Università degli Studi di Bari
Dipartimento di Informatica

LACAM
Machine Learning

CLIPS: Facts, Rules, Inference

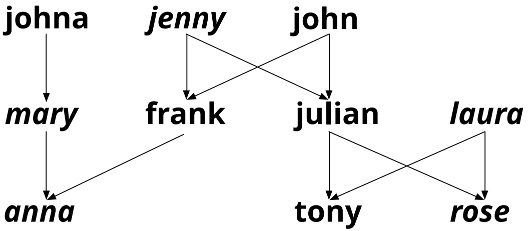
Antonio Vergari

April 19, 2016

A first example

With a family DAG

This is our first take on a very simple **conceptualization** task.
Suppose we want to represent the relationships in a family as sketched in this dag.
The basic info we get are the persons' names, their *gender* (names in italics stand for women) and their *parenthood* (edges).



family DAG I

Facts and rules

We can use ***facts*** to represent what we know about this small world.

Julian is a male. Laura is a female.
Tony is a male. Rose is a female.
...
Julian is Tony's parent. Laura is Tony's parent.

What if now we'd like to assert something about uncle-nephew relationships?

Frank is Tony's uncle.

We can introduce ***rules*** to represent how to automatically derive new facts, i.e. new knowledge.

IF X is Y's brother AND Z's father, THEN Y is Z's uncle.

but we need to specify what brother and father mean in the first place...

family DAG II

more rules

A generic rule to derive fathers’ names is straightforward:

IF X is male AND X is Y’s parent, THEN X is Y’s father.

To derive what a brother is we need to introduce one more variable and

IF X is Y’s parent AND X is Z’s parent AND Z is male,
THEN Z is Y’s brother.

Pretty simple indeed.

Problems: how to formalize this in CLIPS? how to let CLIPS infer new knowledge?

We will dive a bit into CLIPS logics and syntax before answering to it.

family DAG in CLIPS

A simple implementation in CLIPS of a portion of the KB and the father inference rule. This may appear scary...

```
(deffacts family-dag
    (parent julian tony)
    (parent julian rose)
    (parent laura tony)
    (parent laura rose)
    (male tony)
    (male julian)
    (female rose))

(defrule fatherhood
"match-and-assert all father-child relationships"
    (male ?father)
    (parent ?father ?child)
    =>
    (assert (father ?father ?child))
    (printout t ?father " is " ?child "'s father "))
```

A typographical note: keywords are in bold, construct defining keywords are also orange. Nested parentheses are rainbow colored as in emacs rainbow-delim-mode for the sake of legibility.

CLIPS
A LISPy functional programming foretaste

```
; this is a comment in CLIPS
```

Logical blocks and function calls are delimited by parentheses.
Mind the *prefix notation* for operators.

```
(+ 2 3)
```

Functional composition is achieved by nesting parentheses, at any level.

```
(+ (* (* 3 (/ 1 3)) (/ 5 (+ 4 1))) 0)
```

In truth, a pair of parentheses identifies a list.
The first element of a list is a function identifier, the others are the function operands.

```
(+ 2 3 4 5 6) ;; lists of arguments => variadic functions
```

We will return to functional programming aspects later; for the time being, just pay attention to syntax error and evaluation order:

```
(printout t "Beware order" (read) crlf)
```

Command-line REPL

The command-line version of CLIPS lets you write instructions into it (mind nested parentheses and line breaks if you do not want syntax errors).

```
CLIPS> (+ 2 3)
5 ;; if it can be evaluated it is printed on stdout
clisp> (printout t "Beware order" (read) crlf)
Beware order  5
5
CLIPS> (exit) ;; ending the interpreter
```

Like many other interpreted languages, the main interaction happens through a Read-Eval-Print Loop.

```
(loop (print (eval (read)))) ;; this is LISP, not CLIPS!
```

