Adding Recursive Procedures to PROC (REC)

CS496

REC: a Language with Recursive Procedures

- REC = PROC + Recursion
- For this language, just like for LET and PROC, we study:
  - Concrete and Abstract Syntax
  - Specification of the interpreter
  - Implementation of the interpreter

The REC-Language

The Interpreter for REC

REC: Concrete Syntax

⟨*Program*⟩ ::= ⟨*Expression*⟩
⟨*Expression*⟩ ::= ⟨*Number*⟩
⟨*Expression*⟩ ::= ⟨*Identifier*⟩
⟨*Expression*⟩ ::= - (⟨*Expression*⟩ , ⟨*Expression*⟩)
⟨*Expression*⟩ ::= zero? (⟨*Expression*⟩)
⟨*Expression*⟩ ::= if ⟨*Expression*⟩
then ⟨*Expression*⟩ else ⟨*Expression*⟩
⟨*Expression*⟩ ::= let ⟨*Identifier*⟩ = ⟨*Expression*⟩ in ⟨*Expression*⟩
⟨*Expression*⟩ ::= proc (⟨*Identifier*⟩) ⟨*Expression*⟩
⟨*Expression*⟩ ::= (⟨*Expression*⟩ ⟨*Expression*⟩)
⟨*Expression*⟩ ::= letrec ⟨*Identifier*⟩ (⟨*Identifier*⟩) = ⟨*Expression*⟩
in ⟨*Expression*⟩

Example

```
letrec fact(x) =
               if zero?(x)
               then 1
               else *(x,(fact -(x,1)))
in (fact 6)
```

Note: We will assume out language supports multiplication

REC: Abstract Syntax

```
1 (define-datatype program program?
2   (a-program
3     (exp1 expression?)))
```

REC: Abstract Syntax

```
(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?)
    (exp2 expression?))
  (zero?-exp
    (exp1 expression?))
  (if-exp
    (exp1 expression?)
    (exp2 expression?)
    (exp3 expression?))
  (var-exp
    (var identifier?))
  (let-exp
    (var identifier?)
    (exp1 expression?)
    (body expression?))
```

The new variant: letrec-exp

```
1   (proc-exp
2     (var identifier?)
3     (body expression?))
4   (call-exp
5     (rator expression?)
6     (rand expression?))
7   (letrec-exp
8     (p-name identifier?)
9     (b-var identifier?)
10    (p-body expression?)
11    (letrec-body expression?)) )
```

Example - From Concrete to Abstract Syntax

```
letrec fact(x) =
               if zero?(x)
               then 1
               else *(x,(fact -(x,1)))
in (fact 6)
```

```
(a-program
 (letrec-exp
  'fact
  'x
  (if-exp (zero?-exp (var-exp 'x))
          (const-exp 1)
          (mult-exp (var-exp 'x) (call-exp (var-exp
    'fact) (diff-exp (var-exp 'x) (const-exp 1)))))
  (call-exp (var-exp 'fact) (const-exp 6))))
```

The REC-Language

The Interpreter for REC

Discussion on Value of a Recursive Function

- In PROC: what is the value of the highlighted expression?

```
let f =
    proc (x)
        if zero?(x)
        then 1
        else *(x,(f -(x,1)))
in (f 6)
```

- Why would this not work as expected?

Discussion on Value of a Recursive Function

- In PROC: what is the value of the highlighted expression?

```
let f =
    proc (x)
        if zero?(x)
        then 1
        else *(x,(f -(x,1)))
in (f 6)
```

- Why would this not work as expected?
- Summary:
  - The environment included in the closure does not have an association for `f` itself
  - So `f` cannot call itself from the body of the `proc`
  - An easy way out is to introduce a special environment that allows such an association

Expressed Values Remain the Same

- The value of a `letrec` expression is the value of the body in a special environment

```
letrec fact(x) =
               if zero?(x)
               then 1
               else *(x,(fact -(x,1)))
in (fact 6)
```

- So there is no need to change the set of expressed values
- But we will need to extend the interpreter (`value-of`)

What Does a Special Environment Look Like?

```
(define-datatype environment environment?
  (empty-env)
  (extend-env
    (var identifier?)
    (val expval?)
    (env environment?))
  (extend-env-rec
    (p-name identifier?)
    (b-var identifier?)
    (body expression?)
    (env environment?)))
```

