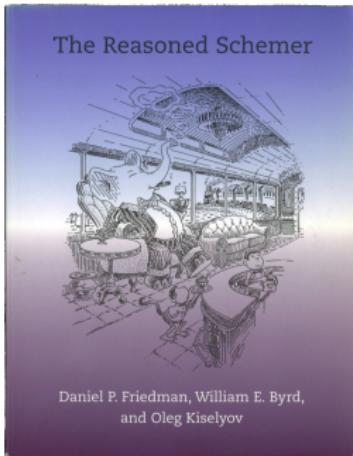
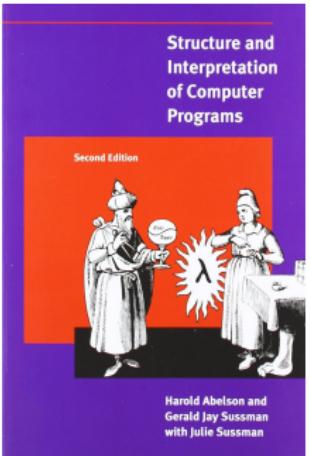




Structure and Interpretation of Definite Clause Grammars (DCGs)



Fundamentalist Declarative Programming with Scheme

State of Pure³ as of January 10, 2014. Peter Kourzanov^{1,2}

²TU Delft (TUD) Parallel & Distributed Systems (PDS) group

¹NXP Research & Development, Eindhoven, Netherlands

[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](#)

Abstract

Pure³ ≡ 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 Revised⁵ 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



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

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](#)])

```
1  (def deintreave (fn '
2    () → () ()
3    | (,x ,y . ,,[~ deinterleave→ '(',a ,b)]) →
4      (,x . ,a) (,y . ,b)
5    ))
```

```
1  (def interleave (fn '
2    () () → ()
3    | (,x . ,a) (,y . ,b) →
4      (,x ,y . ,[apply interleave '(',a ,b)])
5    ))
```

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

Doesn't this sound too good to be true?

Write a parser, get a pretty-printer for free, or equivalently/bidirectionally:

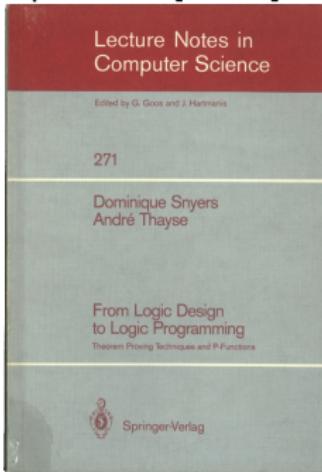
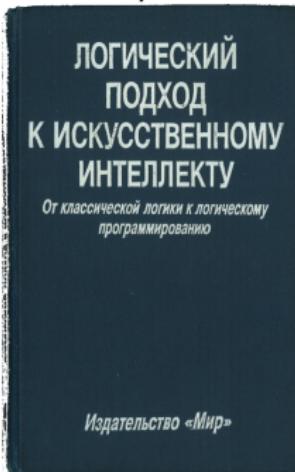
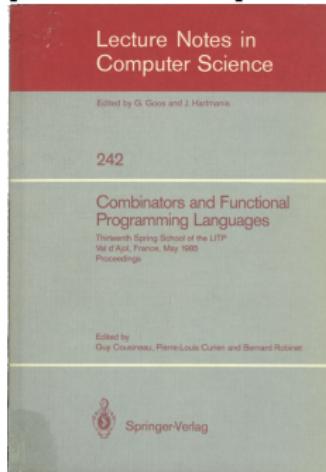
Write a pretty-printer, get a parser for free.

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]



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]

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

Fully reversible syntax \leftrightarrow 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)

```
1 (verify Expr (run* (q) (Expr '(2 ^ 2 * 1 + 3 * 5) `() q)) ==> (+ (* (^ 2 2) 1) (* 3 5)))
2 (verify Expr (run* (q) (Expr q `() `(+ (* 2 a) (* x 5)))) ==> (2 * a + x * 5))
3 (verify Expr (run* (q) (Expr '(1 * 3 + 5) `() `(+ (* 1 3) 5))) ==> _ . 0)
4 (verify Expr (parameterize ([* digits* '42]) [* letters* '#\x]))
5 (run 7 (q) (fresh (x y) (Expr x `() y) (== q `(',x ,y)))) -->
6 ((x + x) (+ x x))
7 ((x) x)
8 (((42 + x) (+ 42 x)))
9 (((x * x) (* x x)))
10 (((x ^ x) (^ x x)))
11 (((x + x * x) (+ x (* x x))))
12 (((x - x) (- x x)))
13 )
```

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

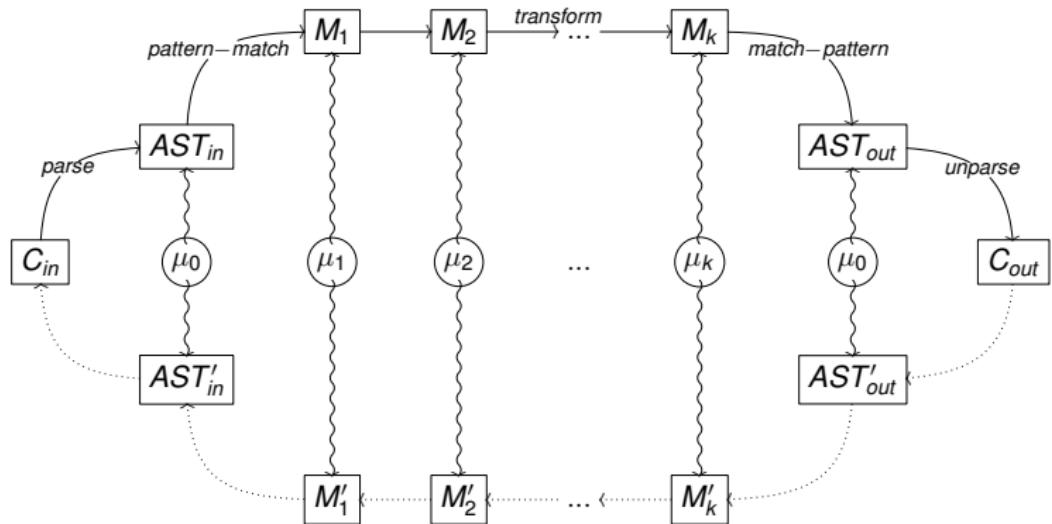
Conclusion

Appendix

Acronyms

References

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-matching is OK

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



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.