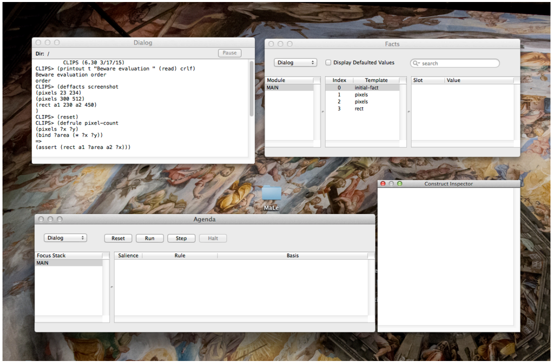
Some handy functions

```
;; math functions
(+ 2 4) ; 6
(* 2 4) ; 8
(** 2 4) ; 16.0
(log10 100) ; 2.0
(div 33 2 3 5) ; 1
(max 3 9.0 -2) ; 9.0
(abs -2.0) ; 2.0
(float 33) ; 33.0
(integer 4.2) ; 4
(round 3.7) ; 4
(cos 0) ; 1
(pi) ; 3.14159...
;; logical ops
(> 1 0) ; TRUE
(and TRUE FALSE) ; FALSE
```

```
;; std I/O
(read) ; reads a symbol
(readline) ; reads a line
(get-char t) ; reads a char
(format t "like in %s\n" C)
(printout t
    "This is a string" crlf)
;; file I/O
(open example.txt
    file-stream "w")
(printout file-stream
    "capercaillie" crlf)
(close file-stream)
(open example.txt
    file-stream)
(get-char file-stream)
```

GUI IDE

For some platforms[1] you can use a little IDE that offers a console and some windows to inspect the working memory, the agenda, object instances and the build constructs.



```
Dir: /
          CLIPS (6.30 3/17/15)
CLIPS> (printout t "Beware evaluation " (read) crlf)
Beware evaluation order
order
CLIPS> (deffacts screenshot
(pixels 23 234)
(pixels 300 512)
(rect a1 230 a2 450)
)
CLIPS> (reset)
CLIPS> (defrule pixel-count
(pixels ?x ?y)
(bind ?area (* ?x ?y))
=>
(assert (rect a1 ?area a2 ?x)))
```

Module	Index	Template
MAIN	0	initial-fact
	1	pixels
	2	pixels
	3	rect

[1] For GNU/Linux distros there should be an old, unmantained, package called xclips. If you manage to get it to work.

Facts

Some facts about facts (a reminder)

A fact is a piece of information we know about the world that we can represent, and on which we can base our inference.

An ***asserted*** fact defines something that is true in our *current* state of the world.

The ***Working Memory*** (WM) is the current collection of all asserted facts. Additional facts can be added to the WM (*asserted*), removed (*retracted*) and modified.

CLIPS implements the ***RETE*** algorithm and its data structure: the WM is modelled as a network of patterns efficiently matching the asserted facts.

Ordered Facts

Facts can be defined by a *name* and a list of properties.

```
(<functor—fact—name> *<properties>)
```

Facts with same name and different properties or*ariety* are allowed (they are functors indeed).

```
(starting—block)    (block 1)    (block red)    (block  "red")
```

They are called *ordered facts* since order in declaring property symbols matters, and it is the only way to bind meanings to symbols (positional semantics).

```
(age julian 32)
```

is different from

```
(age 32 julian)    (age "julian" 32)
```

If you write these in the REPL you will get a function undefined error, why?

Accessing Working Memory I

Facts are meant to be read, written (*asserted*), or deleted (*retracted*) from the ***Working Memory*** (WM), also called fact-list.

To list all facts present in the WM, use the command facts.

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

At start, one shall see only (initial-fact).
Facts in the WM have a fact-index (0), fact-identifier (f-0) and a logical fact-address (<Fact-0>).
To remove all facts from the WM, use

```
CLIPS> (clear)
CLIPS> (facts)
CLIPS>
```

it makes fact-indexes to start from 0 again[2].

[2] The examples shown here are with version 6.24, there the clear command deletes even the initial fact. This is not true anymore in version 6.30, see next slides.

