Master of Technology in Knowledge Engineering

KE4102: Intelligent Systems & Techniques for Business Analytics: Introduction to CLIPS

Charles Pang
Institute of Systems Science
National University of Singapore
E-mail: charlespang@nus.edu.sg

Objectives

- To familiarize students with rule-based programming
- To develop skills and proficiency in rule-based programming using CLIPS shell
- To enable students to carry out the Continuous Assessment (CA) project using CLIPS shell

What is CLIPS ?

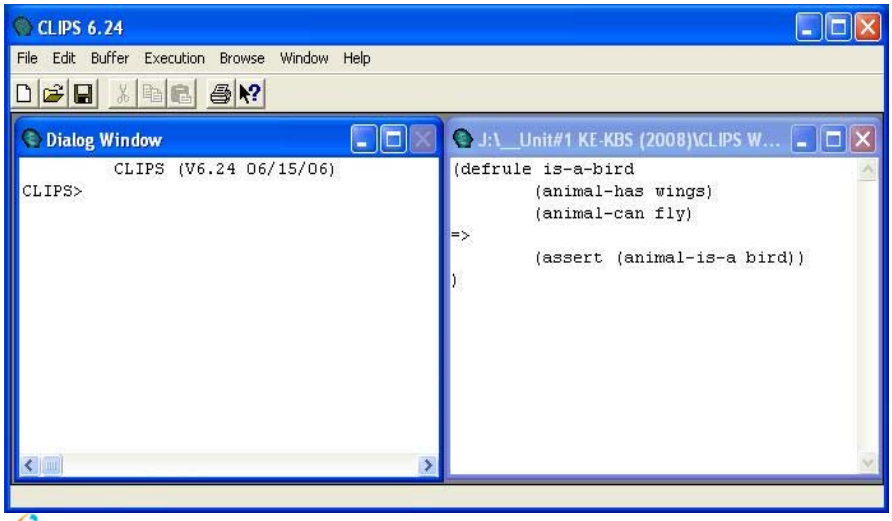
- C Language Integrated Production System (CLIPS)
- An RBS tool written in C
- Inference: Forward chaining
- Developed by A.I. Section, Johnson Space Centre, NASA (1985)
- Runs on PC, MAC, UNIX, VAX VMS
- Top-level interpreter
- Descendants: JESS, FuzzyCLIPS

What is CLIPS ? (cont'd)

- CLIPS is being used by numerous users throughout the public and private community including all NASA sites and branches of the military, numerous federal bureaus, government contractors, universities, and many private companies.
- Now maintained as public domain software by the main program authors who no longer work for NASA
- CLIPS is free!

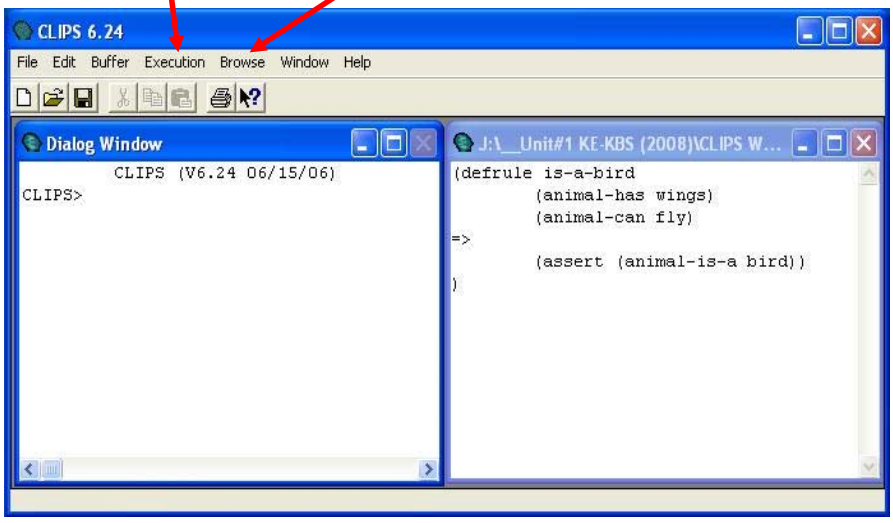
CLIPS programming environment

- CLIPS consists of:
  - Editor – you can edit your program code
  - Interpreter – you can load and run your program

