Bidirectional parsing

a functional/logic perspective

Peter Kourzanov

NXP Eindhoven/TU Delft, Netherlands

kourzanov@acm.org

Abstract

We introduce PURE[3], a pure declarative approach to implementing declarative transformations with declarative tools. This Domain-Specific Language (DSL), inspired by the Definite Clause Grammar (DCG) and Parsing Expression Grammar (PEG) formalisms, is implemented using the Revised[5] Report on the Algorithmic Language Scheme (R5RS). Thanks to its use of the MINIKANREN logic programming system it supports fully reversible and extensible syntax-semantics relations. In this paper we highlight the usability and simplicity of PURE[3]'s approach, address the problem of *left-recursion* and show how its features help in defining custom and extensible typing systems for JavaScript Object Notation (JSON).

Categories and Subject Descriptors D.1.6 [*Programming techniques*]: Logic programming; D.3.2 [*Language Classifications*]: Applicative (Functional) languages; D.3.4 [*Processors*]: Parsing

General Terms (embedded) domain specific languages, automatic program generation, type systems/checking/inferencing

1. Introduction

The declarative approach to programming unifies logic/relational and functional communities in the shared vision of tools that need only be told *what* should be done rather than *how* that must be accomplished.[1] Ideally, these tools should (inter)actively participate in the creative process, i.e., *art* of computer programming by performing *parsing* of the human input, *checking* and *inference* [DM82] of the various properties of such inputs,[2] manipulation, refactoring and optimization of programs [BD77], compilation to machine code [App06] and last but not least, should provide feedback to the user.

Because humans inevitably are still in the loop of this development cycle, it is important that each stage remains palpable - that is, can be understood *semantically* and manipulated using *syntactically* simple terms. First and foremost this concerns parsing, a well-researched domain where many well-established methods exist [ALSU06] and [GBJL02], and yet, very few practical tools possess that elusive mathematical elegance that can immediately appeal to practitioners. Further down the transformation chain, complexity quickly rises and at the level of inference already presents formidable challenges [Wel94] to human understanding.

In this paper we present PURE[3] as a declarative approach to declarative *transformations* using declarative tools. The focus is on parsing as a particular kind of transformation of a linear stream of tokens into a set of Abstract Syntax Tree (AST) instances containing terminals (literal values), non-terminals (expressions, statements), types, assembly etc.

The approach is declarative in that we take the Backus-Naur Formalism (BNF) as a starting point and do not restrict ourselves to a specific way of codifying it. The transformations are declarative because they stay largely independent from the evaluation strategy. The tools are declarative in that we take MINIKANREN [FBK], a logic programming system embedded in R5RS [ABB+98] and abstain from overusing the extra-logical features that are available.

We shall first use a running example of an expression grammar/parser to explain our technical contributions and then switch to an *extensible* JSON grammar/parser to successively illustrate parsing, type checking, type inference and generation of syntax-semantics pairs constrained by types, all within a single framework.

Our main contributions are: the clean-room declarative implementation of PURE[3] (using the hygienic `syntax-rules` macro system and MINIKANREN) relying on *naturally* declarative semantics:

- featuring logical laziness,
- (full) reversibility-by-default,
- on-line behavior for left-recursion, and
- binding schemes for controlled (weak) hygiene "breaking"

This paper is structured as a flow that first addresses the background aspects in the introduction, explains the ideas and the implementation of the new formalism in section 2 and then highlights the use of the formalism by specifying an admittedly simple, yet concise and flexible typing system in section 3. The problems in implementing and using extensible transformations are addressed in section 4. Related work is reviewed in section 5, while the conclusions can be found in section 6. Full implementation using BIGLOO and conforming to R5RS plus two relevant SRFI libraries is available at github [Kou].[3]

1.1 Definite Clause Grammars

DCG is a technique originating in Prolog that allows one to embed a parser for a context-sensitive language into logic programming,

[1] we set machine learning community aside for now

[2] this is commonly known as *type*-checking and *type*-inference

[3] note that our use of R5RS is flavored by macro-expressible pattern-matching as well as a few syntactic liberties for recursive (`def`) and non-recursive (`defn`) bindings, brackets and lexical syntax (viz. *reader-macros*)

via Horn clauses. Logic programming languages such as Prolog and MINIKANREN also support relational programming. Instead of *functions* and *procedures* there are *predicates* that specify relations between terms. Rather than enforcing a particular way of evaluation, these languages specify a *resolution* (i.e., a search) procedure that can be applied and controlled in many ways. We explain the way how this is done in MINIKANREN in section 2.1. These features imply that a carefully designed grammar/parser can be run forwards (i.e., generating semantics from syntax), backwards (i.e., generating syntax from semantics) and sideways (e.g., constrained generation of syntax-semantics pairs).

