Intelligent Systems: Reasoning and Recognition

James L. Crowley

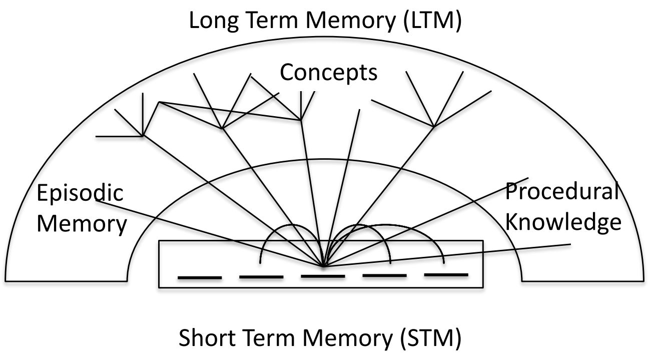
Ensimag 2 Second Semester 2018/2019

Lesson 15 3 April 2019

Knowledge Representation and Reasoning With Rule-Based Production Systems

Knowledge Representation in Rule-Based Systems

Most models of human cognition posit some form of "spreading activation" (Anderson 83) in which activation energy associates cognitive "units" in short term memory with concepts, episodes and procedures in long term memory.



Rule-based production systems provide a programmable implementation for this model.

Three techniques are commonly used to represent knowledge in a rule-based production system

An interpreted language - for procedural knowledge
Schema - to represent concepts and frames
Rules (productions) - for activation of WM from LTM

An interpreted language is a programming language in which instructions are interpreted and executed directly at run time, without previous compiling. Interpreted programs permit programs to be treated as data and to be modified dynamically.
The classic interpreted language is LISP. Popular modern languages include Python and Java.

Schema are patterns (or templates) for representing concepts or data. A schema is a named collection of attribute-value pairs in which the attributes (or slot) names act as a key for indexing and the values may be data elements, lists, or pointers.

Rules associate concepts in working memory with long term memory. Rules take their inspiration from the observation of conditioned reflexes in animals and humans.

Rules are often called “productions”.

Conditioned Reflexes are trained associations of stimulus and response.

Stimulus $\Rightarrow$ Response

In production systems, rules are implemented as condition-action pairs.

Condition $\Rightarrow$ Action

Conditions are conjunctions of patterns that must be true for the rule to be activated. Actions are sequences of procedures that executed by the interpreter when a rule is activated.

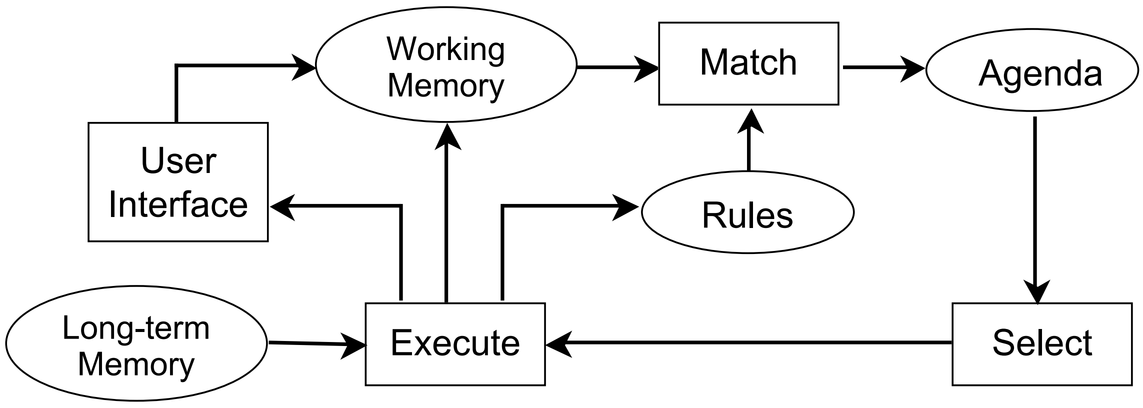
In a production system, working memory (from perception or memory) acts as stimuli that trigger associations with information in long term memory and actions.

The association of working memory and productions poses a problem of algorithmic complexity. If the rules can contain variables, then the algorithmic complexity of matching (activation) is exponential in the number of conditions.

Production System architecture

In a production system, data and concepts in working memory are associated with data and procedures in long-term memory.

Rules are encoded as condition-action pairs: Condition* ⇒ Action*



The system implements an "inference engine" that operates as a 3-phase cycle:

The cycle is called the "recognize act" cycle.
The phases are:

Match: match facts in working memory to conditions of rules to produce "activations" (associations of WM and rules).
Select: Select an activation for execution.
Execute: Execute the actions specified in the activation.

