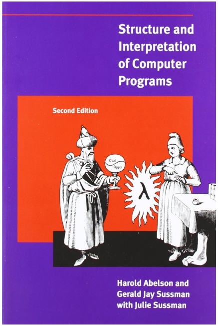
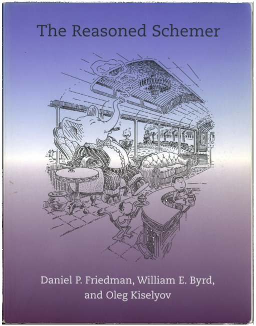
Structure and Interpretation of Definite Clause Grammars (DCGs)



Structure and Interpretation of Computer Programs

Second Edition

Harold Abelson and
Gerald Jay Sussman
with Julie Sussman



The Reasoned Schemer

Daniel P. Friedman, William E. Byrd,
and Oleg Kiselyov

Fundamentalist Declarative Programming with Scheme

State of Pure[3] as of January 10, 2014. Peter Kourzanov[1,2]

[2]TU Delft (TUD) Parallel & Distributed Systems (PDS) group
[1]NXP Research & Development, Eindhoven, Netherlands

Abstract

Pure^3 $\equiv$ Declarative approach to Declarative parsing with Declarative tools

DCGs is a technique that allows one to embed a parser for a context-sensitive language into logic programming, via Horn clauses. A carefully designed grammar/parser can be run forwards, backwards and sideways. In this talk we shall **de-construct** DCGs using `syntax-rules` and MINIKANREN, a library using the Revised^5 Report on the Algorithmic Language Scheme (R5RS) and implementing a compact logic-programming system, keeping reversibility in mind. Parsing Expression Grammars (PEGs) is a related technique that like DCGs also suffers from the inability to express left-recursive grammars. We make a link between DCGs and PEGs by borrowing the mechanism from DCGs, adding meta-syntactic sugar from PEGs and propose a way to run possibly left-recursive parsers using either formalism in a *pure*, on-line fashion. Finally, we **re-interpret** DCGs as executable, bidirectional Domain-Specific Language (DSL) specifications and transformations, perhaps better suited for DSL design than R5RS `syntax-rules`.

The whole presentation is literate Scheme code, including almost everything needed to implement the proposed technique from scratch

How this all started

Our mission

> *“Carefully designed grammar can run backwards, generating text from meaning.” ... seen in the Wild Web in the context of natural language processing [**?**]*

R5RS [ABB^{+}98] allows us to build Domain-Specific Languages (DSLs) embedded in Scheme. E.g., *pattern-matching* and *staging* in the style of Meta Language (ML) as well as the *monads* in the style of Haskell, modulo types of course (more details: [KS13])

```
(def deintreave (fn'
  () → () ()
| (,x ,y . ,[~ deinterleave→ ‘(,a ,b)]) →
  (,x . ,a) (,y . ,b)
))
```

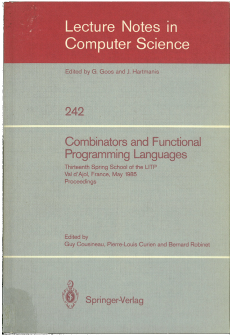
```
(def interleave (fn'
  () () → ()
| (,x . ,a) (,y . ,b) →
  (,x ,y . ,[apply interleave ‘(,a ,b)])
))
```

Doesn’t this sound too good to be true?

Write a parser, get a pretty-printer for free, or equivalently/bidirectionally:
Write a pretty-printer, get a parser for free.
The latter sounds even better.

Some books ...

found in Philips Research Laboratory Eindhoven (PRLE) library [CCR86, ST87] and in an *antique* book shop in Kiev [TG91]



Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

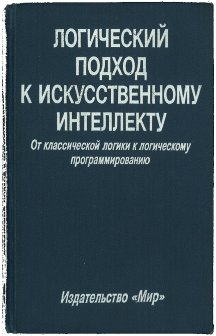
242

Combinators and Functional Programming Languages

Thirteenth Spring School of the LITP
Val d'Ajol, France, May 1985
Proceedings

Edited by
Guy Cousineau, Pierre-Louis Curien and Bernard Robinet

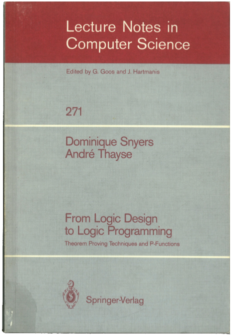
Springer-Verlag



ЛОГИЧЕСКИЙ ПОДХОД К ИСКУССТВЕННОМУ ИНТЕЛЛЕКТУ

От классической логики к логическому программированию

Издательство «Мир»



Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

271

Dominique Snyers
André Thayse

From Logic Design to Logic Programming

Theorem Proving Techniques and P-Functions

Springer-Verlag

Analogy in physics: Landauer's principle

> *"any logically irreversible manipulation of information, such as the erasure of a bit or the merging of two computation paths, must be accompanied by a corresponding entropy increase in non-information bearing degrees of freedom of the information processing apparatus or its environment." [Wikipedia]*

Fully reversible syntax ↔ semantics relation

Why purity?

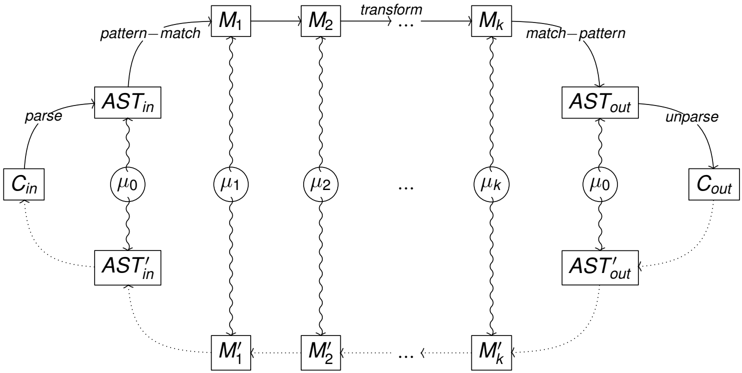
> *“If no information is erased, computation may in principle be achieved without the dissipation of heat, via a thermodynamically reversible process.” [Wikipedia]*

forwards	generating semantics from syntax (c.f., type-inference)
backwards	generating syntax from semantics (c.f. type-inhabitation)
validation	checking the correspondance between syntax and semantics (c.f., type-checking)
sideways	generating all possible syntax-semantics pairs (c.f., generation of typed terms)

Example (REPL)

```
(verify Expr (run* (q) (Expr '(2 ^ 2 * 1 + 3 * 5) '() q)) ===> (+ (* (^ 2 2) 1) (* 3 5)))
(verify Expr (run* (q) (Expr q '() '(+ (* 2 a) (* x 5)))) ===> (2 * a + x * 5))
(verify Expr (run* (q) (Expr '(1 * 3 + 5) '() '(+ (* 1 3) 5))) ===> _.0)
(verify Expr (parameterize ([*digits* '(42)] [*letters* '(#\x)])
 (run 7 (q) (fresh (x y) (Expr x '() y) (== q `(,x ,y))))) --->
 ((x + x) (+ x x))
 ((x) x)
 ((42 + x) (+ 42 x))
 ((x * x) (* x x))
 ((x ^ x) (^ x x))
 ((x + x * x) (+ x (* x x)))
 ((x - x) (- x x))
)
```

The dream: Modular DSLs



Important

① At each stage, the model (M_i) remains executable

② Each transformation $\mu_i, i > 0$ is its own inverse

③ some ad-hoc translation (C_{in}, C_{out}) at the ends is OK

④ some ad-hoc massaging (μ_0) of the Abstract Syntax Tree (AST) by pattern-maching is OK

Killer apps

Besides parsing, of course, which is much fun by itself...

Example (backwards)

Suppose at some stage a transformation detected an error. We need to present an *error message* and, preferably, a *debugger* to the user, both using the DSL understood by the user.

Example (forwards and backwards)

Suppose we have a non-deterministic transformation chain and the tool detected it is not profitable to follow the current path. We have to undo some until the branching point and start over.

Example (sideways)

Now suppose we have an editing system that maintains a notion of semantics in the background (i.e., AST, types). If the system can infer which terms fit with the “hole” being edited now (e.g., by using types) it can suggest a list of possible alternatives or auto-complete if there is only one possibility.