A particularly nice feature of DCGs is its declarative nature and yet *executable* semantics [PW80]. This can be seen in the BNF specification as well as in the following Prolog code for a trivial context-free grammar/recognizer with precedence below.

```
<factor> ::= <literal> | <factor> '^' <literal>
<term> ::= <factor> | <term> '*' <factor>
                    | <term> '/' <factor>
<expr> ::= <term> | <expr> '+' <term>
                  | <expr> '-' <term>
```

Assuming a suitable definition of the `literal` predicate, the BNF can be automatically converted to the corresponding Prolog DCG rules, or, as shall be shown in section 2.2, to R5RS and MINIKANREN using the `syntax-rules`. Both kinds of encodings are "almost" directly executable, modulo *left-recursion* - a problem that plagues many recursive descent systems, and which we address by a novel technique of *logical laziness* in section 2.4.

```
%% An ideal Prolog DCG for a trivial expression grammar
factor --> factor, [^], literal.
factor --> literal.
term --> term, [*], factor.
term --> term, [/], factor.
term --> factor.
expr --> expr, [+], term.
expr --> expr, [-], term.
expr --> term.
```

As shall be become apparent shortly, the DCGs are more powerful than just Chomsky Type-2 systems (context-free grammars, or non-deterministic push-down automata) and in fact can express attribute grammars by allowing the predicates to take arguments that are used to compute variables bottom-up (i.e., a feature identical to synthesized attributes) or to generate and pass around non-instantiated variables (i.e., a feature identical to inherited attributes). This opens the door to concise [FBK05], declarative specification of *typing* systems, relational interpreters [Byr10] as well as a way towards a *practical* DSL for bidirectional transformations.

1.2 Parsing Expression Grammars

This grammar formalism [For04] dispenses with complexities of LL/LR grammars, takes a step back to recursive descent, and then extends it with a few combinators inspired by Type-3, *regular* languages. In addition, the PEG formalism introduces syntactic *and*- and *not*-predicates as well as *prioritized* choice (used for grammar *disambiguation*). This is an improvement over plain recursive descent because explicit recursion is often avoided (by turning it into *primitive* recursion via the Kleene-$*$ and $+$ operators).

One nice aspect of PEGs is better surface syntax for common patterns of programming parsers and transformations. For example, the `expr` predicate from the section above can be concisely specified as the following recognizer.

```
(pcg expr ([] ⇔ [term] [(['+ / '-] : [term]) *]))
```

Looking ahead, we might define a recognizer for a context-sensitive language (using our `pcg` rules introduced in the next section) with PEG combinators and syntactic predicates as follows:

```
;; A context-sensitive grammar with PEG combinators
(defn a^nb^na^n (pcg ⇔ S
  (S ([] ⇔ when([A] 'a) ('a +) [B] unless(['a / 'b])))
  (A ([] ⇔ 'a ([A] ?) 'b))
  (B ([] ⇔ 'b ([B] ?) 'a))
  ))
```

It is apparent that PEGs and DCGs share many of the same benefits and shortcomings. Syntactic predicates as well as the prioritized/ordered choice of PEG are naturally expressible as *committed* choice in logic programming. Left-recursion, however, is still troublesome and has to be either *avoided*, *eliminated* or solved by *ad-hoc* methods such as *cancellation tokens*, *curtailment* or *memoing* (see section 5).

2. Parsing Clause Grammars

In this chapter we introduce our implementation of DCGs which we dub the Parsing Clause Grammar (PCG) as a tribute to the other source of inspiration, the PEG. First, we introduce MINIKANREN and provide a way to specify first-order *predicates* concisely using only the `syntax-rules` of R5RS. Then we show a few examples of the usefulness of *higher-order* predicates. Controlled access to hygiene (i.e., weak hygiene "breaking") is then used to let `syntax-rules` implement an equational theory for name bindings across disparate code fragments. Finally, we highlight PCG's support for left-recursion in a pure, on-line fashion.

