Adding Procedures to LET (PROC)

CS510

PROC: A Language with Procedures

Extending our language LET with procedures:

1. Extending the concrete and abstract syntax.
2. Extending the set of Expressed Values.
3. Specification and Implementation of the interpreter.

The PROC-Language

An Interpreter for PROC

Procedural Representation

PROC: Concrete Syntax

Extending the concrete syntax of LET

$$
\begin{array}{lcl}
\langle \textit{Program} \rangle & ::= & \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Number} \rangle \\
\langle \textit{Expression} \rangle & ::= & \langle \textit{Identifier} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{- } (\langle \textit{Expression} \rangle \text{ , } \langle \textit{Expression} \rangle) \\
\langle \textit{Expression} \rangle & ::= & \text{zero? } (\langle \textit{Expression} \rangle) \\
\langle \textit{Expression} \rangle & ::= & \text{if } \langle \textit{Expression} \rangle \\
 & & \text{then } \langle \textit{Expression} \rangle \text{ else } \langle \textit{Expression} \rangle \\
\langle \textit{Expression} \rangle & ::= & \text{let } \langle \textit{Identifier} \rangle = \langle \textit{Expression} \rangle \text{ in } \langle \textit{Expression} \rangle
\end{array}
$$

PROC: Concrete Syntax

Extending the concrete syntax of LET

⟨*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*⟩)

Examples of Expressions in PROC

```
let f = proc (x) -(x,11)
in (f (f 77))

(proc (f) (f (f 77))    ;; rator
 proc (x) -(x,11))      ;; rand

let x = 200
in let f = proc (z) -(z,x)
   in (f 1)

let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

PROC: Abstract Syntax (1/3)

```
(define-datatype program program?
  (a-program
    (exp1 expression?)))
```

PROC: Abstract Syntax (2/3)

```
(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 symbol?))
  (let-exp
    (var symbol?)
    (exp1 expression?)
    (body expression?))
```

The new variants: proc-exp and call-exp (3/3)

```
1   (proc-exp
2     (var symbol?)
3     (body expression?))
4   (call-exp
5     (rator expression?)
6     (rand expression?)))
```

Concrete Syntax vs Abstract Syntax

- Concrete

```
let f = proc (x) -(x,11) in (f (f 77))
```

- Abstract

```
(a-program
  (let-exp
     'f
      (proc-exp 'x (diff-exp (var-exp 'x)
    (const-exp 11)))
      (call-exp (var-exp 'f)
                (call-exp (var-exp 'f)
                          (const-exp 77)))))
```

Concrete Syntax vs Abstract Syntax

- Concrete

```
let f = proc (x) -(x,11) in (f (f 77))
```

- Abstract

```
(a-program
  (let-exp
     'f
      (proc-exp 'x (diff-exp (var-exp 'x)
    (const-exp 11)))
      (call-exp (var-exp 'f)
                (call-exp (var-exp 'f)
                          (const-exp 77)))))
```

Concrete Syntax vs Abstract Syntax

- Concrete

```
let f = proc (x) -(x,11) in (f (f 77))
```

- Abstract

```
(a-program
  (let-exp
     'f
      (proc-exp 'x (diff-exp (var-exp 'x)
    (const-exp 11)))
      (call-exp (var-exp 'f)
                (call-exp (var-exp 'f)
                          (const-exp 77)))))
```

The PROC-Language

An Interpreter for PROC

Procedural Representation

The Interpreter for PROC

- What is the result of evaluating the expression `2`?

The Interpreter for PROC

- What is the result of evaluating the expression `2`?

  `num-val 2`

- What is the result of evaluating the expression `zero?(2)`?

The Interpreter for PROC

- What is the result of evaluating the expression `2`?

```
num-val 2
```

- What is the result of evaluating the expression `zero?(2)`?

```
bool-val #f
```

- What is the result of evaluating the expression `proc(x) -(7,x)`?

The Interpreter for PROC

- What is the result of evaluating the expression `2`?

```
num-val 2
```

- What is the result of evaluating the expression `zero?(2)`?

```
bool-val #f
```

- What is the result of evaluating the expression `proc(x) -(7,x)`?
  - Its not a number
  - Its not a boolean
  - Its a closure
    - Record that stores the formal parameter and body
    - More details to follow

The Interpreter for PROC

- Before (for the LET-language)

```
value-of: { expression, environment } -> expval
```

- Now

```
value-of: { expression, environment } -> expval
```

- What's the difference?
  - The definition of expressed values
  - Our syntax now supports procedures
- Before

$$\text{ExpVal} \quad = \quad \text{Int} + \text{Bool}$$

- Now

$$\text{ExpVal} \quad = \quad \text{Int} + \text{Bool} + \quad \text{Clos}$$

The Interpreter for PROC

```
value-of: { expression, environment } -> expval
```

- Expressed values before (LET)

```
(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?))
```

- Now (PROC)

```
(define-datatype expval expval?
  (num-val
    (num number?))
  (bool-val
    (bool boolean?))
  (proc-val
    (clos clos?)))
```

Defining Closures

What should the value of the following expression be?

