données relatives au fait (qui peuvent être typées).
- Les valeurs des slots peuvent être changées durant l'exécution.
- Les **Multislots** sont une variante de slot qui peuvent contenir des valeurs de type listes

Partie Jess: Les fonctions en Jess

- **deftemplate** ne peut être utilisée qu'au **toplevel d'un programme Jess**

(**deftemplate** ***template-name***
  [extends ***template-name*****]**
  ["Documentation comment"]
  [(**declare** (slot-specific **TRUE | FALSE**)
    (backchain-reactive **TRUE | FALSE**)
    (from-class *class name*)
    (include-variables **TRUE | FALSE**)
    (ordered **TRUE | FALSE**))]
  (**slot | multislot** *slot-name*
    ([(type **ANY | INTEGER | FLOAT | NUMBER | SYMBOL | STRING | LEXEME | OBJECT | LONG)]**
    [(default *default value*)]
    [(default-dynamic *expression*)]
    [(allowed-values *expression*+)])*)

Partie Jess: Les types de faits en Jess

- **Ordered** :
  - entête seulement; un seul slot
- **Unordered** :
  - slots multiples à l'image des enregistrements dans les BDD
- **Shadow** : (pont vers objets JavaBean)
  - slots correspondent à des propriétés de JavaBean,
  - ces faits permettent à JESS de raisonner à propos de choses qui se passent en dehors de la mémoire de travail,
  - contiennent de facto un slot nommé object,

Partie Jess: Manipulation des faits en Jess

- Certaines fonctions permettent de manipuler les faits en **mémoire de travail**:
  - **reset:** efface de la mémoire de travail tous les faits et les activations des règles et réinsère le **fait initial** et tous les faits de ***deffacts*** et éventuellement les variables globales.
  - **facts:** pour connaître tous les faits de la mémoire de travail,
  - **assert:** pour insérer un fait,
  - **retract**: pour enlever un fait de la mémoire de travail,
  - **modify**: pour modifier un fait de la mémoire de travail,
  - **add**: crée un shadow fact ,
  - **deffacts**: crée plusieurs faits en même temps définis dans une liste,
  - **remove**: **retract** de la mémoire de travail tous les faits utilisant le **template** en question,
  - **list-deftemplates:** retourne les templates de la memoire de travail,

Partie Jess: Manipulation des faits en Jess

Exemple d'utilisation

```
Jess> (deftemplate automobile
            "une voiture spécifique"
                  (slot marque)
                  (slot modele)
                  (slot age (type INTEGER))
                  (slot couleur (default blanche)))
TRUE
Jess> (reset )
Jess> (assert (automobile (modele A4) (marque Audi)
         (age 2000)))
<Fact-1>
Jess> (facts)
f-0    (MAIN::initial-fact)
f-1    (MAIN::automobile (marque Audi) (modele A4)
                         (age 2000) (couleur blanche))
For a total of 2 facts in module MAIN.
```

Partie Jess: Manipulation des faits en Jess

Exemple d'utilisation (suite)

```
Jess> (watch all)
TRUE
Jess> (retract 1)
TRUE
Jess> (facts )
f-0    (MAIN::initial-fact)
For a total of 1 facts in module MAIN.
Jess> (assert (automobile (modele A4) (marque Audi)
         (age 2000)))
<Fact-1>
Jess> (facts)
f-0    (MAIN::initial-fact)
f-1    (MAIN::automobile (marque Audi) (modele A4)
                         (age 2000) (couleur blanche))
For a total of 2 facts in module MAIN.
Jess> (reset )
```

Partie Jess: Manipulation des faits en Jess

Exemple d'utilisation (suite)

```
Jess> (deftemplate voiture-location
        extends automobile
        "une voiture de location"
        (multislot proprietaire)
        (slot kilometrage (type INTEGER)))
TRUE
Jess> (bind ?x (assert (voiture-location (modele Clio) (marque
Renault)(kilometrage 100))))
<Fact-2>
Jess> (modify ?x (kilometrage 300))
<Fact-2>
Jess> (facts)
f-0    (MAIN::initial-fact)
f-1    (MAIN::automobile (marque Audi) (modele A4)
                         (age 2000) (couleur blanche))
f-2    (MAIN::voiture-location (marque Renault) (modele clio)
                   (age nil) (couleur blanche) (proprietaire())
                   (kilometrage 300))
For a total of 3 facts in module MAIN.
```

