Chapter 3. Expressions
(Essentials of Programming Languages)

Kwanghoon Choi

Software Languages and Systems Laboratory
Chonnam National University

Outline

To study the binding and scoping of variables, we introduce a series of small languages to illustrate the concepts,
- writing specifications for the languages, and
- implemeting them using interpreters

- ♢ Specification and Implementation Strategy
- ♢ LET: A Simple Language
- ♢ PROC: A Language with Procedures
- ♢ LETREC: A Language with Recursive Procedures
- ♢ Scoping and Binding of Variables
- ♢ Eliminating Variable names & Implementing Lexical Addressing

3.1 Specification and Implementation Strategy

Our specification will consist of assertions of the form

- (value-of exp ρ) $= val$

meaning the value of expression exp in environment ρ is val.

The overall picture of our implementation

- A figure in the next slide
- Front-end: scanning (lexical analylsis), parsing (syntax analysis)
- Parser generator

: (Input) a lexical specification and a grammar

: (Output) a scanner and a parser

3.1 Spec. and Impl. Strategy(cont.)

The overall picture of our implementation

Figure 3.1(a)

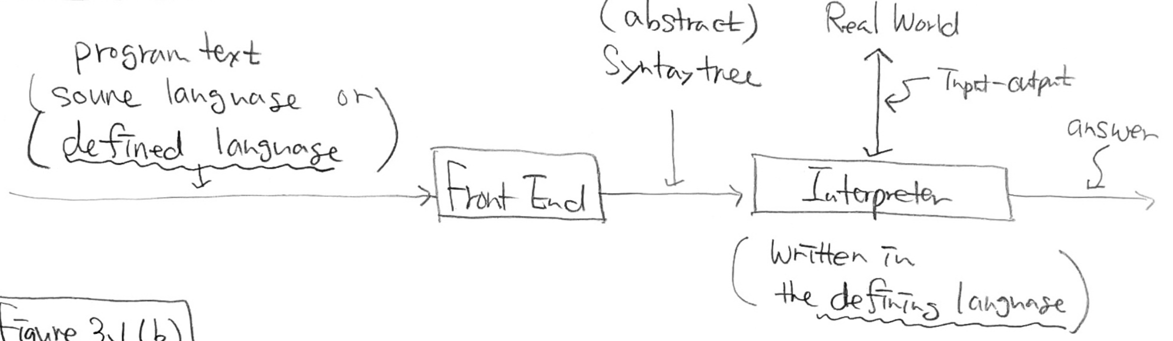
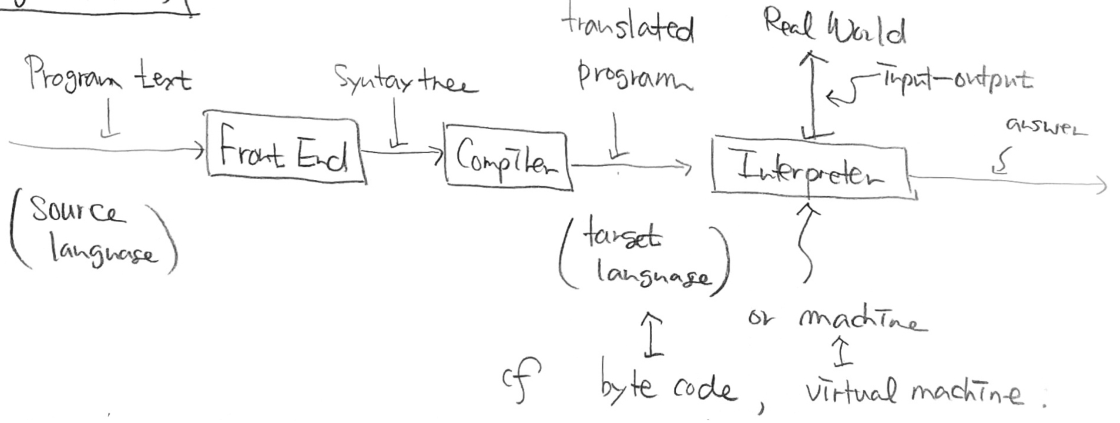


Figure 3.1(b)



3.2 LET: A Simple Language

Four programs (expressions) written in LET:

```
- (-(x,3), -(v,i))

if zero? (-(x,11)) then -(y,2) else -(y,4)

let x = 5 in -(x, 3)

let z = 5 in let x = 3 in let y = -(x, 1) in let x = 4 in -(z, -(x,y))
```