In this architecture, each element in working memory is labeled with an index. In each cycle, the conditions part of the rules are matched to the working memory. The association of working memory and rules are stored in an agenda as "activations". One of these activations is then selected for execution.

The list of associations of facts and rules are stored in an agenda. One of the associations is selected for execution.

There are several different models for how to sort the agenda. The most popular is to use a stack "Last In First Out" or LIFO.

The NASA CLIPS rule based programming environment

We will illustrate rule based programming using the CLIPS 6.0 environment created by NASA. Rule based programming languages went through a number of evolutions in the 1980's

OPS-5 ('78) => ART ('80) => CLIPS ('85) => CLIPS 6 ('94)

The OPS system (invented by Newell at CMU) was used to construct a very successful expert system for configuring VAX computers (R1). R1 gained several hundred million dollars for Digital Equipment Corporation from 1980 to 1985.

The OPS system was commercialized by DEC. However OPS was public domain, and not protected from copyright.

OPS was copied by Teknowledge under the name "ART" Automated Reasoning Tool. Unfortunately ART was built in the LISP language and was very expensive. (50 K$ for the ART license, 50 K$ for a LISP machine).

NASA selected ART as its standard. However, because of the cost, in 1985, NASA engineers reprogrammed ART in C. The result was called CLIPS. (C-language integrated production system). As a public organization, NASA distributed CLIPS for free (Before GNU and ShareWare).

Successive versions of CLIPs have been distributed and a substantial user community has emerged. We will use a version, CLIPS 6.0, that is programmed using object oriented programming. This version integrates many different knowledge representation styles, including Rules, Schema, and a logic based Truth Maintenance system.

For NASA, CLIPS is used to construct Expert Systems for
1) Process Control
2) On-Board Error Diagnostic
3) Mission Planning
4) Logistics Planning.

The CLIPS Program Interpreter

The CLIPS interpreter interprets instruction in a Lisp-like pre-fix notation:

    (operator data*)

Programs are composed of "expressions" enclosed in parentheses. The first symbol after an open parenthesis is an operator. Any remaining symbols are data. The star indicates zero or more elements.

Expressions are both programs and data. They can be created dynamically by other programs. Expressions may be composed recursively:

    (operator (operator (operator data*)))

CLIPS includes a large number of pre-defined operators. These are described in the CLIPS Basic Programming guide.

Any of the predefined operators can be included in the action part of a rule, may be used in a user-defined function, or can be entered directly into the interpreter by the user.

Working Memory: The Facts List

In CLIPS, entities (or concept) in working memory are called "Facts"

Facts are created and destroyed by the commands assert, retract, reset and clear.

```
CLIPS> (ASSERT <<FACT>>)
CLIPS> (RETRACT <<Fact-Index>>)
CLIPS> (RESET) - Empties the working memory
CLIPS> (CLEAR) - reinitialize the entire system
```

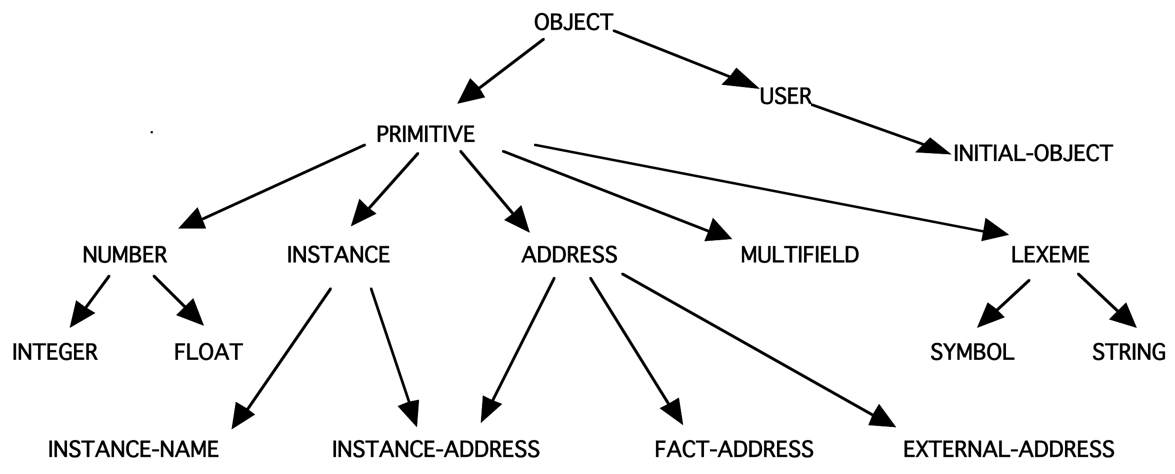
The working memory of CLIPS is called the "Facts List". Facts represent concepts represented as lists or as schema defined by templates.