Wide area

Parsing:

- Attribute grammars (e.g., [Knu68, Knu90])
- Recursive descent (too many references here), e.g.,Packrat parsers [BU73] and Parsing Expression Grammar (PEG) formalism [For02, For04]
- PROLOG's Definite Clause Grammar (DCG) formalism introduced by Colmerauer & Kowalski, see [PW80], [BS08]
- Parser combinators (e.g., [RO10], [FH06, FHC08])

Term-Rewriting System (TRS) implementations:

- STRATEGO [Vis01] and others (RASCAL, ASF+SDF etc.)
- MAUDE [CDE$^+$07]

Bidirectional transformations:

- XML-related and Lenses, e.g., [FGM$^+$05]
- BOOMERANG, e.g., [BFP$^+$08]

Unfortunately

No proposal addresses reversibility "by nature"

Spoiler

This talk is mostly for *lazy* functional programmers ⇒ start by diving into declarative code and explain how we got there later

```
;; A BNF for a trivial expression grammar
<factor> ::= <factor> ^ <literal>
           | <literal>
<term> ::= <term> * <factor>
         | <term> / <factor>
         | <factor>
<expr> ::= <expr> + <term>
         | <expr> - <term>
         | <term>
```

```
;; An ideal DCG for the same ...
factor --> factor, [^], literal.
factor --> literal.
term --> term, [*], factor.
term --> term, [/], factor.
term --> factor.
expr --> expr, [+], term.
expr --> expr, [-], term.
expr --> term.
```

Let's assume Scheme for now...

① the lexer gives us Scheme tokens (viz., R5RS `read`)

- frees us from character munging in this talk
- solves the parenths matching, since (.) are very special
- can reuse native *(quasi-) quotation* ' ` and escapes ,@ ,
- for the rest Scheme tokens are very permissive

② thus, terminals are Scheme data (i.e., anything that is explicitly quoted) `TODO: self-quoted data`

③ non-terminals are Scheme *procedures*

Pure, declarative syntax (i.e., a recognizer)

```
(dcg Factor
    ([Factor] <=> [Factor] '^ [literal])
    ([Factor] <=> [literal])
)
(dcg Term
    ([Term] <=> [Term] '* [Factor])
    ([Term] <=> [Term] '/ [Factor])
    ([Term] <=> [Factor])
)
(dcg Expr
    ([Expr] <=> [Expr] '+ [Term])
    ([Expr] <=> [Expr] '- [Term])
    ([Expr] <=> [Term])
)
```

Yea

This is the 2nd declarative on Slide 2

① `dcg` can/should be a `syntax-rules` macro

② this example will diverge for plain DCG, PEG and LL

③ `dcg` should generate correct code that must *not* diverge

④ OK, putting nail-clippings aside...

How is this done in PROLOG?

In practice

Solved by *left-recursion elimination* via *left-factoring*

1. monadic state threaded by 2 extra arguments representing a difference list
2. user-level functor arguments can express context-sensitive grammars, as well as semantic actions

```
<fact>  ::= <lit > <fact>
<fact>  ::= ^ <lit > <fact>
          | ε
<term>  ::= <fact> <term'>
<term'>  ::= * <fact> <term'>
           | / <fact> <term >
           | ε
<expr>  ::= <term> <expr'>
<expr'>  ::= + <term> <expr'>
           | − <term> <expr'>
           | ε
```

```
factor(F)        --> literal(L), factor_r(L, F).
factor_r(T0, F) --> [^], literal(L),factor_r(exp(T0, L),F).
factor_r(F, F)   --> [].
term(T)          --> factor(F), term_r(F, T).
term_r(T0, T)    --> [*], factor(F), term_r(mul(T0, F), T).
term_r(T0, T)    --> [/], factor(F), term_r(div(T0, F), T).
term_r(T, T)     --> [].
expr(E)          --> term(T), expr_r(T, E).
expr_r(E0, E)    --> [+], term(T), expr_r(pls(E0,T), E).
expr_r(E0, E)    --> [−], term(T), expr_r(min(E0, T), E).
expr_r(E, E)     --> [].
```

Nay

Not nearly on the same declarativeness level as before...

Left-recursion: approaches

Avoidance:

- grammar becomes right-associative
- PEGs do not handle left-recursive rules [For04]

> *"Fortunately, a left-recursive grammar can always be rewritten into an equivalent right-recursive one, and the desired left-associative semantic behavior is easily reconstructed using higher-order functions as intermediate parser results." [For02]*

- on-line behavior
- reversible

Elimination by factoring:

- grammar remains left-associative
- on-line behavior
- needs inherited attributes
- so not reversible in practice

Curtailment [FH06, FHC08] or cancellation tokens (Nederhof & Koster, 1993)

- grammar remains left-associative
- sacrifices on-line behavior
- not reversible

Memoization tricks [WP07, WDM08, BS08]

- grammar remains left-associative
- on-line behavior
- not reversible

Left-recursion avoidance + higher-order patching

Bryan Ford's *non-solution* [For02] using higher-order synthesized attributes to cope with left-recursion in PEGs

```
(dcg Factor
  ([_ (π [λ (z) (y (if [null? z] x ‘(^ ,z ,x)))])]
   <=> [literal x] '^ [Factor y])
  ([_ (π [λ (z) (if [null? z] x ‘(^ ,z ,x))])]
   <=> [literal x])
)
```

① `dcg` introduces a (possibly recursive) grammar

② π maps to MINIKANREN's `project`

③ `project` reifies instantiated logical vars

④ free vars auto-lifted from clause heads by the _ keyword

Nay

This constructs a *huge* closure (c.f. `fold-left` via `fold-right`), reduced to the wanted value only after an *avalanche*, to be triggered from the outside.
Recursion is an effect, after all...

Left-recursion avoidance + declarative patching

```
(dcg factor locals: (x y z)
 ([factor ‘(^ ,x (* . (,y . ,z)))] <=> [literal x] '^ [factor ‘(^ ,y (* . ,z))])
 ([factor ‘(^ ,x (* ,y ,z))]       <=> [literal x] '^ [factor ‘(^ ,y ,z)])
 ([factor ‘(^ ,x ,y)]              <=> [literal x] '^ [factor y])
 ([factor x]                       <=> [literal x])
)
```

```
(dcg term locals: (x y z l)
 ([term ‘(* ,l . ,z)] <=[(pushdown x '* y l)]=> [factor x] '* [term ‘(* ,y . ,z)])
 ([term ‘(* ,x ,y)]   <=[(! sameops  '(*) y)]=> [factor x] '* [term y])
 ([term ‘(/ ,l . ,z)] <=[(pushdown x '/ y l)]=> [factor x] '/ [term ‘(/ ,y . ,z)])
 ([term ‘(/ ,x ,y)]   <=[(! sameops  '(/) y)]=> [factor x] '/ [term y])
 ([term x]                                <=> [factor x])
 )
```

```
(dcg expr locals: (x y)
 ([expr ‘(+ ,x . ,y)] <=> [term x] '+ [expr ‘(+ . ,y)])
 ([expr ‘(+ ,x ,y)]   <=> [term x] '+ [expr y])
 ([expr ‘(- ,x . ,y)] <=> [term x] '- [expr ‘(- . ,y)])
 ([expr ‘(- ,x ,y)]   <=> [term x] '- [expr y])
 ([expr x]            <=> [term x])
)
```

A hack

- this relies on variadic arithmetic operators of Scheme
- fails to respect the duality between syntax and semantics
- right-associativity leaks when run in reverse
- makes use of the Closed World Assumption (CWA), aka “negation as failure”

Left-recursion elimination + inherited attributes (cf. Slide 4)

```
(dcg
(factor locals: (x)
  ([_ y]   <=> [literal x] [factor' x y]))
(factor' locals: (y)
  ([_ x z] <=> '^ [literal y] [factor' `(^ ,x ,y) z])
  ([_ x x] <=> ϵ))
(term locals: (x)
  ([_ y]   <=> [factor x] [term' x y]))
(term' locals: (y)
  ([_ x z] <=> '* [factor y] [term' `(* ,x ,y) z])
  ([_ x z] <=> '/ [factor y] [term' `(/ ,x ,y) z])
  ([_ x x] <=> ϵ))
(expr locals: (x)
  ([_ y]   <=> [term x] [expr' x y]))
(expr' locals: (y)
  ([_ x z] <=> '+ [term y] [expr' `(+ ,x ,y) z])
  ([_ x z] <=> '- [term y] [expr' `(- ,x ,y) z])
  ([_ x x] <=> ϵ)
))
```