We make use of the macro-expressible pattern-matching (introduced in [KS13]) in BIGLOO - a practical implementation of R5RS [SW95]. The Scheme reader is extended with reader macros via `set-sharp-read-syntax!` (see the Chicken Scheme wiki: Unit library [Sch]) and provides a handler for `#h` form (see section 4 for a few examples) in order to obtain a stream of lexical tokens unconstrained by the conventional Scheme syntax.

```
;; A recognizer, each clause a separate predicate
(pcg Factor
   ([] ⇔ [Factor] ^ [literal])
   ([] ⇔ [literal]))
(pcg Term
   ([] ⇔ [Term] * [Factor])
   ([] ⇔ [Term] / [Factor])
   ([] ⇔ [Factor]))
(pcg Expr
   ([] ⇔ [Expr] + [Term])
   ([] ⇔ [Expr] - [Term])
   ([] ⇔ [Term]))
```

The translation of the expression grammar from the previous section is straightforward with the `pcg` macro and is given above. It defines several clause groups and binds a given name to the predicate/function implementing a disjunction for each group. Please see the code in section 2.4 for an illustration of MINIKANREN code that is automatically generated for the `Expr` part of this recognizer.

- we assume that the Scheme `read` procedure has performed lexical analysis on the input, that is, we deal only with syntactic and semantic analysis of tokens produced by the reader
- BNF *terminals* are assumed to be interned Scheme atoms such as literals (`#true` and `#false`), numbers, "strings" and `'symbols`, which might include characters such as `([,]{.})` when wrapped in `|`vertical bars`|`. Terminals are *auto-quoted*.
- BNF *non-terminals* are translated to MINIKANREN predicates (which are just regular, pure Scheme functions), where the first

two arguments represent PCG monadic state, the *difference-list*. Note that unlike original DCGs we *prepend* the pair of `Lin`/`Lout` variables comprising the diff-list at the beginning of the argument list because our predicates are possibly *variadic*

2.1 Declarative logic programming with MINIKANREN

In this section we briefly introduce the way in which we use MINIKANREN's primitives [FBK05] such as *success* and *failure* (`#s` and `#u`), binding of logic variables (`fresh`), unification ($\equiv$), disjunctions (fair choice `conde`, *soft-cutting* `conda`, committed choice `condu`), conjunctions (`all`), impure predicates (`project` for reifying variables) and finally `run/run*` that provide the interface between Scheme and the non-determinism monad that lies at the heart of MINIKANREN.

```
;; the swiss army knife of logic programming
(def append^0 (predicate
 ([_ '() b b])
 ([_ '(,x . ,a1) b '(,x . ,c1)] :- [append^0 a1 b c1])
))
```

Predicates are introduced by either `predicate` or `pcg` macros (these share many design aspects), and may have many clauses inside. Each clause contains a *head* followed by an optional *body*. We borrow the syntax from Prolog, separate the head from the body by a (:-) form and introduce an *implicit* disjunction between all clauses. By convention shared with `syntax-rules`, `predicate` clause heads may begin with any *tag* identifying the clause or with just a wildcard `[_]` while `pcg` clause heads (e.g., for recognizers) may be empty `[]`, in which case they don't unify any passed arguments. If the `pcg` head is not empty but contains only the `[_]` tag then the (thus variadic) predicate will unify exactly one argument with each consumed token in the input, point-wise.

```
;; e is somewhere in t ;; using explicit disjunction
(def member^0 (predicate  (def member^0 (predicate
 ([_ e '()] :- #u)         ([_ e '()] :- #u)
 ([_ e '(,e . ,t)])        ([_ e '(,h . ,t)] :-
 ([_ e '(,h . ,t)] :-        ([≡ e h] / [member^0 e t]))
  [member^0 e t])))        ))
```

By design shared with MINIKANREN, juxtaposition of goals (in the body) and clause attributes (in the head) corresponds to the conjunction. As is observed from 2 versions of the `member`^{0} predicate above, *explicit* PEG-style disjunction in clause bodies is often essential,[4] avoiding duplication of clause bodies and heads.

