Adding Recursive Procedures to PROC (REC)

CS510

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*⟩ ::= (⟨*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

```
type prog = AProg of expr

type expr =
  | Var of string
  | Int of int
  | Sub of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*expr
  | App of expr*expr
  | Letrec of string*string*expr*expr
```

Example - From Concrete to Abstract Syntax

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

```
AProg
(Letrec ("fact", "x",
   ITE (IsZero (Var "x"), Int 1,
    Mul (Var "x", App (Var "fact", Sub (Var "x", Int
        ↪ 1)))),
   App (Var "fact", Int 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 (`eval_expr`)

What Does a Special Environment Look Like?

```
1  type exp_val =
2    | NumVal of int
3    | BoolVal of bool
4    | ProcVal of string*expr*env
5  and
6    env =
7    | EmptyEnv
8    | ExtendEnv of string*exp_val*env
9    | ExtendEnvRec of string*string*expr*env
```

Specification

```
1  eval_expr ρ LetRec(
2                  proc_name,
3                  bound_var,
4                  proc_body,
5                  letrec_body)
6     = eval_expr
7         ExtendEnvRec(proc_name,bound_var,proc_body,ρ)
8         letrec_body
```

- This special environment is

  `ExtendEnvRec(proc_name,bound_var,proc_body,ρ)`
- What is its behavior?

Behavior of `extend-env-rec`

`ExtendEnvRec(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`

`ExtendEnvRec(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     ProcVal(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:

```
1  type exp_val =
2    | NumVal of int
3    | BoolVal of bool
4    | ProcVal of string*expr*env
5  and
6    env =
7    | EmptyEnv
8    | ExtendEnv of string*exp_val*env
9    | ExtendEnvRec of string*string*expr*env
```

Implementing `apply_env`

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

```
1  let rec apply_env (env:env) (id:string):exp_val option
       ↪  =
2    match env with
3    | EmptyEnv -> None
4    | ExtendEnv (key,value,env) ->
5      if id=key
6      then Some value
7      else apply_env env id
8    | ExtendEnvRec(key,param,body,en) ->
9      if id=key
10     then Some (ProcVal(param,body,env))
11     else apply_env en id
```

Depicting the Environment

```
let a=2
in letrec fact(x) =
               if zero?(x)
               then 1
               else x * (fact (x-1))
in (fact 6)    (* breakpoint set *)
```

Draw the environment extant at the breakpoint

a	NumVal 2
f	Rec("fact","x", ITE (IsZero (Var "x"), Int 1, Mul (Var "x", App (Var "fact", Sub (Var "x", Int 1)))))

The Interpreter for REC

- Code available in Canvas `Modules/Interpreters`
- Directory `rec-lang`
- Compile with `ocamlbuild -use-menhir interp.ml`
- Make sure the `.ocamlinit` file is in the folder of your sources
- Run `utop`