CS 565 August 25, 1996
Jens Palsberg Readings: EOPL, Chap. 5

Supplement to Lecture Note 4:

Interpreters

Modeling a Statement-Oriented Language

Let's do a conventional-looking imperative language, with variables and procedures. The moral will be that by following the grammar as slavishly as possible, we will be able to scale up to somewhat larger languages.

Here are some sample programs:

```
(define pgm1 "{print 1}")

(define pgm2 "var x {x = 1 print x; x = x+1 print x}")

(define pgm3
"var x
 proc incx(y){x = x + y}
 {x = 1  incx(2)  print x}")

(define pgm4
"var x y z
 proc mult(x,y)
   var acc, tx
   {acc = 0; tx = x
    while (tx > 0) {acc = acc + y; tx = tx - 1}
    z = acc}
 {read x; read y; mult(x,y); print z;}")
```

Lexical Specification

```
(define lexical-specification
  '((var proc if while else print read)
    (start-state
      (whitespace #f)
      (#\+ plus-sym)
      (#\- minus-sym)
      (#\/ div-sym)
      (#\* mult-sym)
      (#\< lt-state)
      (#\> gt-state)
      (#\= equal-sym)
      (#\! #\= unequal-sym)
      (#\,  comma)
      (#\;  semicolon)
      (#\(  lparen)
      (#\)  rparen)
      (#\{  lbrace)
      (#\}  rbrace)
      ((alphabetic)
        (arbno (alphabetic #\_ numeric))
        identifier)
      (numeric (arbno numeric) number)
      (#\%  comment-state)
      (#\^  end-marker))
    (lt-state
      (#\=  le-sym)
      (else lt-sym))   ; emit the < and reprocess next char
```

```
(gt-state
  (#\=  ge-sym)
  (else gt-sym))
(comment-state
  (#\newline #f)
  (any comment-state))))
```

Grammatical Specification

$\langle \text{pgm} \rangle ::= \langle \text{block} \rangle$

`pgm`

$\langle \text{block} \rangle ::= \langle \text{var-decl} \rangle \ \{\langle \text{proc-decl} \rangle\}^* \ \langle \text{compound-statement} \rangle$

`block`

A program consists of just a block; a block consists of a `var` declaration, some procedure declarations, and a compound statement.

$\langle \text{var-decl} \rangle ::= \epsilon$

`empty-var-decl`

$\langle \text{var-decl} \rangle ::=$ `var` $\langle \text{identifier-list} \rangle$

`non-empty-var-decl`

$\langle \text{proc-decl} \rangle ::=$ `proc identifier (` $\langle \text{identifier-list} \rangle$ `)` $\langle \text{block} \rangle$

`proc-decl`

$\langle \text{compound-statement} \rangle ::=$ `lbrace` $\{\langle \text{statement} \rangle \ \langle \text{optional-semicolon} \rangle\}^*$ `rbrace`

`compound-statement`

A `var` declaration is either empty or is the keyword `var` followed by an identifier list. Lists of identifiers are separated by optional commas. A `proc` declaration looks like what you think; the body of a proc is a block.

Variables declared by a `var` denote mutable (assignable) cells. The procedures declared in a block are mutually recursive and within the scope of the `var`'s. Otherwise, the scope of thee declarations is the compound statement.

A compound statement is a list of statements surrounded by braces and separated by optional semicolons.

Here's the grammar for the optional-commas, etc. This was done VERY CAREFULLY to stay within the LL restriction.

⟨identifier-list⟩	::=	$\{$ `identifier` ⟨optional-comma⟩ $\}^*$
		`identifier-list`
⟨operand-list⟩	::=	`(` $\{$⟨expression⟩ ⟨optional-comma⟩ $\}^*$ `)`
		`operand-list`
⟨optional-comma⟩	::=	`comma`
		`comma`
⟨optional-comma⟩	::=	
		`no-comma`
⟨optional-semicolon⟩	::=	`semicolon`
		`semicolon`
⟨optional-semicolon⟩	::=	
		`no-semicolon`

Statements

We have conditionals, whiles, prints, and reads. Read is supposed to read from the input and assign the result to the given variable. Notice how `if-tail` distinguishes 1-sided from 2-sided if's. Similarly, how `primitive-tail` distinguishes assignment statements from procedure calls. This is a bit of trickery needed to stay within the LL(1) framework; otherwise you'd have two or more productions for `statement` starting with the same non-terminal.