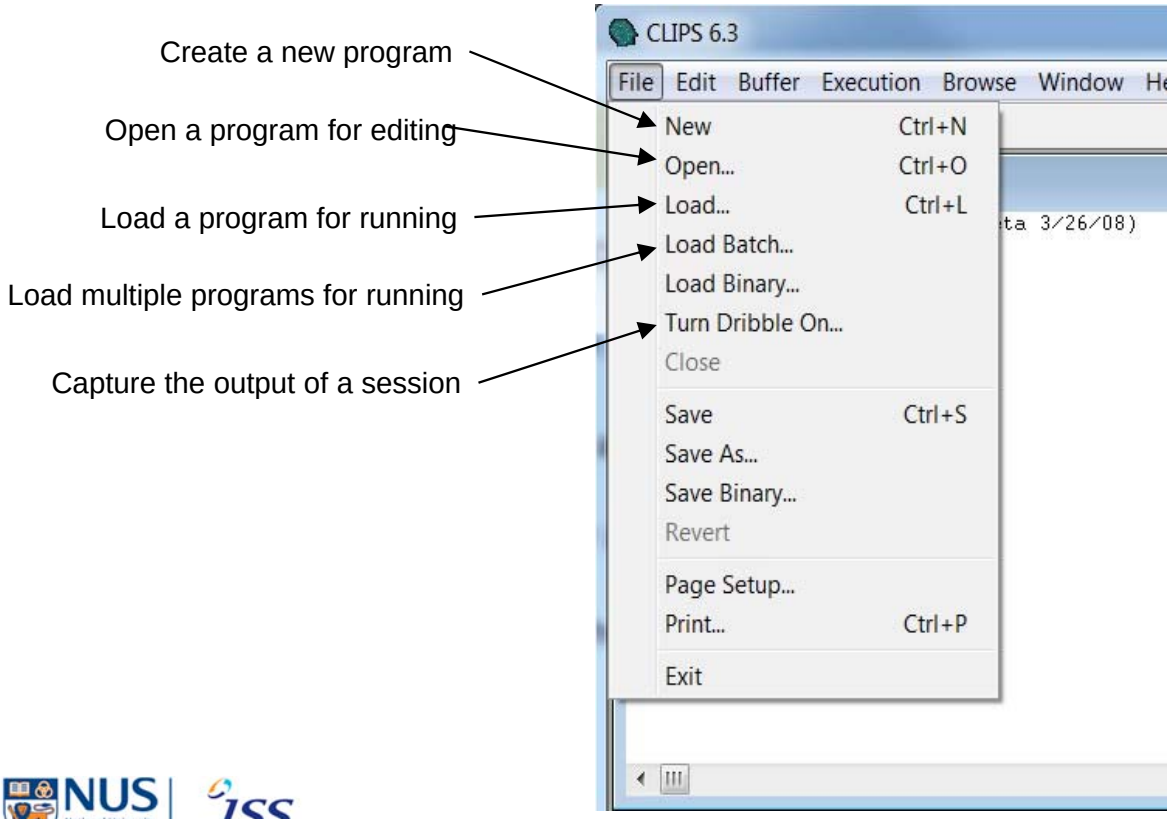


CLIPS programming environment cont'd

Debugging utilities Constructs viewing utilities



Basic I/O



Basic I/O

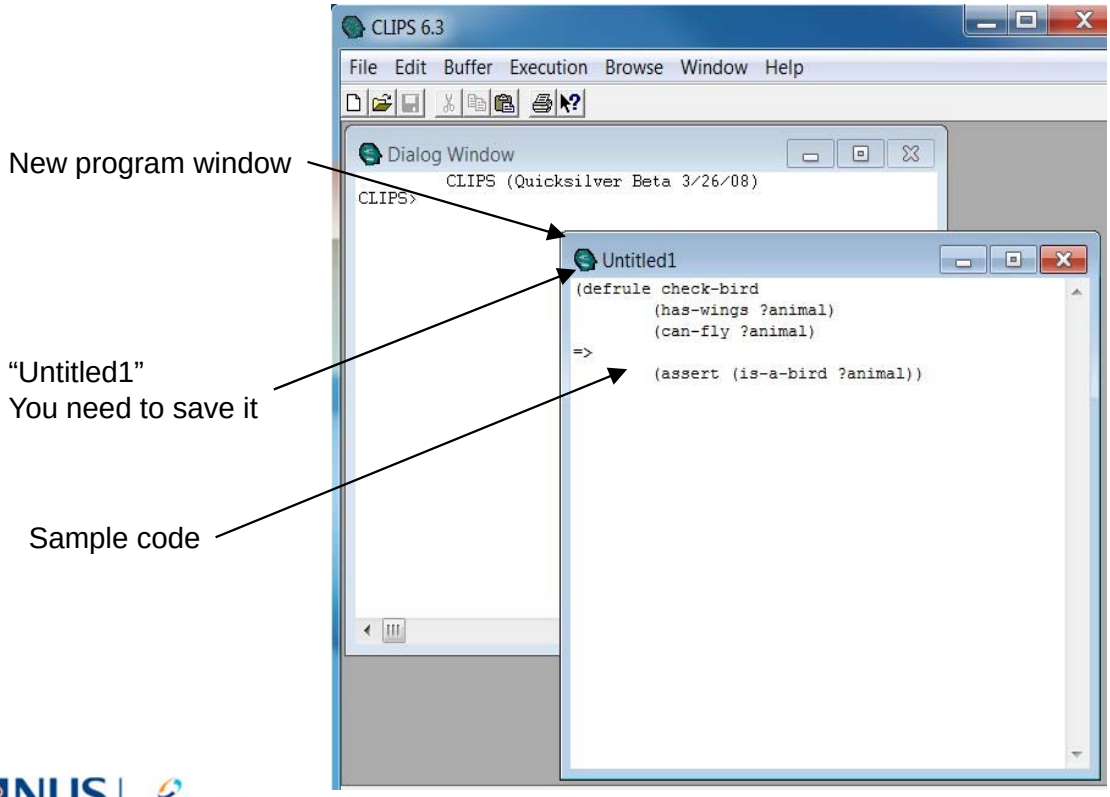
- To load a program file:

```
File->Load …  /* OR */
CLIPS> (load "myprogram.clp")
```

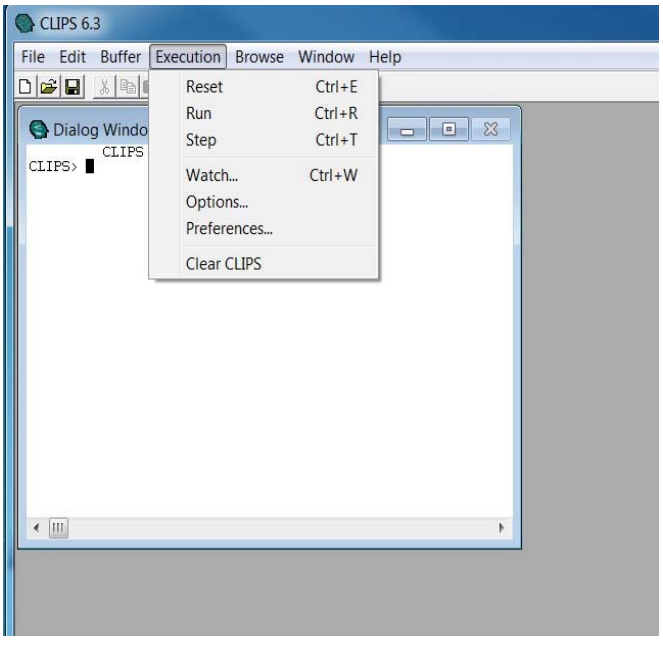
- To load more than one file:
  - Create a batch file – myprograms.bat
  - In CLIPS environment, use **File->Load Batch**
- To capture your interpreter session:

```
File->Turn Dribble on
```

Creating a new program



Running Clips program



Important Basic Commands

- Before running your program, you need to add facts into working memory using the "reset" command:

```
Execution->Reset
Execution->Run
```

- Reset command:-

  (1) It removes existing facts from WM, and activated rules from the agenda

  (3) It asserts facts from existing (deffacts) statements

- Reset is used to re-run your program with new inputs

Important Basic Commands

- To clean out the working memory (e.g. after you have modified your code):

```
Execution->Clear Clips
```