Accessing Working Memory II

Asserting and reading

To add a fact to the WM, use assert. It returns the fact-address of the added fact.

```
CLIPS> (assert (male john))
<Fact-1>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (male john)
For a total of 2 facts.
```

Assert multiple facts at the same time (only the last address is returned)

```
CLIPS> (assert (block red) (is-on red white) (female rose))
<Fact-4>
```

Nested functions will be evaluated before asserting.

```
(assert (age (* 12 5))) ; <Fact-5>
(assert (John (age 5))) ; [EXPRNPSR3] Missing function declaration for age.
```

Loading a KB from file

We can put facts to be asserted into a deffacts construct in a file.

```
(deffacts blocks
    (block red) (block green) (is-on green red))
;; file blocks.clp
```

Once file is loaded in the REPL, all facts are asserted automatically when WM is reset.

```
CLIPS> (clear)
CLIPS> (load blocks.clp)
CLIPS> (facts) ;; no facts yet
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (block red)
f-2     (block green)
f-3     (is-on green red)
For a total of 4 facts.
```

In version 6.30 a reset command is now performed after a clear command (the initial fact is restored).

More on assert

Generally you cannot assert the same fact more than once in the WM:

```
CLIPS> (clear)
CLIPS> (assert (block red))
<Fact-0>
CLIPS> (assert (block red))
FALSE
CLIPS> (facts)
f-0     (block red)
For a total of 1 fact.
CLIPS> (assert (block red on-top)) ;; this specifies a totally different fact
<Fact-1>
```

The same goes for the asserting through a deffacts:

```
CLIPS> (clear)
CLIPS> (deffacts some-facts (first-fact) (second-fact) (first-fact))
CLIPS> (reset) ;; asserting after resetting
CLIPS> (facts)
f-0     (initial-fact)
f-1     (first-fact)
f-2     (second-fact)
For a total of 3 facts.
```

family DAG III
CLIPS ordered facts

Back to the family example, we can build a very simple knowledge base (kb) by introducing the facts *parent*, *male* and *female*.

We are thus modeling only the nodes and the direct edges in the DAG.

```
(parent julian tony)    (parent frank anna)    (female jenny)
(parent julian rose)    (parent mary anna)     (female anna)
(parent laura tony)     (parent johna mary)    (female mary)
(parent laura rose)     (male frank)           (female rose)
(parent john julian)    (male julian)          (female laura)
(parent john frank)     (male john)
(parent jenny julian)   (male johna)
(parent jenny frank)    (male tony)
```

We can assert them all together or use a deffacts construct in a file.

More on conceptualizations

with ordered facts

We will use (in the next lecture) one long fact to represent a 5×5 grid as the Game of Life world state.

```
(deffacts initial-world
    (world FALSE FALSE FALSE FALSE FALSE
           FALSE FALSE TRUE  FALSE FALSE
           FALSE FALSE TRUE  FALSE FALSE
           FALSE FALSE TRUE  FALSE FALSE
           FALSE FALSE FALSE FALSE FALSE))
```

A very flat representation, what are the dis/advantages?

Much of the effort is done in preserving intra-facts relationships. One should account for future facts to *match* the exact parts.

Unordered Facts
aka templated facts

Structured or unordered facts allow for more flexibility when conceptualizing.
For each named property, *slot* or *multislot*, in a fact template, you can specify the type, default value and allowed values (definition by enumeration).

```
(deftemplate <template-name>
    *(slot <field-name>
        ?(type SYMBOL|STRING|NUMBER|INTEGER|FLOAT)
        ?(default <default-value>)
        ?(allowed-symbols <symbol-list>))
    *(multislot <field-name>
        ?(type SYMBOL|STRING|NUMBER|INTEGER|FLOAT)
        ?(default <default-value>)
        ?(allowed-symbols <symbol-list>)))
```

It is a way to declare constraints on the facts with a certain name allowed in the WM.

They offer also a some kind of *introspection* at run time (like getting slot cardinality, template names, and query on templated fact sets).