① non-terminal clauses can be variadic

② `dcg` can introduce a bunch of mutually recursive clauses

③ `locals:` declare (possibly) inherited attributes

④ *unification* in MINIKANREN ensures structural (`equal?`) rather than just numeric (`eqv?`) or pointer (`eq?`) equality

Pure, declarative syntax + semantics (i.e., a parser)

```
(defn Expr (dcg  <=> Expr
(Factor
  ([_ `(^  ,x ,y)] <=> [Factor x] '^ [literal y])
  ([_ x]           <=> [literal x]))
(Term
  ([_ `(*  ,x ,y)] <=> [Term x] '* [Factor y])
  ([_ `(/  ,x ,y)] <=> [Term x] '/ [Factor y])
  ([_ x]           <=> [Factor x]))
(Expr
  ([_ `(+  ,x ,y)] <=> [Expr x] '+ [Term y])
  ([_ `(-  ,x ,y)] <=> [Expr x] '- [Term y])
  ([_ x]           <=> [Term x])
)))
```

Some revelations

1. `dcg` can introduce encapsulated clauses
2. attributes are just like functor arguments in PROLOG
3. declarative binding style for syntesized attributes
4. logical vars are not reified by constructors (`list`, `cons`)
5. no leakage of monadic state (diff-lists) thanks to hygiene
6. direct recursion is by default prevented by `defn`

Which one do you prefer?

Obviously, we want the pure, declarative one:

- natural syntax (on the right)
- natural semantics (on the left)
- direct-style associativity and precedence
- inverse for free (mind the <=>)
- no fuzz, no noise

Looks like we're in trouble

Have to solve the left-recursion. Hang on.

Why R5RS? Because syntax rules!

homoiconicity+`syntax-rules`≡declarative compile-time TRS

- referentially transparent substitution semantics
- preserving hygiene (more on this later)
- normal-order (head-first) evaluation strategy
- sub-language decoupled from base Scheme
- pattern-matching syntax

Yea

This is the 1st declarative on Slide 2

However

- need to break hygiene sometimes (*anaphora*, `gensym`)
- Continuation Passing Style (CPS) for applicative order
- `syntax-rules` are not easily reversible

Some simple examples

```
(def-syntax λ lambda)
(def-syntax ⊥ (syntax-rules ()))
(def-syntax [id a] a)
(def-syntax [hd a . _] a)
(def-syntax [tl _ . b] b)
(def-syntax zip2 (syntax-rules ... ()
  ([_ (k ...)  ()       () . a] (k ... . a))
  ([_ k (x . xs) (y . ys) . a] (zip2 k xs ys (x y) . a))
 ))
(def-syntax revs (syntax-rules ()
  ([_           () () . r] r)
  ([_ (k args ...) () . r] (k args ... . r))
  ([_ k       (h . t) . r] (revs k t h . r))
))
```

```
;; A poor man's device to prevent recursion in the absence of types
;; Redefine f to ⊥ just for inside the body itself. Result is
;; that f can neither be reified as a first-class value, nor can
;; it be applied, since the expansion, ⊥ has no alternatives.
(def-syntax defn (syntax-rules ()
 ([_ (f . args) . body]                       ;; recursive functions
  (define f (λ args
     (let-syntax ([f ⊥])
        (begin . body)))
   ))
 ([_ f . exprs]                               ;; recursive CAFs
  (define f
     (let-syntax ([f ⊥])
        (begin . exprs))
   ))
))
```

Recursive macros

Extracting free variables from Scheme terms:

```
(def−syntax w (syntax−rules .. (qq quote unquote quasiquote unquote−splicing λ)
 ([_ q   (k ..) b [] . a] (k .. . a))
 ([_ q        k b ‘t . a] (w [qq . q] k b t . a))
 ([_ [qq . q] k b ,t . a] (w q k b t . a))
 ([_ []       k b ,t . a] (bad−unquote k b ,t))
 ([_ q        k b ’t . a] (w q k b [] . a))
 ([_ []  k b [λ (var ..) . body] . a] (w [] k (var .. . b) body . a))
 ([_ q  k b [t . ts] . a] (w q (w q k b t) b ts . a))
 ([_ [] k b t a ..]
      (symbol?? t
       (member?? t (a .. . b)
          (w [] k b [] a ..)
          (w [] k b [] a .. t))
          (w [] k b [] a ..)
    ))
 ([_ [qq . q] k b  t . a] (w q k b [] . a))
))
```

Removing one level of quasi-quotation:

```
(def−syntax qs (syntax−rules .. (qq quote unquote quasiquote unquote−splicing)
 ([_ q   (k ..) ()  . a] (k .. . a))
 ([_ []       k ‘y  . a] (qs [qq] k y . a))
 ([_ q        k ‘y  . a] (qs    q k () ’y . a))
 ([_ [qq]     k ,‘y . a] (qs [] k () [qq y] . a))
 ([_ [qq]     k ,y  . a] (qs [] k () y . a))
 ([_ []       k ,y  . a] (bad−unquote k ,y))
 ([_ q        k ’y  . a] (qs q k () ’y . a))
 ([_ [qq]     k (y . ys) . a] (qs [qq] (qs [qq] k y) ys . a))
 ([_ [qq . q] k  y  . a] (qs  q k ’y . a))
))
```

Reflection, breaking hygiene

Courtesy Al Petrofsky and [Kis02b, Kis02a]

```
(def-syntax [extract s body _k]
 (letrec-syntax ([tr (syntax-rules (s)
     ([_ x s tail (k sl . args)]
      (k (x . sl) . args))
     ([_ d (x . y) tail k]
      (tr x x (y . tail) k))
     ([_ d1 d2 () (k  sl . args)]
      (k (s . sl) . args))
     ([_ d1 d2 (x . y) k]
      (tr x x y k))
    )])
  (tr body body () _k)
))
(def-syntax extract* (syntax-rules ()
 ([_ (s) body k] (extract s body k))
 ([_ _ss _body _k]
  (letrec-syntax ([ex (syntax-rules ()
     ([_ fs () body k] (revs k fs))
     ([_ fs (s . ss) body k]
      (extract s body (ex fs ss body k)))
    )])
   (ex () _ss _body _k)))
))
```

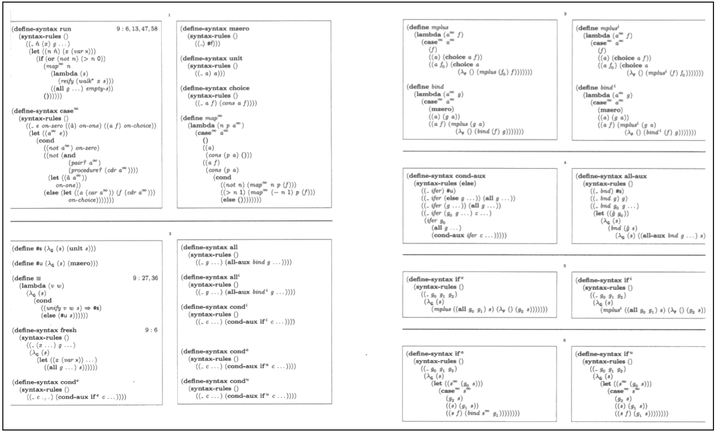
```
(def-syntax symbol?? (syntax-rules ()
  ([_ (x . y) kt kf] kf)
  ([_ #(x ...) kt kf] kf)
  ([_ maybe-symbol kt kf]
    (let-syntax ([test (syntax-rules ()
          ([_ maybe-symbol t f] t)
          ([_ x t f] f)
        )])
      (test abracadabra kt kf)
   ))
))

(def-syntax member?? (syntax-rules ()
  ([_ id () kt kf] kf)
  ([_ (id . ids) xs kt kf] kf)
  ([_ id (x . r) kt kf]
   (let-syntax ([test (syntax-rules (id)
          ([_ id t f] t)
          ([_ xx t f] f)
        )])
      (test x kt (member?? id r kt kf))
   ))
))
```

Enough reflection?

