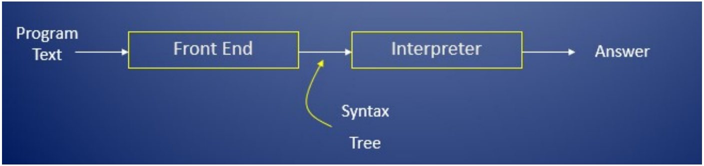
Part III: Expressions

Marco T. Morazán

Seton Hall University

Outline

Automatic Parsing

- We will now study the semantics (or meaning) of some fundamental programming languages features
- Our primary tools will be interpreters
- 

Automatic Parsing

- Adapted and Extended Lambda Calculus

<exp>	::=	<number>
	::=	true
	::=	false
	::=	<id>
	::=	-(<exp>, <exp>)
	::=	zero?(<exp>)
	::=	if <exp> then <exp> else <exp>
	::=	let {<id> = <exp>}* in <exp>
	::=	proc(<id>*) <exp>
	::=	(<exp> <exp>*)
	::=	letrec {identifier (<id>*) = <exp>}* in <exp>

- We will use a parser generator system: sllgen
- Read: Appendix B

Automatic Parsing

- Boilerplate Code

```
#lang eopl

(require rackunit "../eopl-extras.rkt")

;;;;;;;;;;;;;;;; grammatical specification ;;;;;;;;;;;;;;;;

(define the-lexical-spec
  '((whitespace (whitespace) skip)
    (comment ("%" (arbno (not #\newline))) skip)
    (identifier
      (letter (arbno (or letter digit "_" "-" "?"))) symbol)
    (number (digit (arbno digit)) number)
    (number ("-" digit (arbno digit)) number)))
```

Automatic Parsing

- The Grammar (not boilerplate)

  ```
  '((program (expression) a-program)
  ```

- ```
    (expression (number) const-exp)
    (expression ("true") true-exp)
    (expression ("false") false-exp)
    (expression (identifier) var-exp)
  ```

- ```
    (expression("-" "(" expression "," expression ")")diff-exp)
  ```
- ```
    (expression ("zero?" "(" expression ")") zero?-exp)
  ```
- ```
    (expression
      ("if" expression "then" expression "else" expression) if-exp)
  ```
- ```
    (expression
      ("let"(arbno identifier"="expression)"in" expression) let-exp)
  ```
- ```
    (expression
      ("proc" "(" (arbno identifier) ")" expression) proc-exp)
  ```
- ```
    (expression ("(" expression (arbno expression) ")") call-exp)))
  ```
- ```
  (expression
    ("letrec"(arbno identifier"("(arbno identifier)")""="expression)
     "in" expression) letrec-exp)
  ```

Automatic Parsing

- Boilerplate

```
(sllgen:make-define-datatypes the-lexical-spec the-grammar)

(define show-the-datatypes
  (lambda ()
    (sllgen:list-define-datatypes the-lexical-spec the-grammar)))

(define scan&parse
  (sllgen:make-string-parser the-lexical-spec the-grammar))

(define just-scan
  (sllgen:make-string-scanner the-lexical-spec the-grammar))
```

Automatic Parsing

- ```
  > (show-the-datatypes)
  ((define-datatype program program? (a-program (a-program21 expression?)))
   (define-datatype
    expression
    expression?
    (const-exp (const-exp22 number?))
    (true-exp)
    (false-exp)
    (var-exp (var-exp23 symbol?))
    (diff-exp (diff-exp24 expression?) (diff-exp25 expression?))
    (zero?-exp (zero?-exp26 expression?))
    (if-exp (if-exp27 expression?) (if-exp28 expression?) (if-exp29 expression?))
    (let-exp
     (let-exp30 (list-of symbol?))
     (let-exp31 (list-of expression?))
     (let-exp32 expression?))
    (letrec-exp
     (letrec-exp33 (list-of symbol?))
     (letrec-exp34 (list-of (list-of symbol?)))
     (letrec-exp35 (list-of expression?))
     (letrec-exp36 expression?))
    (proc-exp (proc-exp37 (list-of symbol?)) (proc-exp38 expression?))
    (call-exp (call-exp39 expression?) (call-exp40 (list-of expression?)))))
  ```
- ```
  > (scan&parse "if 0 then 1 else 2")
  #(struct:a-program
    #(struct:if-exp #(struct:const-exp 0) #(struct:const-exp 1) #(struct:const-exp 2)))
  ```