- To view loaded facts in WM:

```
CLIPS> (facts)
```

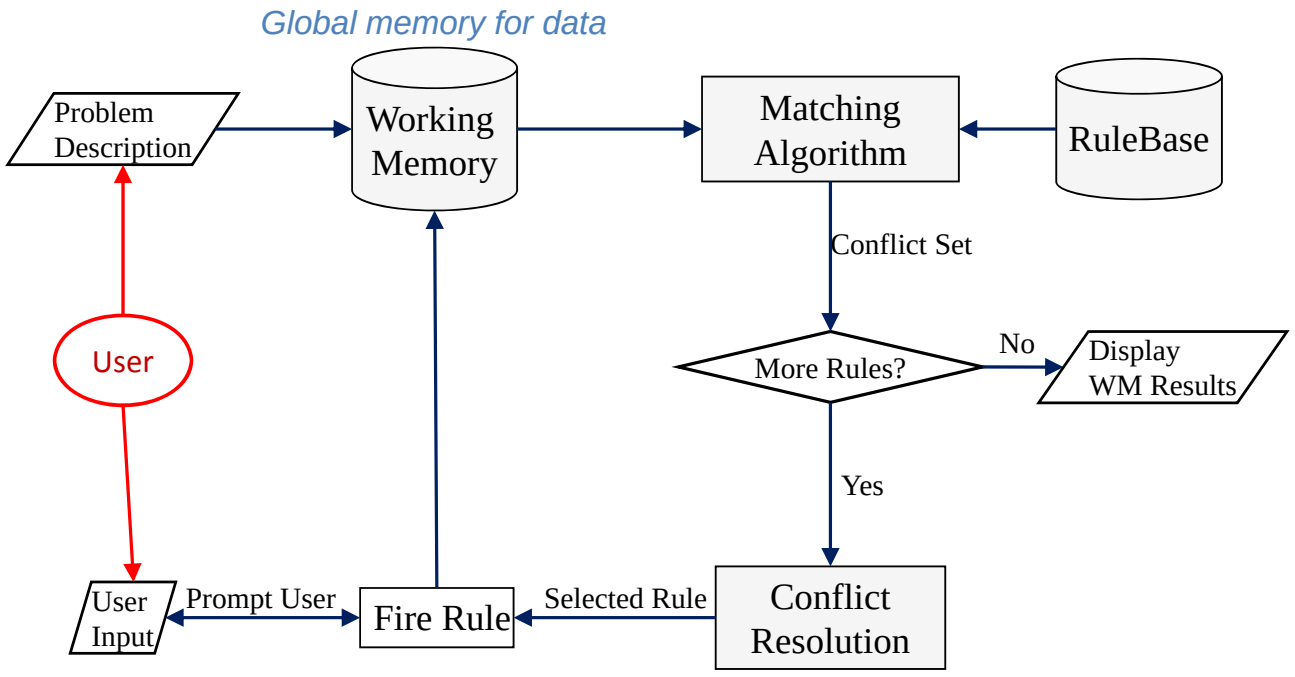
- To view loaded rules :

```
CLIPS> (rules)
```

- To view activated rules (conflict set):

```
CLIPS> (agenda)
```

Recognise-Act Control Cycle



Program Example

- Rule#1:

  IF time=low THEN pet=mice, fish
- Rule#2:

  IF budget=low AND time=mid THEN pet=rabbit, cat
- Rule#3:

  IF budget=high AND time=high THEN pet=dog
- Fact#1:

  budget=low
- Fact#2:

  time=mid

CLIPS Program

```
(deffacts user-preference
        (budget low)
        (time mid)
)

;IF time=low THEN pet=mice, fish
(defrule Rule1
        (time low)
        => (assert (pet mice fish))
)

;IF budge=low and time=mid THEN pet=rabbit, cat
(defrule Rule2
        (budget low)
        (time mid)
        => (assert (pet rabbit cat))
)

;IF budget=high and time=high THEN pet=dog
(defrule Rule3
        (budget high)
        (time high)
        => (assert (pet dog))
)
```


Defining Facts

- Facts are defined inside a set of matching parenthesis:

```
(budget low)
(time high)
```

- Facts are usually grouped together:

```
(deffacts user-preference
     (budget low)
     (time high))
```

- Facts are added to working memory:

```
(assert (pet rabbit cat))
```

  - *NOTE: CLIPS is case-sensitive*

About Facts

- Facts can be stored in a separate file and then loaded into CLIPS:

```
File->Load … "myfacts.clp"
Execution->Reset
```

