LET: A Simple Language – The 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⟩

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

(define-datatype expression expression?
;;  ... continues in next slide ...
```

LET: Abstract Syntax (cont.)

```
(define-datatype expression expression?
(const-exp (num number?))
(var-exp (var identifier?))
(diff-exp
  (exp1 expression?)
  (exp2 expression?))
(zero?-exp
  (exp1 expression?))
(if-exp
  (exp1 expression?)
  (exp2 expression?)
  (exp3 expression?))
(let-exp
  (var identifier?)
  (exp1 expression?)
  (body expression?)))
```

Let Σ be the set of strings formed from symbols

( ) z s

Example of elements in Σ

z zz $zsss$ s $s(($ $(())$

Example of inductive definition

Let S be the smallest subset of Σ that satisfies:

1. $z \in S$,
2. $s(n) \in S$ whenever $n \in S$.

Notation 1: Prose (already seen)

Let S be the smallest set that satisfies:

1. $z \in S$,
2. $s(n) \in S$ whenever $n \in S$.

Notation 2: Rule notation

$$\frac{}{z \in S} \qquad \frac{n \in S}{s(n) \in S}$$

Specification of Values (1/2)

Expressed Values: possible values of expressions.

$$ExpVal = Int + Bool$$

- Here is a datatype declaration for ExpVal

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

- Example expressions of this type are `(num-val 2)` and `(bool-val #t)`

1

PROC: abstract syntax (1/3)

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

Specification of Values (2/2)

Interface for the datatype of expressed values

`num-val`	:	Int	→	ExpVal
`bool-val`	:	Bool	→	ExpVal
`expval->num`	:	ExpVal	→	Int
`expval->bool`	:	ExpVal	→	Bool

```
;; expval->num : ExpVal -> Int
;; Page: 70
(define expval->num
  (lambda (v)
    (cases expval v
      (num-val (num) num)
      (else (expval-extractor-error 'num v)))))
```

9

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?))
```

```
  (proc-exp
    (var symbol?)
    (body expression?))
  (call-exp
    (rator expression?)
    (rand expression?)))
```

The Interpreter for PROC

- Before (for the LET-language)

  `value-of` : Exp → ExpVal

- Now

  `value-of` : Exp → ExpVal

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

  ExpVal = Int + Bool

- Now

  ExpVal = Int + Bool + Proc

Proc = Let +

⟨Expression⟩	::=	proc (⟨Identifier⟩) ⟨Expression⟩
⟨Expression⟩	::=	(⟨Expression⟩ ⟨Expression⟩)

expal = let +

```
(proc-val
    (proc proc?)))
```

- 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)))))
```

LETREC = Proc +

⟨Expression⟩ ::= Letrec ⟨Identifier⟩ (⟨Identifier⟩) = ⟨Expression⟩ in ⟨Expression⟩

```
(letrec-exp
  (p-name identifier?)
  (b-var identifier?)
  (p-body expression?)
  (letrec-body expression?)) )
```

```scheme
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
    (syms (list-of symbol?))
    (vals (list-of scheme-value?))
    (env environment?)))

(define scheme-value? (lambda (v) #t))
```

```scheme
(define empty-env
  (lambda ()
    (empty-env-record)))

(define extend-env
  (lambda (syms vals env)
    (extended-env-record syms vals env)))