Partie Jess: Manipulation des faits en Jess

Les faits '**shadow facts**' sont des faits non ordonnés '**unordered**' dont les slots correspondent à des propriétés de JavaBean:

- **Defclass: crée un *deftemplate* à partir de bean, peut être utilisé dans un programme, mais avec moins de possibilités que deftemplate,**

```
Jess> (import java.util.Calendar)
TRUE
Jess> (defclass calendrier Calendar)
Java.util.Calendar
Jess>
```

Partie Jess: Les règles en Jess

- Constituent la **Base de Connaissance** du système,
- Se déclenchent une seule fois relativement à un ensemble donné de faits,
- Utilisent les **pattern constraints** pour la mise en correspondance avec les faits
- Sont beaucoup plus rapides que les constructions **IF-THEN**.

Partie Jess: Les règles en Jess

- Une règle dispose de:
  - côté gauche: ‘Left Hand Side’ (LHS)
  - côté droit: ‘Right Hand Side’ (RHS),

- LHS contient des faits correspondant à certains patterns,
- RHS contient des appels de fonctions

Partie Jess: Les règles en Jess

- Création d'une règle par appel de la fonction **defrule**,
- La partie LHS **doit** pouvoir être mise en correspondance avec **des faits** et **ne peut être un appel de fonctions**,

```
Jess> (defrule bienvenue-retraite
        "clin d'oeil spécial pour les personnes accédant à la retraite"
        (personne {age > 60})
        =>
        (printout t "Bonjour, nous vous souhaitons une bonne
retraite!" crlf))
Jess>
```

Partie Jess: Les règles en Jess

Exemple d'utilisation

```
Jess> (watch all)
Jess> (deftemplate personne (slot prenom) (slot nom) (slot age))
Jess> (reset)
f-0    (MAIN::initial-fact)
TRUE
Jess> (defrule bienvenue-retraite
          "clin d'oeil spécial pour les personnes accédant à la retraite"
          (personne {age > 60})
          =>
          (printout t "Bonjour, nous vous souhaitons une bonne retraite!" crlf))
Bienvenue-retraite: +1+1+1+t
Jess> (assert (personne (nom Tounsi) (prenom Ali) (age 65)))
f-1    (MAIN::personne (prenom Ali) (nom Tounsi) (age 65))
Activation: MAIN:: bienvenue-retraite : f-1
Jess> (run)
Bonjour, nous vous souhaitons une bonne retraite!
<== Focus MAIN
1
Jess> (run)
```

Partie Jess: Les règles en Jess

- Le déclenchement effectif d'une règle dépend de sa priorité: critère de résolution de conflits,
- Chaque règle a une propriété nommée « **salience** » qui définie sa priorité dans le déclenchement parmi les règles déclenchables,
- Pour forcer certaines règles à être déclenchées les premières, on introduit dans la règle une déclaration « salience » avec une valeur haute. Une telle approche ne doit être utilisée que quand c'est nécessaire: exemple des feux de croisement,
- La valeur par défaut est zéro, la valeur peut être INTEGER, variable globale,

```
Jess> (defrule bienvenue
        "Souhaiter la bienvenue à la première personne définie"
        (declare (salience 100))
        (personne {age >= 0})
        =>
        (printout t "Bienvenue !"  crlf))
Jess>
```

Partie Jess: Les règles en Jess

- Si égalité dans la priorité « salience » des règles, JESS dispose de deux stratégies de résolution de conflits:
  - **Depth strategy**: la règle la plus récemment activée sera déclenchée avant toutes les autres, *c'est la stratégie par défaut*
  - **Breadth strategy**: les règles sont déclenchées selon l'ordre de leur activation (de la plus ancienne à la plus récente),
  - **(set-strategy <strategy-name>):** avec l'argument *depth* ou *breadth*
  - **(get-strategy)**: retourne la stratégie courante

