Properly Tail Recursive Interpreter. Some timings of compilation vs interpretation

Lecture 16

Why do we want to eliminate recursion?

Uses up stack
Required by Scheme language specification
 (not required in CL but generally done if optimizing)
Makes compiling more obvious
Applicable to Java, too

```
(defun (mydo  f j lim)
  (cond ((> j lim) 'done)
        (t  (funcall f j)(mydo f (+ j 1) lim))))

;; or equivalent in Scheme
```

Tail recursion elimination

```
;; A PROPERLY TAIL-RECURSIVE INTERPRETER  (approx for mj)

(defun mj-statement (x &optional env) ;;
 (block nil  ;; block establishes a tagbody with possible labels.
   :TOP    ;; this is a label for a goto statement (yes, Lisp has "go" within block!)
 (return
    (cond ((symbolp x) (get-var x env))
          ((atom x) x);; integers and strings are atoms.
          (t
           (case
              (first x)

             (Block
              ;;remove Block to get at args
              (pop x)
              ;; interpret all but the last arg
              (loop while (rest x) do (mj-statement (pop x) env))
              ;; finally, rename the last expression as x    < The KEY
              (setf x (first x))
              (go :TOP))

             (Assign ( mj-set-var (idname (elt x 1))(elt x 2) env))

 ;;;; ..more
```

Tail recursion elimination

```
;; more of the above program
(defun mj-statement (x &optional env)
;;;……snipped out
            (IfExp
             (setf x
               (if (equal 'true (mj-exp-eval (elt x 1) env))
                       (elt x 2) (elt x 3)))
                       (go :TOP))

;; more snipped out, not changed..
```

Tail recursive elimination (more)

Much simpler in a more functional-programming setting; one way of looking at this is to have "continuations" that look like this ..evaluate-with-continuation

```
(defun eval-w-c (expr env more)
  …  ;; (Times x y) →
       evaluate x to X, then
         (eval-w-c y env #'(lambda(R)(* X R))
```

But don't call eval-w-c, just reset expr, more. Also what about previous "more")

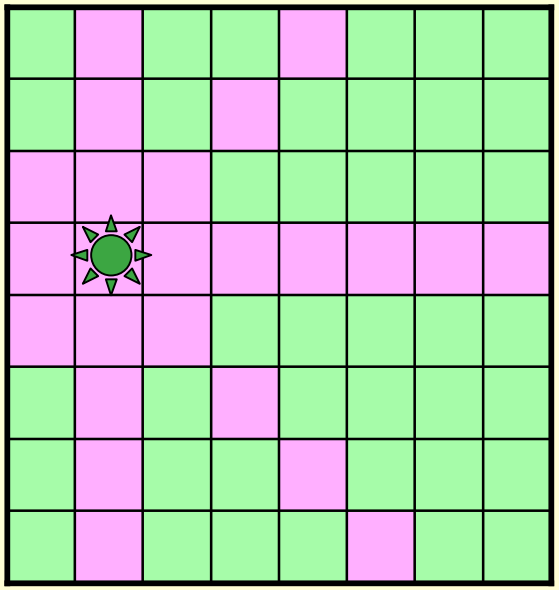
```
#'(lambda(R)(more (* X r)).
```

A change of pace

- Simple-interp.lisp is a collection of lisp programs that can be compiled or interpreted
- Figure that compiling a lisp program makes it about 10 or 20 times faster, but somewhat harder to debug.
- We can also implement MJ by rewriting the AST into lisp via simple-translate and Interpreting that as a lisp program.
- Or we can compile that rewritten program with the common lisp compiler.
- Semantics should be the same. What differences are there?

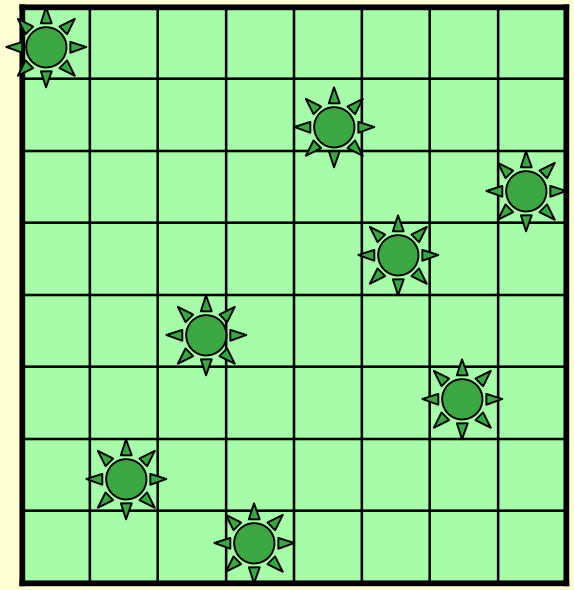
8 Queens program benchmark

- Classic computing problem based on a chess game: how many ways can 8 queens be placed on a board without attacking each other?



8 Queens program benchmark

- Classic computing problem based on a chess game: how many ways can 8 queens be placed on a board without attacking each other?



One of 92 solutions

8 Queens program benchmark (We did this 3 years ago in CS164. This is a Tiger program)

```
let var N := 8
   type intArray = array of int
    var row := intArray [ N ] of 0
    var col := intArray [ N ] of 0
    var diag1 := intArray [N+N-1] of 0
    var diag2 := intArray [N+N-1] of 0

      function printboard() =
      (for i := 0 to N-1
            do (for j := 0 to N-1
                 do print(if col[i]=j then " O" else " .");
               print("\n"));
        print("\n"))
    function try(c:int) =
( /*  for i:= 0 to c do print("."); print("\n"); flush();*/
     if c=N
     then printboard()
     else for r := 0 to N-1
            do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
                  then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
                        col[c]:=r;
                      try(c+1);
                        row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)

) in try(0) end
```

8 Queens program benchmark

Remove almost all printing of the chess board,

so that it runs at CPU speed.

Do you expect these results?

Speed/ compiled/ interpreted (8 Queens)

running the tiger interpreter interpreted in common lisp:
; cpu time (total) 67,837 msec (00:01:07.837) 68 sec
using 56,000,000 bytes of cons cells, 1.1 gigabytes of other memory

compiling the interpreter but still interpreting queensc.tig
cpu time (total) 721 msec user 0.7 sec
using 181,000 bytes of cons cells

translating the queensc.tig program into a lisp program
; cpu time (total) 2,795 msec user, 0 msec system 2.8 sec
using 2,000,000 bytes of cons cells

"compiling" the resulting lisp program
; cpu time (total) 10 msec user, 0 msec system 0.01 sec
using 193 cons cells.

Some typechecking hints..

Don't be afraid of adding some more information fields, e.g. who uses this variable?

The notion of "same" type is not "equal" type fields.

Keep looking for type errors after the first one,even if they are likely to be bogus.

Make sure that you are checking every piece of the code.

Assume the AST is "well-formed" i.e. you don't need to check that the parser was correct.

Reminder: Exam coming up!!

- Nov 2, in class, Would you like 2 pages of notes? (2-sides?) this time?
- Topics: everything so far, but emphasis on material since exam 1. Certainly LR, LALR, interpreter, typechecker, subtleties of MiniJava semantics.