- we don't need the full power of Scheme
- avoiding impure `syntax-case` and unsafe List Processing (LISP) macros

Why MINIKANREN? Because of its compactness!

[FBK05]: “Connecting the wires”



1

```
(define-syntax run                                9 : 6, 13, 47, 58
  (syntax-rules ()
    ((_ n̂ (x) g ...)
     (let ((n n̂) (x (var x)))
       (if (or (not n) (> n 0))
         (map∞ n
           (lambda (s)
             (reify (walk* x s)))
           ((all g ...) empty-s))
         ())))))

(define-syntax case∞
  (syntax-rules ()
    ((_ e on-zero ((â) on-one) ((a f) on-choice))
     (let ((a∞ e))
       (cond
         ((not a∞) on-zero)
         ((not (and
                 (pair? a∞)
                 (procedure? (cdr a∞))))
          (let ((â a∞))
            on-one))
         (else (let ((a (car a∞)) (f (cdr a∞)))
                 on-choice)))))))
```

```
(define-syntax mzero
  (syntax-rules ()
    ((_) #f)))

(define-syntax unit
  (syntax-rules ()
    ((_ a) a)))

(define-syntax choice
  (syntax-rules ()
    ((_ a f) (cons a f))))

(define map∞
  (lambda (n p a∞)
    (case∞ a∞
      ()
      ((a)
       (cons (p a) ()))
      ((a f)
       (cons (p a)
         (cond
           ((not n) (map∞ n p (f)))
           ((> n 1) (map∞ (− n 1) p (f)))
           (else ())))))))
```

2

```
(define #s (λ_G (s) (unit s)))

(define #u (λ_G (s) (mzero)))

(define ≡                                         9 : 27, 36
  (lambda (v w)
    (λ_G (s)
      (cond
        ((unify v w s) ⇒ #s)
        (else (#u s))))))

(define-syntax fresh                              9 : 6
  (syntax-rules ()
    ((_ (x ...) g ...)
     (λ_G (s)
       (let ((x (var x)) ...)
         ((all g ...) s))))))

(define-syntax condᵉ
  (syntax-rules ()
    ((_ c .,.) (cond-aux ifᵉ c ...))))
```

```
(define-syntax all
  (syntax-rules ()
    ((_ g ...) (all-aux bind g ...))))

(define-syntax allⁱ
  (syntax-rules ()
    ((_ g ...) (all-aux bindⁱ g ...))))

(define-syntax condⁱ
  (syntax-rules ()
    ((_ c ...) (cond-aux ifⁱ c ...))))

(define-syntax condᵃ
  (syntax-rules ()
    ((_ c ...) (cond-aux ifᵃ c ...))))

(define-syntax condᵘ
  (syntax-rules ()
    ((_ c ...) (cond-aux ifᵘ c ...))))
```

3

```
(define mplus
  (lambda (a∞ f)
    (case∞ a∞
      (f)
      ((a) (choice a f))
      ((a f₀) (choice a
                (λ_F () (mplus (f₀) f)))))))

(define bind
  (lambda (a∞ g)
    (case∞ a∞
      (mzero)
      ((a) (g a))
      ((a f) (mplus (g a)
               (λ_F () (bind (f) g)))))))
```

```
(define mplusⁱ
  (lambda (a∞ f)
    (case∞ a∞
      (f)
      ((a) (choice a f))
      ((a f₀) (choice a
                (λ_F () (mplusⁱ (f) f₀)))))))

(define bindⁱ
  (lambda (a∞ g)
    (case∞ a∞
      (mzero)
      ((a) (g a))
      ((a f) (mplusⁱ (g a)
               (λ_F () (bindⁱ (f) g)))))))
```

4

```
(define-syntax cond-aux
  (syntax-rules (else)
    ((_ ifer) #u)
    ((_ ifer (else g ...)) (all g ...))
    ((_ ifer (g ...)) (all g ...))
    ((_ ifer (g₀ g ...) c ...)
     (ifer g₀
       (all g ...)
       (cond-aux ifer c ...)))))
```

```
(define-syntax all-aux
  (syntax-rules ()
    ((_ bnd) #s)
    ((_ bnd g) g)
    ((_ bnd g₀ g ...)
     (let ((ĝ g₀))
       (λ_G (s)
         (bnd (ĝ s)
           (λ_G (s) ((all-aux bnd g ...) s)
```

5

```
(define-syntax ifᵉ
  (syntax-rules ()
    ((_ g₀ g₁ g₂)
     (λ_G (s)
       (mplus ((all g₀ g₁) s) (λ_F () (g₂ s)))))))
```

```
(define-syntax ifⁱ
  (syntax-rules ()
    ((_ g₀ g₁ g₂)
     (λ_G (s)
       (mplusⁱ ((all g₀ g₁) s) (λ_F () (g₂ s))
```

6

```
(define-syntax ifᵃ
  (syntax-rules ()
    ((_ g₀ g₁ g₂)
     (λ_G (s)
       (let ((s∞ (g₀ s)))
         (case∞ s∞
           (g₂ s)
           ((s) (g₁ s))
           ((s f) (bind s∞ g₁))))))))
```

```
(define-syntax ifᵘ
  (syntax-rules ()
    ((_ g₀ g₁ g₂)
     (λ_G (s)
       (let ((s∞ (g₀ s)))
         (case∞ s∞
           (g₂ s)
           ((s) (g₁ s))
           ((s f) (g₁ s))))))))
```

- runs in the Bigloo interpreter via `alexpander`
- can be compiled to native (via C)
- can be compiled to run on the Java Virtual Machine (JVM)
- can work in the browser through Javascript (JS)

MINIKANREN is the only system fitting this slide that I know of...

Example (pure MINIKANREN)

```
(set-sharp-read-syntax! 's succeed)
(set-sharp-read-syntax! 'u fail)
(def (nullo? x) [≡ x '()])
(def (pairo? x) (fresh (x0 x1) [≡ x `(,x0 . ,x1)]))
(def (caro x y) (fresh (t) [≡ x `(,y . ,t)]))
(def (cdro x y) (fresh (h) [≡ x `(,h . ,y)]))
(def (conso h t l) (≡ l `(,h . ,t)))
(def take-from
   (λ () _ => #u
    | `(,head . ,tail) f =>
      (conde
       ([≡ f head])
       (else (take-from tail f)))
    | db _ => (error 'take-from "bad database" db)))
(def *digits* (make-parameter (list-tabulate 10 values)))      ;; dynamic binding
(def *letters* (make-parameter                                 ;; dynamic binding
  (unfold [_ char>? #\z]                                       ;; just for the purpose
     values                                                    ;; of exposition
     (∘ integer->char
        [_ + 1]
        char->integer)
     #\a)))
(def (numbers? x) (take-from [*digits*] x))
(def (symbols? x) (take-from
     (map (∘ string->symbol
             list->string
             list)
          [*letters*]) x))
(def (! p . args)                                              ;; that is where
   (condu                                                      ;; real impurity is
      ([apply p args] #u)
      (else #s)))
```

... we shall avoid using *impure* MINIKANREN (`project`, `conda` and `condu` that dissipate information) for *pure* applications

Why DCGs? Because of declarative semantics!

Yea

This is the 3rd declarative on Slide 2

DCG formalism of PROLOG has supplanted the Augmented Transition Network (ATN) from the LISP world [PW80]

1. declarative semantics
2. reversibility “by nature”
3. built-in support for non-determinism (resolution)
4. incremental instantiation (aka delaying by unification ≡ “spooky action at distance” from quantum mechanics)

Last but not least

- easy, *macro-expressible* [Fel91] translation to diff-lists
- it fits naturally with both R5RS and MINIKANREN
- the only difference to PROLOG is the order of args

```
literal(x) --> symbol(x).
literal(x) --> [x],
    {(number(x)
     ;var(x))
    ,between(0,9,x)}
```

```
(def (number Lin Lout x) (all (numbers? x) (consᵒ x Lout Lin)))
(def (symbol Lin Lout x) (all (symbols? x) (consᵒ x Lout Lin)))
(def (literal Lin Lout x)
   (conde ([symbol Lin Lout x])
          ([number Lin Lout x])))
```

Problem

What was our mission again?