3.2.1 Specifying the Syntax

Syntax for the LET language

Program ::= Expression $\boxed{\text{a-program (exp1)}}$

Expression :: = Number $\boxed{\text{const-exp (num)}}$
Expression ::= -(Expression , Expression) $\boxed{\text{diff-exp (exp1 exp2)}}$
Expression ::= zero? (Expression) $\boxed{\text{zero?-exp (exp1)}}$
Expression ::= if Expression then Expression else Expression
$\boxed{\text{if-exp (exp1 exp2 exp3)}}$
Expression ::= Identifier $\boxed{\text{var-exp (var)}}$
Expression ::= let Identifier = Expression in Expression
$\boxed{\text{let-exp (var exp1 body)}}$

cf. Concrete syntax $\boxed{\text{Abstract syntax}}$

3.2.2 Specification of Values

Each programming language has at least two set of values
- Expressed values: the possible values of expressions
- Denoted values: the values bound to variables

Values and interfaces in the LET language,

$$\begin{aligned} ExpVal &::= Int + Bool \\ DenVal &::= Int + Bool \end{aligned}$$

- num-val : $Int \rightarrow ExpVal$
- bool-val : $Bool \rightarrow ExpVal$
- expval->num : $ExpVal \rightarrow Int$
- expval->bool : $ExpVal \rightarrow Bool$

3.2.3 Environments

Environments keep track of the meaning of each variable in the expression benig evaluated.

Some abbreviations in writing environments

- $[x = 3]\ [y = 7]\ [u = 5]\rho$
  denotes (extend-env ‘x 3 (extend-env ‘y 7 (extend-env ‘u 5 ρ))).

Formally,

- ρ ranges over environments
- $[]$ denotes the empty environments
- $[var = val]\rho$ denotes (extend-env $var\ val\ \rho$)
- $[var_1 = val_1, var_2, = val_2]\rho$ abbreviates $[var_1 = val_1]([var_2 = val_2]\rho)$
- $[var_1 = val_1, \cdots, var_n, = val_n]$ denotes the environment in which the value of var_1 is val_1, etc.

3.2.4 Specifying the Behavior of Expressions

Constructor interfaces

- const-exp : $Int \rightarrow Exp$
- zero?-exp : $Exp \rightarrow Exp$
- if-exp : $Exp \times Exp \times Exp \rightarrow Exp$
- diff-exp : $Exp \times Exp \rightarrow Exp$
- var-exp : $Var \rightarrow Exp$
- let-exp : $Var \times Exp \times Exp \rightarrow Exp$

(See Figure 3.2)

Observer interfaces

- value-of : $Exp \times Env \rightarrow ExpVal$

(Recall : value-of $exp\ \rho = val$)

3.2.4 Specifying the Behavior of Exprs (Cont.)

Before starting on an implementation, we write down a specification for the behaviors of the procedures in the previous slide

```
(value-of (const-exp $n$) $\rho$) $=$ (num-val $n$)

(value-of (var-exp $var$) $\rho$) $=$ (apply-env $\rho$ $var$)

(value-of (diff-exp $exp_1$ $exp_2$) $\rho$) $=$
  (num-val
    (-
      (expval->num (value-of $exp_1$ $\rho$))
      (expval->num (value-of $exp_2$ $\rho$))))
```

(See Figure 3.3 for “an execution of” this specification)

3.2.5 Specifying the Behavior of Programs

In the LET language, a whole program is just an expression.

$$\text{(value-of-program } exp\text{)} = \text{(value-of } exp\ \rho_{initial}\text{)}$$

3.2.6 Specifying Conditionals

The LET language has one constructor of boolean, zero?, and one observer of booleans, the if expression.

$$\frac{(\text{value-of } exp_1\ \rho) = val_1}{\begin{array}{l}(\text{value-of } (\text{zero?-exp } exp_1\ \rho) \\ \quad = \begin{cases} (\text{bool-val \#t}) & \text{if } (\text{expval->num } val_1) = 0 \\ (\text{bool-val \#f}) & \text{if } (\text{expval->num } val_1) \neq 0 \end{cases}\end{array}}$$

$$\frac{(\text{value-of } exp_1\ \rho) = val_1}{\begin{array}{l}(\text{value-of } (\text{if-exp } exp_1\ exp_2\ exp_3\ \rho) \\ \quad = \begin{cases} (\text{value-of } exp_2\ \rho) & \text{if } (\text{expval->num } val_1) = \text{\#t} \\ (\text{value-of } exp_3\ \rho) & \text{if } (\text{expval->num } val_1) \neq \text{\#f} \end{cases}\end{array}}$$