This grammar is incomplete, check the manual

Ordered vs Unordered Facts

Each ordered fact can is an unordered fact composed by a single multislot field, whose template is implicitly defined by CLIPS (with no restriction on type, range, etc...)

```
CLIPS> (clear)
CLIPS> (assert (ordered—fact the—answer " is "  42))
CLIPS> (list—deftemplates)
 initial—fact
ordered—fact
For a total of 2 deftemplates.
```

Unordered Facts: asserting

To model a record for the results of the exam for this class we could use a simple template:

```
(deftemplate icse-exams
    (slot student
        (type STRING)
        (default ?NONE))
    (slot score
        (type INTEGER)
        (default 18)
        (range 0 30)))
```

Type info associated to slots ensure asserts are done properly:

```
(assert (icse-exams (score 18) (student "A. Ver"))) ;; ok!
(assert (icse-exams (score 22))) ;; [TMPLTRHS1] Slot student
;; requires a value because of its (default ?NONE) attribute.
(assert (icse-exams (student A-Ver))) ;; a string is needed, not a symbol
(assert (icse-exams (student "A. Ver"))) ;; ok, default score: 18, but...
```

Unordered Facts: more examples

When we will model the search space of the Cannibals and Missionaries game we will use a structured fact to keep track of the state, search depth and tree path:

```
(deftemplate status
    (slot search-depth (type INTEGER) (range 1 ?VARIABLE))
    (slot parent (type FACT-ADDRESS SYMBOL) (allowed-symbols no-parent))
    (slot shore-1-missionaries (type INTEGER) (range 0 ?VARIABLE))
    (slot shore-1-cannibals (type INTEGER) (range 0 ?VARIABLE))
    (slot shore-2-missionaries (type INTEGER) (range 0 ?VARIABLE))
    (slot shore-2-cannibals (type INTEGER) (range 0 ?VARIABLE))
    (slot boat-location (type SYMBOL) (allowed-values shore-1 shore-2))
    (slot last-move (type STRING)))
```

Again we can use a deffacts to assert unordered facts (e.g. the initial state):

```
(deffacts initial-positions
    (status (search-depth 1)
        (parent no-parent) (shore-1-missionaries 3)
        (shore-2-missionaries 0) (shore-1-cannibals 3)
        (shore-2-cannibals 0) (boat-location shore-1) (last-move
        nil)))
```

This example is taken from Giarratano’s code.

family DAG IV

Refactoring our KB

We could restructure the family example by defining a new template for a *person*, embedding *gender* and *parenthood* information at once.

```
(deftemplate person
    "Representing a person as a compound type"
    (slot name
        (type STRING))
    (slot gender
        (allowed—symbols male female))
    (multislot children
        (type STRING)))
```

Here is a way to assert a part of our KB via deffacts:

```
(deffacts templated—family—dag
    (person (name "tony") (gender male))
    (person (name "julian") (gender male) (children "tony" "rose"))
    (person (name "john") (gender male) (children "julian" "frank"))
    (person (name "johna") (gender male) (children "mary"))
    (person (name "frank") (gender male) (children "anna")))
```

Accessing Working Memory II

Deleting by retracting

Dual to asserting, there is *retracting*.

retract needs a fact-index or fact-address as argument.

```
CLIPS> (facts)
f—0      (initial—fact)
f—1      (icse—exams (student "A. Ver") (score 18))
For a total of 2 facts.
CLIPS> (retract 1)
CLIPS> (facts)
f—0      (initial—fact)
For a total of 1 fact.
CLIPS> (retract 0)
CLIPS> (facts) ;; empty WM
```

It can be used on multiple arguments at once:

```
CLIPS> (clear)
CLIPS> (retract (assert (a fact) (another fact)) (assert (a third fact)))
CLIPS> (facts)
f—0      (a fact)
For a total of 1 fact. ;; why?
```

Accessing Working Memory III

Duplicating and modifying facts

modify retracts and assert an unordered fact in order to modify one of its slots value:

```
CLIPS> (facts)
f-3     (icse-exams (student "A. Ver") (score 18))
For a total of 1 fact.
CLIPS> (modify 3 (score 5))
<Fact-4>
CLIPS> (facts)
f-4     (icse-exams (student "A. Ver") (score 5)) ;; note new index
For a total of 1 fact.
```

duplicate differs from it in that the original fact is not retracted.

```
CLIPS> (duplicate 4 (student "F. Cen"))
<Fact-5>
CLIPS> (facts)
f-4     (icse-exams (student "A. Ver") (score 5))
f-5     (icse-exams (student "F. Cen") (score 5))
For a total of 2 facts.
```

Rules I

Rules I

Rules model the knowledge needed for inference (modifying our knowledge base, the WM). In CLIPS we do not care about how inference is carried out, we get it for free from the inference engine implementation.