Evaluation Wrapper Functions and Tests

- ```
  ;; string → a-program
  ;; Purpose: Parse the given extended LC-program
  (define (parse p) (scan&parse p))
  ```
- ```
  ;; string → expval
  ;; Purpose: Evaluate the given extended LC-program
  (define (eval string)
    (value-of-program (parse string)))
  ```

Evaluation Wrapper Functions and Tests

- ```
  (check-equal? (eval "if zero?(1) then 1 else 2") (num-val 2))
  (check-equal? (eval "-(15, 10)") (num-val 5))
  (check-equal?
   (eval "let x = 10 in if zero?(-(x, x)) then x else 2")
   (num-val 10))
  (check-equal? (eval "let decr = proc (a) -(a, 1) in (decr 30)")
                (num-val 29))
  (check-equal? (eval "( proc (g) (g 30) proc (y) -(y, 1))")
                (num-val 29))
  (check-equal? (eval "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))")
                (num-val -100))
  (check-equal? (eval "let sum = proc (x) proc (y) -(x, -(0, y)) in ((sum 3) 4)")
                (num-val 7))
  (check-equal? (eval "let sum = proc (x) proc (y) -(x, -(0, y))
                        in letrec sigma (n) = if zero?(n)
                                              then 0
                                              else ((sum n) (sigma -(n, 1)))
                           in (sigma 5)")
                (num-val 15))
  (check-equal? (eval "letrec even(n) = if zero?(n)
                                        then zero?(n)
                                        else if zero?(-(n, 1))
                                             then zero?(n)
                                             else (even -(n, 2))
                        in (even 501)")
                (bool-val #f))
  ```
  ```

Expressed and Denoted Values

- An important part of the specification of any programming language is the set of values the language manipulates
- Each language has at least two sets:

  Expressed values Possible values of expressions
  Denoted values Values bound to variables

- In our extended λ-calculus
  - Expressed values: Int, Bool, and Proc
  - Denoted values: Int, Bool, Proc
- We need an interface for expressed values

Constructors	• num-val: int $\rightarrow$ expval • bool-val: Boolean $\rightarrow$ expval • proc-val: (listof symbol) expression env $\rightarrow$ expval
Observers	• expval->int : expval $\rightarrow$ int • expval->bool : expval $\rightarrow$ boolean • expval->proc : expval $\rightarrow$ proc

Expressed and Denoted Values

- Expressed values

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

- Extractors

```
;; expval → Int throws error
;; Purpose: Extract number from given expval
(define (expval2num v)
  (cases expval v
    (num-val (num) num)
    (else (expval-extractor-error 'num-val v))))
```

- ```
  ;; expval → Bool throws error
  ;; Purpose: Extract Boolean from given expval
  (define (expval2bool v)
    (cases expval v
      (bool-val (bool) bool)
      (else (expval-extractor-error 'bool-val v))))
  ```

- ```
  ;; expval → proc throws error
  ;; Purpose: Extract proc from given expval
  (define (expval2proc v)
    (cases expval v
      (proc-val (proc) proc)
      (else (expval-extractor-error 'proc-val v))))
  ```

- ```
  ;; symbol expval → throws error
  ;; Purpose: Throw expval extraction error
  (define (expval-extractor-error variant value)
    (eopl:error 'expval-extractors "Looking for a ~s, given ~s"
                variant value))
  ```
  ```

