

The CLIPS environment

- CLIPS is an environment to develop knowledge based systems
- It defines a programming language that allows to represent declarative and procedural knowledge
- This language define a syntax to define production rules and frames
- The execution environment is an inference engine that uses forward chaining
- This inference engine is implemented over an interpreter of the CLIPS language

Notes

The CLIPS programming language

- The CLIPS programming language derives its syntax from the LISP language
- It is a parenthesized language that uses prefix notation
- The predefined data types are: real, integer, string, symbol, pointer to fact, instance name and pointer to instance
- The usual data types have the usual operators
- CLIPS programming language joins three programming paradigms: rule programming, functional programming, object oriented programming

Notes

CLIPS production rules language - facts

- The two elements that allow to represent problems using production rules are facts and rules
- In CLIPS facts can be of two kinds *ordered facts* and *deftemplate facts*
- Ordered Facts have a free syntax, therefore they have not a fixed structure, they follow the pattern:

$$(\text{relation } p_1 \ p_2 \ \dots \ p_n)$$
- relation has to be a symbol, the rest of parameters can be of any kind, for examples

$$(\text{father john peter})$$

$$(\text{num-children john } 2)$$

Notes

CLIPS production rules language - deftemplates

- The structure of Deftemplate facts has to be declared, they could be assimilated to a frame-like representation
- The definition of this facts is done using (*slots*) that define their characteristics. Each slot can have constraints such as data type, cardinality or a default value that can be a constant or a function that can compute it

```
(deftemplate template-name "comment"
  (slot slot-name)
  (multislot slot-name))
```

- For example:

```
(deftemplate person
  (slot name (type STRING))
  (slot age (type INTEGER) (default 0)))
```

Notes

CLIPS production rules language - creating facts

- The creation of new facts are performed by using the construction `assert` (just one) or `deffacts` (a set), for example:

```
(assert (father joseph john))
(assert (person (name "peter") (age 25)))
(deffacts my-facts
  (house red) (ball green)
  (person (name "Louis") (age 33)))
```

Notes

CLIPS production rules language - facts

- (`facts`) allows to know what facts has been created
- (`clear`) deletes all existing facts
- (`retract <fact-index>`) deletes the fact identified by the given index
- (`get-deftemplate-list`) returns the list of defined deftemplates

Notes

CLIPS production rules language - rules

- CLIPS rules are defined by:
 - A left hand side (LHS) that defines the conditions to hold
 - A right hand side (RHS) that defines the actions to execute

- Syntax:

```
(defrule rule-name "comment"
  (condition-1) (condition-2) ...
  =>
  (action-1) (action-2) ...)
```

Notes

CLIPS production rules language - variables

- To build patterns in the conditions of the rules we need variables
- CLIPS variables are denoted by a question mark before their name (?variable) if the variable is for just one value or by a dollar sign and a question mark if the variable is for a list of values \$?variable
- There are anonymous variables or wild-cards (their value is immaterial) for just one value ? or for a list of values \$?
- In running time inference engine looks for the values that can instantiate the variables that allow the conditions of the rules to hold
- Rule variables are local variables, if we want to define global values we must use the constructor defglobal (global variables are denoted as ?*variable*)

Notes

CLIPS production rules language - LHS

- In the left hand side of a rule can appear different kinds of conditions
 - Constants, **Patterns** with variables or wild-cards: they can be instantiated with facts from the knowledge base
 - Expressions using logic connectives as not, and, or, exist and forall with patterns
 - Test for expressions using the variables instantiated by the patterns (test)
- Patterns** can also indicate what facts can instantiate the rules by using constraints over the variables or with constant values
- Simple constraints can be stated by using logic connectives ~ (not), & (and) and | (or)
- More complex constraints can be also defined by using the operator :

Notes

CLIPS production rules language - examples

- Person over 18: `(person (age ?x&:(> ?x 18)))`
- Person named john or peter: `(person (name john|peter))`
- Two persons with different name: `(person (name ?x)) (person (name ?y&~?x))`
- Nobody has the name peter: `(not (person (name peter)))`
- Everybody is over 18: `(forall (person (name ?n) (age ?x)) (test (> ?x 18)))`

Notes

CLIPS production rules language

- The address of a fact that matches a pattern can be obtained using the operator `<-`, for example:


```
(defrule my-rulea
  ?x <- (person (name john))
=>
  (retract ?x)
)
```
- In the right hand side of a rule we can use any valid CLIPS construction (see the manual)

Notes

CLIPS production rules language - modules

- CLIPS rules can be organized in modules
- The main advantage is to be able to structure the knowledge and to focus the rules execution depending on their goal
- The definition of a module uses the constructor `(defmodule <name> comment«export-import»)`
- Nothing that is defined inside a module is visible if it is not exported explicitly
- To use constructions from other module they have to be imported explicitly
- There is a default module named MAIN that has all that is not defined in other module