What Does a Special Environment Look Like?

```
1  (define-datatype environment environment?
2    (empty-env)
3    (extend-env
4      (var identifier?)
5      (val expval?)
6      (env environment?))
7    (extend-env-rec
8      (p-name identifier?)
9      (b-var identifier?)
10     (body expression?)
11     (env environment?)))
```

Specification

```
(value-of (letrec-exp
                proc-name
                bound-var
                proc-body
                letrec-body)
          ρ)
   = (value-of
       letrec-body
       (extend-env-rec proc-name bound-var proc-body ρ))
```

- This special environment is

  `(extend-env-rec proc-name bound-var proc-body`ρ`)`
- What is its behavior?

Behavior of `extend-env-rec`

`(extend-env-rec proc-name bound-var proc-body`ρ`)`

- Let us call the environment `special-env`
- In order to describe its behavior we must establish how look-up works in its presence

```
1 (apply-env special-env var) = ???
```

- There are two cases depending on whether `var` is equal to `proc-name` or not
- If not, then we simply keep looking in ρ

```
1 (apply-env special-env var) = (apply-env ρ var)
```

- If `var` is equal to `proc-name`?

Behavior of `extend-env-rec`

`(extend-env-rec proc-name bound-var proc-body`ρ`)`

- In that case `(apply-env special-env var)` should produce a closure
  1. whose bound variable is bound-var,
  2. whose body is proc-body, and
  3. with an environment in which proc-name is bound to this procedure (there may be more recursive calls!).
- But we already have such an environment: `special-env`.

```
1 (apply-env special-env proc-name) =
2    (proc-val
3       (closure bound-var proc-body special-env))
```

Implementing `extend-env-rec`

- We can implement extend-env-rec in any way that satisfies these requirements
- We choose the abstract-syntax representation
- First we extend the environment datatype as follows:

```
(define-datatype environment environment?
  (empty-env)
  (extend-env
    (var identifier?)
    (val expval?)
    (env environment?))
  (extend-env-rec
    (p-name identifier?)
    (b-var identifier?)
    (body expression?)
    (env environment?)))
```

Implementing `extend-env-rec`

- We can implement extend-env-rec in any way that satisfies these requirements
- We choose the abstract-syntax representation
- First we extend the environment datatype as follows:

```
1  (define-datatype environment environment?
2    (empty-env)
3    (extend-env
4      (var identifier?)
5      (val expval?)
6      (env environment?))
7    (extend-env-rec
8      (p-name identifier?)
9      (b-var identifier?)
10     (body expression?)
11     (env environment?)))
```

Implementing `extend-env-rec`

We now need to show how to apply such a recursive environment.

```
(define apply-env
  (lambda (env search-var)
    (cases environment env
      (empty-env ()
        (report-no-binding-found search-var))
      (extend-env (saved-var saved-val saved-env)
        (if (eqv? saved-var search-var)
          saved-val
          (apply-env saved-env search-var)))
      (extend-env-rec (p-name
                       b-var
                       p-body
                       saved-env)
        (if (eqv? search-var p-name)
          (proc-val (closure b-var p-body env))
          (apply-env saved-env search-var))))))
```

Implementing `extend-env-rec`

We now need to show how to apply such a recursive environment.

```
1  (define apply-env
2    (lambda (env search-var)
3      (cases environment env
4        (empty-env ()
5          (report-no-binding-found search-var))
6        (extend-env (saved-var saved-val saved-env)
7          (if (eqv? saved-var search-var)
8            saved-val
9            (apply-env saved-env search-var)))
10       (extend-env-rec (p-name
11                        b-var
12                        p-body
13                        saved-env)
14         (if (eqv? search-var p-name)
15           (proc-val (closure b-var p-body env))
16           (apply-env saved-env search-var))))))
```

The Interpreter for REC

- Code available from `http://www.eopl3.com`
- Directory `chapter3/letrec-lang`
- Open `top.scm` in Racket
- There are a number of tests in `tests.scm`
- You can run them with `run-one`. Eg.

```
(run-one 'simple-letrec-3)
(num-val 20)
```