Environment

- Having vars → need for an environment
- Discussed earlier in this course
- Notation
  - $[x = 3][y = 7][u = 5]\rho$
  - $[x = 3, y = 7, u = 5]\rho$
  - (extend-env 'x 3 (extend-env 'y 7 (extend-env 'u 5 ρ)))
-
```
(define-datatype environment environment?
  (empty-env)
  (extend-env
   (bvar symbol?)
   (bval expval?)
   (saved-env environment?)))
```
-
```
(define (apply-env env search-sym)
  (cases environment env
    (empty-env ()
         (eopl:error 'apply-env "No binding for ~s" search-sym))
    (extend-env (var val saved-env)
         (if (eqv? search-sym var)
             val
             (apply-env saved-env search-sym)))))
```

Language Specification

- Specifying the behavior of programs

Constructor	a-program: expression $\rightarrow$ program
Observer	(value-of-program e) = (value-of e ρ_{init})

- Specifying the behavior of expressions

Constructors	const-exp, true-exp, false-exp, diff-exp, etc.
Observer	value-of: expression env $\rightarrow$ expval

Language Specification

- `value-of`:
- (value-of (const-exp n) ρ) = (num-val n)
- (value-of (true-exp n) ρ) = (bool-val #t)
- (value-of (false-exp n) ρ) = (bool-val #f)
- (value-of (var-exp n) ρ) = (apply-env ρ x)
- $$\frac{v1\text{=(expval2num (value-of e1 }\rho\text{))} \wedge v2\text{=(expval2num (value-of e2 }\rho\text{))}}{\text{(diff-exp e1 e2) = (num-val (- v1 v2))}}$$
- $$\frac{v\text{=(expval2num (value-of e)}\rho\text{)}}{\text{(value-of (zero?-exp n) }\rho\text{)=}\begin{cases}\text{(bool-val \#t)}, & \text{if } v = 0\\ \text{(bool-val \#f)}, & \text{if } v \neq 0\end{cases}}$$
- $$\frac{cval\text{=(expval2bool (value-of }c\ \rho\text{))}}{\text{(value-of (if-exp c t e}\rho\text{)=}\begin{cases}\text{(value-of } t\ \rho\text{)}, & \text{if cval = \#t}\\ \text{(value-of } e\ \rho\text{)}, & \text{if cval = \#f}\end{cases}}$$

Language Specification

- To evaluate a let-expression the environment must be extended with the bindings for the local variables before evaluation the body
- $$\frac{(v1 \ldots vn)\text{=(map (lambda (e) (value-of e } \rho\text{))}(e1 \ldots en))}{\text{(value-of (let-exp (s1...sn) (e1...en) body)}\rho\text{)} \;=\; \text{(value-of } \textit{body} \; [s1\text{=}v1 \ldots sn\text{=}vn]\rho\text{)}}$$

Language Specification

- For a language to be useful, it must allow the creation of new procedures
-
```
expval = int + boolean + proc
denval = int + boolean + proc
```
- proc is a set of values representing procedures
- What should this evaluate to?
```
let decr = proc (a) -(a, 1)
in (decr 30)
```
- What should this evaluate to?
```
(proc (g) (g 30) proc (y) -(y, 1))
```
- What should this evaluate to?
```
      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))
```

Language Specification

- What should this evaluate to?

```
      let x = 200
      in let f = proc (z) −(z, x)
in let x = 100         Same expression behaves differently
   in let g = proc (z) −(z, x)
      in −((f 1), (g 1))
```

- Variables must obey the lexical binding rule
- The value of a proc-exp depends on the environment
- (value-of (proc-exp (p1. . .pn) b) ρ) $=$ (proc-val (procedure (p1. . .pn) b ρ))

Language Specification

- What needs to happen to evaluate a call-exp?

```
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))
```

- 1. Evaluate f
  2. Evaluate 1
  3. Apply the proc to its argument(s)

- $$\frac{p = (expval\text{->}proc(\texttt{value-of e0 } \rho)) \wedge \texttt{args = (map (lambda (e) (value-of e env)) } (e1 \ldots en))}{\texttt{(value-of (call-exp e0 (e1...en))} \rho\texttt{) = (apply-procedure p args)}}$$

Language Specification

- Recursion

```
letrec fact(n) = if zero?(n)
                 then 1
                 else *(n, (fact {(n, 1)))
in (fact 5)
```