Rules are formed by an *antecedent* to be matched in the WM, the Left Hand Side (LHS), and a series of actions in the Right Hand Side, RHS, the *consequent*.

They can be defined in the REPL and from a file through the function defrule:

```
(defrule <rule-name>
?<comment-as-string>
?<LHS>
=>
?<RHS>)
```

Defined rules can be listed (their names) with the function rules. clear can undefine all defined rules as well.

Rules II

All rules whose LHS is matched by at least one fact are stored in a priority queue called ***agenda***.
All rules are ordered by a priority factor called **salience**.
The salience determines their order of execution.

If the state of the WM changes (new facts are asserted or retracted), then the state of the agenda can change as well (rules whose LHS is not matched by any fact are removed from the agenda).

At each inference cycle, the top most rule in the agenda (the one with highest salience) is selected, removed from the Agenda and its RHS is executed, possibly altering the WM.
To start rule firing in CLIPS, first reset the environment (to properly assert all the facts), then call the run command that will execute up to <limit> rules:

```
(run [<limit >])
```

Rules III

A rule with no LHS will always *fire* (as long as there is at least one fact)

```
CLIPS> (defrule always-fire
                =>
                (printout t "Firing"))
```

while a rule without a RHS will produce no effects (but not no side-effects[3])

```
CLIPS> (defrule no-effect-rule
                (initial-fact) ;; to fire needs (initial-fact)
                =>)
```

Rules get activated when something changes in the working memory (assert, retract, modify, duplicate).
Rules that are not active when defined can get activated when a new fact is asserted for instance (remember how RETE works...).

[3]We will see later how functions can be called in the LHS of a rule, imagine now a function with a side effect (e.g. printing to stdout.). However this would probably be considered bad programming.

The Agenda

In CLIPS the agenda stores not only the rules that are active, i.e. whose LHS is matched by one or more fact sets, but also lists these facts.

To be more precise, when there is a set of facts matching the LHS of a rule, then *an instance of that rule* is added to the agenda. That is, for a single rule, multiple facts matching will result in multiple instances in the agenda.

The command agenda lists the active rule instances waiting to be fired and the facts matching them:

```
CLIPS> (agenda) ;; agenda is empty now
CLIPS> (reset)  ;; asserting initial—fact
CLIPS> (agenda)
0       no—effect—rule: f—0
0       always—fire: f—0
For a total of 2 activations
```

Matching I

Patterns and Pattern Matching

The simplest constrained pattern to appear in a LHS of a rule are called **pattern CE** (Conditional Element) or simply **pattern**.

If more than a pattern CE is present, all of them shall match some facts for the rule to fire (they are conditions in a logical AND).

Back to the family DAG example, we could write a very specific rule to asses that John is Julian’s father (is this useful to anyone?):

```
(defrule john—julian—father
    (parent john julian)
    (male john)
    =>
    (assert (father john julian)))
```

Matching II

Modifying the WM in the RHS

In the RHS we can call functions like assert, retract, modify, duplicate.

```
CLIPS> (clear)
CLIPS> (defrule duck
            (animal-is duck)
            =>
            (assert (sound-is quack)))
CLIPS> (assert (animal-is duck))
==> f-1 (animal-is duck)
CLIPS> (run)
==> f-2 (sound-is quack)
```

This is the first basic way we have to govern and manage the control flow: WM changes will activate other rules that will modify the WM again,...