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



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”

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

Spoiler

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

```
1  ;; A BNF for a trivial expression grammar
2  <factor> ::= <factor> ^ <literal>
3  | <literal>
4  <term> ::= <term> * <factor>
5  | <term> / <factor>
6  | <factor>
7  <expr> ::= <expr> + <term>
8  | <expr> - <term>
9  | <term>
```

```
1  ;; An ideal DCG for the same ...
2  factor -> factor, [^], literal.
3  factor -> literal.
4  term -> term, [*], factor.
5  term -> term, [/], factor.
6  term -> factor.
7  expr -> expr, [+], term.
8  expr -> expr, [-], term.
9  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 parens 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*



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

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

```
1  (dcg Factor
2    ([Factor] <=> [Factor] '^' [literal])
3    ([Factor] <=> [literal]))
4  )
5  (dcg Term
6    ([Term] <=> [Term] '*' [Factor])
7    ([Term] <=> [Term] '/' [Factor])
8    ([Term] <=> [Factor]))
9  )
10 (dcg Expr
11   ([Expr] <=> [Expr] '+' [Term])
12   ([Expr] <=> [Expr] '-' [Term])
13   ([Expr] <=> [Term]))
14 )
```

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...



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

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

| | | | |
|----|------------------------------|----|---|
| 1 | <fact> ::= <lit> <fact> | 1 | factor(F) → literal(L), factor_r(L, F). |
| 2 | <fact> ::= ^ <lit> <fact> | 2 | factor_r(T0, F) → [^], literal(L), factor_r(exp(T0, L), F). |
| 3 | ε | 3 | factor_r(F, F) → []. |
| 4 | <term> ::= <fact> <term> | 4 | term(T) → factor(F), term_r(F, T). |
| 5 | <term'> ::= * <fact> <term'> | 5 | term_r(T0, T) → [*], factor(F), term_r(mul(T0, F), T). |
| 6 | / <fact> <term'> | 6 | term_r(T0, T) → [/], factor(F), term_r(div(T0, F), T). |
| 7 | ε | 7 | term_r(T, T) → []. |
| 8 | <expr> ::= <term> <expr'> | 8 | expr(E) → term(T), expr_r(T, E). |
| 9 | <expr'> ::= + <term> <expr'> | 9 | expr_r(E0, E) → [+], term(T), expr_r(pls(E0, T), E). |
| 10 | - <term> <expr'> | 10 | expr_r(E0, E) → [-], term(T), expr_r(min(E0, T), E). |
| 11 | ε | 11 | expr_r(E, E) → []. |

Nay

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

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



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

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



Left-recursion avoidance + higher-order patching

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

```
1 (dkg Factor
2   ([_ (π [λ (z) (y (if [null? z] x '(^ ,z ,x))))])
3   <=> [literal x] '^(Factor y)]
4   ([_ (π [λ (z) (if [null? z] x '(^ ,z ,x))))]
5   <=> [literal x])
6 )
```

- ① `dkg` 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...

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

Left-recursion avoidance + declarative patching



```
1 (dcg factor locals: (x y z))
2 ([factor `(^ ,x (* . ,y . ,z))]) <=> [literal x] `^` [factor `(^ ,y (* . ,z))])
3 ([factor `(^ ,x (* . ,y ,z))]) <=> [literal x] `^` [factor `(^ ,y ,z)])
4 ([factor `(^ ,x ,y)]) <=> [literal x] `^` [factor y])
5 ([factor x]) <=> [literal x])
6 )
```

```
1 (dcg term locals: (x y z l))
2 ([term `(* ,l . ,z)] <=[(pushdown x `* y l)] => [factor x] `*` [term `(* ,y . ,z)])
3 ([term `(* ,x ,y)] <=[(! sameops `(*) y)] => [factor x] `*` [term y])
4 ([term `(/ ,l . ,z)] <=[(pushdown x `/ y l)] => [factor x] `/` [term `(/ ,y . ,z)])
5 ([term `(/ ,x ,y)] <=[(! sameops `(/) y)] => [factor x] `/` [term y])
6 ([term x]) <=> [factor x])
7 )
```

```
1 (dcg expr locals: (x y))
2 ([expr `(+ ,x . ,y)] <=> [term x] `+` [expr `(+ . ,y)])
3 ([expr `(+ ,x ,y)] <=> [term x] `+` [expr y])
4 ([expr `(- ,x . ,y)] <=> [term x] `-` [expr `(- . ,y)])
5 ([expr `(- ,x ,y)] <=> [term x] `-` [expr y])
6 ([expr x]) <=> [term x])
7 )
```

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

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)