- (value-of letrec-exp(p-names params p-bodys letrec-body) ρ) = ???
- (value-of letrec-body ???)
- What env is needed to evaluate the body?
- ρ?
- `(value-of (fact 5)` ρ`)`
  `= error fact is not in the env`
- Body of letrec-exp cannot be evaluated using ρ

Language Specification

- Recursion
```
letrec fact(n) = if zero?(n) then 1 else *(n, (fact {(n, 1)))
in (fact 5)
```
- (value-of (letrec-exp p-names params p-bodys letrec-body) ρ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the procedure first?
```
(value-of
 (fact 5)
 [fact=(procedure '(n) (if zero?(n) then 1 else *(n, (fact {(n, 1)))) ?)] ρ)
```
- ```
  = (value-of
      (fact 5)
      [fact=(procedure
              '(n)
              (if zero?(n) then 1 else *(n, (fact {(n, 1)))
              ρ)]
      ρ)
  ```
- ```
  = (value-of
      *(n, (fact {(n, 1)))
      [n=5]
      [fact=(procedure
               '(n)
               (if zero?(n) then 1 else *(n, (fact {(n, 1)))
               ρ)]
      ρ)
  ```
- `= (value-of *(n, (fact {(n, 1))) [n=5]ρ)`
- ⋮
- `= (value-of (fact {(n, 1)) [n=5]ρ)`
- `= error fact is not in the env; We can't create the procedure first`

Language Specification

- Recursion
  ```
  letrec fact(n) = if zero?(n) then 1 else *(n, (fact {(n, 1)))
  in (fact 5)
  ```
- (value-of letrec-exp(p-names params p-bodys letrec-body) ρ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the needed env first?
  `[fact=?]ρ)]`
- Create the needed env first?
  `[fact=(procedure ... ... ?)]ρ)]`
- `[fact=(procedure ... ... [fact=...]ρ)]ρ)]`
- The required env needs the procedure for fact
- The procedure for fact needs the required env
- What is this known as?
- The Chicken and the Egg Paradox
- How do we solve it?

97 Solving the Paradox

Given that neither a client nor an account can be created first using the constructors for the respective structures, a new type of constructor is needed. A *generalized constructor* builds incorrect structure instances. Mutation is used to correct the values in the instances. As problem-solvers, we need to decide which structure type is returned by a general constructor.

A generalized constructor may be written for any structure in a circular dependency, and later fields are mutated to correct the structure instances. For example, to build a **client**, the client's name and the initial balance of the first account may be used to build a **client** with no accounts. Observe that this violates the data definition for a **client**. Later the client's list of accounts is mutated to add the first account. We now proceed to design and implement a generalized constructor for **clients**.

Language Specification

- We need an env that contains a binding for all the local functions defined
- $$\frac{\rho_r = [n1 = (\texttt{procedure p1 b1 } \rho_r)] \ldots [nn = (\texttt{procedure pn bn } \rho_r)]\,\rho}{(\text{value-of (letrec-exp (n1\ldots nn) (p1\ldots pn) (b1\ldots bn) body) } \rho) = (\text{value-of } \textit{body } \rho_r)}$$

Implementation

- Procedure representation

```
(define-datatype proc proc?
  (procedure
   (var (list-of symbol?))
   (body expression?)
   (envv voenv?)))
```

- ```
  ;; Any → Boolean
  ;; Purpose: Determine if given value is a vector with a single environment
  (define (voenv? penv)
    (and (vector? penv)
         (= (vector-length penv) 1)
         (environment? (vector-ref penv 0))))
  ```
  ```

Implementation

