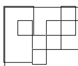
Environment Passing Interpreter

CS395T - Domain Specific Languages
Greg Lavender
Department of Computer Sciences
The University of Texas at Austin



The Read-Eval-Print Loop

- The "read" is done by the SLLGEN lexer+syntax analyzer

```
(define read-eval-print
 (sllgen:make-rep-loop
    "--> "  ; prompt string
    (lambda (pgm) (eval-program pgm)) ; eval proc
    (sllgen:make-stream-parser the-lexical-spec the-grammar)))
```

- The "eval" is done on the parsed expression using the current environment

```
(define eval-program
   (lambda (pgm)
    (cases program pgm  ; program is symbol passed by parser
           (a-program (body)
           (eval-expression body (init-env))))))
```

SSLGEN Lexical Grammar Rules

- Regular expressions are defined as strings
  - a single character string, the recursive concatenation of two strings, a (union) set of alternative strings, a recursive string (Kleene closure), or negation of a character

```
<R> ::= <character> | <R><R> | (<R> | <R>) | <R>* | ~<character>
```

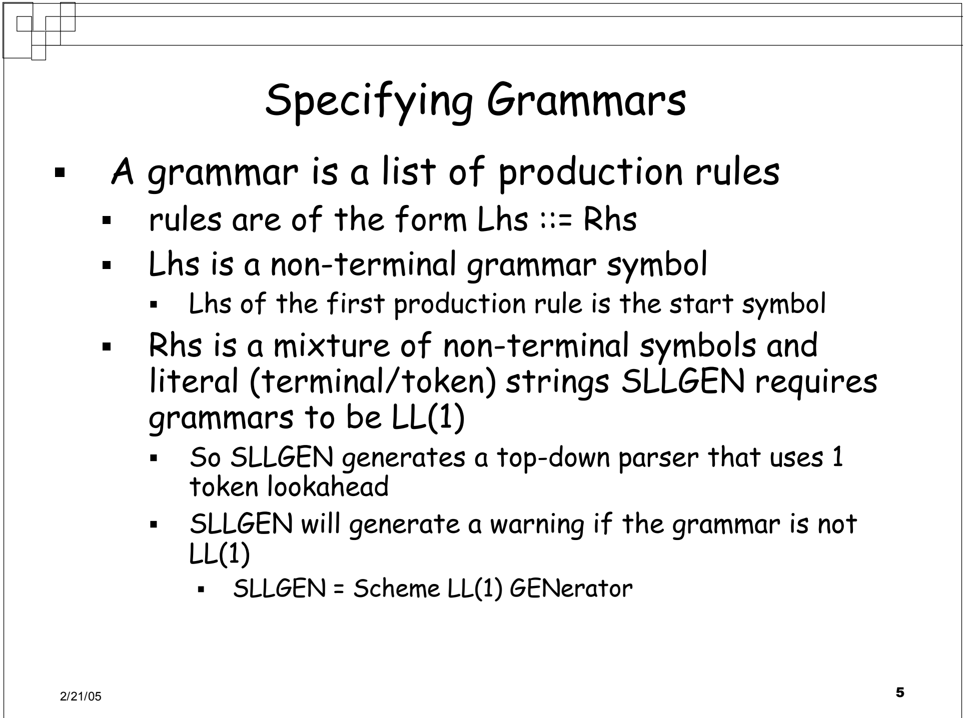
- Also called a scanner spec

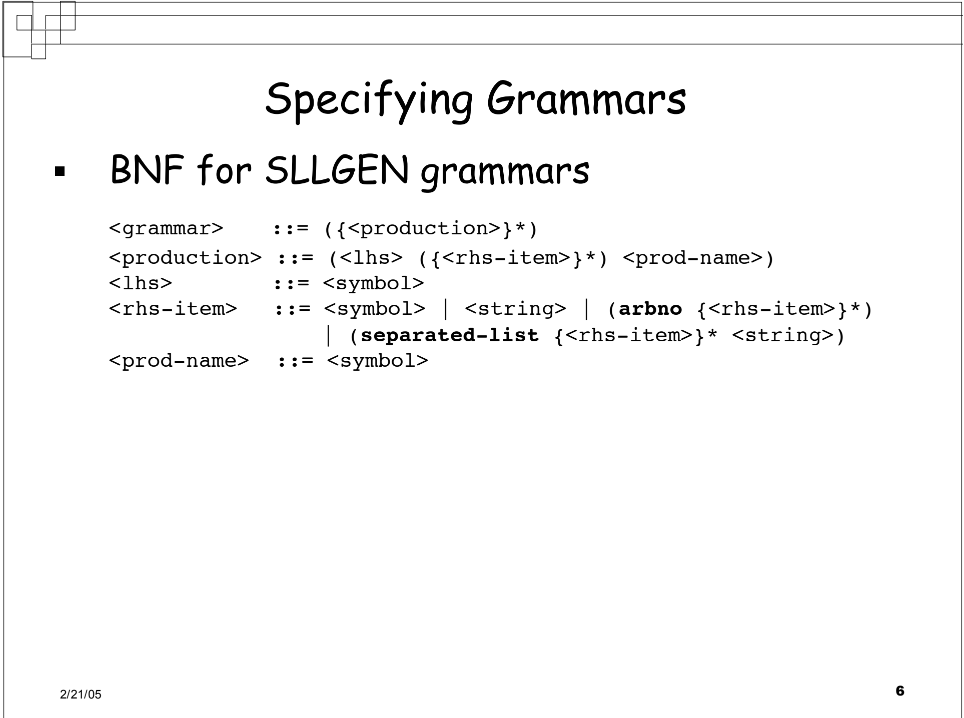
```
<scanner-spec>        ::= {  {<regexp-and-action>}* }

<regexp-and-action> ::= ( <name> ({<regexp>}* <outcome> )

<name>       ::= <symbol>

<regexp>     ::= <string> | letter | digit | whitespace | any
                       | (not <character> | (or {<regexp>}*)
                       | (arbno <regexp>) | (concat {<regexp>}*)

<outcome>    ::= skip | symbol | number | string
```