Notes

CLIPS production rules language - modules

- We can assign a construction to a module by using as a prefix the name of the module followed by two colons ::, for example:

```
(deftemplate A::cube (slot size))
```
- Export of constructions from a module is done by including the sentence `export` in its definition. Anything that is defined can be exported, for example:

```
(defmodule A (export deftemplate cube))
(defmodule A (export deftemplate ?ALL))
```
- Import of constructions from other modules is done by including the sentence `import` in its definition. Anything that is visible from other modules can be imported, for example:

```
(defmodule B (import A deftemplate cube))
```

Notes

CLIPS production rules language - focus

- We can restrict the modules that are used to execute rules using the sentence `(focus <module>*)`
- This sentence can be used in the right hand side of a rule to explicitly change the module
- It also allows to focus on the module of the last executed rule declaring the rule property `auto-focus`, for example:

```
(defrule JOHN::my-rule
  (declare (auto-focus TRUE))
  (person (name john))
  => ...
```

Notes

Conflict Resolution Strategies

Notes

The inference engine has defined some conflict resolution strategies

- Depth-first, newest rules have priority
- Breadth-first, oldest rules have priority
- Simplicity, before same priority the less specific rule is preferred (specificity is measured respect the complexity of the conditions of the rules)
- Complexity, the more specific rules have priority

Conflict Resolution Strategies

- LEX strategy, priority to the rules instantiated with the newest facts, ordering the facts that instantiate the rule by recency and following a lexicographic order
- MEA strategy, rules are ordered by the recency of the fact that instantiates the first condition, ties are break using LEX strategy
- Random, rules are fired in random order

Notes

CLIPS programming language

- CLIPS includes a pseudo-functional programming language
- It can be used to define new functions or to implement the actions to be performed on the right hand side of the rules
- Each sentence or control structure is a function that receives parameters and returns a result (functional paradigm)

Notes

CLIPS programming language - syntax

These are the more commonly used sentences and control structures:

- (bind <var> <value>): assigns a value to a variable, returns the value assigned
- (if <exp> then <action>* [else <action>*]): alternative control structure, returns the value of the last evaluated expression
- (while <exp> do <action>*): conditional iteration control structure, returns false, except if a return sentence breaks the iteration
- (loop-for-count (<var> <val-i> <val-f>) do <action>*): iteration over a range of values, returns false, except is there is a return sentence

Notes

CLIPS programming language - syntax

- (progn <action>*) : Executes a set of actions sequentially, returns the value of the last one
- (return <expr>) : Breaks the execution of the current control block and returns the value of the expression
- (break) : Breaks the execution of the current control block
- (switch <expr> (case (<comp>) then <action>*)* [(default <action>*)]) : Case structure each case compares with the value of the expression. Returns the last evaluated expression or false if no case clause holds

Notes

CLIPS programming language - defining functions

- The constructor `deffunction` allows to define new functions


```
(deffunction <name> "Comment"
  (<?parameter>* [<$?wildcard-parameter>])
  <action>*)
```
- The list of parameters can be variable in size, the wild-card parameter puts in a list the remaining parameters
- The function returns the last evaluated expression

Notes

CLIPS Object Oriented language

- CLIPS defines also an object oriented language that complements their ability to represent knowledge
- It could be considered an extension of the constructor `deftemplate` that allows to use *frames* as a representation language
- As in Object Oriented languages/Frame languages we can define slots and methods
- CLIPS defines an initial set of classes that organize the CLIPS predefined data types in a hierarchy

Notes

CLIPS Object Oriented language

- The constructor `defclass` allows to define a class
- In order to define a class has to be specified:
 - 1 The name of the class
 - 2 A lists of its superclasses (it will inherit their slots and methods)
 - 3 If the class is abstract or not (if it is not abstract we can define instances of the class)
 - 4 If it is allowed to use the instances of the class as patterns of the LHS of a rule
 - 5 Definition of the slots of the class (slot, multi-slot)
- All classes must have at least a superclass

Notes

CLIPS Object Oriented language

For example:

```
(defclass living-being
  (is-a USER)
  (role abstract)
  (pattern-match non-reactive)
  (slot breathes (default si)))

(defclass person
  (is-a living-being)
  (role concrete)
  (pattern-match reactive)
  (slot name))
```

Notes

CLIPS Object Oriented language - slots

The definition of a slot includes a set of properties, such as:

- `(default ?DERIVE|?NONE|<exp>*)`
- `(default-dynamic <expr>*)`
- `(access read-write|read-only|initialize-only)`
- `(propagation inherit|no-inherit)`
- `(visibility public|private)`
- `(create-accessor ?NONE|read|write|read-write)`
- We can also declare the data type, cardinality, ...