- ```
  ;; value-of-program : program → expval
  ;; Purpose: Evaluate the given program
  (define (value-of-program pgm)
    (cases program pgm
      (a-program (exp1)
                 (value-of exp1 (empty-env)))))
  ```
- ```
  ;; value-of : expression env → expval
  ;; Purpose: Evaluate the given expression in the given env
  (define (value-of exp env)
    (cases expression exp
      (const-exp (num) ...)
      (true-exp () ...)
      (false-exp () ...)
      (var-exp (var) ...)
  ```
- ```
      (diff-exp (exp1 exp2) ... (value-of exp1 env) (value-of exp2 env))
      (zero?-exp (exp1) ... (value-of exp1 env))
      (if-exp (exp1 exp2 exp3) ...
                               (value-of exp1 env)
                               (value-of exp2 env)
                               (value-of exp3 env))
  ```
- ```
      (let-exp (vars exps body) ...
                                (map (lambda (e) (value-of e env)) exps)
                                (value-of body ...))
  ```
- ```
      (proc-exp (params body) ...)
      (call-exp (rator rands) ...
                              (value-of rator env)
                              (map (lambda (rand) (value-of rand env)) rands))
  ```
- ```
      (letrec-exp (p-names params p-bodys letrec-body) (value-of letrec-body ...))))
  ```

Implementation

- Specializing `value-of`
- `(const-exp (num) (num-val num))`
- (value-of (true-exp) ρ) = (bool-val #t)
- (value-of (false-exp) ρ) = (bool-val #f)
- `(var-exp (var) (apply-env env var))`

Implementation

- $$\frac{v1\text{=(expval2num (value-of e1 } \rho\text{))} \wedge v2\text{=(expval2num (value-of e2 } \rho\text{))}}{\text{(diff-exp e1 e2) = (num-val (- v1 v2))}}$$
- ```
  (diff-exp (exp1 exp2)
    (let ((num1 (expval2num (value-of exp1 env)))
          (num2 (expval2num (value-of exp2 env))))
      (num-val (- num1 num2))))
  ```
- $$\frac{v\text{=(expval2num (value-of e)}\rho\text{)}}{\text{(value-of (zero?-exp n) }\rho\text{)=}\begin{cases}\text{(bool-val \#t)}, & \text{if } v = 0\\ \text{(bool-val \#f)}, & \text{if } v \neq 0\end{cases}}$$
- ```
  (zero?-exp (exp1)
    (let ((val1 (expval2num (value-of exp1 env))))
      (if (zero? val1)
          (bool-val #t)
          (bool-val #f))))
  ```
- $$\frac{cval\text{=(expval2bool (value-of } c\ \ \rho\text{))}}{\text{(value-of (if-exp c t e}\rho\text{)=}\begin{cases}\text{(value-of } t\ \rho\text{)}, & \text{if cval} = \#\text{t}\\ \text{(value-of } e\ \rho\text{)}, & \text{if cval} = \#\text{f}\end{cases}}$$
- ```
  (if-exp (exp1 exp2 exp3)
    (let ((val1 (value-of exp1 env)))
      (if (expval2bool val1)
          (value-of exp2 env)
          (value-of exp3 env))))
  ```
  ```

Implementation

- $$\frac{(v1 \ldots vn)\text{=(map (lambda (e) (value-of e } \rho\text{))}(e1 \ldots en))}{\text{(value-of (let-exp (s1...sn) (e1...en) body)}\rho\text{) = (value-of } \textit{body}\ [s1{=}v1 \ldots sn{=}vn]\rho\text{)}}$$
- ```
  (let-exp (vars exps body)
    (let [(vals (map (lambda (e) (value-of e env)) exps))]
      (value-of
       body
       (foldr (lambda (var val acc) (extend-env var val acc))
              env
              vars
              vals))))
  ```
- (value-of (proc-exp (p1...pn) b) ρ) = (proc-val (procedure (p1...pn) b ρ))
- ```
  (proc-exp (params body)
    (proc-val (procedure params body (vector env))))
  ```
- $$\frac{p\text{=}(\textit{expval->proc}\text{(value-of e0 } \rho\text{))} \wedge \text{args = (map (lambda (e) (value-of e env)) }(e1 \ldots en))}{\text{(value-of (call-exp e0 (e1...en))}\rho\text{) = (apply-procedure p args)}}$$
- ```
  (call-exp (rator rands)
    (let [(proc (expval2proc (value-of rator env)))
          (args (map (lambda (rand) (value-of rand env)) rands))]
      (apply-procedure proc args)))
  ```
  ```

Implementation

