CSc 335 Class 18 Part 2 April 4 2024

- An interpreter for Aexp — using the Aexp data structure, as well as tree-recursion

This is a parameterized postcondition — or a higher order postcondition — because it has function parameters

pre: inputs e ∈ Aexp

post: outputs the correct value of e for the definitions of the operators @, #, !

we name the interpreter value —
the interpreter is the entity which defines the semantics (ie, the meaning) of expressions in Aexp

global variable — should it be passed as param?

Reference is again TLS

Again — the structure of the code mirrors that of its data

This is an example of (interpretation by) recursive

```
(define value
  (lambda (aexp)
    (cond ((natnum? aexp) aexp)
          (else (cond ((@-exp? aexp)
                       ((@-function)
                        (value (first-operand aexp))
                        (value (second-operand aexp)))))
                      ((#-exp? aexp)
                       (#-function
```

descent

```
            (value (first-operand aexp))
            (value (second-operand aexp))))

        ((!-exp? aexp)
            (!-function
            (value (first-operand aexp))
            (value (second-operand aexp))))
      )))
  ))
```

What are some possible definitions for @-function, #-function, and !-function?

While we could simply

```
(define @-function +)
(define #-function *)
(define !-function expt)
```

It is important — and demonstrative of the basic power of interpreter technology — to realize ~~that~~ these functions could be defined arbitrarily within the constraints imposed by the value function. That is: they each need to be binary functions returning numbers.

even this could be relaxed if we changed the basis step to return some type other than scheme-number.

In my notes you will find my-plus, my-times, my-expt and you should to produce other examples on your own. To that end, I ask you to spend some time with the 2ND exercise of hw 7 — I ask you to discover and carry out a minimal set of modifications on the value function to allow to deal with Aexp modified so that its basis case is simply lists of ones. Eg

(1) @ ( ) = (1)

(1) @ (1 1) = (1 1 1)

IE — simple "stroke notation". One goal is to

NOT use base-10 numbers anywhere in your design. Another is to preserve data abstraction - you should not "reach behind" the interface functions to work directly with the low-level representations.

To solidify your understanding of function parameters, you should also rewrite the value function so that it takes @-function, #-function and !-function as parameters. Notice that doing so brings the function into alignment with the spec we gave.

Can we in fact make even greater use of higher-order functions to further improve the value function?

Now consider the following function:

```
(define atom-to-function
  (lambda (x)
    (cond ((eq? x '@) @-function)
          ((eq? x '#) #-function)
          (else !-function))))
```

We can use this to improve value:

```
(define value
  (lambda (aexp)
    (cond ((natnum? aexp) aexp)
          (else
          ((atom-to-function (operator aexp))
            (value (first-operand aexp))
            (value (second-operand aexp)))
          ))))
```