sgf.clj

;; Predefined lists

(def uc-letters (set (doall (map char (range (int \A) (+ (int \Z) 1))))))

(def uc-board-letters '(\A \B \C \D \E \F \G \H \J \K \L \M \N \O \P \Q \R \S \T))

(def board-letters (doall (map #(char (+ (int \a) (- (int %) (int \A)))) uc-board-letters)))

(def numbering (zipmap board-letters (range 19)))

;; Working with SGF-files

(defn lazy-read [file len]

(let [rdr (clojure.java.io/reader file)

is-closed? (atom false)

array (char-array len)]

(fn next-char []

(if @is-closed?

nil

(let [result (.read rdr array)]

(if (= -1 result)

(do (.close rdr)

(swap! is-closed? complement)

nil)

array))))))

(defn unread-char [chr stack]

(swap! stack (partial cons chr))

chr)

(defn read-next-char [lazy-read-fn stack]

(if-let [result (first @stack)]

(do

(swap! stack rest)

result)

(do

(let [string (lazy-read-fn)

result (first string)]

(swap! stack concat (rest string))

result))))

(defn lexer

([input-chars stack]

(reverse (lexer input-chars stack nil)))

([input-chars stack result]

(letfn [(process-chars [input-chars stack result]

(if-let [next-char (read-next-char input-chars stack)]

(condp = next-char

\( :treestart

\) :treeend

\; :nodestart

\] :propvalueend

\[ (list :propvalue

(apply str (process-prop-value input-chars stack)))

(if (contains? uc-letters next-char)

(list :propident

(apply str (cons next-char

(process-prop-ident input-chars stack))))

(recur input-chars stack result)))

nil))

(process-prop-ident [input-chars stack]

(if-let [next-char (read-next-char input-chars stack)]

(if (some #{next-char} uc-letters)

(cons next-char (process-prop-ident input-chars stack))

(do (unread-char next-char stack) nil))))

(process-prop-value [input-chars stack]

(if-let [next-char (read-next-char input-chars stack)]

(condp = next-char

\\ (cons (read-next-char input-chars stack) (process-prop-value input-chars stack))

\] (do (unread-char next-char stack) nil)

(cons next-char (process-prop-value input-chars stack)))))]

(if-let [next-token (process-chars input-chars stack result)]

(recur input-chars stack (cons next-token result))

result))))

(defn nilcons [tree subtree]

(if (nil? tree)

subtree

(cons tree subtree)))

(defn append [col1 col2]

(into col2 (reverse col1)))

(defn parser- [lexems]

(letfn [(process-file [lexems tree]

(let [lexem (first lexems)

lexems (rest lexems)]

;(println "process-file: " lexem)

(case lexem

:treestart (let [[lexems subtree] (process-tree lexems nil)]

(recur lexems (nilcons tree subtree)))

nil tree

(throw (Exception. (str "File can't be started with " lexem))))))

(process-tree [lexems tree]

(let [lexem (first lexems)

lexems (rest lexems)]

;(println "process-tree: " lexem)

(case lexem

:treestart (let [[lexems subtree] (process-tree lexems nil)]

(recur lexems (nilcons tree subtree)))

:nodestart (let [[lexems subtree] (process-nodes lexems nil)]

(recur lexems (nilcons tree subtree)))

:treeend (list lexems tree)

nil (list lexems tree)

(throw (Exception. (str "Tree can't be started with " lexem))))))

(process-nodes [lexems tree]

;(println "process-nodes: " (first lexems))

(if (empty? lexems)

(list lexems tree)

(if (and

(instance? clojure.lang.PersistentList (first lexems))

(= (ffirst lexems) :propident))

(let [propident (second (first lexems))

[lexems subtree] (process-values (rest lexems) nil)]

(recur lexems (cons (cons propident subtree) tree)))

(list lexems tree))))

(process-values [lexems tree]

(if (and

(instance? clojure.lang.PersistentList (first lexems))

(= (ffirst lexems) :propvalue)

(= (second lexems) :propvalueend))

(recur (nnext lexems) (cons (second (first lexems)) tree))

(list lexems tree)))]

(process-file lexems nil)))

;; Parsing to tree form

(defn sgf2tree [file-name]

(let [lazy-read-fn (lazy-read file-name 512)

stack (atom nil)

lexems (reverse (lexer lazy-read-fn stack nil))

tree (parser- lexems)]

tree))

;; Simplifying functions

(defn simplify [tree]

(if (instance? java.lang.String (first tree))

;; process property ident/value pair

(let [color (first tree)

move (second tree)]

(if (or

(= color "W")

(= color "B"))

move))

(filter (complement (and nil? empty?)) (pmap simplify tree))))

(defn tree2board [tree]

(let [board (vec (repeatedly 19 (fn [] (vec (repeat 19 :empty)))))

player (mkplayer)]

(reduce (fn [board move]

(let [y (char2index (first move))

x (char2index (second move))]

(assoc board x (assoc (board x) y (stone (player))))))

board tree)))

monte-carlo.clj

(defn- heuristic [board player]

(- (rand-int 100) 50))

(defn- make-move [board move player]

(let [x (char2index (first move))

y (char2index (second move))]

(assoc board x (assoc (board x) y player))))

(defn- movegen [board player]

(take 10 (shuffle

(filter (complement nil?)

(flatten

(map (fn [[row x]]

(map (fn [[element y]]

(if (= element :empty)

(str x y)))

(map (partial list) row board-letters)))

(map (partial list) board board-letters)))))))

(defn- monte-carlo [board depth player firstrun]

(let [moves (movegen board player)]

(if (or (empty? moves) (zero? depth))

[(heuristic board player) nil]

(do

;(println "Processing moves...")

(apply max-key first

(cons

[-9999 nil]

((if firstrun pmap map) (fn [move]

;(println "---> " move)

(let [[value path]

(monte-carlo (make-move board move player) (dec depth)

(opposite player) false)]

;(println "<--- " value)

[(- value) (cons move path)]))

moves)))))))

(defn- make-moves [board moves player]

(reduce (partial apply make-move)

board

(map list

moves

(flatten (repeat (/ (count moves) 2) [player (opposite player)])))))}