A Fact can represent:
- A list of items (symbols, strings, integers or floats)
- A schema template composed of a "name" and a list of "attribute-value" pairs.

• An object from the CLIPS object oriented system.

CLIPS objects are used to represent Frames.

Data primitives (items) belong to a hierarchy of 13 primitive types.



It is also possible to define new types. Version 6.0 of CLIPS contains an object-oriented environment named COOL. ("C-Language Object Oriented Language"). User defined types belong to the class COOL class USER

In later lessons we will use the COOL. For now, we will work with lists and structures.

Typing (RESET) creates an "initial fact" with index 0.
Typing (Clear) deletes all templates, rules, facts, classes etc..
And then creates an "initial fact" with index 0.

f-0 (initial-fact)

There are 5 keywords that cannot be used in the first field in a fact:
"test", "and", "or", "not", "declare".

Each fact is identified by a unique index number. Indices are allocated in order. They are fundamental to the logic of CLIPS.

```
CLIPS> (assert (a b c))
```

ATTN (assert (1 2 3)) will generate an error. The first item in a fact must be a symbol

Defining Schema with Deftemplates

A template is a schema for a concept. Templates are named sets of "attribute - value" pairs. Such structures are defined with deftemplate.

A template makes it possible to manipulate attributes of a fact without having to scan through a list. Attributes are called "slots" in clips. Example:

```
(deftemplate person            ;  A template for a person
   "template for a person"     ;  Optional Comment
   (slot name(default "John Doe"))   ; Default name
   (slot age (type INTEGER))  ; default will be 0
)
```

Instances of templates are created by (assert).

```
(assert (person))
(assert (person (name "joe")))
(assert (person (name "john") (age 20)))
```

A number of commands exist inspecting templates

(list-deftemplates)
(ppdeftemplate <template-name>)
(undeftemplate <template-name>)

examples:

```
CLIPS> (list-deftemplates)
initial-fact
person
For a total of 2 deftemplates.
CLIPS> (ppdeftemplate person)
(deftemplate MAIN::person "template for a person"
   (slot name (default "John Doe"))
   (slot age (type INTEGER)))
CLIPS>
```

We can define types and default values for slots.
We can also define a list of allowed symbols for a slot value.

example :

```
(deftemplate person        ;  A template for a person
"template for a person" ;  Optional Comment
   (slot name          ; Names of a person (string)
   (type STRING)(default "Pierre Dupont"))
   (slot age           ; age of the person
      (type INTEGER)   ; default will be 0
      (range 0 120))   ; Possible ages - 0 to 120
   (slot Profession      ;
      (type SYMBOL)
      (allowed-values  artist  engineer  salesman  manager
         other none)
 )
)
```

For numbers we can define a range

```
   (slot age(type NUMBER) (range 0 120))
```

Attempts to assign a value outside the range will cause an error.

deffacts

A predefined list of facts can be created by the expression "deffacts".

(deffacts <NOM> ["<comment>"]
[(<<FAIT-1>>) (<<FAIT-2>>) ... (<<FAIT-N>>))

the command (reset) will empty the facts list and create an "initial-fact" and to create the list of default facts.

examples :

```
(deftemplate place
   (slot name (type SYMBOL) (default NIL))
   (slot x (type NUMBER) (default -1))
   (slot y (type NUMBER)(default -1))
   (multislot neighbors (default NIL))
)

(deffacts network-of-places
   (place (name A) (x 0) (y 0) (neighbors B C))
   (place (name B) (x 0) (y 1) (neighbors A D))
   (place (name C) (x 1) (y 0) (neighbors A I))
)

CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (place (name A) (x 0) (y 0) (neighbors B C))
f-2     (place (name B) (x 0) (y 1) (neighbors A D))
f-3     (place (name C) (x 1) (y 0) (neighbors A I))
For a total of 4 facts.
```

Rules in CLIPS

CLIPS rules allow programming of reactive knowledge that associate facts.
Rules are defined by the "defrule" command.

```
(defrule <rule-name> [<comment>]
   [<declaration>]          ; Rule Properties
   <conditional-element>*   ; Left-Hand Side (LHS)
=>
   <action>*)               ; Right-Hand Side (RHS)
```

Defrule defines a rule.
If the rule with the same name exists, it is replaced with the new rule.