$\langle$statement$\rangle$	::=	`if` $\langle$expression$\rangle$ $\langle$compound-statement$\rangle$ $\langle$if-tail$\rangle$
		`if-statement`
$\langle$statement$\rangle$	::=	`while` $\langle$expression$\rangle$ $\langle$compound-statement$\rangle$
		`while-statement`
$\langle$statement$\rangle$	::=	`print` $\langle$expression$\rangle$
		`print-statement`
$\langle$statement$\rangle$	::=	`read identifier`
		`read-statement`
$\langle$statement$\rangle$	::=	`identifier` $\langle$primitive-tail$\rangle$
		`primitive-statement`
$\langle$if-tail$\rangle$	::=	
		`empty-if-tail`
$\langle$if-tail$\rangle$	::=	$\langle$compound-statement$\rangle$
		`non-empty-if-tail`
$\langle$primitive-tail$\rangle$	::=	`equal-sym` $\langle$expression$\rangle$
		`assignment-statement`
$\langle$primitive-tail$\rangle$	::=	$\langle$operand-list$\rangle$
		`proc-call-statement`

Arithmetic Expressions

We have 4 levels of precedence: expressions (linked by relational operators), sums, terms (or products), and factors. This is like the previous problem set.

$\langle\text{expression}\rangle ::= \langle\text{sum}\rangle\ \{\langle\text{relational-op}\rangle\ \langle\text{sum}\rangle\ \}^*$
  `expression`

$\langle\text{sum}\rangle ::= \langle\text{term}\rangle\ \{\langle\text{additive-op}\rangle\ \langle\text{term}\rangle\ \}^*$
  `sum`

$\langle\text{term}\rangle ::= \langle\text{factor}\rangle\ \{\langle\text{mult-op}\rangle\ \langle\text{factor}\rangle\ \}^*$
  `product`

$\langle\text{factor}\rangle ::=$ `number`
  `const-factor`

$\langle\text{factor}\rangle ::=$ `(` $\langle\text{expression}\rangle$ `)`
  `paren-exp`

$\langle\text{factor}\rangle ::=$ `identifier`
  `identifier-exp`

Arithmetic Operators

⟨additive-op⟩ ::= `plus-sym`
`plus-op`
⟨additive-op⟩ ::= `minus-sym`
`minus-op`
⟨mult-op⟩ ::= `mult-sym`
`times-op`
⟨mult-op⟩ ::= `div-sym`
`div-op`
⟨relational-op⟩ ::= `equal-sym`
`equal-op`
⟨relational-op⟩ ::= `unequal-sym`
`unequal-op`
⟨relational-op⟩ ::= `lt-sym`
`lt-op`
⟨relational-op⟩ ::= `gt-sym`
`gt-op`
⟨relational-op⟩ ::= `le-sym`
`le-op`
⟨relational-op⟩ ::= `ge-sym`
`ge-op`

The Interpreter

To build the interpreter, we structure our program to follow the grammar. We will use only a very small amount of cleverness, almost entirely for dealing with arbnos.

We have 19 nonterminals and 37 productions. We will have to write about 19 procedures (about one per non-terminal). These will have to cover the 37 possibilities.

There are no primops in this language, because all the arithmetic operators are built into the grammar. So we can start with the empty environment:

```
;    (pgm
;        (block)
;        pgm)

(define eval-pgm
  (lambda (pgm)
    (record-case pgm
      (pgm (block)
           (let ((env (empty-env)))
             (eval-block block env)))
      (else (raise-error 'eval-pgm pgm)))))
```

Blocks

For a block, we process the declarations, and then perform the compound statement in the resulting environment. We'll use a slightly different interface for environments: `extend-env` will take a rib as an argument. `eval-var-decl` and `eval-proc-decl*` need to return ribs.

```
;      (block
;        (var-decl (arbno proc-decl) compound-statement)
;        block)

(define eval-block
  (lambda (block env)
    (record-case block
      (block (var-decl proc-decl* compound-statement)
             (let*
               ((new-vars (eval-var-decl var-decl))
                (new-env  (extend-env new-vars env))
                (new-procs (eval-proc-decl* proc-decl*))
                (new-env-2 (extend-env new-procs new-env)))
               (eval-compound-statement
                 compound-statement new-env-2)))
      (else (raise-error 'eval-block block)))))
```