- View facts in the working memory:

```
CLIPS> (facts)
f-1 (budget low)
f-2 (time mid)
```


Defining Rules

- Conditions are AND by default:

```
(defrule rule-name
                (cond-1)...(cond-n)
                =>
                (action-1)...(action-n))
```

- "OR" conditions:

```
(defrule rule-name
                 (or (cond-1)(cond-2))
                 =>
                 (action-1)...(action-n))
```

- "not" condition:

```
(defrule rule-name
                  (not (cond-1))...(cond-n))
                  =>
                  (action-1)...(action-n))
```

About Rules

- View loaded rules in the working memory:

```
CLIPS> (rules)
Rule1
Rule2
Rule3
```

- View activated rules in the working memory:

```
CLIPS> (agenda)
0    Rule2 : f-1, f-2
```


CLIPS Interactive Program

```
;Rule to obtain user input
(defrule user-input-budget
        => (printout t crlf "Enter budget (low, mid, high): " crlf)
        (bind ?user-budget (read))
        (assert (budget ?user-budget))
)
(defrule user-input-time
        => (printout t crlf "Enter Time available for pet (low, mid,
high): " crlf)
        (bind ?user-time (read))
        (assert (time ?user-time))
)
;IF time=low THEN pet=mice, fish
(defrule Rule1
        (time low)
        => (assert (pet mice fish))
        (printout t "We recommend mice and fish" crlf)
)

;IF budge=low and time=mid THEN pet=rabbit, cat
(defrule Rule2
        (budget low)
        (time mid)
        => (assert (pet rabbit cat))
        (printout t "We recommend rabbit and cat" crlf)
)

;IF budget=high and time=high THEN pet=dog
(defrule Rule3
        (budget high)
        (time high)
        => (assert (pet dog))
        (printout t "We recommend dog" crlf)
)
```

comments, variable, bind

- Comments:
  - In-line documentation is preceded by one or more";"
- Variable:
  - store values
  - syntax requires preceding with a question mark (?)
- **(bind <variable> <value>)**
  - Sometimes it is advantageous to store a value in a temporary variable to avoid recalculation.
  - The *bind* function can be used to bind the value of a variable to the value of an expression

read, printout

- (bind ?user-budget (read))
  - The *read* function can only input a single field at a time
  - Data must be followed by a carriage return to be read
  - To input, say a first and last name, they must be delimited with quotes, "xxx xxx"
- (printout <logical-name> <expression>)
  - allows output to a device attached to a logical name. The logical name *must* be specified and the device must have been prepared previously for output (e.g., a file must be opened first)
  - To send output to stdout, use a t for the logical name.
  - crlf = carriage return line feed

Execution->Watch (facts & rules)

```
         CLIPS (Quicksilver Beta 3/26/08)
CLIPS> (load "G:/Pet-Select-Interactive.clp")
Defining defrule: user-input-budget +j+j
Defining defrule: user-input-time +j
Defining defrule: Rule1 +j+j
Defining defrule: Rule2 +j+j+j
Defining defrule: Rule3 +j+j+j
TRUE
CLIPS> (reset)
<== f-0     (initial-fact)
==> f-0     (initial-fact)
CLIPS> (run)
FIRE    1 user-input-budget: *

Enter budget (low, mid, high):
low
==> f-1     (budget low)
FIRE    2 user-input-time: *

Enter Time available for pet (low, mid, high):
mid
==> f-2     (time mid)
FIRE    3 Rule2: f-1,f-2
==> f-3     (pet rabbit cat)
We recommend rabbit and cat
```


Arithmetic Operations

```
CLIPS>(+ 22 7)  ; 22+7
29
CLIPS>(- 22 7)  ; 22-7
15
CLIPS>(/ 22 7) ; 22/7
3.14285714285714
CLIPS>(* 22 7)  ; 22*7
154
CLIPS>(/ (* (+ 3 5) 7) (- 9 4))   ;((3+5)*7)/(9-4)
11.2
```

NOTE: Evaluation is from left to right.

Parenthesis give precedence.