There is no limit to the number of conditions or actions (* means 0 or more).
Actions are executed sequentially.

Rules with no condition are activated by (Initial-Fact). This can be used for initialisation.

The syntax for condition elements is complex:

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

A condition element (CE) can be a list or a template or user defined object.

List: (<constant-1> ... <constant-n>)
Deftemplate:

```
   (<deftemplate-name> (<slot-name-1> <constant-1>)
                                     •
                                     •
                                     •
                       (<slot-name-n> <constant-n>))
```

A CE can contain constant values or variables.

Variables

Variables are represented by ?x
There are two sorts of variables in CLIPS:

Index Variables: are assigned the index of a fact that matches a CE.
Attribute Variables: Contain the value of an item that matches a CE.

Index Variables

Variable : ?x
Index variables are used to identify a fact that has matched a CE
This can be used to retract of modify the fact.

```
(defrule rule-A
   ?f <- (a)
=>
   (printout t "Retracting  " ?f  crlf)
   (retract ?f)
)
```

Attribute Variables

Attribute variables are assigned the value of an item that matched a CE.
These can be used to
1) Recover the value for computation
2) Detect matching facts.

Syntax for attribute variables.

?var - Defines a variable named var.
    The matching value is assigned to ?var.
$?list - Defines a list of variables named list
? - An unnamed variable. No data is stored.
$? - An unnamed list. no data is stored.

WITHIN condition elements, values are implicitly bound to variables.

Examples :

```
(assert (a b c))
(assert (a b c d e f))
(assert (d e f))

(defrule a
   (a)  ;; matches only facts with a single item. (a)
=>
(printout t "a" found crlf)
)

(defrule abc
   (a b c)  ;; matches only facts of the form (a b c)
=>
(printout t "a b c" crlf)
)

(defrule abx
   (a b ?x)
=>
(printout t "a b  and ?x = " ?x crlf)
)

(defrule ax-
   (a ?x  ?)
=>
(printout t "x = " ?x crlf)
)

(defrule process-a-list
   (a $?x)
=>
(printout t "The list is " $?x crlf)
)

(defrule extract-every-element-from-a-list
   (a $?x $?)
=>
(printout t "The list is " $?x crlf)
)
```

Scanning a list

The following "trick" is used to obtain each item from a list:

```
(defrule Extract-every-element-from-list
   (a $? ?x $?)
=>
(printout t "x = " ?x crlf)
)

$?   Matches nothing, or 1 item, or 2 items etc...
?x   matches the next item.
$?    Matches the rest of the list.

This rule will provide one activation for each item after
the first item in the fact.


(defrule increment-x-example
  ?f <- (a ?x)
=>
   (printout t "x = " ?x crlf)
   (bind ?x (+ ?x 1))
   (printout t "now x = " ?x crlf)
   (modify ?f (a ?x))
)

(clear)

(deftemplate a (slot x))

(defrule increment-x-example
  ?f <- (a (x ?x))
=>
   (printout t "x = " ?x crlf)
   (bind ?x (+ ?x 1))
   (printout t "now x = " ?x crlf)
   (modify ?f (x ?x))
)
```

WITHIN the action part of a rule, values may be assigned by
(bind ?Var Value)
e.g. (bind ?x 3) assigns 3 to ?x

ATTN: DO NOT use (bind) in condition elements

```
(defrule test
?c <- (a b c)
=>
(bind ?c oops)
(printout t ?c crlf)
)
```

Rule Activations (associations of a rule with facts that match conditions)
are placed on the agenda.

```
(deftemplate person
    "A record for a person"
    (slot family-name)
    (slot first-name)
)

(assert (person (family-name DOE) (first-name John)))
(assert (person (family-name DOE) (first-name Jane)))

(defrule Find-same-name
    ?P1 <- (person (family-name ?f) (first-name ?n1))
    ?P2 <- (person (family-name ?f) (first-name ?n2))
=>
    (printout t  ?n1 " " ?f" and  " ?n2 " " ?f " have the
same family name" crlf)
)

CLIPS>  (assert  (person  (family-name  DOE)  (first-name
John)))
<Fact-1>
CLIPS>  (assert  (person  (family-name  DOE)  (first-name
Jane)))
<Fact-2>
CLIPS> (defrule Find-same-name
    ?P1 <- (person (family-name ?f) (first-name ?n1))
    ?P2 <- (person (family-name ?f) (first-name ?n2))
=>
    (printout t  ?n1 " " ?f" and  " ?n2 " " ?f " have the
same family name" crlf)
)
```

```
CLIPS> (run)
Jane DOE and  Jane DOE have the same family name
Jane DOE and  John DOE have the same family name
John DOE and  Jane DOE have the same family name
John DOE and  John DOE have the same family name
```