Matching III

Pattern Matching with Vars

Variables can be introduced as fields with a $?$ to bind different pattern CE, or the LHS and the RHS.

```
(defrule is-father
    "match-and-assert all fathers' names"
    (male ?father)
    (parent ?father ?child)
    =>
    (assert (father ?father ?child)))
```

For a multifield we could specify all properties at once with $\$?$.
For the family DAG KB with deftemplates we could rewrite the fatherhood rule this way:

```
(defrule is-father
    "asserting fatherhood"
    (person (name ?father) (gender male) (children $?children))
    =>
    (assert (father (name ?father) (children $?children))))
```

Matching IV

Binding matching facts addresses

Now that we have introduced variables, we can use the arrow operator $< -$ to bind fact-addresses to them.

In the RHS we can **retract**, **modify**, **duplicate** the matching facts.

```
(defrule is-father-II
    "match-and-assert all fathers' names"
    (male ?father)
    ?f <- (parent ?father ?child)
    =>
    (retract ?f)
    (assert (father ?father ?child)))
```

Matching V

Refraction

A rule cannot be triggered by the same fact set twice.

Refraction prevents the inference engine to go into trivial and infinite loops (how could that happen?).

However, retracting and asserting new facts (they are considered as different facts) can:

```
(defrule duck—quack
     ?d <— (animal—is duck)
     =>
     (retract ?d)
     (printout t "if it is a duck, then it does quack")
     (assert (sound—is quack)))
(defrule quack—duck
     ?q <— (sound—is quack)
     =>
     (retract ?q)
     (printout t "if it does quack, then it is a duck")
     (assert (animal—is duck)))
```

family DAG

More examples

Back to the family DAG use case, we now want to embed knowledge in rules to automatically assert new facts for the relationship motherhood, e.g. we are interested in facts in the form:

```
(mother mary anna)    (mother laura rose)
```

Very similarly to inferring fatherhood relationships, we can write:

```
(defrule motherhood
    "match—and—assert all mother—child relationships"
    (female ?mother)
    (parent ?mother ?child)
    =>
    (assert (mother ?mother ?child))
    (printout t ?mother "is the mother of" ?child))
```

family DAG

More examples

If we are interested in the granparent or grandchildren facts like this:

```
(grandparent johna anna) (grandparent john tony)
(grandparent john rose)
```

we could infer them by expressing the transitive property of the parenthood relationship:

```
(defrule grandparent
    "match—and—assert all grand parent relationships"
    (parent ?grandparent ?parent)
    (parent ?parent ?child)
    =>
    (assert (grandparent ?grandparent ?child))
    (printout t ?grandparent "is the grandparent of" ?child))
```

Rules activation
Control flow

As you may have noticed: you do not control how rules are activated.

That is the inference engine job.

The role of the programmer is to provide knowledge in the form of rules and facts.

You describe what a father and a uncle are, not how to match which family facts nor when to fire a rule.
Nevertheless you shall have a deep knowledge about what happens behind the curtains (in the WM, agenda, and so on). Go back to the RETE algorithm.

Moreover, once you'd get a grasp of how CLIPS fully works, we could start to bend it to program what we need, e.g. implementing *backward chaining inference*.

Rules activation
From Pattern Matching

When rules are matched by facts in the WM, they are instantiated and go to the agenda.

To see which facts are activating which pattern in a rule we could also use the function matches:

```
CLIPS> (matches fatherhood)
Matches for Pattern 1
f—12
f—13
...
Matches for Pattern 2
f—1
...
f—11
Partial matches for CEs 1 — 2
f—16,f—9
...
Activations
f—16,f—9
...
```

Rule activation

Back to the Agenda

More instances of a rule are put in the agenda at the same time, but what it the activation order, i.e. which rule instances in the agenda will fire before the others?

Activation order matters, since some rules actions can influence other rules (e.g. invalidate other activated rules or activate new ones).

The agenda is implemented like a *priority queue*, but we can think of it like a sequence ordered according two criterions: rules **salience** and the **conflict resolution strategy** set.