VAR declarations

`eval-var-decl` just returns a suitable rib. It calls `eval-identifier-list` to find the list of names. What are the cells initialized with?
How could we change this?

```
;      (var-decl
;        ()
;        empty-var-decl)
;      (var-decl
;        (var identifier-list)
;        non-empty-var-decl)

(define-record var-rib (names cells))

;;; returns rib for use by extend-env

(define eval-var-decl
  (lambda (var-decl)
    (record-case var-decl
      (empty-var-decl () (make-var-rib '() '()))
      (non-empty-var-decl (identifier-list)
        (let ((names (eval-identifier-list identifier-list)))
          (make-var-rib names (map make-cell names))))
      (else (raise-error 'eval-var-decl var-decl)))))
```

`eval-identifier-list` strips out the optional commas and returns the list of identifiers in the identifier-list.

```
;     (identifier-list
;       ((arbno identifier optional-comma))
;       identifier-list)

(define eval-identifier-list
  (lambda (identifier-list)
    (record-case identifier-list
      (identifier-list (identifier-and-optional-comma-list)
        (letrec
          ((loop (lambda (l)
                   ;; l is of form (identifier optional-comma ...)
                   ;; strip off the optional-commas and return the list of
                   ;; identifiers
                   (if (null? l) '()
                     (cons (car l) (loop (cddr l)))))))
          (loop identifier-and-optional-comma-list)))
      (else (raise-error 'eval-identifier-list identifier-list)))))
```

`eval-proc-decl*` takes a list of proc-decls (no optional semi-colons here. Is this good or bad?) and produces a rib of `procdef`s. These are closures without environments; `apply-env` will turn them into real closures.

```
;      (proc-decl
;        (proc identifier lparen identifier-list rparen block)
;        proc-decl)

(define-record proc-rib (procdefs))
(define-record procdef (name formals body))

(define eval-proc-decl*
  (lambda (proc-decl*)
    (make-proc-rib (map eval-proc-decl proc-decl*))))

(define eval-proc-decl
  (lambda (proc-decl)
    (record-case proc-decl
      (proc-decl (name identifier-list body)
        (make-procdef
          name
          (eval-identifier-list identifier-list)
          body)))))
```

Compound Statements

For a compound statement, we have a list of alternating statements and optional-semicolons. Evaluate the statements in order.

```
;     (compound-statement
;       (lbrace (arbno statement optional-semicolon) rbrace)
;       compound-statement)

(define eval-compound-statement
  (lambda (compound env)
    (record-case compound
      (compound-statement (statement-and-semicolon-list)
        (letrec
          ((loop (lambda (statement-and-semicolon-list)
                   (if (null? statement-and-semicolon-list)
                     #t                ; statements work by side-effect
                     (begin
                       (eval-statement
                         (car statement-and-semicolon-list) env)
                       (loop (cddr statement-and-semicolon-list)))))))
          (loop statement-and-semicolon-list)))
      (else (raise-error 'eval-compound-statement compound)))))
```

Statements

Now for the statements. These are pretty routine. Note how the grammar loops back to `compound-statement` for all the structured statements.

```
;      (statement
;        (if expression compound-statement if-tail)
;        if-statement)
;      (statement
;        (while expression compound-statement)
;        while-statement)
;      (statement
;        (print expression)
;        print-statement)
;       (statement
;         (read identifier)
;         read-statement)
;      (statement
;        (identifier primitive-tail)
;        primitive-statement)
```

```
(define eval-statement
  (lambda (statement env)
    (record-case statement
      (if-statement
        (expression compound-statement if-tail)
        (if (true-value? (eval-expression expression env))
          (eval-compound-statement env)
          (eval-if-tail if-tail env)))
      (while-statement
        (expression compound-statement)
        (letrec
          ((loop (lambda ()
                   (if (true-value? (eval-expression expression env))
                     (begin
                       (eval-compound-statement compound-statement env)
                       (loop))
                     #t))))
          (loop)))
      (print-statement
        (expression)
        (display (eval-expression expression env))
        (printf "~%"))
      (read-statement
        (identifier)
        (let ((cell (apply-env env identifier))
              (value (prompt-read "input>")))
          (set-cell! cell value)))
      (primitive-statement
        (identifier primitive-tail)
        (eval-primitive-tail identifier primitive-tail env))
      (else (raise-error 'eval-statement statement)))))
```

```
;     (if-tail
;       ()
;       empty-if-tail)
;     (if-tail
;       (compound-statement)
;       non-empty-if-tail)

(define eval-if-tail
  (lambda (if-tail env)
    (record-case if-tail
      (empty-if-tail () #t)             ; do nothing
      (non-empty-if-tail (compound-statement)
        (eval-compound-statement compound-statement env))
      (else (raise-error 'eval-if-tail if-tail)))))
```