Question: Why does the rule execute 4 times?
Answer: Twice because a fact can match itself and twice because the rule matches Jane with John as well as John with Jane

```
(defrule Find-same-name
   ?P1 <- (person (family-name ?f) (first-name ?n1))
   ?P2 <- (person (family-name ?f) (first-name ?n2))
   (test (neq ?n1 ?n2))
=>
   (printout t  ?n1 " " ?f" and  " ?n2 " " ?f " have the
same family name" crlf)
)
```

Rule Syntax: Constraints

Variable assignment and matching in conditions can be "constrained" by constraints. There are two classes of constraints: "Logic Constraints" and Predicate Functions

Logic Constraints are composed using"&", "|", "~"

"&"- AND - Conjunctive constraint
"|" - OR - Disjunctive Constraint
"~" - NOT - Negation

```
(defrule Find-same-name
   ?P1 <- (person (family-name ?f) (first-name ?n1))
   ?P2 <- (person (family-name ?f)
      (first-name ?n2&~?n1))

=>
   (printout t  ?n1 " " ?f" and  " ?n2 " " ?f " have the
same family name" crlf)
)
```

example :

```
(?x & green | blue)  - ?x must be green or blue for the
condition to match
(?x & ~red)  -  ?x cannot match red.

(defrule Stop-At-Light
   (color ?x & red | yellow)
=>
   (assert (STOP))
   (printout t "STOP! the light is " ?x crlf)
)
 (assert (color red))
```

Predicates

Predicates provide functions for defining constraints.
For Predicate functions, the variable is followed by ":".

`(?x&:(<predicate> <<arguments>>)`	The condition is satisfied if 1) a value is assigned to ?x , and 2) the predicate is true for arguments
`(?x|:(<predicate> <<arguments>>)`	The condition is satisfied if 1) a value is assigned to ?x , or 2) the predicate is true for arguments
`(?x&~(<predicate> <<arguments>>)`	The condition is satisfied if 1) a value is assigned to ?x , and 2) the predicate is false for arguments

```
(defrule Find-same-name
   ?P1 <- (person (family-name ?f) (first-name ?n1))
   ?P2 <- (person (family-name ?f)
      (first-name ?n2&:(neq ?n1 ?n2)))

=>
   (printout t  ?n1 " " ?f" and  " ?n2 " " ?f " have the
same family name" crlf)
)
```

There are many predefined predicates. For example.
(numberp <arg>) - true if <arg> is a PRIMITIVE of type NUMBER
(stringp <arg>) - true if <arg> is a PRIMITIVE of type STRING
(wordp <arg>) - true if <arg> is a PRIMITIVE of type WORD
Additional functions can be found in the manual

```
(defrule example-1
  (data ?x&:(numberp ?x))
  =>)

(defrule example-2
  (data ?x&~:(symbolp ?x))
  =>)
```

```
(defrule example-3
  (data ?x&:(numberp ?x)&:(oddp ?x))
  =>)

(defrule example-4
  (data ?y)
  (data ?x&:(> ?x ?y))
  =>)

(defrule example-4
     (data ?y)
 not (data ?x&:(> ?x ?y))
  =>)


(defrule example-5
  (data $?x&:(> (length$ ?x) 2))
  =>)
```

The ACTION part (RHS) of a rule

In the action part (or RHS) the rule contains a sequence of actions.
Any command recognized by the interpreter can be placed in the action part of a rule.
Each action enclosed in parentheses (<fonction> <<args>>*)
The first symbol in parentheses is interpreted as a function.

New variables can be defined and assigned with bind: (bind ?x 0).
Values may be read from a file or from ttyin by read and readline.

example :
```
(defrule ask-user
    (person)
=>
    (printout t "first name? ")
    (bind ?firstname (read))
    (printout t "Family name? ")
    (assert (person ?firstname (read)))
)
```

System Actions

1) assert : facts are created with "ASSERT"
Syntax : (assert (<<fait>>) [(<<faits>>)])

```
(defrule I-Think-I-Exist
     (I think)
=>
     (assert-string "(I exist)")
)
```

2) retract - Facts are deleted with retract

```
(defrule I-dont-think-I-Exits
     ?me <- (I do not think)
=>
    (printout t "oops!" CRLF)
    (retract ?me)
)
```

3) Str-assert Assert a string

```
(defrule I-Think-I-Exist
     (I think)
=>
     (str-assert "I Think therefore I exist")
)
```

4) Halt : Stop execution.