Mathematical Operations

```
CLIPS> (max 3.5 4 2.3 6)
6
CLIPS> (min 3.5 4 2.3 6)
2.3
CLIPS> (abs -4)
4
CLIPS> (float 5)
5.0
CLIPS> (integer 3.142)
3
CLIPS> (sqrt 4)
2.0
```

```
CLIPS> (round 3.65)
4
CLIPS> (** 3 2)
9.0
CLIPS> (pi)
3.141592653589793
CLIPS> (cos 0)
1.0
CLIPS> (< 5 6)
TRUE
CLIPS> (> 5 6)
FALSE
```


Logical Test

```
CLIPS>(numberp 5.32)
TRUE
CLIPS>(numberp 5)
TRUE
CLIPS>(floatp 5.32)
TRUE
CLIPS>(floatp 5)
FALSE
CLIPS>(integerp 5)
TRUE
CLIPS>(stringp hello)
FALSE
CLIPS>(stringp “hello”)
TRUE
```

```
CLIPS>(eq hello hello)
TRUE
CLIPS>(eq “hello” “hello”)
TRUE
CLIPS>(eq 5 5)
TRUE
CLIPS>(eq 5 5.0)
FALSE
CLIPS>(neq “hello” “hello”)
FALSE
CLIPS>(neq “hello” hello)
TRUE
```

& Constraint

```
;Select persons who are older than 21
(defrule adulthood
        (person ?name ?age&:(> ?age 21))
        =>
        (assert (adult ?name ?age)))

;Select persons who are between 21 and 30 years
(defrule between-21-and-30
        (person ?name ?age&:(and (> ?age 21) (< ?age 30)))
        =>
        (printout t "Between 21 and 30 : " ?name ?age crlf))

;Check if a=b+2
(defrule comparison
        (compare ?a ?b&:(eq ?a (+ ?b 2)))
        =>
        (printout t ?a "->" ?b crlf))

;Select all fruits except durians
(defrule exception
        (fruits ?c&~ "Durians")
        =>
        (printout t "Fruits that I love : " ?c crlf))
```


Salience

- Salience provides a way for the programmer to prioritize rules and to resolve rule conflicts
- Salience range from -10,000 to +10,000
- Default salience is 0 (zero)
- Be careful not to overuse this feature since it will lead to a controlled sequential program and defeats the purpose of a rule-based program

Example of Salience

```
(defrule high-priority-rule
   (declare (salience 1)) (a) (b)
   =>
   (c))

(defrule default-priority-rule
   (d) (e)
   =>
   (f))

(defrule lower-priority-rule
   (declare (salience -1)) (g) (h)
   =>
   (i))

(defrule lowest-priority-rule
   (declare (salience -2)) (j) (k)
   =>
   (m))
```


Salience

In expert systems, one use of the term strategy is in conflict resolution of activations. Now you might say, "Well, I'll just design my expert system so that only one rule can possibly be activated at one time. Then there is no need for conflict resolution." The good news is that if you succeed, conflict resolution is indeed unnecessary. The bad news is that this success proves that your application can be well represented by a sequential program. So you should have coded it in Ada, C, or Pascal in the first place and not bothered writing it as an expert system.

CLIPS offers seven different modes of conflict resolution: depth, breadth, LEX, MEA, complexity, simplicity, and random. It's difficult to say that one is clearly better than another without considering the specific application. Even then, it may be difficult to judge which is "best." For more information on the details of these strategies, see the CLIPS Reference Manual.

CLIPS UserGuide v6.20 page 33

Ordered and non-ordered Facts

- Ordered facts:

```
(Person "John Tan" 45 Male)
```

  - Fields must be referenced in a specific order
- Non-ordered facts:

```
(person (name Mary) (age 35) (sex female)))
```

  - Fields can be referenced in any order by their slot-names

```
(person (age 35) (name Mary) (sex female)))
```


Record Structures

- CLIPS can represent PASCAL like records:

```
(deftemplate person
        (slot name)
        (slot age)
        (slot sex (default Male))
        (multislot itskills))

(deffacts people
        (person (name Brian)(age 35)(itskills java c++ python)
        (person (name Angie)(age 27)(sex Female)))

;; Modifying a record
(defrule change_age
        ?u <- (change ?name ?new-age)
        ?p <- (person (name ?name))
        =>
        (modify ?p (age ?new-age))
        (retract ?u))
```

Wildcards

- $? => a list with zero or many members
- E.g., “Allen” is a member of the list

```
(name-list $? Allen $?)
```