Interpreter Lexical Specification

- Tokens are whitespace, comments, identifiers and numbers
  - SSLGEN defines some special operators to allow us to write lexical rules to match tokens using common regular expression patterns
  - patterns match the longest possible token in the input stream

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


Specifying Grammars
A grammar is a list of production rules
rules are of the form Lhs ::= Rhs
Lhs is a non-terminal grammar symbol
Lhs of the first production rule is the start symbol
Rhs is a mixture of non-terminal symbols and literal (terminal/token) strings SLLGEN requires grammars to be LL(1)
So SLLGEN generates a top-down parser that uses 1 token lookahead
SLLGEN will generate a warning if the grammar is not LL(1)
SLLGEN = Scheme LL(1) GENerator
2/21/05
5


Specifying Grammars
BNF for SLLGEN grammars
<grammar> ::= ({<production>}*)
<production> ::= (<lhs> ({<rhs-item>}*) <prod-name>)
<lhs> ::= <symbol>
<rhs-item> ::= <symbol> | <string> | (arbno {<rhs-item>}*)
| (separated-list {<rhs-item>}* <string>)
<prod-name> ::= <symbol>
2/21/05
6

Interpreter Grammar Specification

```
(define the-grammar
  '((program (expression) a-program)
    (expression (number) lit-exp)
    (expression (identifier) var-exp)
    (expression
      (primitive "(" (separated-list expression ",") ")") primapp-exp)
    (expression
      ("if" expression "then" expression "else" expression) if-exp)
    (expression
      ("let" (arbno  identifier "=" expression) "in" expression) let-exp)
    (expression
      ("proc" "(" (separated-list identifier ",") ")" expression) proc-exp)
    (expression
      ("(" expression (arbno expression) ")") app-exp)
    (expression
      ("begin" expression (arbno ";" expression) "end") begin-exp)
    (primitive ("+")     add-prim)
    (primitive ("-")     subtract-prim)
    (primitive ("*")     mult-prim)
    (primitive ("add1")  incr-prim)
    (primitive ("sub1")  decr-prim)
    (primitive ("zero?") zero-test-prim)
    ))
```


Building the Scanner/Parser

- SSLGEN will use the scanner/parser grammar specifications to generate an executable parser that we call from the REPL code
  - we need the following boilerplate code to do this

```
(define the-lexical-spec …)  ;; as before
(define the-grammar …)       ;; as before

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

(define show-the-datatypes
  (lambda () (sllgen:list-define-datatypes the-lexical-spec
                                           the-grammar))
(define just-scan
  (sllgen:make-string-scanner the-lexical-spec the-grammar))

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

(define read-eval-print …) ;; as before
```

Interpreter Semantics

- The semantics for the interpreter are associated with each production rule in the grammar
  - when we match the rhs of a rule, we want to execute a procedure associated with that rule, relative to the current environment
  - The semantic action "computes" the semantic value of the parsed syntactic expression
    - relative to the current environment
  - the semantic action associated with a production rule may also update the environment
    - So we have to pass the environment to each procedure that evaluates the semantics of an expression

Expression Evaluation

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      (lit-exp (datum) datum)
      (var-exp (id) (apply-env env id))
      (primapp-exp (prim rands)
        (let ((args (eval-rands rands env)))
          (apply-primitive prim args)))
      (if-exp (test-exp true-exp false-exp)
         (if (true-value? (eval-expression test-exp env))
          (eval-expression true-exp env)
          (eval-expression false-exp env)))
      (begin-exp (exp1 exps)
        (let loop ((acc (eval-expression exp1 env))
                   (exps exps))
          (if (null? exps) acc
            (loop (eval-expression (car exps) env) (cdr exps)))))
      (let-exp (ids rands body)
        (let ((args (eval-rands rands env)))
          (eval-expression body (extend-env ids args env))))
      (proc-exp (ids body) (closure ids body env))
      (app-exp (rator rands)
        (let ((proc (eval-expression rator env))
              (args (eval-rands rands env)))
          (if (procval? proc) (apply-procval proc args)
            (eopl:error 'eval-expression
                        "Attempt to apply non-procedure ~s" proc))))
      (else (eopl:error 'eval-expression "Not here:~s" exp)))))
```

- The simplest expressions are literals (constants) and identifiers (variables)
  - the semantic value of a literal is itself