```
1  (dcg
2   (factor locals: (x)
3     ([_ y]    <=> [literal x] [factor' x y]))
4   (factor' locals: (y)
5     ([_ x z] <=> '^' [literal y] [factor' ' (^ ,x ,y) z])
6     ([_ x x] <=> ε))
7   (term locals: (x)
8     ([_ y]    <=> [factor x] [term' x y]))
9   (term' locals: (y)
10    ([_ x z] <=> '*' [factor y] [term' ' (* ,x ,y) z])
11    ([_ x z] <=> '/' [factor y] [term' ' (/ ,x ,y) z])
12    ([_ x x] <=> ε))
13   (expr locals: (x)
14     ([_ y]    <=> [term x] [expr' x y]))
15   (expr' locals: (y)
16     ([_ x z] <=> '+' [term y] [expr' ' (+ ,x ,y) z])
17     ([_ x z] <=> '-' [term y] [expr' ' (- ,x ,y) z])
18     ([_ x x] <=> ε))
19 ))
```



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

- 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)

```
1  (defn Expr (dcg <=> Expr
2  (Factor
3    ([_ `(^ ,x ,y)] <=> [Factor x] `^ [literal y])
4    ([_ x]           <=> [literal x])))
5  (Term
6    ([_ `(*) ,x ,y)] <=> [Term x] `* [Factor y])
7    ([_ `(/ ,x ,y)] <=> [Term x] `/ [Factor y])
8    ([_ x]           <=> [Factor x])))
9  (Expr
10   ([_ `(+ ,x ,y)] <=> [Expr x] `+ [Term y])
11   ([_ `(- ,x ,y)] <=> [Expr x] `- [Term y])
12   ([_ x]           <=> [Term x]))
13 )))
```



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

Some revelations

- ① `dcg` can introduce encapsulated clauses
- ② attributes are just like functor arguments in PROLOG
- ③ declarative binding style for synthesized 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.

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

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



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

Some simple examples

```
1 (def-syntax λ lambda)
2 (def-syntax ⊥ (syntax-rules ()))
3 (def-syntax [id a] a)
4 (def-syntax [hd a . _] a)
5 (def-syntax [tl _ . b] b)
6 (def-syntax zip2 (syntax-rules ... ())
7   ([_ (k ...) () () . a] (k ... . a))
8   ([_ k (x . xs) (y . ys) . a] (zip2 k xs ys (x y) . a))
9   ))
10 (def-syntax revs (syntax-rules ())
11   ([_ () () . r] r)
12   ([_ (k args ...) () . r] (k args ... . r))
13   ([_ k (h . t) . r] (revs k t h . r))
14 ))
```

```
1 ; A poor man's device to prevent recursion in the absence of types
2 ; Redefine f to ⊥ just for inside the body itself. Result is
3 ; that f can neither be reified as a first-class value, nor can
4 ; it be applied, since the expansion, ⊥ has no alternatives.
5 (def-syntax defn (syntax-rules ())
6   ([_ (f . args) . body] ; recursive functions
7     (define f (λ args
8       (let-syntax ([f ⊥])
9         (begin . body))))
10    ))
11  ([_ f . exprs] ; recursive CAFs
12    (define f
13      (let-syntax ([f ⊥])
14        (begin . exprs)))
15    ))
16 ))
```



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

Recursive macros



Extracting free variables from Scheme terms:

```
1 (def-syntax w (syntax-rules .. (qq quote unquote quasiquote unquote-splicing λ)
2   ([_ q (k ..) b [] . a] (k .. a))
3   ([_ q k b 't . a] (w [qq . q] k b t . a))
4   ([_ [qq . q] k b ,t . a] (w q k b t . a))
5   ([_ [] k b ,t . a] (bad-unquote k b ,t))
6   ([_ q k b 't . a] (w q k b [] . a))
7   ([_ [] k b [λ (var ..) . body] . a] (w [] k (var .. b) body . a))
8   ([_ q k b [t . ts] . a] (w q (w q k b t) b ts . a))
9   ([_ [] k b t a ..]
10    (symbol?? t
11      (member?? t (a .. b)
12        (w [] k b [] a ..)
13        (w [] k b [] a .. t))
14        (w [] k b [] a ..)
15      )))
16   ([_ [qq . q] k b t . a] (w q k b [] . a))
17 ))
```

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

Removing one level of quasi-quoting:

```
1 (def-syntax qs (syntax-rules .. (qq quote unquote quasiquote unquote-splicing)
2   ([_ q (k ..) () . a] (k .. a))
3   ([_ [] k 'y . a] (qs [qq] k y . a))
4   ([_ q k 'y . a] (qs q k () 'y . a))
5   ([_ [qq] k ,y . a] (qs [] k () [qq y] . a))
6   ([_ [qq] k ,y . a] (qs [] k () y . a))
7   ([_ [] k ,y . a] (bad-unquote k ,y))
8   ([_ q k 'y . a] (qs q k () 'y . a))
9   ([_ [qq] k (y . ys) . a] (qs [qq] (qs [qq] k y) ys . a))
10  ([_ [qq . q] k y . a] (qs q k 'y . a)))
11 ))
```

Reflection, breaking hygiene

Courtesy Al Petrofsky and [Kis02b, Kis02a]



```
1 (def-syntax [ extract s body _k]
2   (letrec-syntax ([tr (syntax-rules (s)
3     ([_ x s tail (k sl . args)]
4      (k (x . sl) . args))
5     ([_ d (x . y) tail k]
6      (tr x x (y . tail) k))
7     ([_ d1 d2 () (k sl . args)])
8      (k (s . sl) . args))
9     ([_ d1 d2 (x . y) k]
10       (tr x x y k)))
11    )))
12   (tr body body () _k)
13 ))
14 (def-syntax extract* (syntax-rules ())
15   ([_ (s) body k] (extract s body k))
16   ([_ _ss _body _k]
17    (letrec-syntax ([ex (syntax-rules ()
18      ([_ fs () body k] (revs k fs))
19      ([_ fs (s . ss) body k]
20        (extract s body (ex fs ss body k))))
21    )))
22   (ex () _ss _body _k)))
23 ))
```