(define apply-env
  (lambda (env sym)
    (cases environment env
      (empty-env-record ()
        (error 'apply-env "No binding for ~s" sym))
      (extended-env-record (syms vals env)
        (let ((pos (list-find-position sym syms)))
          (if (>= pos 0)
              (list-ref vals pos)
              (apply-env env sym))))
  )))
```

Free Variables Defined Formally

$$
\begin{array}{rcl}
\langle\text{exp}\rangle & ::= & \langle\text{identifier}\rangle \\
 & | & (\texttt{lambda}\ (\langle\text{identifier}\rangle)\ \langle\text{exp}\rangle) \\
 & | & (\langle\text{exp}\rangle\ \langle\text{exp}\rangle)
\end{array}
$$

$FV(\cdot) : \langle\text{exp}\rangle \to \wp\langle\text{var}\rangle$

$$
\begin{array}{rcl}
FV(x) & = & x \\
FV((\texttt{lambda}\,(x)\,E)) & = & FV(E)\ -\ \{x\} \\
FV((E_1\,E_2)) & = & FV(E_1) \cup FV(E_2)
\end{array}
$$

Bound Variables Defined Formally

$$
\begin{array}{rcl}
\langle\text{exp}\rangle & ::= & \langle\text{identifier}\rangle \\
 & | & (\texttt{lambda}\ (\langle\text{identifier}\rangle)\ \langle\text{exp}\rangle) \\
 & | & (\langle\text{exp}\rangle\ \langle\text{exp}\rangle)
\end{array}
$$

$BV(\cdot) : \langle\text{exp}\rangle \to \wp\langle\text{var}\rangle$

$$
\begin{array}{rcl}
BV(x) & = & \emptyset \\
BV((\texttt{lambda}\,(x)\,E)) & = & BV(E) \cup (\{x\} \cap FV(E)) \\
BV((E_1\,E_2)) & = & BV(E_1) \cup BV(E_2)
\end{array}
$$

```scheme
(define-datatype binTree binTree?
     (leaf-node
        (datum number?))
     (interior-node
        (left binTree?)
        (right binTree?)))

;; binTree? -> number?
(define leaf-sum
  (lambda (tree)
    (cases binTree tree
       (leaf-node (datum) datum)
       (interior-node (left right)
          (+ (leaf-sum left) (leaf-sum right))))))
```

```scheme
;; a, listof a -> boolean?
(define list-find-position
  (lambda (sym los)
    (list-index
      (lambda (sym1) (eqv? sym1 sym)) los)))

;; (a -> boolean?), listof a -> number?
(define list-index
  (lambda (pred ls)
    (cond
      ((null? ls) -1)
      ((pred (car ls)) 0)
      (else (let ((list-index-r
                  (list-index pred (cdr ls))))
              (if (>= list-index-r 0)
                  (+ list-index-r 1)
                  -1
              )
     )))))
```

```scheme
(define (po t)
  (cases BTree t
    (leaf-t (n) (list n))
    (node-t (n l r) (append (list n) (po l) (po r)))))

(define (io t)
  (cases BTree t
    (leaf-t (n) (list n))
    (node-t (n l r) (append (io l) (list n) (io r)))))

(define (pto t)
  (cases BTree t
    (leaf-t (n) (list n))
    (node-t (n l r) (append (pto l) (pto r) (list n) ))))
```

`(extend-env 'x 3 (empty-env))` has type `environment`.

`(procedure 'x (diff-exp (const-exp 11) (var-exp x)) extend-env 'x 3 (empty-env))` has type proc.

```scheme
(define-datatype BTree BTree?
  (leaf-t
     (data number?))
  (node-t
     (data number?)
     (left BTree?)
     (right BTree?)))
```

Define three functions `preorder`, `inorder` and `pos`
sals of a tree. For example, if `t` is the tree:

```scheme
(define t
  (node-t 6
     (node-t 2 (leaf-t 1)
               (node-t 4 (leaf-t 3)
                         (leaf-t 5)))
     (node-t 7 (leaf-t 8) (leaf-t 9)))
)
```

$$
\begin{array}{rl}
 & <E> \\
\to & <E> + <E> \\
\to & <E> + (<E>) \\
\to & <E> + (<E> - <E>) \\
\to & <Number> + (<E> - <E>) \\
\to & 1 + (<E> - <E>) \\
\to & 1 + (<Number> - <E>) \\
\to & 1 + (4 - <E>) \\
\to & 1 + (4 - <Number>) \\
\to & 1 + (4 - 7)
\end{array}
$$