Today's Lecture Notes for cs173

sk and dbtucker

October 2, 2000

Topic: If's & But's

If's

Exercise #1: Add an `if` expression to our language.

You need to extend the datatype to include booleans, add an *ifE* variant to the datatype, and then add something that returns a boolean. (Our solution has a *eqE* construct which tests the equivalence of two numbers.)

But's

Recall our function *real-prod* from yesterday. How would you write it in a language like Java? Easy—you would raise an exception:

```
(define (prod L)
  (cond
    [(empty? L) 1]
    [(cons? L) (cond
                 [(zero? (first L)) (raise 0)]
                 [else (* (first L) (prod (rest L)))])]))
```

```
(define (real-prod L)
  (try (prod L)
   catch ...))
```

What should you do when you catch an exception (i.e. the ... above)? We could just evaluate some expression:

```
(define (real-prod L)
  (try (prod L)
   catch (3 + 4)))
```

But what if we want the exception handler to use the value contained in the exception? Instead of introducing a new binding construct, we'll can specify a function of one argument as the exception handler. This function will be applied

to the value contained in the exception. In the case of *real-prod*, we want to return the value in the exception (0), so the exception handler is just the identiy function:

```
(define (real-prod L)
  (try (prod L)
   catch (lambda (v) v)))
```

Exercise #2: Add exceptions to our language.

We can support exceptions by creating an exception value, then always checking the return value of a computation. For example, the *addE* case is:

```
[addE (le re) (let ([lv (interp le d)])
                (cond
                  [(exn-value? lv) lv]
                  [else (let ([rv (interp re d)])
                          (cond
                            [(exn-value? rv) rv]
                            [else (numV+ lv rv)]))]))]
```

The *tryE* case then executes the handler if an exception is returned:

```
[tryE (e handler) (let ([val (interp e d)])
                    (cases AFunVal val
                      [exnV (v) (let ([hv (interp handler d)])
                                  (apply-fun hv v))]
                      [else val]))]
```