```
1 (def-syntax symbol?? (syntax-rules ())
2   ([_ (x . y) kt kf] kf)
3   ([_ #(x ...) kt kf] kf)
4   ([_ maybe-symbol kt kf]
5    (let-syntax ([test (syntax-rules ()
6      ([_ maybe-symbol t f] t)
7      ([_ x t f] f)
8    )))
9      (test abracadabra kt kf)
10    )))
11 )))
12
13 (def-syntax member?? (syntax-rules ())
14   ([_ id () kt kf] kf)
15   ([_ (id . ids) xs kt kf] kf)
16   ([_ id (x . r) kt kf]
17    (let-syntax ([test (syntax-rules (id)
18      ([_ id t f] t)
19      ([_ xx t f] f)
20    )))
21      (test x kt (member?? id r kt kf)))
22    )))
23 ))
```

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

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”

- runs in the Bigloo interpreter via `alexander`
 - 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...



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

Example (pure MINIKANREN)

```
1 (set-sharp-read-syntax! 's succeed)
2 (set-sharp-read-syntax! 'u fail)
3 (def (null0? x) [≡ x '()])
4 (def (pair0? x) (fresh (x0 x1) [≡ x '(,x0 . ,x1)]))
5 (def (car0 x y) (fresh (t) [≡ x '(,y . ,t)]))
6 (def (cdr0 x y) (fresh (h) [≡ x '(,h . ,y)]))
7 (def (cons0 h t !) (≡ ! '(',h . ,t)))
8 (def take-from
9   (λ () _ => #u
10    | '(.head . ,tail) f =>
11      (conde
12        ([≡ f head])
13        (else (take-from tail f)))
14    | db _ => (error 'take-from "bad database" db)))
15 (def *digits* (make-parameter (list-tabulate 10 values)))
16 (def *letters* (make-parameter
17   (unfold [_ char>? #\z]
18     values
19     (o integer->char
20       [_ + 1]
21       char->integer)
22     #\a)))
23 (def (numbers? x) (take-from [* digits *] x))
24 (def (symbols? x) (take-from
25   (map (o string->symbol
26     list->string
27     list)
28     [* letters *]) x))
29 (def (! p . args)
30   (cond
31     ((apply p args) #u)
32     (else #s)))
```

;; dynamic binding
;; dynamic binding
;; just for the purpose
;; of exposition

;; that is where
;; real impurity is



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

... 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, flexible ordering of instantiation (aka delaying by unification \equiv “spooky action at distance”)

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

```
1 literal(x) --> symbol(x). 1
2 literal(x) --> [x],           2
3   {(number(x)                 3
4     ;var(x))                4
5   ,between(0,9,x)}           5
```

```
(def (number Lin Lout x) (all (numbers? x) (cons0 x Lout Lin)))
(def (symbol Lin Lout x) (all (symbols? x) (cons0 x Lout Lin)))
(def (literal Lin Lout x)
  (conde ([symbol Lin Lout x])
         ([number Lin Lout x])))
```



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



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...

```
1 (def-syntax make-fresh (syntax-rules ()
2   ([_ () head . body] (head . body))
3   ([_ vars _ . body] (fresh vars . body)))
4 ))
```

Let's address the “how” first...

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

implementing multi-rule clauses (DCG excerpts)



```
1  ([_ head heads: heads locals: locals condo: condo . rules]
2   [define head (λ (Lin Lout . result)
3     (letrec-syntax
4       ([p (syntax-rules (<=> <= =>)
5          ([p k acc] (revs (k) acc)) ;; implementing pure clauses
6          ([p k acc ((x args ...) <=> . goals) . rest]
7            (p k (((all [= result '(,args ...)])
8              (seq Lin Lout k () () [heads acc] . goals)
9                )) . acc) . rest))) ;; implementing logical "effects"
10         ([p k acc ((x args ...) <=[actions ...]=> . goals) . rest]
11          (p k (((all [= result '(,args ...)])
12            (project (Lin)
13              (if [ground? Lin]
14                #s
15                  (begin actions ...)))
16                (seq Lin Lout k () () [heads acc] . goals)
17                  (project (Lin)
18                    (if [ground? Lin]
19                      (begin actions ...)
20                        #s))
21                      )] . acc) . rest))))])
22        (make-fresh locals begin
23          (p condo () . rules)
24        )))
25      ))))
```

Declare recursive clauses that are mutually aware (slide 4)

```
1  ([_ (head . args) ..]
2   (let-syntax-rule ([k . heads]
3     (begin (dcg head heads: (rev: . heads) . args) ..)
4     (k head ..)
5   )))
```

Encapsulate clauses and make them mutually aware (slide 1)

```
1  ([_ <=> start (head . args) ..] (dcg start [] (rev: head .. => (head . args) ..))
2  ([_ => start (head . args) ..] (dcg start () (ref: head .. => (head . args) ..))
3  ([_ <= start (head . args) ..] (dcg start () (reb: head .. => (head . args) ..))
4  ([_ <=: start (head . args) ..] (dcg start () (reu: head .. => (head . args) ..))
5  ([_ start [acc ..] ach =>] (let () acc .. start))
6  ([_ start acc ach => [head . args] . rest]
7    (dcg start ((dcg head heads: ach . args) . acc) ach => . rest)))
```

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

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

