Desugaring LetRec

CIS400 (Compiler Construction)

Kris Micinski, Fall 2021

Project 2

- In this project, you will “desugar” a big language into a smaller language
- Subsequent projects have you handle this (smaller) language
- Today we will go through the forms of the big language
- **Please start early, this project has a lot of forms, many of them are easy—but some take time to learn**

Input Language

```
e ::= (letrec* ([x e] ...) e)
    | (letrec ([x e] ...) e)
    | (let* ([x e] ...) e)
    | (let ([x e] ...) e)
    | (let x ([x e] ...) e)
    | (lambda (x ...) e)
    | (lambda x e)
    | (lambda (x ...+ . x) e)
    | (if e e e)
    | (and e ...)
    | (or e ...)
    | (when e e)
    | (unless e e)
    | (begin e ...+)
    | (cond cond-clause ...)
    | (case e case-clause ...)
    | (delay e)
    | (force e)
    | (call/cc e)
    | (set! x e)
    | (apply e e)
    | (e e ...)
    | x
    | op
    | (quote dat)
```

Output Language

```
e ::= (let ([x e] ...) e)
    | (lambda (x ...) e)
    | (lambda x e)
    | (apply e e)
    | (e e ...)
    | (prim op e ...)
    | (apply-prim op e)
    | (if e e e)
    | (set! x e)
    | (call/cc e)
    | x
    | (quote dat)
```

A few notes

- **Copy tester.py from p1 into your p2 folder.**

- Make sure you understand all of the forms of the input language —otherwise how can you translate!

- Your project should be very pattern-match heavy

    - All forms of the input lang should have a corresponding match pattern / statement!

- You don’t need to translate every form

    - E.g., don’t translate set! or call/cc, just desugar their arguments

(and e_0 e_1)

(if e_0 e_1 (quote #f))

Can you come up with a similar encoding for **or**?

`(or e0 e1)`

…?

(when e_0 e_1)

(if e_0 e_1 (void))

What about unless…?

`(unless e0 e1)`

…

What about begin

(begin e_0 … e_n)


```
(let ([x0 e0])
 (let ([x1 e1])
  …
   (let ([xn en]) (void))))
```

Cond gets translated into a long sequence

(cond [g_0 e_0] … [else e_n])


```
(if g0
    e0
    (if g1 … en)…)
```

The case form

```
(case val-expr case-clause ...)
```

case-clause	=		`[(`*datum* `...)` *then-body* `...+]`
	|		`[else` *then-body* `...+]`

```
> (case (+ '1 '8)
    [(a b c d) '77]
    [else (case '8 [() '97] [(7 8 9) '98] [else '99])])
98
```

Allows a lightweight (but not full) form of pattern matching

(Encoding this is part of the project..)

Force and Delay

Create lazily-evaluated **thunks**

If you **delay** a value, it may later be *forced*

When a value is forced, it's value will be computed unless it has been previously computed

```
> (define x (delay (displayln "hello world") 23))
> x
#<promise:x>
> (force x)
hello world
23
> (force x)
23
```

Implementing delay/force

For this project, you decide on the representation of promises

Basic idea: represent promises as a **vector** like…
`#(promise … …)

Where you store (a) whether the answer has been evaluated or not and (b) either the result or a lambda (to evaluate the first time)

You want to use a vector because you will want to forcibly update various values, which you can do using vector-set! (a primitive in the output lang).

Implementing call/cc

You don’t have to do it for this project! But you **do** have to desugar everything under it…

```
(call/cc (lambda (k) (when #t (k (displayln “hello”)))))
```



```
(call/cc (lambda (k) (if #t (k (displayln “hello”)) (void))))
```

set!

You also don’t have to desugar set! We will handle that in subsequent passes.

Therefore, you may use set! in your desugarings

```
(let loop ([x ivx] [y ivy] …)
  body)
```



```
(letrec ([loop (lambda (x y …)
                 body)])
  (loop ivx ivy …))
```

```
(define (sum from to)
  (define total 0)
  (let loop ()
    (set! total (+ total from))
    (set! from (+ from 1))
    (if (<= from to)
        (loop)
        total)))
```

```
(define (sum from to)
  (let loop ([i from]
             [total 0])
    (if (<= i to)
        (loop (+ i 1) (+ total i))
        total)))
```

```
(letrec* ([x0 e0] …) body)
```



```
(let ([x0 'undefined] …)
  (set! x0 e0)
  ...
  body)
```

```
(letrec ([x0 e0] …) body)
```



```
(let ([x0 'undefined] …)
  (let ([t0 e0])
    (set! x0 t0)
    ...
    body))
```