- E.g., Allen is the first member of the list

```
(name-list Allen $?)
```

- Last member …

```
(name-list $? Allen)
```

- Third member …

```
(name-list ? ? Allen $?)
```


Field Constraints

- Used to restrict the values that a field may have in the LHS of a rule
- Connective constraints
  - Not

```
(colour  ~red)
```

  - Or

```
(colour red | green)
```

Field Constraints

```
(deftemplate tropical-fruits
        (slot name)
        (slot color) )

(deffacts my_fruits
        (tropical-fruits (name Apple) (color Red))
        (tropical-fruits (name Grape) (color Purple))
        (tropical-fruits (name Banana) (color Yellow))
        (tropical-fruits (name Durian) (color Green)) )

;Select fruits that are not red
(defrule fruit-not-red-color
        (tropical-fruits (name ?name) (color ~Red))
=>
        (printout t ?name " is not red in color" crlf))

;Select either red or green fruits
(defrule fruit-red-or-green-color
        (tropical-fruits (name ?name) (color Red | Green))
=>
        (printout t ?name " is red or green in color" crlf))
```


Program efficiency

```
(defrule cousin
        (parent ?p1 ?c1)            ; if p1 is the parent of c1
        (parent ?p2 ?c2)            ; and p2 is the parent c2
        (sibling ?p1 ?p2)           ; and p1 and p2 are siblings
        =>
        (assert (cousin ?c1 ?c2))   ; then c1 and c2 are cousins
)
```

Reducing Match combinations

```
(defrule cousin
        (parent ?p1 ?c1)          ; if p1 is the parent of c1
        (sibling ?p1 ?p2)         ; and p1 and p2 are siblings
        (parent ?p2 ?c2)          ; and p2 is the parent c2
        =>
        (assert (cousin ?c1 ?c2)) ; then c1 and c2 are cousins
)
```