```
1 ([_ in out c acc ts hs do(as ...) . rest] (seq in out c (as ... . acc) ts hs . rest))
2 ([_ in out c acc tmpls heads ε . rest] (seq in out c ([== in out] . acc) tmpls heads . rest))
3 ([_ in out c acc tmpls heads (quote datum) . rest]
4   (let ([temp #FALSE]) ;; just to generate a new temporary
5     (seq temp out c ([== in '(datum . ,temp)] . acc)
6       (temp . tmpls) heads . rest)
7   ))
8 ([_ in out c acc tmpls heads (qq dat) . rest] (seq in out c acc tmpls heads 'dat . rest))
9 ([_ in out c acc tmpls [hs (ac ...)] (quasiquote datum) . rest]
10  (let ([temp #FALSE][data #FALSE]) ;; just to generate new temporaries
11    (seq temp out c ((qs []) (seq data '() c () () [hs (ac ... . acc)]) (quasiquote datum))
12      (temp data . tmpls) [hs (ac ...)] . rest)
13    ))
14 ))
```

Handling sequencing, non-terminals

```
1 ([_ in out c acc temps [heads (ac ...)] (: goals ...) . rest]
2   (let ([temp #FALSE]) ;; just to generate a new temporary
3     (seq temp out c ((all (seq in temp c () () [heads (ac ... . acc)] goals ...)) . acc)
4       (temp . temps) [heads (ac ...)] . rest)
5   ))
6 ([_ in out c acc temps heads (goal . args) . rest]
7   (let ([temp #FALSE]) ;; just to generate a new temporary
8     (seq temp out c ([goal in temp . args] . acc)
9       (temp . temps) heads . rest)
10    ))
```

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



Finalization, and optimizations for singular rules

```

1  ([_ in out _ acc ts hs do(as ...)] (revs (make-fresh ts begin) (as ... . acc)))
2  ([_ in out c acc ts hs e] (seq in out c acc ts hs do[==(in out)]))
3  ([_ in out c acc ts hs (quote d)] (seq in out c acc ts hs do[==(in `'(d . ,out))]))
4  ([_ in out c acc ts [heads (ac ...)] (quasiquote datum)])
5  (let ([data #FALSE]) ; just to generate a new temporary
6    (seq in out c acc (data . ts) [heads (ac ... . acc)])
7    do[(qs [] (seq data '() c () () [heads (ac ... . acc)]) (quasiquote datum))
8      (== in `'(,data . ,out))])
9  )
10 ([_ in out c acc temps [heads (ac ...)] (: goals ...)]
11  (seq in out c acc temps [heads (ac ...)])
12  do[(all (seq in out c () () [heads (ac ... . acc)] goals ...))]
13  ))
14 ([_ in out c acc temps heads (goal . args)]
15  (seq in out c acc temps heads do[(goal in out . args)]))

```

Example (Wasn't too difficult was it?)

```

1  (dcg O
2    ([O] <=> '*)
3    ([O] <=> '/))
4  (dcg O1..3
5    ([O1..3] <=> [O]))
6    ([O1..3] <=> [O] 'o [O])
7    ([O1..3] <=> [O] 'o [O] 'o [O]))
8
9  (dcg S
10   ([S 'z] <=> e)
    ([_ '(S ,x)] <=> (: 'a 'a [S x])))

```

Oops it still is

```

1  (dcg SS
2    ([SS 'z] <=> e)
3    ([_ '(S ,x)] <=> [SS x] 'a 'a))

```

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

Solving left-recursion, lazily



Let's be logically lazy

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

```
1  append([],L,L).  
2  append([X|A1],B,[X|C1]) :- append(A1,B,C1).
```

or, with MINIKANREN [Byr10]

```
1  (def (appendo a b c)  
2    (conde  
3      ([≡ a '()] (≡ b c))  
4      (else (fresh (x a1 c1)  
5                    (≡ a '(,x . ,a1))  
6                    (≡ c '(,x . ,c1))  
7                    (appendo a1 b c1)))  
8    ))
```

Example (`appendo` is fully reversible)

```
1  (verify A (run* (q) (appendo q '(3) '(1 2 3))) ==> (1 2))  
2  (verify A (run* (q) (appendo '(1 2) q '(1 2 3))) ==> (3))  
3  (verify A (length (run* (q) (fresh (x y) (appendo x y '(1 2 3)) (== q '(,x ,y)))))) = 4)  
4  (verify A (run* (q) (fresh (x y) (appendo '(1 2) x y) (== q '(,x ,y)))))  
5  ==> (_.0 (1 2 . _.0)))  
6  (verify A (run 2 (q) (fresh (x y) (appendo x '(3) y) (== q '(,x ,y))))  
7  ==> (( (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^o as dummy
- ④ delay the unlifting goal resolution to the very end
- ⑤ tie the knot by unifying difference list components

```
1  ([_ in out c acc temps heads (unlift goal . args)]          ;; tie the knot
2   (seq in out c acc temps heads do[(goal . args) (== in out)]))
3   ([_ in out c acc temps heads (lift goal . args)])           ;; already the "end"
4   (seq in out c acc temps heads do[(goal in out . args)])
5   ([_ in out c acc temps heads (unlift goal . args) . rest]
6   (seq in out c ((goal . args) . acc) temps heads . rest))
7   ([_ in out c acc temps [(ref: . heads) ac] (lift goal . args) rest ...]; untie the knot
8   (let ([temp #FALSE][data #FALSE])                                ; just to generate new temporaries
9     (seq temp out c ((appendo data temp in) . acc)
10    (temp data . temps) [(ref: . heads) ac] rest ...
11    (unlift goal data '() . args)))
12 ))
```

Yea

- “solves” left-recursion
- this often makes the parser also tail-recursive
- 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.