```
(lit-exp (datum) datum)
```

  - the semantic value of a variable is its value in the current environment, if any

```
(var-exp (id) (apply-env env id))
```

- The environment is simply a vector of name-to-values bindings. The initial environment represents the default set of name-to-value bindings for the interpreter

```
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
    (syms (list-of symbol?))
    (vec vector?)              ; can use this for anything.
    (env environment?)))

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

(define extend-env
  (lambda (syms vals env)
    (extended-env-record syms (list->vector vals) env)))

(define init-env
    (lambda () (extend-env '(i v x) '(1 5 10) (empty-env))))
```

Symbol Lookup in the Environment

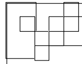
- When evaluating operands to an expression, we have to lookup up identifiers to determine their values before evaluation.

```
(define apply-env
  (lambda (env sym)
    (cases environment env
      (empty-env-record ()
        (eopl:error 'apply-env "No binding for ~s" sym))
      (extended-env-record (syms vals env)
        (let ((position (rib-find-position sym syms)))
          (if (number? position)
              (vector-ref vals position)
              (apply-env env sym)))))))

(define rib-find-position
  (lambda (sym los) (list-find-position sym los)))

(define list-find-position
  (lambda (sym los)
    (list-index (lambda (sym1) (eqv? sym1 sym)) los)))
```


Primitive Expression Evaluation

- The semantic value of a primitive expression is computed by applying the primitive operator to its operand(s)
  - rator = operator
  - rand = operand, rands = list of operands

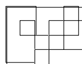
```
(primapp-exp (prim rands)
     (let ((args (eval-rands rands env)))
       (apply-primitive prim args)))

(define eval-rands
  (lambda (rands env)
    (map (lambda (x) (eval-rand x env)) rands)))

(define eval-rand
  (lambda (rand env)
    (eval-expression rand env))) ;will invoke var-exp
```



Primitive Expression Evaluation

- Once we resolve all literals and identifiers to their values, we can invoke the semantic operation corresponding to a primitive operator
  - note that these primitives do not check number of arguments

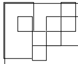
```
(define apply-primitive
  (lambda (prim args)
    (cases primitive prim
      (add-prim  () (+ (car args) (cadr args)))
      (subtract-prim () (- (car args) (cadr args)))
      (mult-prim  () (* (car args) (cadr args)))
      (incr-prim  () (+ (car args) 1))
      (decr-prim  () (- (car args) 1))
      (zero-test-prim () (if (zero? (car args)) 1 0))
      )))
```

IF Expression Evaluation

- The IF expression semantically evaluates a syntactic conditional using a short-circuited conditional
  - First evaluate the test expression relative to the current environment
    - If true evaluate the true-exp in the current environment
    - If false, evaluate the false-exp in the current environment

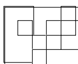
```
(if-exp (test-exp true-exp false-exp)
   (if (true-value? (eval-expression test-exp env))
      (eval-expression true-exp env)
      (eval-expression false-exp env)))

(define true-value?
   (lambda (x)
    (not (zero? x))))
```



Let Expression

- The let expression creates a temporarily extended set of environment bindings for evaluating an expressions and then evaluates an expression relative to that temporary environment

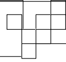
```
(let-exp (ids rands body)
    (let ((args (eval-rands rands env)))
      (eval-expression body (extend-env ids args env))))
```

Expression Application

- An “apply” expression applies an operator to a list of operands
  - If the operator is a procedure, then the corresponding procedure is applied, otherwise its an error

```
(app-exp (rator rands)
        (let ((proc (eval-expression rator env))
              (args (eval-rands rands env)))
          (if (procval? proc) (apply-procval proc args)
            (eopl:error 'eval-expression
                   "Attempt to apply non-procedure ~s" proc))))

(define apply-procval
  (lambda (proc args)
    (cases procval proc
      (closure (ids body env)
        (eval-expression body (extend-env ids args env))))))
```



Homework

- Read the rest of Chapter 3
  - See the book's website to download code examples
    - http://www.cs.indiana.edu/eopl/
  - Due Friday, Feb 25th.
    - Extend the interpreter according to these exercises:
      - 3.7, 3.8, 3.10, 3.11, 3.12, 3.13, 3.15
      - I recommend doing 3.15 before 3.10 to add boolean expressions rather than using 0/1 to represent false/true
  - Due Monday, Feb 28th
    - Using the extended interpreter do the following Exercises
      - 3.21, 3.22
    - Add the following additional features to the interpreter:
      - 3.29 and also trace the factorial proc you wrote in 3.21
      - 3.40