

Adding Recursive Procedures to PROC (REC)

CS496

REC: a Language with Recursive Procedures

- ▶ $\text{REC} = \text{PROC} + \text{Recursion}$
- ▶ 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

$\langle \text{Program} \rangle ::= \langle \text{Expression} \rangle$
 $\langle \text{Expression} \rangle ::= \langle \text{Number} \rangle$
 $\langle \text{Expression} \rangle ::= \langle \text{Identifier} \rangle$
 $\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle - \langle \text{Expression} \rangle$
 $\langle \text{Expression} \rangle ::= \text{zero? } (\langle \text{Expression} \rangle)$
 $\langle \text{Expression} \rangle ::= \text{if } \langle \text{Expression} \rangle$
 $\text{then } \langle \text{Expression} \rangle \text{ else } \langle \text{Expression} \rangle$
 $\langle \text{Expression} \rangle ::= \text{let } \langle \text{Identifier} \rangle = \langle \text{Expression} \rangle \text{ in } \langle \text{Expression} \rangle$
 $\langle \text{Expression} \rangle ::= (\langle \text{Expression} \rangle)$
 $\langle \text{Expression} \rangle ::= \text{proc } (\langle \text{Identifier} \rangle) \{ \langle \text{Expression} \rangle \}$
 $\langle \text{Expression} \rangle ::= (\langle \text{Expression} \rangle \langle \text{Expression} \rangle)$
 $\langle \text{Expression} \rangle ::= \text{letrec } \langle \text{Identifier} \rangle (\langle \text{Identifier} \rangle) = \langle \text{Expression} \rangle$
 $\text{in } \langle \text{Expression} \rangle$

Example

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

Note: We will assume our language supports multiplication

REC: Abstract Syntax

```
1 type prog = AProg of expr
2
3 type expr =
4   | Var of string
5   | Int of int
6   | Sub of expr*expr
7   | Let of string*expr*expr
8   | IsZero of expr
9   | ITE of expr*expr*expr
10  | Proc of string*expr
11  | App of expr*expr
12  | Letrec of string*string*expr*expr
```

Example - From Concrete to Abstract Syntax

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

```
1 AProg  
2 (Letrec ("fact", "x",  
3     ITE (IsZero (Var "x"), Int 1,  
4     Mul (Var "x", App (Var "fact", Sub (Var "x", Int  
5     ↪ 1)))))  
6     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?

```
1  let f =  
2      proc (x) {  
3          if zero?(x)  
4              then 1  
5              else x*(f (x-1)) }  
6  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?

```
1  let f =  
2      proc (x) {  
3          if zero?(x)  
4              then 1  
5              else x*(f (x-1)) }  
6  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**

```
1 letrec fact(x) =  
2     if zero?(x)  
3     then 1  
4     else x * (fact (x-1))  
5 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  $\rho$  LetRec(  
2      proc_name ,  
3      bound_var ,  
4      proc_body ,  
5      letrec_body)  
6  = eval_expr  
7      ExtendEnvRec(proc_name , bound_var , proc_body ,  $\rho$ )  
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  $\rho$  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
```

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`