Notes

CLIPS Object Oriented language - instances

- To create an instance of a class the constructor `make-instance` is used
- When we create an instance we must give the values of their slots, for example:

```
(make-instance john of person (name "john"))
```
- We can create sets of instances with the constructor `definstances`, for examples:

```
(definstances persons
  (john of person (name "john"))
  (mary of person (name "mary"))
)
```

Notes

CLIPS Object Oriented language- messages

- All the interaction with objects is perform using *messages*
- These messages has handlers that process the messages and perform the indicated task
- These *handlers* are defined using the constructor `defmessage-handler`, their syntax is identical to the syntax of the functions

```
(defmessage-handler <class> name <hand-type> (<param>*) <expr>*)
```

- There are different kinds of handlers `primary`, `before`, `after`, `around`, we will only use `primary` handlers

Notes

CLIPS Object Oriented language- messages

- By default each class has defined a set of handlers, for example: `init`, `delete`, `print`
- Defining `create-accessor` in a slot we are creating two messages, `get-name_slot`, `set-name_slot` to access and modify the slot
- The access to a slot inside the code of a handler is done using the variable `?self`, and putting `:` before the name of the slot, for example:

```
(defmessage-handler person print-name ()
  (printout t "Name:" ?self:name crlf))
```

Notes

CLIPS Object Oriented language - messages

- To send a message to an instance is done using the sentence `send`, the name of the instance has to be between square brackets, for example:


```
(send [john] print-name)
(send [john] set-name "peter")
```
- Handlers can be defined in each class, hence each subclass can run the handlers of their superclasses. For the handlers of type `primary` it is used the handler of the most specific class, to execute the handlers of the superclasses the sentence `call-next-handler` as to be used
- It must be at least a `primary` handler for each message

Notes

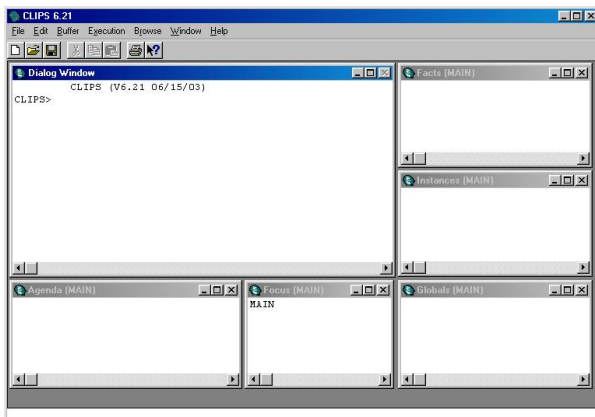
CLIPS Object Oriented language - instances and rules

- In order to use instances on the LHS of a rule the sentence `object` has to be used, for example:


```
(defrule rule-person
  (object (is-a person) (name ?x))
  =>
  ...
)
```
- The class must be declared as `pattern-match reactive`
- Modifying a slot allows an instance to be instantiated again by same rule if its conditions hold

Notes

CLIPS - The IDE



Notes

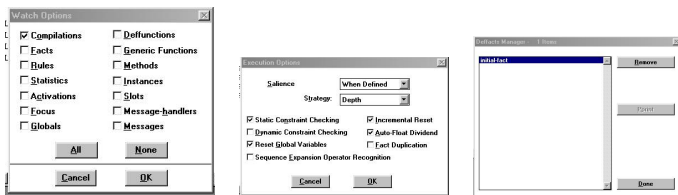
Using the CLIPS interpreter - windows

- Interpreter window
- Facts window (Fact defined and obtained during execution)
- Instances window (instances from the defined classes)
- Defglobals windows (global variables)
- Focus window (modules in the current focus)
- Agenda window (possible rule activations)

Notes

Using the CLIPS interpreter

Using the menus the most common commands can be accessed (run, stop, debugging, definitions, configuration)



Notes

Using the CLIPS interpreter - commands

- (facts), lists the facts in the facts base
- (reset), deletes all the facts in the fact base
- (clear), deletes all the facts and rules
- (rules), lists the rules defined
- (load "name.clp"), loads a CLIPS file
- (save "name.clp"), saves the deffacts, defrule and deffunction to a file

Notes

Using the CLIPS interpreter - running

- `(run [<integer>])`, runs the inference engine, a number of steps or until it stops if no parameter is passed on
- `(agenda [<module>])`, shows all the possible rule activations o the activations from a module
- `(focus <module>+)`, changes the execution focus in the given modules
- `(halt)`, halts the inference engine
- `(set-strategy <strategy>)`, changes the conflict resolution strategy

Notes

Notes

Notes