- When, exactly, is a grammar “carefully designed”?
- How, exactly, can a parser run *backwards*?

Maybe this is reversibility “by construction” (as in [RO10])?

① only reversible compositions (isomorphisms)

② of reversible building blocks (bijections)

No, its even better: reversibility “by nature” and by default

Lets start by making some fresh vars...

```
(def-syntax make-fresh (syntax-rules ()
  ([_ () head . body] (head . body))
  ([_ vars _ . body] (fresh vars . body))
))
```

Let’s address the “how” first...

implementing multi-rule clauses (DCG excerpts)

```
([_ head heads: heads locals: locals condo: condo . rules]
  [define head (λ (Lin Lout . result)
  (letrec—syntax
   ([p (syntax—rules (<=> <= =>)
      ([p k acc] (revs (k) acc))                          ;; implementing pure clauses
      ([p k acc ((x args ...) <=> . goals) . rest]
        (p k (((all [== result ‘(,args ...)]
                    (seq Lin Lout k () () [heads acc] . goals)
                )) . acc) . rest))                        ;; implementing logical ‘‘effects’’
      ([p k acc ((x args ...) <=[actions ...]=> . goals) . rest]
       (p k ([(all [== result ‘(,args ...)]
                (project (Lin)
                 (if [ground? Lin]
                     #s
                     (begin actions ...)))
                (seq Lin Lout k () () [heads acc] . goals)
                (project (Lin)
                 (if [ground? Lin]
                     (begin actions ...)
                     #s))
        )] . acc) . rest)))])
     (make—fresh locals begin
       (p condo () . rules)
     ))
  )])
```

Declare recursive clauses that are mutually aware (slide 4)

```
([_ (head . args) ..]
 (let—syntax—rule ([k . heads]
    (begin (dcg head heads: (rev: . heads) . args) ..))
    (k head ..)
   ))
```

Encapsulate clauses and make them mutually aware (slide 1)

```
([_ <=> start (head . args) ..] (dcg start [] (rev: head ..) => (head . args) ..))
([_ => start (head . args) ..] (dcg start () (ref: head ..) => (head . args) ..))
([_ <= start (head . args) ..] (dcg start () (reb: head ..) => (head . args) ..))
([_ <=: start (head . args) ..] (dcg start () (reu: head ..) => (head . args) ..))
([_ start [acc ..] ach =>] (let () acc .. start))
([_ start acc ach => [head . args] . rest]
 (dcg start ((dcg head heads: ach . args) . acc) ach => . rest))
```

implementing rules (seq macro excerpts)

Threading monadic state around

1. pure `gensym`, as `alexpander` is not aware of the MINIKANREN's `fresh` construct and `syntax-rules` are expanded head-first
2. sparing nested `fresh` intros (see next slide)

Handling {escapes}, ϵ, terminals and quasi-data

```
([_ in out c acc ts hs do(as ...) . rest] (seq in out c (as ... . acc) ts hs . rest))
([_ in out c acc tmps heads ϵ . rest] (seq in out c ([== in out] . acc) tmps heads . rest))
([_ in out c acc tmps heads (quote datum) . rest]
 (let ([temp #FALSE])                                ;; just to generate a new temporary
   (seq temp out c ([== in `(datum . ,temp)] . acc)
       (temp . tmps) heads . rest)
  ))
([_ in out c acc tmps heads (qq dat) . rest] (seq in out c acc tmps heads `dat . rest))
([_ in out c acc tmps [hs (ac ...)] (quasiquote datum) . rest]
  (let ([temp #FALSE][data #FALSE])                  ;; just to generate new temporaries
    (seq temp out c ((qs [] (seq data '() c () () [hs (ac ... . acc)]) (quasiquote datum))
                    [== in `(,data . ,temp)] . acc)
        (temp data . tmps) [hs (ac ...)] . rest)
   ))
```

Handling sequencing, non-terminals

```
([_ in out c acc temps [heads (ac ...)] (: goals ...) . rest]
  (let ([temp #FALSE])                               ;; just to generate a new temporary
    (seq temp out c ((all (seq in temp c () () [heads (ac ... . acc)] goals ...)) . acc)
        (temp . temps) [heads (ac ...)] . rest)
   ))
([_ in out c acc temps heads (goal . args) . rest]
  (let ([temp #FALSE])                               ;; just to generate a new temporary
    (seq temp out c ([goal in temp . args] . acc)
        (temp . temps) heads . rest)
   ))
```

Finalization, and optimizations for singular rules

```
([_ in out _ acc ts hs do(as ...)] (revs (make−fresh ts begin) (as ... . acc)))
([_ in out c acc ts hs ϵ] (seq in out c acc ts hs do[(== in out)]))
([_ in out c acc ts hs (quote d)] (seq in out c acc ts hs do[(== in ‘(d . ,out))]))
([_ in out c acc ts [heads (ac ...)] (quasiquote datum)]
  (let ([data #FALSE])                                ;; just to generate a new temporary
    (seq in out c acc (data . ts) [heads (ac ... . acc)]
     do[(qs [] (seq data ’() c () () [heads (ac ... . acc)]) (quasiquote datum))
        (== in ‘(,data . ,out))])
  ))
([_ in out c acc temps [heads (ac ...)] (: goals ...)]
  (seq in out c acc temps [heads (ac ...)]
   do[(all (seq in out c () () [heads (ac ... . acc)] goals ...))]
  ))
([_ in out c acc temps heads (goal . args)]
  (seq in out c acc temps heads do[(goal in out . args)]))
```

Example (Wasn’t too difficult was it?)

```
(dcg O
   ([O] <=> ’∗)
   ([O] <=> ’/))
(dcg O1..3
   ([O1..3] <=> [O])
   ([O1..3] <=> [O] ’∘ [O])
   ([O1..3] <=> [O] ’∘ [O] ’∘ [O]))
(dcg S
 ([S ’z] <=> ϵ)
 ([_ ‘(S ,x)] <=> (: ’a ’a [S x])))
```

Oops it still is

```
(dcg SS
 ([SS ’z] <=> ϵ)
 ([_ ‘(S ,x)] <=> [SS x] ’a ’a))
```

Solving left-recursion, lazily

Let’s be logically lazy

Enter `append°` , the “swiss army knife” of logic programming.

```
append([],L,L).
append([X | A1],B,[X | C1]) :— append(A1,B,C1).
```

or, with MINIKANREN [Byr10]

```
(def (append° a b c)
   (conde
    ([≡ a '()] (≡ b c))
    (else (fresh (x a1 c1)
       (≡ a `(,x . ,a1))
       (≡ c `(,x . ,c1))
       (append° a1 b c1)))
    ))
```

Example (append° is fully reversible)

```
(verify A (run* (q) (append° q '(3) '(1 2 3))) ===> (1 2))
(verify A (run* (q) (append° '(1 2) q '(1 2 3))) ===> (3))
(verify A (length (run* (q) (fresh (x y) (append° x y '(1 2 3)) (== q `(,x ,y))))) = 4)
(verify A (run* (q) (fresh (x y) (append° '(1 2) x y) (== q `(,x ,y))))
 ===> (_.0 (1 2 . _.0)))
(verify A (run 2 (q) (fresh (x y) (append° x '(3) y) (== q `(,x ,y))))
 ——> (() (3)) ((_.0) (_.0 3)))
```

Introduction
Vision
Why we want this?
Related work
Motivation
What we do not want
What we do want
Background
Why, why, why ...
Problem statement
Solution
Solving left-recursion
Meta-syntactic sugar
Conclusion
Appendix
Acronyms
References

Now, what if...