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

Left-recursion and reversibility

Now lets use the trick from slide 2

```
1  ([_ in out c acc temps heads (lift goal . args) rest ...]
2   (let ([temp #FALSE][data #FALSE]) ; just to generate new temporaries
3     (seq temp out c ([append0 data temp in]
4       (project (in)
5         (if [ground? in]
6           #s
7             (goal data '() . args)))
8           . acc)
9     (temp data . temps) heads rest ...
10    (unlift project (in)
11      (if [ground? in]
12        (goal data '() . args)
13        #s
14      )))))
```

Example (Yea!)

slide 1 + grammar from slide 1 finally work together

Looks like mission accomplished

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



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

On-line?

```
1 (def (fresh0 x)
2   (conde ([== x '()])
3     (else (fresh (y z)
4       (fresh0 z)
5       (== x '(,y . ,z)))
6     ))))
7 (def (prefix0 a b)
8   (fresh (x)
9     (fresh0 x)
10    (append0 a x b)
11  )))
```



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

Example (Infinite input stream)

```
1 ;; testing prefix0
2 (verify fresh0 (run 4 (q) (fresh0 q))
3   ____> (_.0 _.1 _.2) (_.0 _.1) (_.0) ())
4 (verify prefix0 (run 4 (q) (prefix0 '(1 2 3) q))
5   ____> (1 2 3) (1 2 3 _.0) (1 2 3 _.0 _.-1) (1 2 3 _.0 _.-1 _.-2))
6 ;; testing infinitary parsing
7 (verify SS (run 3 (q) (fresh (l) (prefix0 '() l) (SS l '() q)))
8   ____> z (S z) (S (S z)))
9 (verify SS (run 2 (q) (fresh (l) (prefix0 '(a a) l) (SS l '() q)))
10  ____> (S (S z)) (S z))
11 (verify SS (run 2 (q) (fresh (l) (prefix0 '(a a a a) l) (SS l '() q)))
12  ____> (S (S (S z))) (S (S z)))
13 (verify SS (run 2 (q) (fresh (l) (prefix0 '(a a a a a a) l) (SS l '() q)))
14  ____> (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

```
1  ([_ in out c () temps [(r . heads) ()] (goal . args)]
2   (member?? goal heads
3    (seq in out c () temps [(r . heads) ()] (lift goal . args)))
4    (seq in out c () temps [(r . heads) ()] do[(goal in out . args)])
5   ))
6  ([_ in out c () temps [(r . heads) acc] (goal . args) . rest]
7   (member?? goal heads
8    (seq in out c () temps [(r . heads) acc] (lift goal . args) . rest)
9     (let ([temp #FALSE]) ; just to generate a new temporary
10      (seq temp out c ([goal in temp . args])
11        (temp . temps) [(r . heads) acc] . rest)
12     )))
```

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]:

```
1  ([_ in out c () temps [(reu: . heads) ()] (lift goal . args)] ; just diverge
2   (seq in out c () temps [(reu: . heads) ()] do[(goal in out . args)]))
3  ([_ in out c acc temps [(reu: . heads) ac] (lift goal . args) . rest] ; just diverge
4   (let ([temp #FALSE]) ; just to generate a new temporary
5    (seq temp out c ([goal in temp . args] . acc)
6      (temp . temps) [(reu: . heads) ac] . rest)))
7  ([_ in out c () temps [(x . heads) ()] (lift goal . args)] ; recursive singleton goal
8   (seq in out c () temps [(x . heads) ()] do[#u (== in out)]))
9  ([_ in out c acc temps [(reb: . heads) ac] (lift goal . args) . rest] ; degenerate loop
10   (let ([temp #FALSE]) ; just to generate a new temporary
11    (seq temp out c (#u [== in temp] . acc)
12     (temp . temps) [(reb: . heads) ac] . rest)))
```

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

Sort of a CAP theorem



Theorem (A conjecture, really, about relational programming)

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

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

"An atomic PEG consists of: any terminal symbol, any nonterminal symbol, or the empty string ϵ . Given any existing PEGs e , e_1 , and e_2 , a new PEG can be constructed using the following operators: (1) sequence: $e_1\ e_2$, (2) ordered choice: $e_1\ /e_2$, (3) zero-or-more: e^* , (4) one-or-more: e^+ , (5) optional: $e?$, (6) and-predicate: $\&e$, (7) not-predicate: $\text{!}e$." [Wikipedia]



Atomic PEGs and sequencing we already had

Now lets add the rest (skipping some details now)

```

1 (def-syntax proc-/ (syntax-rules (/)
2   ([_ in out c (k ...) heads] (k ... #u))
3   ([_ in out c (k ...) heads a] (k ... ((seq in out c () () heads a))))
4   ([_ in out c (k ...) heads a / . as] (proc-/ in out c (k ... ((seq in out c () () heads a))) heads . as)))
5 ))

```

```

1 ([_ in out c acc temps heads (alt / . alts)]
2   (seq in out c acc temps heads do[(proc-/ in out c (c) heads alt / . alts)]))
3   ([_ in out c acc temps [heads (ac ...)] (goals ... *)]
4     (seq in out c acc temps [heads (ac ...)])
5       do[([let loop ([|in in|][|out out|])
6         (c ([|= lin lin out]))
7           ((let ([temp #FALSE]) ; just to generate a new temporary
8             (fresh (temp)
9               (seq lin temp c () () [heads (ac ... . acc)] goals ...))
10              (loop temp [out]))))))]
11   ([_ in out c acc temps [heads (ac ...)] (goals ... +)]
12     (seq in out c acc temps [heads (ac ...)])
13       do[([let loop ([|in in|][|out out|])
14         (let ([temp #FALSE]) ; just to generate a new temporary
15           (fresh (temp)
16             (seq lin temp c () () [heads (ac ... . acc)] goals ...))
17             (c ([|= temp temp out]))
18               (loop temp [out]))))))]
19   ([_ in out c acc temps [heads (ac ...)] (goals ... ?)]
20     (seq in out c acc temps [heads (ac ...)])
21       do[(c ((seq in out c () () [heads (ac ... . acc)] goals ...)) ([== in out])))])
22   ([_ in out c acc temps [heads (ac ...)] when guards]
23     (seq in out c acc temps [heads (ac ...)])
24       do[(fresh (temp) (c ((seq in temp c () () [heads (ac ... . acc)] . guards) #s) (else #u))))])
25   ([_ in out c acc temps [heads (ac ...)] unless guards]
26     (seq in out c acc temps [heads (ac ...)])
27       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...*