Primitive Statements

Primitive statements are the ones that operate on variables. They all start with an identifier, so this gets carried down to `eval-primitive-tail` where it will be used.

```
;      (primitive-tail
;        (= expression)
;        assignment-statement)
;      (primitive-tail
;        (operand-list)
;        proc-call-statement)

(define eval-primitive-tail
  (lambda (identifier primitive-tail env)
    (record-case primitive-tail
      (assignment-statement (expression)
        (set-cell!
          (apply-env env identifier)
          (eval-expression expression env)))
      (proc-call-statement
        (operand-list)
        (apply-proc
          (apply-env env identifier)
          (eval-operand-list operand-list env)))
      (else (raise-error 'eval-primitive-tail
                         primitive-tail)))))
```

Operand Lists

For an operand list, evaluate the expressions in turn and form the list of values. This involves the same sort of looping as `eval-identifier-list`.

```
;     (operand-list
;       (lparen (arbno expression optional-comma) rparen)
;       operand-list)

(define eval-operand-list
  (lambda (operand-list env)
    (record-case operand-list
      (operand-list (expression-and-comma-list)
        (letrec
          ((loop (lambda (list)
                   (if (null? list) '()
                     (cons (eval-expression (car list) env)
                           (loop (cddr list)))))))
          (loop expression-and-comma-list)))
      (else (raise-error 'eval-operand-list operand-list)))))
```

Arithmetic Expressions

Now we get to expressions. These are like the ones in the problem set. `rest` is a list of alternating `relational-op`s and `sum`s. We call `apply-relational-op` to decode the relational operator and apply it to the arguments. Since the pieces are `sum`'s, we call `eval-sum` to evaluate each piece, following the grammar.

```
;      (expression
;        (sum (arbno relational-op sum))
;        expression)

(define eval-expression
  (lambda (expression env)
    (record-case expression
      (expression (seed rest)
        (letrec
          ((loop (lambda (acc rest)
                   (if (null? rest) acc
                     (loop (apply-relational-op (car rest)
                            acc
                            (eval-sum (cadr rest) env))
                           (cddr rest))))))
          (loop (eval-sum seed env) rest)))
      (else (raise-error 'eval-expression expression)))))
```

```
;      (sum
;        (term (arbno additive-op term))
;        sum)

(define eval-sum
  (lambda (sum env)
    (record-case sum
      (sum (seed rest)
        (letrec
          ((loop (lambda (acc rest)
                   (if (null? rest) acc
                     (loop (apply-additive-op (car rest)
                            acc
                            (eval-term (cadr rest) env))
                           (cddr rest))))))
          (loop (eval-term seed env) rest)))
      (else (raise-error 'eval-sum sum)))))
```

```
;      (term
;        (factor (arbno mult-op factor))
;        product)

(define eval-term
  (lambda (term env)
    (record-case term
      (product (seed rest)
        (letrec
          ((loop (lambda (acc rest)
                   (if (null? rest) acc
                     (loop (apply-mult-op (car rest)
                            acc
                            (eval-factor (cadr rest) env))
                           (cddr rest))))))
          (loop (eval-factor seed env) rest)))
      (else (raise-error 'eval-term term)))))
```

For factors, we have several cases. Notice the `identifier-exp` line– this is where we dereference a cell from the environment.

```
;      (factor
;        (number)
;        const-factor)
;      (factor
;        (lparen expression rparen)
;        paren-exp)
;      (factor
;        (identifier)
;        identifier-exp)
```

```
(define eval-factor
  (lambda (factor env)
    (record-case factor
      (const-factor (num) num)
      (paren-exp (expression) (eval-expression expression env))
      (identifier-exp (ident) (deref (apply-env env ident)))
      (else (raise-error 'eval-factor factor)))))
```