- ```
  ;; apply-procedure : proc (listof expval) → expval
  ;; Purpose: Apply the given procedure to the given values
  (define (apply-procedure f vals)
    (cases proc f
      (procedure (params body envv)
        (let [(saved-env (vector-ref envv 0))]
          (value-of body
                    (foldr (lambda (binding acc)
                      (extend-env (car binding)
                                  (cadr binding)
                                  acc))
                      saved-env
                      (map (lambda (p v) (list p v))
                           params
                           vals))))))))
  ```
  ```

Implementation

- $$\frac{\rho_r = [n1 = (\texttt{procedure p1 b1 } \rho_r)] \ldots [nn = (\texttt{procedure pn bn } \rho_r)]\rho}{(\text{value-of (letrec-exp (n1}\ldots\text{nn) (p1}\ldots\text{pn) (b1}\ldots\text{bn) body) } \rho) = (\text{value-of } \mathit{body} \ \rho_r)}$$
-

```
;; (listof symbol) (listof (listof symbol)) (listof expression) env
;; Purpose: Add proc-vals for given procedures in given environment
(define (mk-letrec-env ns ps bs env) Generalized constructor
  (let* [(temp-proc-vals
           (map (lambda (p b)          Temporary wrong proc-vals
                  (proc-val (procedure p b (vector (empty-env)))))
                ps
                bs))
```

- ```
         (new-env (foldl (lambda (name proc env)
                           (extend-env name proc env))
                         env
                         names
                         temp-proc-vals))]
  ```
- ```
    (begin
      (for-each (lambda (p)   Correcting proc-vals
                  (cases proc p
                    (procedure (p b ve) (vector-set! ve 0 new-env))))
                (map (lambda (p) (expval2proc p)) temp-proc-vals))
  ```
- ```
      new-env)))
  ```
  ```

Implementation

HOMEWORK

- Problems: 3.6–3.10, 3.12, 3.16–3.17, 3.21, 3.23–3.24, 3.26, 3.32–3.33, 3.55 (using the interpreter developed in class)
- Some problems we have already solved!

Variable Names Elimination

- Several ways to declare vars: let, letrec, and proc (so far!)
- In most PLs, declarations have limited scope

```
(let [(pi 3.14)
      (e 2.71)]
  (+ (let [(pi 3.4)]      (+ pi e) )     pi) )
```

- Every programming language must have *scoping rules*
- Rules for determining the declaration for a variable reference
- In many PLs, search outward from the reference to the declaration
- This is called lexical scoping and it is a static property

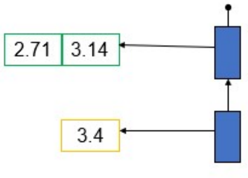
Variable Names Elimination

- ```
  (let [(pi 3.14)
        (e 2.71)]
  (+ (let [(pi 3.4)]     (+ pi e) )     pi) )
  ```
- Holes in the scope of a var may be created by nested declarations
- The number of boxes crossed is the lexical depth of a var
- The position in the declarations in the lexical position
- Lexical address is both the lexical depth and lexical position
- pi: 0 0
- e: 1 1
- pi: 0 0
- Why is this important?
- Using lexical addresses eliminates the need to search for a binding in an environment.
  ```

Variable Names Elimination

```
(let [(pi 3.14)
      (e 2.71)]
(+ (let [(pi 3.4)]        (+ pi e) )      pi) )
```

- Implement an env as a list of ribs, where a rib is a list of expvals




```
(let [(3.14 2.71)]
(+ (let [(pi 3.4)]        (+ (0 0) (1 1)) )      (0 0)) )
```

- The same is done for: proc, letrec
- A variable becomes a lexical address
- Add nameless versions for var, proc, let, and letrec to extended LC grammar
- Change env representation