More specifically, when a new rule is activated it is put in the agenda:

- after all rules with a higher salience and before all rules with a lower one
- the order of rules with the same salience is determined by the current conflict resolution strategy
- if some rules are activated by the same assertion/retraction of a fact set, then they are arbitrarily inserted

Salience

The salience of a rule is an integer value in the range $[-10000, 10000]$ that can be specified in CLIPS with the function declare in its LHS:

```
CLIPS> (defrule first-to-fire
            (declare (salience 100))
            =>
            (printout t "The next sentence is true" crlf))
CLIPS> (defrule second-to-fire
            (declare (salience 99))
            =>
            (printout t "The previous sentence is false" crlf))
CLIPS> (reset)
CLIPS> (run)
The next sentence is true
The previous sentence is false
```

Decliring rule salience is one of the weapons for programmers to suggest one execution order, not to govern it. It shall not be misused.

Conflict Resolution Strategy

The conflict resolution strategy can be set with the function set-strategy and the current one checked with get-strategy.

```
CLIPS> (get-strategy)
depth ;; default value
CLIPS> (set-strategy breadth)
depth ;; returns the previous value
```

CLIPS provides several strategies:

depth	new rules (with same salience) put before other
breadth	new rules put after
simplicity	less specific rules are put before
complexity	more specific rules are put before
random	randomly put
lex/mea	time ordering as in OPS5 (see manual)

Can you tell why the first two recall the graph search variants?

The MU Game

The formal system MIU has only three *symbols*: M, I, U.
Here are some possible strings, you could compose with these symbols:

MIU, UIM, MUUMUU, UIIUMIUUIMUIIUMIUUIIUUMIUM

There is only one *axiom*, the starting string MI.

There are only four rules to derive *theorems*, as strings, from the single axiom:

I. If you have string terminating with I, you can add U to it;
II. If you have a string in the form M*x*, with *x* being any string, you can add M*xx*;
III. If you have a string containing III as substring, you can add a new string having III replaced by U;
IV. If a word contains UU, you can remove it.

Douglas R. Hofstadter, *Gödel, Escher, Bach* - Chapter 1.

The MU Game

A derivation as an example

1.	MI	the axiom
2.	MII	by applying II
3.	MIIII	by applying II
4.	MIIIIU	by applying I
5.	MUIU	by applying III
6.	MUIUUIU	by applying II
7.	MUIIU	by applying IV
8.	…	

Question: can we derive the theorem MU?

Douglas R. Hofstadter, *Gödel, Escher, Bach* - Chapter 1.

The MU Game

a simple CLIPS realization

The simplest conceptualization we can design is to use an ordered fact as a multifield of symbols to represents the strings letters. The first string (axiom) could be defined as:

```
(deffacts axiom
    (miu-string M I))
```

To implement rules we can exploit the power of multifield pattern matching by using $?prefix and $?suffix variables:

```
(defrule rule-I
    "If a string ends with I, then you can add a U at the end"
    ?s <- (miu-string $?prefix I)
    =>
    (retract ?s)
    (assert (miu-string $?prefix I U))
    (printout t (implode$ $?prefix) I U " (by applying rule I)" crlf)
    (assert (key-command (get-char t)))) ;; why is this needed?
```

The MU Game

more rules

Other rules can be defined in a very similar fashion:

```
(defrule rule-II
    "If a string is in the form Mx, then you can rewrite it as Mxx"
    ?s <- (miu-string M $?x)
    =>
    (retract ?s)
    (assert (miu-string M $?x $?x))
    (printout t M (implode$ $?x) (implode$ $?x) " (by applying rule II)" crlf)
    (assert (key-command (get-char t))))
```

We can then add a rule to end the activation by calling the function halt:

```
(defrule halt-mu
    "Halts the inference if 'e' is entered on stdin "
    (declare (salience 10000))
    (key-command 101)
    =>
    (halt))
```

The MU Game

more rules

```
(defrule rule-III
    "If a string contains III ,
        then you can write a rule that has U instead of it"
    ?s <- (miu-string $?prefix I I I $?suffix)
    =>
    (retract ?s)
    (assert (miu-string $?prefix U $?suffix))
    (printout t (implode$ $?prefix) U (implode$ $?suffix)
        " (by applying rule III)" crlf)
    (assert (key-command (get-char t))))

(defrule rule-IV
    "If a string contains UU, then you can remove it"
    ?s <- (miu-string $?prefix U U $?suffix)
    =>
    (retract ?s)
    (assert (miu-string $?prefix $?suffix))
    (printout t (implode$ $?prefix) (implode$ $?suffix)
        " (applying rule IV)" crlf)
    (assert (key-command (get-char t))))
```