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

Committed choice via PEG macro

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



```
1 (def-syntax peg (syntax-rules (<= <=> =>)
2   ([_ <=> start (head . args) ...]
3    (dcg <=> start (head condo: conde . args) ...))
4   ([_ => start (head . args) ...]
5    (dcg => start (head condo: conde . args) ...))
6   ([_ <= start (head . args) ...]
7    (dcg <= start (head condo: conde . args) ...))
8   ([_ <:= start (head . args) ...]
9    (dcg <:= start (head condo: conde . args) ...))
10  ([_ (head . args) ...]
11   (dcg (head condo: conde . args) ...))
12  ([_ head . args]
13   (dcg head condo: conde . args)))
14 ))
```

Example (Dangling else)

```
1 (define ife (peg <=> if
2   (if
3     ([_ '([if ,x ,y ,z]) <=> 'if [Expr x] 'then [Expr y] 'else [Expr z])
4     ([_ '([if ,x ,y #f]) <=> 'if [Expr x] 'then [Expr y])
5     ([_ '([if ,x ,y ,z]) <=> 'if [Expr x] 'then [if y] 'else [Expr z])
6     ([_ '([if ,x ,y #f]) <=> 'if [Expr x] 'then [if y])
7   )))
8   (verify ife.nest (run* (q) (ife '(if 1 then if 2 then 3 else 4 else 5) '() q))
9   ===> (if 1 (if 2 3 4) 5))
10  (verify ife.dangling (run* (q) (ife '(if 1 then if 2 then 3 else 4) '() q))
11  ===> (if 1 (if 2 3 4) #FALSE))
```

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

Returning to slide 2...

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

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



```
1 (define scheme-bindings (syntax-rules ... ())
2   ([_ (k a b [s ..] . d)]
3    (k a b [s .. list first second pair? car cdr null? if cond begin + - * /] . d))
4 ))
```



```
1 ([p k acc ([] <=> . goals) . rest] ;; the user wants to get all results as a single list
2   (p k (((fresh
3     (append0 results Lout Lin)
4     (== result '(,results)))
5     (seq Lin Lout k () () [heads acc] . goals)) . acc) . rest))
6   ([p k acc ([] <=> . goals) . rest] ;; the user wants to get each result separately
7     (p k (((all (append0 result Lout Lin)
8       (seq Lin Lout k () () [heads acc] . goals)) . acc) . rest)))
9     ;; the user wants to use higher-order and infer synthesized attributes
10  ([p k acc ([] (π args ...) ] <=> . goals) . rest]
11    (let-syntax-rule ([K . vars] ;; collect the free vars
12      (let-syntax-rule ([K vars pats terms] ;; use extracted vars
13        (make-fresh vars all
14          (seq Lin Lout k () () [heads acc] . terms)
15          (project vars [= result pats]))
16        )))
17        (extract* vars (args ... . goals) (K () '(,args ...) goals)))
18      ))
19    (p k (((scheme-bindings (w [] (K) [] (args ...))) . acc) . rest)))
20      ;; the user wants to just infer synthesized attributes
21  ([p k acc ([] args ...) <=> . goals) . rest]
22    (let-syntax-rule ([K . vars] ;; collect the free vars
23      (let-syntax-rule ([K vars pats terms] ;; use extracted vars
24        (make-fresh vars all
25          [= result pats]
26          (seq Lin Lout k () () [heads acc] . terms)
27        )))
28        (extract* vars (args ... . goals) (K () '(,args ...) goals)))
29      ))
30    (p k (((scheme-bindings (w [] (K) [] (args ...))) . acc) . rest)
31 )))
```

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

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...



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

```
1  ([_ in out c acc temps [(r . heads) (ac ...) ] (goals ... +)]
2   (let-syntax ([K (syntax-rules () ()))
3     ([_ in out vars ..] ;; we need to explicitly get [[in]] and [[out]] from the caller
4      (let loop ([lin in] [lout out] [vars '()] ..) ;; 1st bunch
5        (let-syntax ([K (syntax-rules () ()))
6          ([_ res ...] ;; 2nd bunch
7            (let ([res #FALSE] ...)
8              (make-fresh (res ...)) begin
9                (letrec-syntax ([K (syntax-rules () ()))
10                  ([_ gls (v v1 v2 v3) ..] ;; and now declare the 3rd bunch of vars
11                    ;; and substitute it for the original var in [[gls]]==[[goals]]
12                    (c ([== lin lout]
13                        ([== v1 v] ...))
14                      (let ([temp #FALSE][v3 #FALSE] ...)
15                        (fresh (temp v3 ...)) ;; rename original var to a local temporary
16                        (let-syntax ([(_ v3) ...])
17                          (seq lin temp c () () [(r . heads) (ac ... . acc)] . gls)))
18                        (append0 v1 `,(,v3) v2) ...
19                        [loop temp lout v2 ..)))))))
20
21  [K1 (syntax-rules ())
22    ([_ var gls .. args] ;; zip all the vars together
23      (zip4 (K gls) var . args)
24    ))]
25
26  [K0 (syntax-rules ()
27    ([_ . vs] ;; 3rd bunch
28      ;; extract the 4th bunch of free-vars
29      ;; now with the same colour as in [[goals]]
30      (extract* vs (goals ...))
31      (K1 [] (goals ...) (vars ..) (res ...) vs)
32      ))))) ;; retrieve the 3rd bunch of free-vars
33
34  (scheme-bindings (w []) (K0) heads (goals ...)))
35  (scheme-bindings (w []) (K) heads (goals ...)))
36  ))))) ;; retrieve the 1st bunch of free-vars
37  ;; we need to pass [[in]] and [[out]] as they would
38  ;; otherwise be renamed
39  (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)