Variable Names Elimination

- ```
  ;;;;;     ENVIRONMENT

  #|

  A rib is a (listof expval)

  An environment is a (listof rib)

  |#
  ```
- ```
  (define (environment? e)
    (list-of (list-of expval?)))
  ```
- ```
  ;;  → environment
  ;; Purpose: Build the empty env
  (define (empty-env) '())
  ```
- ```
  ;; (listof expval) environment → environment
  ;; Purpose: Build an environment from given expvals and env
  (define (extend-env vals env) (cons vals env))
  ```
- ```
  ;; environment natnum natnum → expval throws error
  ;; Purpose: Return expval at given lexical address in given env
  (define (apply-env env depth pos)
    (if (empty? env)
        (eopl:error 'apply-env "No binding for lexical address: ~s ~s
        (list-ref (list-ref env depth) pos)))
  ```
  ```

Variable Names Elimination

- New grammar rules
- ```
  (expression ("%lexvar" number number) nameless-var-exp)

  (expression ("%nameless-proc" "(" expression ")") nameless-proc-exp

  (expression ("%let" (arbno expression) "in" expression) nameless-le

  (expression
    ("%letrec" (arbno expression) "in" expression) nameless-letrec-ex
  ```
  ```

Variable Names Elimination

- procs no longer need the store the parameter names

```
(define-datatype proc proc?
  (procedure
   (body expression?)
   (envv voenv?)))
```

- To evaluate we must translate a program to an equivalent nameless version

```
;; string → expval
;; Purpose: Evaluate the given extended LC-program
(define (eval string)
  (value-of-program (translate-program-nameless (parse string))))
```

Variable Names Elimination

- Refinements to value-of

```
(nameless-var-exp (d p) (apply-env env d p))

(nameless-let-exp (exps body)
  (let [(vals (map (lambda (e) (value-of e env)) exps))]
    (value-of body (extend-env vals env))))

(nameless-proc-exp (body)
  (proc-val (procedure body (vector env))))

(nameless-letrec-exp (bodies letrec-body)
  (value-of letrec-body (mk-letrec-env bodies env)))
```

Variable Names Elimination

- We must now design `translate-program-nameless`
- The design idea:
  - Convert `var-exps` to `nameless-var-exps`
  - Convert `proc-exps` to `nameless-proc-exps`
  - Convert `let-exps` to `nameless-let-exps`
  - Convert `letrec-exps` to `nameless-letrec-exps`
  - Perform the above transformations for all expressions in a program's parse tree

Variable Names Elimination

- ```
  ;; expression senv → expression
  ;; Purpose: Change var references to lexical-addresses
  (define (translate-exp-nameless exp senv)
    (cases expression exp
  ```
- ```
      (var-exp (var)
               (let [(lex-addr (apply-senv senv var))]
                 (nameless-var-exp (first lex-addr) (second lex-addr))))
  ```
- ```
      (diff-exp (exp1 exp2)
                (diff-exp (translate-exp-nameless exp1 senv)
                          (translate-exp-nameless exp2 senv)))
  ```
- ```
      (zero?-exp (exp1)
                 (zero?-exp (translate-exp-nameless exp1 senv)))
  ```
- ```
      (if-exp (exp1 exp2 exp3)
              (if-exp
               (translate-exp-nameless exp1 senv)
               (translate-exp-nameless exp2 senv)
               (translate-exp-nameless exp3 senv)))
  ```
- ```
      (let-exp (vars exps body)
               (nameless-let-exp
                (map (lambda (e) (translate-exp-nameless e senv))
                     exps)
                (translate-exp-nameless body (extend-senv vars senv))))
  ```
- ```
      (proc-exp (params body)
                (nameless-proc-exp
                 (translate-exp-nameless body (extend-senv params senv))))
  ```
  ```

Variable Names Elimination

- ```
  (call-exp (rator rands)
            (call-exp
             (translate-exp-nameless rator senv)
             (map (lambda (e) (translate-exp-nameless e senv))
                  rands)))
  ```
- ```
  (letrec-exp (names params bodies body)
              (nameless-letrec-exp
               (map (lambda (e ps)
                      (translate-exp-nameless
                       e
                       (extend-senv ps (extend-senv names senv))))
                    bodies
                    params)
               (translate-exp-nameless body (extend-senv names senv))))
  (else exp)))
  ```

- Problems: 3.38, 3.41 (using the interpreter developed in class)