① we have (possibly) recursive clause heads (see slide 2)

② lets lift the calls to such procedures in the `seq` macro

③ *untie* the knot for those by inserting `append`$^{\circ}$ as dummy

④ delay the resolution of the unlifted goals until the very end

⑤ *tie* the knot by unifying difference list components

```
([_ in out c acc temps heads (unlift goal . args)]                ;; tie the knot
  (seq in out c acc temps heads do[(goal . args) (== in out)]))
([_ in out c acc temps heads (lift goal . args)]                  ;; already the ''end''
  (seq in out c acc temps heads do[(goal in out . args)]))
([_ in out c acc temps heads (unlift goal . args) . rest]
  (seq in out c ([goal . args] . acc) temps heads . rest))
([_ in out c acc temps [(ref: . heads) ac] (lift goal . args) rest ...];; untie the knot
  (let ([temp #FALSE][data #FALSE])                   ;; just to generate new temporaries
    (seq temp out c ([append° data temp in] . acc)
         (temp data . temps) [(ref: . heads) ac] rest ...
      (unlift goal data '() . args))
  ))
```

Yea

- “solves” left-recursion
- this often makes the parser also tail-recursive ⇒ linear parsers
- a form of predictiveness (only *possible* `data` shall be considered)

But...

not reversible, since when running backwards (the input is *unknown*, but the result is), the recursive must be called before input unparsing.

Left-recursion and reversibility

Now lets use the trick from slide 2

```
([_ in out c acc temps heads (lift goal . args) rest ...]
  (let ([temp #FALSE][data #FALSE])                  ;; just to generate new temporaries
    (seq temp out c ([append^O data temp in]
                       (project (in)
                          (if [ground? in]
                              #s
                              (goal data '() . args)))
                         . acc)
        (temp data . temps) heads rest ...
             (unlift project (in)
                 (if [ground? in]
                     (goal data '() . args)
                     #s
       )))))
```

Example (Yea!)

slide 1 + grammar from slide 1 finally work together

Looks like mission accomplished

1. declarative syntax ↔ semantics relations
2. handling left-recursion
3. pure, on-line behavior
4. fully reversible execution model

On-line?

```
(def (fresh^o x)
  (conde ([== x '()])
         (else (fresh (y z)
                 (fresh^o z)
                 (== x `(,y . ,z))
      ))))
(def (prefix^o a b)
   (fresh (x)
      (fresh^o x)
      (append^o a x b)
      ))
```

Example (Infinite input stream)

```
;; testing prefix^o
(verify fresh^o (run 4 (q) (fresh^o q))
 ---> (_.0 _.1 _.2) (_.0 _.1) (_.0) ())
(verify prefix^o (run 4 (q) (prefix^o '(1 2 3) q))
 ---> (1 2 3) (1 2 3 _.0) (1 2 3 _.0 _.1) (1 2 3 _.0 _.1 _.2))
;; testing infinitary parsing
(verify SS (run 3 (q) (fresh (l) (prefix^o '() l) (SS l `() q)))
 ---> z (S z) (S (S z)))
(verify SS (run 2 (q) (fresh (l) (prefix^o '(a a) l) (SS l `() q)))
 ---> (S (S z)) (S z))
(verify SS (run 2 (q) (fresh (l) (prefix^o '(a a a a) l) (SS l `() q)))
 ---> (S (S (S z))) (S (S z)))
(verify SS (run 2 (q) (fresh (l) (prefix^o '(a a a a a a) l) (SS l `() q)))
 ---> (S (S (S (S z)))) (S (S (S z))))
```

As observed, the infinite stream of fresh vars can be instantiated as many times as needed for the parser to succeed...

Lets make reversibility the default

Now, can we auto-lift left-recursive clauses?

Yea, we can ⇒ frees us from annotating `dcg` rules

```
([_ in out c () temps [(r . heads) ()] (goal . args)]
  (member?? goal heads
    (seq in out c () temps [(r . heads) ()] (lift goal . args))
    (seq in out c () temps [(r . heads) ()] do[(goal in out . args)])
))
([_ in out c () temps [(r . heads) acc] (goal . args) . rest]
  (member?? goal heads
    (seq in out c () temps [(r . heads) acc] (lift goal . args) . rest)
      (let ([temp #FALSE]) ; just to generate a new temporary
        (seq temp out c ([goal in temp . args])
            (temp . temps) [(r . heads) acc] . rest)
   )))
```

But, if the user wants to diverge, or if a rule has only one (recursive) sub-goal, or if they want to express “degenerate loops that are actually unreachable” [For04]:

```
([_ in out c () temps [(reu: . heads) ()] (lift goal . args)]          ;; just diverge
  (seq in out c () temps [(reu: . heads) ()] do[(goal in out . args)]))
([_ in out c acc temps [(reu: . heads) ac] (lift goal . args) . rest]   ;; just diverge
  (let ([temp #FALSE])                            ;; just to generate a new temporary
    (seq temp out c ([goal in temp . args] . acc)
        (temp . temps) [(reu: . heads) ac] . rest)))
([_ in out c () temps [(x . heads) ()] (lift goal . args)]    ;; recursive singleton goal
  (seq in out c () temps [(x . heads) ()] do[#u (== in out)]))
([_ in out c acc temps [(reb: . heads) ac] (lift goal . args) . rest] ;; degenerate loop
  (let ([temp #FALSE])                            ;; just to generate a new temporary
    (seq temp out c (#u [== in temp] . acc)
        (temp . temps) [(reb: . heads) ac] . rest)))
```

Sort of a CAP theorem

Theorem (A conjecture, really, about relational parsing)

only 2 of the following 3 properties can hold simultaneously:

reversible	*the ability to run forwards and backwards*
complete	*the ability to terminate on finite input or output*
generative	*the ability to run sideways, i.e., enumerate all possible input-output pairs (for Chomsky type-0's)*

① reversibility+completeness: we use `dcg <=>` rules for this (expanding to `rev:` annotation)

② reversibility+generativity: we use `dcg =>` rules for this (expanding to `ref:` annotation)

③ reversibility+completeness−degenerates: we use `dcg <=` rules for this (expanding to `reb:` annotation)

④ reversibility+completeness−degenerates−safety: we use `dcg <=:` rules for this (expanding to `reu:` annotation)

⑤ completeness+generativity: not interesting

> *"An atomic PEG consists of: any terminal symbol, any nonterminal symbol, or the empty string ϵ. Given any existing PEGs e, e1, and e2, a new PEG can be constructed using the following operators: (1) sequence: e1 e2, (2) ordered choice: e1 / e2, (3) zero-or-more: e*, (4) one-or-more: e+, (5) optional: e?, (6) and-predicate: &e, (7) not-predicate: !e." [Wikipedia]*

Atomic PEGs and sequencing we already had

Now lets add the rest (skipping some details now)

```
(def-syntax proc-/ (syntax-rules (/)
  ([_ in out c (k ...) heads] (k ... #u))
  ([_ in out c (k ...) heads a] (k ... ((seq in out c () () heads a))))
  ([_ in out c (k ...) heads a / . as] (proc-/ in out c (k ... ((seq in out c () () heads a))) heads . as))
))
```

```
([_ in out c acc temps heads (alt / . alts)]
  (seq in out c acc temps heads do[(proc-/ in out c (c) heads alt / . alts)]))
([_ in out c acc temps [heads (ac ...)] (goals ... *)]
  (seq in out c acc temps [heads (ac ...)]
    do[(let loop ([lin in][lout out])
         (c ([== lin lout])
            ((let ([temp #FALSE])                         ;; just to generate a new temporary
                (fresh (temp)
                  (seq lin temp c () () [heads (ac ... . acc)] goals ...)
                  (loop temp lout)))))))]))
([_ in out c acc temps [heads (ac ...)] (goals ... +)]
  (seq in out c acc temps [heads (ac ...)]
    do[(let loop ([lin in][lout out])
         (let ([temp #FALSE])                             ;; just to generate a new temporary
              (fresh (temp)
                (seq lin temp c () () [heads (ac ... . acc)] goals ...)
                (c ([== temp lout])
                   ([loop temp lout])))))]))
([_ in out c acc temps [heads (ac ...)] (goals ... ?)]
  (seq in out c acc temps [heads (ac ...)]
    do[(c ((seq in out c () () [heads (ac ... . acc)] goals ...)) ([== in out]))]))
([_ in out c acc temps [heads (ac ...)] when guards]
  (seq in out c acc temps [heads (ac ...)]
    do[(fresh (temp) (c ((seq in temp c () () [heads (ac ... . acc)] . guards) #s) (else #u)))]))
([_ in out c acc temps [heads (ac ...)] unless guards]
  (seq in out c acc temps [heads (ac ...)]
    do[(fresh (temp) (c ((seq in temp c () () [heads (ac ... . acc)] . guards) #u) (else #s)))]))
```

Huh?