- La liste des règles activées et non encore déclenchée peut être connue à travers la fonction: **agenda**,

```
Jess> (agenda )
Jess>
```

Partie Jess: Les motifs des règles en Jess

- Un motif « pattern » est une liste dont la tête est le nom d'un fait, suivie éventuellement des descriptions des slots

```
(personne (age ?x) (prenom ?y) (nom ?z))
```

- Ce motif peut être mis en correspondance avec tout fait personne
- Lors du déclenchement d'une règle, Jess affecte le contenu du slot:
  - age à ?x
  - prenom à ?y
  - nom à ?z
- Ces variables peuvent être utilisées n'importe où dans la même règle

```
?personne <- (personne {age >10} && {prenom != "ali"} (nom ?z))
```

- Les operateurs booléens sont: == « EGAL », && « ET », || « OU », ……

Partie Jess: Les motifs des règles en Jess

Exemple d'utilisation

```
Jess> (defrule adulte
        ?ad <- (personne {age >= 20} (nom ?nom))
       =>
       (printout t ?nom "est  âgé de " ?ad.age  " années " crlf))
```

```
Jess> (deftemplate point-coordonnees (slot x) (slot y) (slot nom))
Jess> (reset)
Jess> (assert  (point-coordonnees (x 20) (y 5) (nom point-depart)))
Jess> (assert  (point-coordonnees (x 50) (y 10) (nom point-arrivee)))
Jess> (defrule lecture-coordonnees
        (point-coordonnees (x ?x) (y ?y) (nom ?n))
        =>
        (printout t  " Le point " ?n " a pour abscisse " ?x  " et pour
ordonnée "  ?y crlf))
Jess> (run)
```

Partie Jess: Les motifs des règles en Jess

- Un motif « pattern » peut contenir des contraintes à satisfaire :
  - Contraintes de prédicat:

```
; mise en correspondance de faits dont la valeur d'un slot satisfait une contrainte:

(point-coordonnees (x ?x&:(> ?x 10)))
```

  - Contraintes de valeur:

```
; mise en correspondance de faits dont la valeur d'un slot est égale à une valeur connue ou rendue par un appel de fonction:

(point-coordonnees (x =(+ 2 10)))
```

Partie Jess: Stratégies d'inférence

- Les règles vues jusqu'ici sont utilisées dans le sens du chainage avant : du LHS vers RHS, si LHS est vérifié, on exécute le RHS,
- Jess autorise aussi le chainage arrière ou recherche guidée par le but,
- Pour faire du chainage arrière avec Jess, il faut le déclarer en avance avant même de déclarer les règles qui utilisent ce type de faits, soit:
  - dans la définition du **template** du fait en question, en activant l'option "***backward chaining reactive***",
  - En le déclarant par la fonction:

(**do-backward-chaining** fact)

Partie Jess: Stratégies d'inférence

- Les règles qui utilisent des faits avec option "***backward chaining reactive***", sont traités par l'interpréteur de façon spéciale:

Exemple:

```
Jess>  (do-backward-chaining factoriel)
Jess>  (defrule print-factoriel-10
         (factoriel 10 ?r1)
        =>
         (printout t "La factorielle de 10 est " ?r1 crlf))
```

Partie Jess: Stratégies d'inférence

- En traitant cette règle, l'interpréteur va insérer dans la mémoire de travail un fait fictif: (need-factoriel 10 nil)

```
Jess>  (defrule do-factoriel
       (need-factoriel ?x ?)
       =>
       (bind ?r 1)  (bind ?n ?x)
       (while (> ?n 1)
           (bind ?r (* ?r ?n))
              (bind ?n (- ?n 1)))
              (assert (factoriel ?x ?r)))
Jess  >  (reset)
Jess  > (watch all)
Jess  > (run)
.....
```

Amusez-vous bien!!

Bon travail