```
proc (x) -(x,11)
```

- A datatype that records the parameter and its body
- However, is that enough?
- What about the procedures `f` and `g` below?

```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

- The value of `f` and `g` depends on the value of `x`.

Defining Closures

What should the value of the following expression be?

```
proc (x) -(x,11)
```

- A datatype that records the parameter and its body
- However, is that enough?
- What about the procedures `f` and `g` below?

```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

- The value of `f` and `g` depends on the value of `x`.

Defining Closures

```
let x = 200
in let f = proc (z) -(z,x)
   in let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))
```

- The value of a procedure depends on the environment in which it is evaluated.
  - `f` and `g` only differ in the value of `x`.
- Summary of what a closure should contain:
  1. The formal parameter of the procedure
  2. The body of the procedure
  3. The environment extant at the point where the procedure was evaluated

Representing Closures

Datatype Representation

```
(define-datatype clos clos?
  (closure
    (var symbol?)
    (body expression?)
    (saved-env environment?)))
```

- An alternative representation using Scheme procedures is also possible

Specifying the Behavior of the Interpreter – `proc-exp`

```
1 (value-of (proc-exp var body) ρ) =
2       (proc-val (closure var body ρ))
```

- Recall from above: closure is a constructor with arguments:
  - a formal parameter var,
  - an expression body,
  - and an environment ρ.
- It constructs elements of type `clos?`
- Note: In the EOPL book this constructor is called procedure
- `proc-val` tags the procedure so that it can be considered of type `expval?`
  - Remember that value-of returns an expressed value.

Implementation

New clauses for `value-of`

```
(define value-of
  (lambda (exp env)
    (cases expression exp
      (proc-exp (var body)
          (proc-val (closure var body env)))

      (call-exp (rator rand)
          ... to be defined ...

 )))
```

Specifying the Behavior of the Interpreter – `call-exp`

- A procedure call is represented as

  `(call-exp rator rand)`

  in the abstract syntax
- We must therefore give meaning to

  `(value-of (call-exp rator rand)`ρ`)`

  1. Evaluate rator to a procedure (check it is `proc-val`)
  2. Evaluate rand to an argument
  3. Pass argument to procedure
- Lets specify its behavior using equations

Evaluating Procedure Calls

- We must give meaning to `(value-of (call-exp rator rand)`ρ`)`
  1. Evaluate rator to a procedure (check it is `proc-val`)
  2. Evaluate rand to an argument
  3. Pass argument to procedure

```
1 (value-of (call-exp rator rand) ρ)
2   = (let ((proc (expval->proc (value-of rator ρ)))
3           (arg (value-of rand ρ)))
4     (apply-procedure proc arg))
```

- expval->proc checks argument is actually a procedure (i.e. it was constructed with proc-val)
- What does apply-procedure do?

Evaluating Procedure Calls

Consider the example:

```
(apply-procedure
  (closure 'x
           (diff-exp
                (var-exp 'x)
                (num-val 11))
           ρ)
  (num-val 20))
```

- The value of this expression is the value of the body `-(x,11)` where `x`'s value is 20.

In general:

```
(apply-procedure (closure var body ρ) val)
    = (value-of body [var=val]ρ)
```

Evaluating Procedure Calls

```
(apply-procedure (closure var body ρ) val)
    = (value-of body [var=val]ρ)
```

- The code corresponding to this specification is:

```
(define apply-procedure
  (lambda (clos1 val)
    (cases clos clos1
      (closure (var body saved-env)
        (value-of body
                  (extend-env var val saved-env))))))
```

Implementation

New clauses for `value-of`

```
(define value-of
 (lambda (exp env)
   (cases expression exp
     (proc-exp (var body)
       (proc-val (closure var body env)))

     (call-exp (rator rand)
       (let ((proc (expval->proc (value-of rator
    env)))
             (arg (value-of rand env)))
       (apply-procedure proc arg)))

)))
```

The PROC-Language

An Interpreter for PROC

Procedural Representation

Representing Procedures using Procedures

- We have defined clos?, closure, and apply-procedure for the procedures represented as inductive types
- We'll see one other implementation: using the procedures of Scheme

```
1 (define clos?
2  (lambda (val)
3    (procedure? val)))
```

- `procedure?` is Scheme's built-in predicate

Representing Procedures

procedural representation

procedure : Var × Exp × Env → Proc

```
(define closure
  (lambda (var body env)
    (lambda (val)
      (value-of body (extend-env var val env)))))
```

apply-procedure : Proc × ExpVal → ExpVal

```
(define apply-procedure
  (lambda (clos1 val)
    (clos1 val)))
```

The Interpreter for PROC

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

```
> (run-one 'nested-procs)
(num-val -1)
```