And what about ordered choice, you might ask. *Another effect, if you ask me...*

Committed choice via PEG macro

Delegates to the `DCG` macro, replacing `conde` by `condu`.

```
(def-syntax peg (syntax-rules (<= <=> =>)
  ([_ <=> start (head . args) ...]
   (dcg <=> start (head condo: condu . args) ...))
  ([_ => start (head . args) ...]
   (dcg => start (head condo: condu . args) ...))
  ([_ <= start (head . args) ...]
   (dcg <= start (head condo: condu . args) ...))
  ([_ <=: start (head . args) ...]
   (dcg <=: start (head condo: condu . args) ...))
  ([_ (head . args) ...]
   (dcg (head condo: condu . args) ...))
  ([_ head . args]
   (dcg head condo: condu . args))
))
```

Example (Dangling else)

```
(defn ife (peg <=> if
(if
 ([_ `(if ,x ,y ,z)] <=> 'if [Expr x] 'then [Expr y] 'else [Expr z])
 ([_ `(if ,x ,y #f)] <=> 'if [Expr x] 'then [Expr y])
 ([_ `(if ,x ,y ,z)] <=> 'if [Expr x] 'then [if y] 'else [Expr z])
 ([_ `(if ,x ,y #f)] <=> 'if [Expr x] 'then [if y])
)))
(verify ife.nest (run* (q) (ife '(if 1 then if 2 then 3 else 4 else 5) '() q))
 ===> (if 1 (if 2 3 4) 5))
(verify ife.dangling (run* (q) (ife '(if 1 then if 2 then 3 else 4) '() q))
 ===> (if 1 (if 2 3 4) #FALSE))
```

Returning to slide 2...

And what about [], [_] and [π] magic in clause heads?

Lets break hygiene for fun and profit (look ma, no `gensym`)

```
(define scheme-bindings (syntax-rules .. ()
 ([_ (k a b [s ..] . d)]
  (k a b [s .. list first second pair? car cdr null? if cond begin + - * /] . d))
))
```

```
([p k acc ([] <=> . goals) . rest] ;; the user wants to get all results as a single list
  (p k ([(fresh (results)
           (appendo results Lout Lin)
           (== result `(,results))
           (seq Lin Lout k () () [heads acc] . goals))] . acc) . rest))
([p k acc ([_] <=> . goals) . rest]      ;; the user wants to get each result separately
  (p k ([(all (appendo result Lout Lin)
              (seq Lin Lout k () () [heads acc] . goals))] . acc) . rest))
                  ;; the user wants to use higher-order and infer synthesized attributes
([p k acc ([_ (π args ...)] <=> . goals) . rest]
  (let-syntax-rule ([K . vars]                                  ;; collect the free vars
  (let-syntax-rule ([K vars pats terms]                            ;; use extracted vars
    (make-fresh vars all
      (seq Lin Lout k () () [heads acc] . terms)
      (project vars [== result pats])
     ))
     (extract* vars (args ... . goals) (K () `(,args ...) goals))
   ))
   (p k (([scheme-bindings (w [] (K) [] (args ...))]) . acc) . rest)))
                                  ;; the user wants to just infer synthesized attributes
([p k acc ([_ args ...] <=> . goals) . rest]
  (let-syntax-rule ([K . vars]                                  ;; collect the free vars
  (let-syntax-rule ([K vars pats terms]                            ;; use extracted vars
    (make-fresh vars all
       [== result pats]
       (seq Lin Lout k () () [heads acc] . terms)
     ))
     (extract* vars (args ... . goals) (K () `(,args ...) goals))
    ))
    (p k (([scheme-bindings (w [] (K) [] (args ...))]) . acc) . rest)
))
```

Returning to slide 1...

And what if (goals ... *) constructs do binding?

We have to collect the results in a (possibly, empty) list. *More pure hygiene breaking* because we need 4 but have only 1 binding to start with...

```
([_ in out c acc temps [(r . heads) (ac ...)] (goals ... *)]
 (let-syntax ([K (syntax-rules .. ()
  ([_ in out vars ..]     ;; we need to explicitly get [[in]] and [[out]] from the caller
   (let loop ([lin in][lout out] [vars '()] ..)                                ;; 1st bunch
      (let-syntax ([K (syntax-rules ... ()
         ([_ res ...]                                                          ;; 2nd bunch
          (let ([res #FALSE] ...)
            (make-fresh (res ...) begin
             (letrec-syntax ([K (syntax-rules .... ()
               ([_ gls (v v1 v2 v3) ....]           ;; and now declare the 3rd bunch of vars
                             ;; and substitute it for the original var in [[gls]]==[[goals]]
               (c ([== lin lout]
                    (== v1 v) ....)
                   ((let ([temp #FALSE][v3 #FALSE] ....)
                    (fresh (temp v3 ....)           ;; rename original var to a local temporary
                        (let-syntax ([v v3] ....)
                          (seq lin temp c () () [(r . heads) (ac ... . acc)] . gls))
                        (appendo v1 `(,v3) v2) ....
                        [loop temp lout v2 ....])))))))]
               [K1 (syntax-rules ()
                ([_ var gls . args]                                 ;; zip all the vars together
                 (zip4 (K gls) var . args)
                ))]
               [K0 (syntax-rules ()                  ;; 3rd bunch
                ([_ . vs]                            ;; extract the 4rd bunch of free-vars
                                                     ;; now with the same colour as in [[goals]]
                 (extract* vs (goals ...)
                   (K1 [] (goals ...) (vars ..) (res ...) vs)
                 )))])                                   ;; retrieve the 3rd bunch of free-vars

                (scheme-bindings (w [] (K0) heads (goals ...)))
                ))))])                                   ;; retrieve the 2nd bunch of free-vars
             (scheme-bindings (w [] (K) heads (goals ...)))
          ))))])                            ;; retrieve the 1st bunch of free-vars
                                            ;; we need to pass [[in]] and [[out]] as they would
                                            ;; otherwise be renamed
         (seq in out c acc temps [(r . heads) (ac ...)]
           do[(scheme-bindings (w [] (K in out) heads (goals ...)))])
         ))
```

It's like 4 stage rocket piercing levels of abstraction

Still no `gensym` in sight...
I shall spare you the details of `(goals ... +)`, but its very similar.

Final words

Example (assorted)

```
;; higher-order rules
(defn [R p] (dcg <=> c
 (c ([_ `(,x . ,y)] <=> [c y] 'comma [p x])
    ([_ `(,x)]      <=> [p x]))))
;; regular grammars
(dcg A ([] <=> '< ('a *) '>))
(dcg B ([] <=> (('a / 'b) +)))
(dcg C ([] <=> '< (('a / 'b / 'c) +) '>))
(dcg O1..3 ([O1..3] <=> [O] ((: 'o [O] ((: 'o [O]) / ϵ)) / ϵ)))
;; Dyck language
(dcg D ([_] <=> 'x)
       ([_ `(D ,x)] <=> `(,[D x])))
;; context-free grammar a^nb^n
(dcg a^nb^n (['cfg] <=> 'a ([a^nb^n] ?) 'b))
;; context-free, non-packrat grammar
(defn s (dcg <=> S
 (S ([_] <=> 'x)
    ([_ `(s ,x)] <=> 'x [S x] 'x))))
;; non context-free, packrat grammar
(defn a^nb^nc^n (dcg <=> S
  (S ([S] <=> when([A] 'c) ('a +) [B] unless(['a / 'b / 'c])))
  (A ([A] <=> 'a ([A] ?) 'b))
  (B ([B] <=> 'b ([B] ?) 'c))
  ))
;; bastardized λ-calulus
(defn Λ (dcg <=> S
  (S ([_ x] <=> ([L x] / [A x] / [T x]))))
  (L ([_ `(λ (,x) ,y)] <=> 'λ [T x] '· [S y]))
  (A ([_ x] <=> `(! ,([S x] +)))
     ([_ x] <=> '! ([S x] +)))
  (T ([_ x] <=> [symbol x]))
  ))
```

Conclusions

Code: `https://github.com/kourzanov/purecube`

We've addressed the "how"

So what does it mean for a grammar to be carefully designed?