```
1 ;; higher-order rules
2 (defn [R p] (dcg => c
3   (c ([_ `(. ,x , .y)]) <=> [c y] `comma [p x])
4   ([_ `(. ,x)]) <=> [p x)))))
5 ;; regular grammars
6 (dcg A ([] <=> '< ('a *) '>))
7 (dcg B ([] <=> ((`a / `b) +)))
8 (dcg C ([] <=> '< ((`a / `b / `c) +) '>))
9 (dcg O1..3 ([O1..3] <=> [O] (: `o [O] ((: `o [O]) / `ε)) / `ε)))
10 ;; Dyck language
11 (dcg D ([_] <=> 'x)
12   ([_ `(D ,x)] <=> `([D x]))) )
13 ;; context-free grammar  $a^n b^n$ 
14 (dcg anbn ([`cfg] <=> 'a ([anbn] ?) `b))
15 ;; context-free, non-packrat grammar
16 (defn s (dcg <=> S
17   (S ([_] <=> 'x)
18     ([_ `(`s ,x)] <=> 'x [S x] `x)))) )
19 ;; non context-free, packrat grammar
20 (defn anbncn (dcg <=> S
21   (S ([S] <=> when([A] `c) (`a +) [B] unless([`a / `b / `c])) )
22   (A ([A] <=> 'a ([A] ?) `b))
23   (B ([B] <=> 'b ([B] ?) `c)))
24 ))
25 ;; bastardized λ-calculus
26 (defn Λ (dcg <=: S
27   (S ([_ x] <=> ([L x] / [A x] / [T x])))
28   (L ([_ `(`λ ,(_ ,x) ,y)]) <=> `λ [T x] `· [S y]))
29   (A ([_ x] <=> `(! ,([S x] +)))
30     ([_ x] <=> `! ([S x] +)))
31   (T ([_ x] <=> [symbol x])) )
32 ))
```

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



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 Landauer!
- 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)

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

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

Future work



extensibility (ala *CoCoCo* [KSV14])

efficiency (downscale it to do character-based parsing)

memoization for efficient incremental parsing

proof the conjecture on slide 6

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

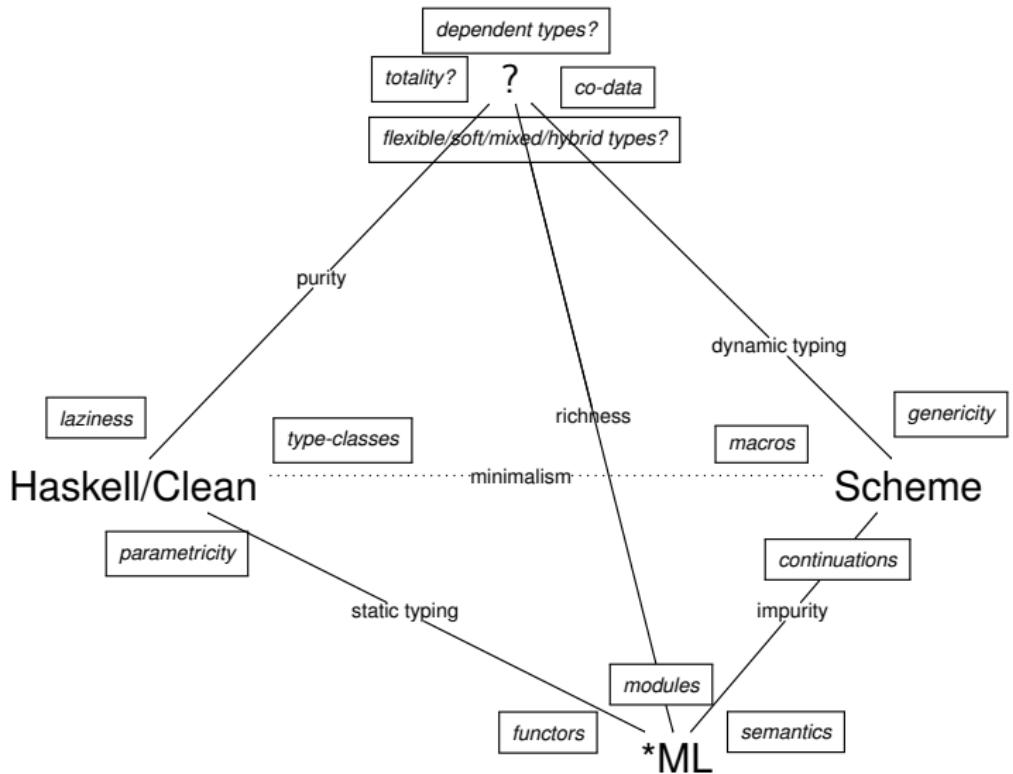
Conclusion

Appendix

Acronyms

References

Grand unified theory of Functional Programming (FP)?



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



6. The Fun Never Ends...



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

| | |
|------|---------------------------------------|
| 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 |



[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](#)

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



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



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.



Revised⁵ 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.

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

 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.

[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](#)

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.

[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](#)



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.



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



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.

[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](#)

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.



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