Be strategic

Forcing just one string in the WM at a time, we must concentrate on the rule activation order.

```
CLIPS> (reset)
CLIPS> (set—strategy breadth)
depth
CLIPS> (run)
MII (by applying rule II)

MI II I (by applying rule II)

M IU (by applying rule III)

MI UI U (by applying rule II)

MI U I UI U I U (by applying rule II)
e ;; ending generation
CLIPS>
```

Question: are you able to reproduce the trace shown some slides ago by setting an appropriate strategy only?

Misc

List and print constructs

CLIPS provides several functions to list all the construct defined, may they be rules, facts (via a deffacts) or templates:

```
(list-deffacts)
(list-deftemplates)
(list-defrules)
```

There are also routine to print to string a prettified version of the defined constructs, referenced by their names:

```
(ppdeffact <deffact-name>)
(ppdefrule <rule-name>)
(ppdeftemplate <template-name>)
```

Undefine constructs

For each def* function, there is one undef* that removes the definition from the current environment. This enhances runtime flexibility.

```
(undeffacts <deffacts—name>)
(undefrule <defrule—name>)
(undeftemplate <deftemplate>)
```

To undefine all constructs at once one can use the already introduced clear function.

Debugging I

The watch function allows to monitor the WM, the agenda and the rule base, at each change in them, e.g. a fact being asserted or retracted from the fact-list, a print is sent to the stdout as an alert.

```
(watch facts)     (watch rules)     (watch activations)
```

To disable this mode, call the corresponding unwatch.

```
CLIPS> (clear)
CLIPS> (watch facts)
CLIPS> (assert (a) (b) (c) (d))
==> f—0      (a)
==> f—1      (b)
==> f—2      (c)
==> f—3      (d)
<Fact—3>
CLIPS> (retract 0 1 2 3 )
<== f—0      (a)
<== f—1      (b)
<== f—2      (c)
<== f—3      (d)
```

Debugging II

A useful way to inspect what is going on is to set a breakpoint to a rule with the function set-break. Execution will halt before executing the rule RHS. The breakpoint can be deleted with remove-break and all the breakpoints set listed with show-breaks.

```
 CLIPS> (clear)
 CLIPS> (defrule no-age
             "fever dreaming"
             (no age)
             =>
             (assert (an-object "!") (an-object "?") (an-object
             "."))))
        (defrule an-object
             (an-object ?x)
             =>
             (printout t an-object ?x))
CLIPS> (assert (no age))
CLIPS> (set-break no-age)
CLIPS> (run)
Breaking on rule no-age.
CLIPS> (run)
an-object.an-object?an-object!
```

Exercises

Blocks World

In a blocks mini world there are the blocks A, B, C, D, E, F.
Each block can be stacked only on top of another block.
They are *stacked* in two piles of three, from the top: A, B and C (first one) and D on E, on F (second stack).

One can move only one block at a time, putting it on the floor or on another block that has no blocks on itself.
Conceptualize a KB in CLIPS to represent this mini world; then write the rules to obtain block C on top of block E starting from the above configuration.

A possible deffacts for the initial configuration:

```
(deffacts initial—state
   (stack A B C)
   (stack D E F)
   (stack) ;; why is this useful?
   (goal (move C) (on—top—of E))) ;; why is an explicit fact?
```

As in Giarratano& Riley’s Blocks World Problem in Chapter 8

more on family DAG

Using the KB for the family DAG implemented with ordered facts, implement some rules in CLIPS to infer new knowledge about these relationships:

uncle/aunt to derive facts like

```
(uncle frank tony) (aunt laura anna)
```

married

```
(married julian laura)
```

cousins

```
(cousins tony anna)
;; what about (cousins tony anna rose)
```

orphan

```
(orphan mary)    (orphan johna)
```

Write them for the KB with templated facts as well.