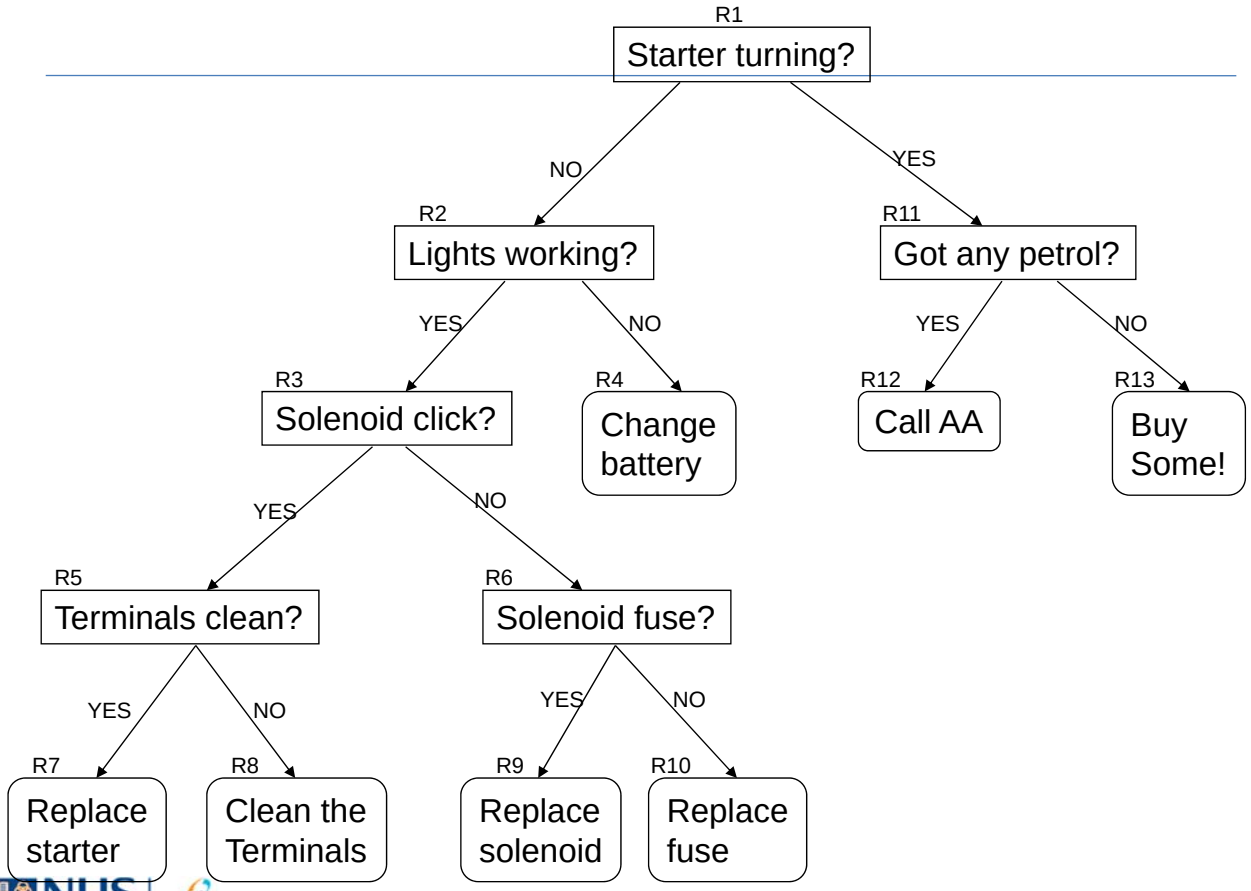

Exercise

- Download the CLIPS Windows installer onto your workstation or thumbdrive

  http://clipsrules.sourceforge.net
- Install CLIPS on your machine
- Familiarise yourself with CLIPS by typing in all the examples given in class, and running them.

Exercise (con't)

- Study the Decision tree given in the next slide.
  - Download the file "engine.clp" from the lecturer drive
  - load, reset & run the file "engine.clp"
  - Examine the output and note the facts assertions and rule activation/firing
  - The code given in engine.clp is ***not complete***. Find out what is missing.
  - ***Modify*** the program (by editing engine.clp) and test it for completeness.



CLIPS Homework

- See handout

Programming Guide

- http://math.hws.edu/eck/cs453/s03/about_clips.html
- The major part of a typical CLIPS program consists of rules. Each rule specfies conditions and actions. Conditions are patterns which are matched against facts in working memory. Actions are commands. When the conditions of a rule match the current state of working memory, it is said to be activated and it is put on the agenda, which is simply the list of currently activated rules. An activated rule is eligible to be executed, but it is not actually executed until it gets to the top of the agenda. When that happens, the rule fires -- that is, its actions are executed. This can result in new facts being asserted, which can cause new rules to be be activated and placed on the agenda. Also, facts can be retracted, which might deactivate some of the active rules because their conditions are no longer satisfied. These rules are removed from the agenda. The (reset) command clears the current agenda; when the initial set of facts is then asserted, some rules might become active and be placed on the agenda. The (run) command causes rules to be fired, with any resulting changes in the agenda, until the agenda is empty (or until the firing of a rule causes a (halt) command to be executed as one of its actions).
- The ordering of rules on the agenda determines the order in which they will be fired. Newly activated rules are not necessarily placed at the end of the agenda. First of all, each rule can have a salience, which is an integer in the range -10000 to 10000. Rules of higher salience are always placed above rules of lower salience on the agenda, and therefore are executed first. Salience allows you to exert some influence over the order in which rules will be applied. For example, suppose your program occasionally produces some values for output, and suppose you want to make sure that a value is output as soon as it is generated. You could give your output rules higher salience than other rules. When an output rule is activated because some output has become available, its higher salience will ensure that it is placed at the top of the agenda and is executed immediately. Salience could also be used to encode heuristic knowledge about which rules should be tried first.
- Within rules of the same salience, the placement of rules on the agenda is determined by the conflict-resolution strategy. The default strategy is depth, which means that new rules are placed above rules that are already on the agenda. Another strategy is breadth, which places newly activated rules below existing rules. These two strategies result, respectively, in a depth-first or breadth-first search. To change from the default depth-first strategy to breadth-first, use the command:
- (set-strategy breadth)
- There are other strategies, but depth and breadth are probably the most common.
- (Note: A rule can appear on the agenda several times simultaneously. It appears once for each fact or combination of facts that are matched by its patterns. Note also that a given match can only cause the rule to fire once. This is one reason why the facts are ordered and numbered. You can think of the rule as marching down the list of facts, not looking back after it fires. If you retract and then re-assert a fact, it gets a new number and can be used in new matches.)

Programming Guide & Help

http://clipsrules.sourceforge.net/OnlineDocs.html

http://www.cs.oswego.edu/~odendahl/manuals/clips/help/con.html