(See Figure 3.4 for a simple calculation of a conditional expression)

3.2.7 Specifying let

In the LET language, a let expression creates a new variable binding.

$$\frac{(\text{value-of } exp_1\ \rho) = val_1}{\begin{array}{l}(\text{value-of } (\text{let-exp } var\ exp_1\ body\ \rho) \\ \quad = (\text{value-of } body\ [var = val_1]\ \rho\end{array}}$$

3.2.8 Implementing the Specification of LET

In Scheme, a data type for abstract syntax for LET

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

(define-datatype expression expression?
  (const-exp
    (num number?))
  (diff-exp
    (exp1 expression?))
    (exp2 expression?))
  . . .
(See Figure 3.6)
```

3.2.8 Implementing the Spec. of LET (Cont.)

Our implementation uses SLLGEN as a front-end to parse,

- e.g., an input text "(- x 123)" into
  (a-program (diff-exp (var-exp ‘x) (const-exp ‘123))) in Scheme

In Scheme, a data type for expressed values

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

(Exercise: Implement expval->num and expval->bool.)

We can write down the interpreter, shown in Figure 3.8 and 3.9, in Scheme

- run : $String \rightarrow ExpVal$
- value-of-program : $Program \rightarrow Expval$
- value-of : $Exp \times Env \rightarrow Expval$

3.3 PROC: A Language with Procedures

In PROC, one can create new procedures as:

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

(proc (f) (f (f 77)))
proc (x) - (x, 11)

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))
```

(Note that the two identical procedures behave differently.)

3.3 PROC: A Language with Procedures (Cont.)

Procedures are new expressed values

$$\begin{aligned} ExpVal &::= Int + Bool + Proc \\ DenVal &::= Int + Bool + Proc \end{aligned}$$

$Proc$ is a set of values representing procedures with a constructor and an observer

- procedure : $Var \times Exp \times Env \rightarrow Proc$
- apply-procedure : $Proc \times ExpVal \rightarrow ExpVal$

3.3 PROC: A Language with Procedures (Cont.)

Syntax for procedure creation and calling

Expression :: = proc (Identifier) Expression $\boxed{\text{proc-exp (var body)}}$
Expression :: = (Expression Expression) $\boxed{\text{call-exp (rator rand)}}$

var: bound variable or formal parameter
rator: operator
rand: operand or actual parameter
(argument: the value of an actual parameter)

3.3 PROC: A Language with Procedures (Cont.)

(value-of (proc-exp $var\ body$) ρ)
    = (proc-val (procedure $var\ body\ \rho$))

[Exercise] Explain what proc-val is.

(value-of (call-exp $rator\ rand$) ρ)
    = (let ((proc (expval->proc (value-of $rator\ \rho$)))
            (arg (value-of $rand\ \rho$)))
          (apply-procedure proc arg))

(apply-procedure (procedure $var\ body\ \rho$) val)
    = (value-of $body\ [var = val]\rho$)

3.3.1 An Example

[Exercise] Execute the specification defined in the previous slide with procedural examples.

3.3.2 Representing Procedures

In Scheme, a new data type for expressed values including proc-val

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

Two alternative implementations of proc are in the textbook.

- A data structure representation is explained in the next slide.
  (See Section 3.3.2 for a procedural representation)

3.3.2 Representing Procedures

We define procedure as a data structure representation

```
(define-datatype proc proc?
    (procedure
      (var identifier?)
      (body expression?)
      (saved-env environment?)))

(define apply-procedure
    (lambda (proc1 val)
      (cases proc proc1
        (procedure (var body saved-env)
          (value-of body (extend-env var val saved-env))))))
```

Procedures in the data structure representation are called *closures*.

