The Analyze Evaluator

In the analyze evaluator (A-EVAL), the analyze procedure creates an execution object, which is a procedure that can be used to complete an evaluation by applying it to an environment. For example, the analyze-self-evaluating procedure is

\[
\text{(define (analyze-self-evaluating exp)} \\
\text{\hspace{1cm} (lambda (env) exp))}
\]

1. Draw a picture representing the execution object used in \((\text{analyze '1} \text{ env})\).

The AMB Evaluator

In the amb evaluator, the execution object is extended. Now it becomes a procedure that can be used to complete an evaluation, and can also be used to try for more values from an evaluation, as we shall see in a moment. This execution object takes an environment, a succeed procedure, and a fail procedure. For example, the analyze-self-evaluating procedure now becomes

\[
\text{(define (analyze-self-evaluating exp)} \\
\text{\hspace{1cm} (lambda (env succeed fail)} \\
\text{\hspace{2cm} (succeed exp fail)))}
\]

2. Draw a picture showing the execution object used in \((\text{analyze '1} \text{ env mainloop-succeed mainloop-fail})\).
The mainloop-succeed and mainloop-fail are part of the main driver loop for the amb evaluator. These describe what should happen when a value is found (tell the user, and then go back for another value if the user wants one), and when no value is found (tell the user “no more values”).

```
(define (mainloop-succeed value fail-continuation)
  (write-line "got a value" value)
  (let ((exp (read)))
    (if (eq? exp 'try-again)
        (fail-continuation)
        (amb-eval exp t-g-e mainloop-succeed mainloop-fail)))))

(define (mainloop-fail)
  (write-line "no more values"))
```

3. Assuming the user does in fact request a second value, use the execution object on the previous page and enumerate the steps that occur in the interpretation of ((analyze '1) env mainloop-succeed mainloop-fail).
The `amb` expression is at the heart of the backtracking abilities of the `amb` evaluator.

```lisp
(define (analyze-amb exp)
  (let ((alternatives (map analyze (operands exp))))
    (lambda (env succeed fail)
      (define (try-next choices)
        (if (null? choices)
            (fail)
            ((car choices) env
             succeed
             (lambda () (try-next (cdr choices))))))
      (try-next alternatives))))
```

4. Draw a picture representing the execution object resulting from

```lisp
((analyze '(amb 1 2)) env mainloop-succeed mainloop-fail)
```
5. Assuming the user does in fact request a second and third value, enumerate the steps that occur in the interpretation of
\[
((\text{analyze } \text{'}(\text{amb } 1 \ 2)) \ \text{env mainloop-succeed mainloop-fail}).
\]

Using the backtracking interpreter

6. Despite (because of?) the fact that we’re doing “declarative programming” with amb, efficiency is an important issue. Can you write a more efficient version of sum-to?

\[
\begin{align*}
\text{(define (sum-to n)} \\
\text{ (let ((x (a-number-between 1 n))} \\
\text{ (y (a-number-between 1 n))))} \\
\text{ (require (= n (+ x y)))} \\
\text{ (list x y)))}
\end{align*}
\]