Now we can get to the application of the primitive operators:

```
;       (additive-op (+) plus-op)
;       (additive-op (-) minus-op)


(define apply-additive-op
  (lambda (additive-op x y)
    (record-case additive-op
      (plus-op () (+ x y))
      (minus-op ()(- x y))
      (else (raise-error 'apply-additive-op additive-op)))))

;       (mult-op (*) times-op)
;       (mult-op (/) div-op)

(define apply-mult-op
  (lambda (mult-op x y)
    (record-case mult-op
      (times-op () (* x y))
      (div-op () (/ x y))
      (else (raise-error 'apply-mult-op mult-op)))))
```

```
;       (relational-op (equal-sym) equal-op)
;       (relational-op (unequal-sym) unequal-op)
;       (relational-op (lt-sym) lt-op)
;       (relational-op (gt-sym) gt-op)
;       (relational-op (le-sym) le-op)
;       (relational-op (ge-sym) ge-op)


(define apply-relational-op
  (lambda (relational-op x y)
    (record-case relational-op
      (equal-op () (if (= x y) true-value false-value))
      (unequal-op () (if (= x y) false-value true-value))
      (gt-op () (if (> x y) true-value false-value))
      (lt-op () (if (< x y) true-value false-value))
      (ge-op () (if (>= x y) true-value false-value))
      (le-op () (if (<= x y) true-value false-value))
      (else (raise-error 'eval-relational-op relational-op)))))
```

Procedures

Procedures are routine. Note that `apply-proc` has to construct the rib to give to `extend-env`, as per our previous discussion of `extend-env`.

```
(define-record closure (formals body env))

(define apply-proc
  (lambda (proc args)
    (record-case proc
      (closure (formals body env)
        (eval-block body
          (extend-env
            (make-var-rib formals (map make-cell args))
            env)
         ))
      (else (raise-error 'apply-proc proc)))))
```

Environments

An environment is a list of ribs. Each rib is either a var-rib or a proc-rib. vars are in cells, procs are not. This means you can't pass a procedure. [Is this a bug or a feature?]

```
(define empty-env
  (lambda () '()))

(define extend-env
  (lambda (rib old-env)
    (cons rib old-env)))

(define apply-env
  (lambda (ribs var)
    (if (null? ribs)
      (error 'apply-env "unbound variable ~s" var)
      (record-case (car ribs)
        (var-rib (names cells)
          (lookup-in-var-rib var names cells (cdr ribs)))
        (proc-rib (procs)
          (lookup-in-proc-rib procs var ribs))
        (else (raise-error 'apply-env ribs))))))
```

Here is where the work gets done:

```
(define lookup-in-var-rib
  (lambda (var names cells old-ribs)
    (cond
      ((null? names) (apply-env old-ribs var))
      ((null? cells) (error 'lookup-in-var-rib
                       "not enough cells in rib: names = ~s cells = ~s"
                       names cells))
      ((eq? var (car names)) (car cells))
      (else (lookup-in-var-rib var (cdr names) (cdr cells) old-ribs)))))

(define lookup-in-proc-rib
  (lambda (procdefs var ribs)
    (cond
      ((null? procdefs) (apply-env (cdr ribs) var))
      ((eq? var (procdef->name (car procdefs)))
       (make-closure
         (procdef->formals (car procdefs))
         (procdef->body    (car procdefs))
         ribs                           ; the recursive environment
         ))
      (else
        (lookup-in-proc-rib (cdr procdefs) var ribs)))))
```

```
;;; Cells

(define make-cell
  (lambda (value)
    (cons '*cell value)))

(define deref
  (lambda (cell)
    (cdr cell)))

(define set-cell!
  (lambda (cell value)
    (set-cdr! cell value)))

;;; booleans

(define true-value 1)
(define false-value 0)

(define true-value?
  (lambda (val) (not (zero? val))))

;;; miscellany

(define raise-error
  (lambda (name value)
    (error name "unknown structure ~s" value)))

(define prompt-read
  (lambda (msg)
    (printf "~d " msg)
    (read)))
```

Testing

```
(define scan&parse
  (sllgen:string->tree lexical-specification
    syntactic-specification))

(define run
  (lambda (string)
    (eval-pgm (scan&parse string))))

> (run pgm1)
1
#t
> (run pgm2)
1
2
#t
> (run pgm3)
3
#t
> (run pgm4)
input> 3
input> 4
12
#t
```