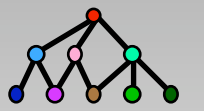
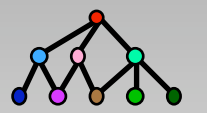
Procedural Programming



Outline of talk:

- **Deductive Kb with Multiple Paradigms**
- **Production rules**
- **Methods**
- **Lisp-to-Loom Interface**
- **Interpretations of Updates**

Multiple Paradigm Programming

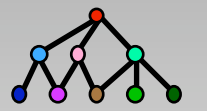


Idea: Suite of programming paradigms that each exploit a dynamically changing deductive knowledge base.

Loom paradigms:

`Data driven`	`(production rules, monitors)`
`Methods`	`(pattern-directed dispatch)`
`Procedural`	`(Lisp)`

Upgrading Traditional Paradigms



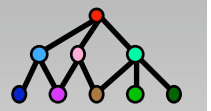
```
(defproduction P1
   :when (:detects (Foo ?x))
   :do ((print "New Foo")))
(defmethod M1 (?self)
   :situation (Foo ?self)
   :response ((print "It's a Foo all right")))
```

Innovations:

- **"Foo" can expand to an arbitrarily complex description;**
- **"Edge-triggered" productions;**
- **Pattern-based method dispatching.**

USC ISI

Production Rule Semantics



```
(defproduction <name>
    :when <condition> :perform <action>)
```

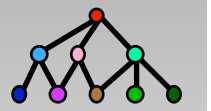
Semantics: Whenever a set of variable bindings in <condition> becomes true (provable), call <action> with that set of bindings.

Example:

```
(defproduction P2
    :when (and (Switch ?s)
               (:detects (status ?s 'on)))
  :perform (turn-on (appliance-of ?s)))
```

The `:when` condition of a production must include at least one of the transition operators `:detects`, `:undetects`, or `:changes`.

Semantics of :detects



```
(:detects (A ?x))
```

is defined as

```
      (and (A ?x)
           (:previously (:fail (A ?x))))
```

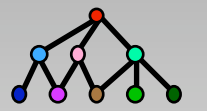
```
(:previously (B ?x))
```

is defined as

```
      (:at-agent-time (- *now* 1)
          (B ?x)))
```

USC ISI

Semantics of Detects (cont.)



```
(:detects (:and (A ?x) (B ?x)))
```

will trigger if A and B become true simultaneously

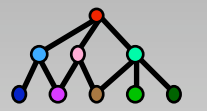
or if A becomes true and B is already true

or if B becomes true and A is already true

```
(:and (:detects (A ?x))
      (:detects (B ?x)))
```

will trigger only if A and B become true simultaneously

Production Rule Semantics (cont).



All production rule instantiations at the end of an update cycle are fired in parallel.

- **No conflict resolution (this is a feature!)**
- **Effects of one production cannot inhibit firing of another (parallel) production.**

Rationale:

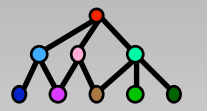
- **We want productions to be "well-behaved" (no race conditions);**
- **Preference semantics is the province of the method paradigm.**

Division of responsibility:

- **Production determines <u>when</u> to perform task;**
- **Method determines <u>how</u> to perform task.**

Task Scheduling

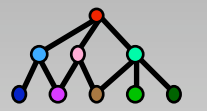
Productions can post tasks on a queue rather than executing them immediately.



```
(defproduction P5
   :when (and (:changes (home-team-score ?game))
              (basketball-game ?game))
   :schedule (celebrate)
   :priority :low)
(defproduction P6
   :when (and (:changes (home-team-score ?game))
              (football-game ?game))
   :schedule (celebrate)
   :priority :high)
```

USC ISI

Monitors



Monitors are productions that fire only when specifically designated instances undergo property transitions.

```
(defmonitor Watch-for-Redraw
   :when (or (:changes (color ?object))
             (:changes (size ?object)))
   :do ((redraw (slot-value ?object 'window)))
(tellm (color Thing5 'Red))
```

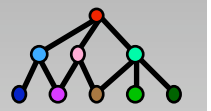
nothing happens

```
(attach-monitor 'Thing' Watch-for-Redraw)
(tellm (color Thing5 'Green))
```

calls redraw

Monitors generalize the active value paradigm

Methods



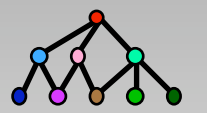
defaction: *Defines Loom equivalent of ``generic function''.*

defmethod: *Defines procedurally-invoked situation-response rule.*

```
(defmethod <name> (<parameters>)
   :situation <situation>
   :response <response>)
```

USC ISI

Method Filters



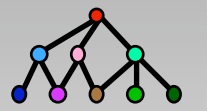
Most frequent modes of method use. Given a call to invoke an action M:

(1) execute all methods named M whose situations are satisfied, or

(2) execute the most specific among those methods named M whose situations are satisfied.

A ``filter sequence'' determines the criteria for choosing which methods to fire (among those that are eligible).

Method Filters Example



```
(defaction M2 (?x ?y) :filters (:perform-all))
(defmethod M2 (?x ?y)
   :situation (= ?x ?y)
   :response ((print "EQ")))
(defmethod M2 (?x ?y)
   :situation (<= ?x ?y)
   :response ((print "LE")))

(perform (M2 3 4))
   --> "LE"

(perform (M2 4 4))
   --> "LE"
       "EQ"          both methods fire

(defaction M2 (?x ?y) :filters (:most-specific))
(perform (M2 4 4))
   --> "EQ"          only the most specific method fires
```