- a closed procedure + (its creation) environment

3.3.2 Representing Procedures (Cont.)

An implementation of the interpreter
- should extend one in Figure 3.8 and 3.9
- by adding value-of two new clauses as :

```
(proc-exp (var body)
  (proc-val (procedure var body env)))

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

3.4 LETREC: A Lang. with Recursive Procs.

In LETREC, one can create a recursive procedure as:

```
letrec double (x)
        = if zero? (x) then 0 else - ((double -(x, 1)), -2)
in (double 6)
```

Issue: When a closure for the recursive procedure double is created, the creation environment must contain a binding for double, which is the closure itself!

☐ proc (x) −(x, 1)

proc-val (procedure "x" (−("x", 1)) savedenv)

☐ proc (f) (x) ... f (−(x, 1)) ...

f = proc-val (procedure "x" (... "f" (−("x", 1)) ...) ("f" ↦ f ∪ savedenv))

3.4 LETREC: A Lang. with Rec. Procs. (Cont.)

Syntax for letrec

Expression :: $=$
letrec Identifier (Identifier) $=$ Expression in Expression
$\boxed{\text{letrec-exp (p-name b-var p-body letrec-body)}}$

Specification

(value-of
(letrec-exp $proc\text{–}name\ bound\text{–}var\ proc\text{–}body\ letrec\text{–}body$) ρ)
$=$ (value-of $lectrec\text{–}body$
(extend-env-rec $proc\text{–}name\ bound\text{–}var\ proc\text{–}body\ \rho$))

What is the specification for extend-env-rec then?

3.4 LETREC: A Lang. with Rec. Procs. (Cont.)

Assume ρ_1 is (extend-env-rec $proc\text{--}name$ $bound\text{--}var$ $proc\text{--}body$ ρ) for some ρ.

The behavior of ρ_1 can be specified as this:

(apply-env ρ_1 var)   if $var = proc\text{--}name$
$=$ (proc-val (procedure $bound\text{--}var$ $proc\text{--}body$ ρ_1))

(apply-env ρ_1 var)   if $var \neq proc\text{--}name$
$=$ (apply-env ρ var)

Note that the closure for the recursive procedure contains the creation environment ρ_1, not ρ.

3.4 LETREC: A Lang. with Rec. Procs. (Cont.)

Implementation:

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

(define apply-env (lambda (env search-var)
     (cases environment env
       . . .
       (extend-env-rec p-name b-var p-body saved-env)
         (if (eqv? search-var p-name)
           (proc-val (procedure b-var p-body env))
           (apply-env saved-env search-var))))))
```

See Figure 3.12 (as an extended solution of Exercise 2.21)

3.5 Scoping and Binding of Variables

Variables as references or as declarations in Scheme expressions
- (f x y), (lambda (x) (+ x 3)), (let ((x (+ y 7)) (+ x 3)))

The portion of a program in which a declaration is valid is called the *scope* of the declaration.
- Scoping rules (See Figure 3.13 and Figure 3.14)
- Static scoping rules (or lexical scoping rules)
- cf. Dynamic scoping rules (e.g., for exceptions)

The association between a variable and its value is called a binding
- Procedure variables, let variables, and letrec variables
- See three examples in P.90

The extent of a binding is the time interval during which the binding is maintained.
- Garbage collection (when bindings are no long reachable)

3.6 Eliminating Variable Names

The number of contours crossed is called the lexical (or static) depth of the variable reference.

```
(lambda (x)
  ((lambda (a)
     (x a))          ; depth(x)=1, depth(a)=0
   x))               ; depth(x)=0
```

Using this, we could get rid of variable names entirely as:

```
(nameless-lambda
  ((nameless-lambda
     (#1 #0))
   #0))
```

This number uniquely identifies the declaration to which it refers.