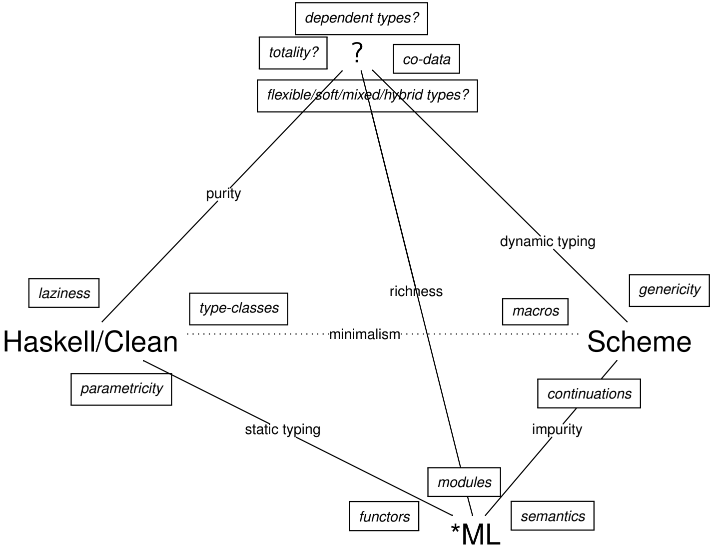
- use physics (Landauer, and quantum mechanics)!
- be more declarative
  - avoid inherited attributes
  - faithfully represent semantics (stuff on the left)
  - faithfully represent syntax (stuff on the right)
- use the syntactic sugar
- but: avoid impure operators (committed choice)

Its interesting to see that in original DCGs, `append` was used to link each sub-goal in a rule. Threading was not used.

Future work

extensibility	(ala *CoCoCo* [KSV14])
efficiency	(downscale it to do character-based parsing)
memoization	for efficient incremental parsing
prove	the conjecture on slide 6

Grand unified theory of Functional Programming (FP)?



dependent types?

totality?

?

co-data

flexible/soft/mixed/hybrid types?

purity

dynamic typing

richness

laziness

type-classes

macros

genericity

Haskell/Clean

minimalism

Scheme

parametricity

continuations

static typing

impurity

modules

functors

semantics

*ML

6.

The Fun Never Ends...

AST	Abstract Syntax Tree
ATN	Augmented Transition Network
CPS	Continuation Passing Style
CWA	Closed World Assumption
DCG	Definite Clause Grammar
DCGs	Definite Clause Grammars
DSL	Domain-Specific Language
DSLs	Domain-Specific Languages
FP	Functional Programming
JS	Javascript
JVM	Java Virtual Machine
LISP	List Processing
ML	Meta Language
NXP	Next Experience Semiconductors
PDS	Parallel & Distributed Systems
PEG	Parsing Expression Grammar
PEGs	Parsing Expression Grammars
PRLE	Philips Research Laboratory Eindhoven

R5RS	Revised^5 Report on the Algorithmic Language Scheme
TRS	Term-Rewriting System
TUD	TU Delft
XML	Extensible Markup Language

N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson.
Revised^5 report on the algorithmic language scheme.
SIGPLAN Not., 33(9):26–76, September 1998.
URL: `http://doi.acm.org/10.1145/290229.290234`, `doi:10.1145/290229.290234`.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt.
Boomerang: resourceful lenses for string data.
In George C. Necula and Philip Wadler, editors, POPL, pages 407–419. ACM, 2008.

Ralph Becket and Zoltan Somogyi.
Dcgs + memoing = packrat parsing but is it worth it?
In Hudak and Warren [HW08], pages 182–196.

Alexander Birman and Jeffrey D. Ullman.
Parsing algorithms with backtrack.
Information and Control, 23(1):1–34, 1973.

William E Byrd.
Relational programming in minikanren: techniques, applications, and implementations.
2010.

Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors.
Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985, Proceedings, volume 242 of Lecture Notes in Computer Science. Springer, 1986.

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott.
All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic.
Springer-Verlag, Berlin, Heidelberg, 2007.

D.P. Friedman, W.E. Byrd, and O. Kiselyov.
Reasoned Schemer.
MIT Press, 2005.

Matthias Felleisen.
On the expressive power of programming languages.
Sci. Comput. Program., 17(1-3):35–75, 1991.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt.
Combinators for bi-directional tree transformations: a linguistic approach to the view update problem.
In Jens Palsberg and Martín Abadi, editors, POPL, pages 233–246. ACM, 2005.

Richard A. Frost and Rahmatullah Hafiz.

A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time.
SIGPLAN Notices, 41(5):46–54, 2006.

Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan.
Parser combinators for ambiguous left-recursive grammars.
In Hudak and Warren [HW08], pages 167–181.

Bryan Ford.
Packrat parsing: : simple, powerful, lazy, linear time, functional pearl.
In Mitchell Wand and Simon L. Peyton Jones, editors, ICFP, pages 36–47. ACM, 2002.

Bryan Ford.
Parsing expression grammars: a recognition-based syntactic foundation.
In Neil D. Jones and Xavier Leroy, editors, POPL, pages 111–122. ACM, 2004.

Paul Hudak and David Scott Warren, editors.
Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008, San Francisco, CA, USA, January 7-8, 2008, volume 4902 of Lecture Notes in Computer Science. Springer, 2008.

Oleg Kiselyov.
How to write seemingly unhygienic and referentially opaque macros with syntax-rules.
In Scheme Workshop, 2002.

Oleg Kiselyov.
Macros that compose: Systematic macro programming.
In Don S. Batory, Charles Consel, and Walid Taha, editors, GPCE, volume 2487 of Lecture Notes in Computer Science, pages 202–217. Springer, 2002.

Donald E. Knuth.
Semantics of context-free languages.
Mathematical Systems Theory, 2(2):127–145, 1968.

Donald E. Knuth.
The genesis of attribute grammars.
In Pierre Deransart and Martin Jourdan, editors, WAGA, volume 461 of Lecture Notes in Computer Science, pages 1–12. Springer, 1990.

Peter Kourzanov and Henk Sips.
Lingua franca of functional programming (fp).
In Hans-Wolfgang Loidl and Ricardo Peña, editors, Trends in Functional Programming, volume 7829 of Lecture Notes in Computer Science, pages 198–214. Springer Berlin Heidelberg, 2013.
URL: `http://dx.doi.org/10.1007/978-3-642-40447-4_13`, `doi:10.1007/978-3-642-40447-4_13`.

Jacco Krijnen, S. Doaitse Swierstra, and Marcos Viera.
Expand: Towards an extensible pandoc system.
In Matthew Flatt and Hai-Feng Guo, editors, PADL, volume 8324 of Lecture Notes in Computer Science, pages 200–215. Springer, 2014.

- Fernando C. N. Pereira and David H. D. Warren.
  Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks.
  Artif. Intell., 13(3):231–278, 1980.

- Tillmann Rendel and Klaus Ostermann.
  Invertible syntax descriptions: unifying parsing and pretty printing.
  In Jeremy Gibbons, editor, Haskell, pages 1–12. ACM, 2010.

- Dominique Snyers and André Thayse.
  From Logic Design to Logic Programming: Theorem Proving Techniques and P-Functions, volume 271 of Lecture Notes in Computer Science.
  Springer, 1987.

- André Thayse and Eric Gregoire.
  Approche Logique de l'Intelligence Artificielle: 1. De la logique classique a la programmation logique.
  Number 44618 in 429. MIR, 1991.

- Eelco Visser.
  Stratego: A language for program transformation based on rewriting strategies.
  In Aart Middeldorp, editor, RTA, volume 2051 of Lecture Notes in Computer Science, pages 357–362. Springer, 2001.

- Alessandro Warth, James R. Douglass, and Todd D. Millstein.
  Packrat parsers can support left recursion.

In Robert Glück and Oege de Moor, editors, PEPM, pages 103–110. ACM, 2008.

- Alessandro Warth and Ian Piumarta.
  Ometa: an object-oriented language for pattern matching.
  In Pascal Costanza and Robert Hirschfeld, editors, DLS, pages 11–19. ACM, 2007.