Example 3

This third example examines the use of General Problem Solver (GPS) style operators used within a blocks world environment. Since first presented by Newell and Simon [8] this has become a classic example of symbolic computation.

At one level of abstraction, commands like (pick-up ?x) and (drop-it-on ?y) are issued to a virtual robot existing in a simple blocks-world environment. The robot *carries out* these commands by effecting changes to the description of its virtual world. At another level, individual operators are defined in terms of their preconditions (what needs to exist in the world for the operator to be used) and their effects. The effects of operators are defined in two parts: what is no longer true about the world after the operator is applied (parts of the world description that the operator deletes) and what becomes true (parts of the world description that the operator adds). In this way simple operators can be described in terms of three sets of facts **pre**conditions **del**etions and **add**itions.

Using the type of world description below, the (pick-up ?x) operator could be defined as shown.

```
(defparameter blocks
 '((isa b1 block) (isa b2 block)
   (isa p1 pyramid) (supports b1 p1)
   (supports floor b1)
   (supports floor b2)
   (cleartop p1) (cleartop b2)
   (cleartop floor) (holds nil)))

(defparameter pick-up    ; pick up object ?x
  '((pre (holds nil) (cleartop ?x)
         (supports ?y ?x))
    (del (holds nil) (supports ?y ?x))
    (add (holds ?x)  (cleartop ?y))
    ))

; note: pick-up is an association list
; so its parts are accessed using ->

> (-> pick-up 'del)
→ ((holds nil) (supports ?y ?x))
```

Given the kind of operator description above, a generalised apply operator function can be built around the use of the matcher.

```
(defun apply-op (op object world)
 (let ((pre (-> op 'pre))
       (del (-> op 'del))
       (add (-> op 'add))
       )
   (all-present
     (pre world `((x ,object)))
       ($+ (match>> add)
           ($- world (match>> del)))
      )))

> (apply-op pick-up 'p1 blocks)
→ ((cleartop b1) (holds p1)
    (cleartop floor) (cleartop b2)
    (cleartop p1) (supports floor b2)
    (supports floor b1)
    (isa p1 pyramid)
    (isa b2 block) (isa b1 block))
```

This example is concluded by creating more GPS operators and a mechanism which will apply a series of commands. Notice below, that with simple operators like the ones used here the problems of defining and using operators have become abstract and conceptual. The difficulties associated with programming the *actions* of operators have largely been removed and are dealt with by the matcher. Learners are able to concentrate on the mechanics of applying these types of operators in symbolic world environments without spending large amounts of time writing Lisp functions. As an additional benefit, the use of matcher patterns encourages a more declarative approach.

```
(defparameter ops
 '((pick-up     ; as defined above
    (pre (holds nil) (cleartop ?x)
         (supports ?y ?x))
    (del (holds nil) (supports ?y ?x))
    (add (holds ?x)  (cleartop ?y))
   )
  (drop-on  ; puts one obj on another
   (pre (holds ?obj) (cleartop ?x))
   (del (holds ?obj) (cleartop ?x))
   (add (holds nil)(supports ?x ?obj))
  )))

;arm-controller takes a series of commands

(defun arm-controller (world commands)
  ;; this provides textual output
  (dolist (com commands)
    (format t "~2&Applying ~a..." com)
    (setf world
      (apply-op (-> ops (first com))
                (second com) world))
    (format t "ok~%")
    (pprint world)
    ))

> (arm-controller blocks
       '((pick-up p1) (drop-on b2)))

→ Applying (pick-up p1)...ok
→ ((cleartop b1) (holds p1)
    (cleartop floor) (cleartop b2)
    (cleartop p1) (supports floor b2)
    (supports floor b1)
    (isa p1 pyramid)
    (isa b2 block) (isa b1 block))

→ applying (drop-on b2)...ok
→ ((supports b2 p1) (holds nil)
    (isa b1 block) (isa b2 block)
    (isa p1 pyramid)
    (supports floor b1)
    (supports floor b2) (cleartop p1)
    (cleartop floor) (cleartop b1))
nil
```