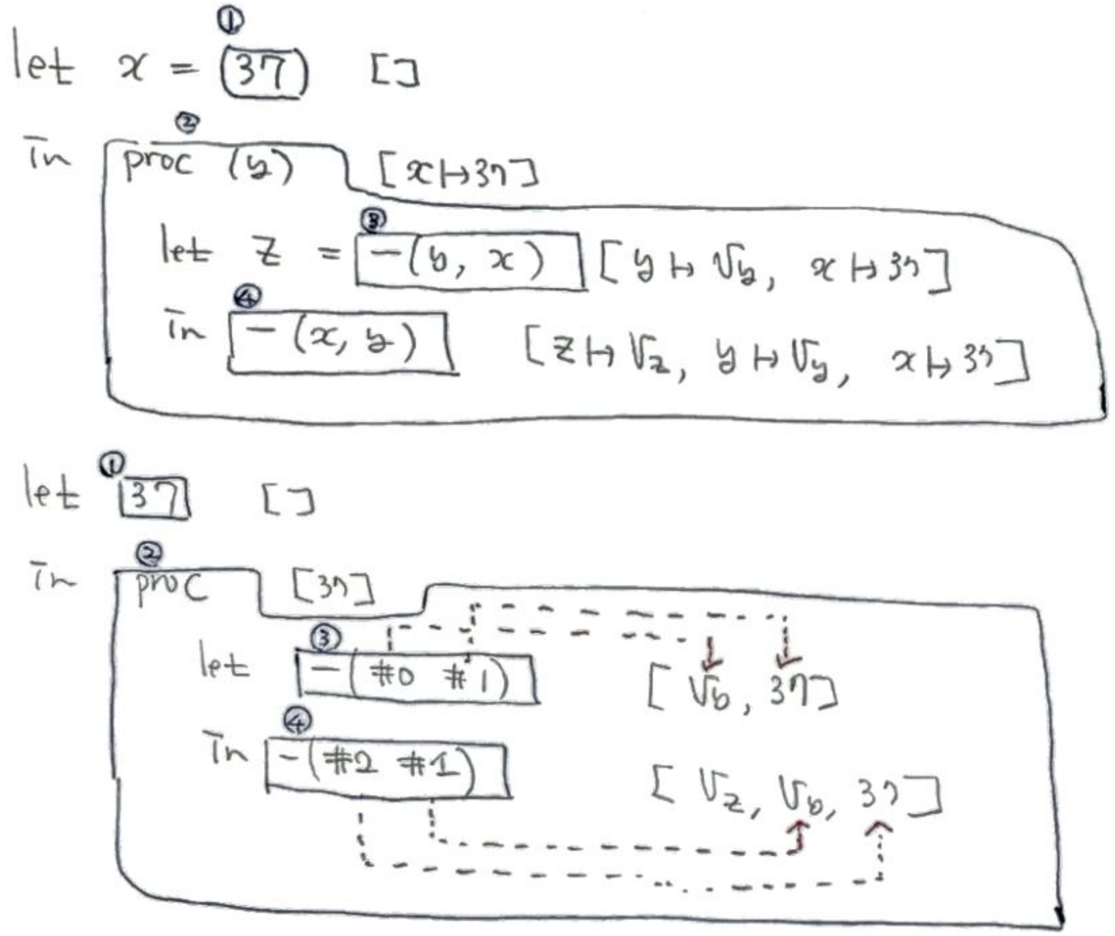
- Lexical addresses or de Bruijn indices

3.6 Eliminating Variable Names (Cont.)

This way of recording the information is useful because the lexical address predicts just where in the environment any particular variable will be found.

Examples in P.92 and P.93

3.6 Eliminating Variable Names (Cont.)



①
let x = 37 []
②
in proc (y) [x ↦ 37]
③
let z = −(y, x) [y ↦ v_y, x ↦ 37]
④
in −(x, y) [z ↦ v_z, y ↦ v_y, x ↦ 37]
①
let 37 []
②
in proc [37]
③
let −(#0 #1) [v_y, 37]
④
in −(#2 #1) [v_z, v_y, 37]

3.7 Implementing Lexical Addressing

Implementation of the lexical address analysis:

```
let x = 37
in proc (y)
   let z = -(y,x)          depth(y)=0, depth(x)=1
   in -(x,y)               depth(x)=2, depth(y)=1
```

is translated into

```
(a-program
   (nameless-let-exp (const-exp 37)
   (nameless-proc-exp
      (nameless-let-exp
         (diff-exp (nameless-var-exp 0) (nameless-var-exp 1))
         (diff-exp (nameless-var-exp 2) (nameless-var-exp 1))
```

3.7.1 The Translator

A procedure translation-of-program that
- takes a program,
- removes all the variables from the declarations, and
- replaces every variable reference by its lexical depth.

cf. translation-of-program vs. value-of-program

Static environment: a list of variables, representing the scopes within which the current expression lies.
- See Figure 3.15
- cf. Senv vs. Env

Implementation: P.96 and Figure 3.16

3.7.2 The Nameless Interpreter

This interpreter takes advantage of the predictions of the lexical-address analyzer to avoid explicitly searching for variables at runtime.

Nameless environment: a list of denoted values
- Figure 3.17 and the figure in P.98

Two modified procedures, value-of and value-of-program, with nameless environment
- Figure 3.18 and P.100