In contrast to MINIKANREN, PURE^3 advocates Prolog-style *automatic* inference of bindings. Unlike Prolog, however, in all `predicate` examples, variable names are extracted from clause heads and then are equated with the corresponding bindings from clause bodies using the Term-Rewriting System (TRS) equational theory that is explained in section 2.3.

Because of this, no binding can be used in a clause body without it being mentioned first in the clause head, which enforces fully reversible predicates which are "correct-by-construction". For some predicates, there may be *fresh* bindings introduced in the head but not used in the body (e.g., `fresh`^{0} in section 2.4) or there may be bindings (see `locals:` spec) that are not explicitly named in the head but used in the body to build some synthesized attribute that is mentioned in the head (e.g., the `prefix`^{0} in section 2.4)

2.2 Macro-expressibility of PCG rules

The `pcg` macro builds upon the structure introduced in the previous section and provides (1) *natural* representation of the syntax for terms of the expression grammar - to the right of $\Leftrightarrow$, (2) *natural* representation of semantics, i.e., an AST - to the left of $\Leftrightarrow$, (3) direct-style operator *associativity* and *precedence* and (4) inverse for free (note that we separate the clause head from the clause body by $\Leftrightarrow$ to indicate full reversibility). Our final version of a reversible syntax-semantics relation for expressions is given in figure 1.

```
(pcg
(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])
))
```

Figure 1. Pure, declarative PCG parser analyzer

The design of PCG (see figure 2) is centered around a set of `syntax-rules` macros: `seq` for processing clause bodies, `process-args` for clause heads, `predicate` and `pcg` that glue everything together.[5]

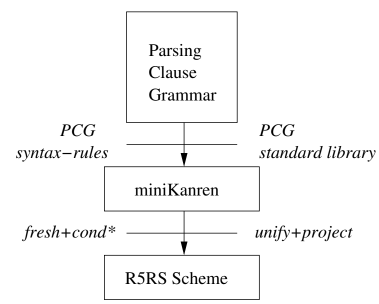

Figure 2. PURE^3 DSL architecture

The `seq` macro (see figure 5) implements the *threading* of a difference-list, per-clause sub-goal sequencing, introduction of a new logical temporary for each step and dispatching on the *shape* of forms encountered as sub-goals (non-terminals, quasi-data, atoms, escapes, ϵ, PEG combinators). By the very nature of hygienic `syntax-rules`, both components of the difference list (i.e., monadic state bindings) as well as all logical temporaries can not leak to user code, making the PCG formalism safe. This macro also performs a few optimizations such as skipping the introduction of a new logical temporary at the end of the sub-goal list.

Since each temporary is introduced by a different invocation of the `seq` macro, and yet 2 bindings get referred to by the generated code at each step (see section 2.4 for an example), an expander[6] compatible with the Scheme Request for Implementation (SRFI)#46: "Basic Syntax-rules Extensions" [Cam05] shall automatically rename it, while gratuitous bindings thus introduced shall be removed by the BIGLOO compiler's constant β-reduction pass, as they are immediately shadowed by the `fresh` binder. The `syntax-rules` therefore give us gratis, *pure*, declarative `gensym`!

Both `predicate` and `pcg` flavors of our `syntax-rules` macros support *named* (see the `pcg expr` from section 1.2) as well as

[4] note that our disjunction is pure (`conde`) by default. Soft-cut resp. committed/ordered choice are introduced explicitly by `*->` resp. `->` combinators

[5] due to space limits we can only refer to snippets of these in the appendix

[6] we use the ALEXPANDER library ported to, and integrated with BIGLOO as it still lacks a native and compatible `syntax-rules` expander

anonymous (see e.g., section 2.1) predicate abstractions. In addition, pcg macros allow specification of a group of possibly mutually recursive predicates where each is named and visible from the top-level (see figure 1), as well as a group where a distinguished predicate is selected as a *start* predicate (see the $a^n b^n a^n$ recognizer from section 1.2) with the rest hidden from the top-level.

2.2.1 Higher-order rules

Since MINIKANREN predicates are represented by normal Scheme functions, all the benefits of working in a Functional Programming (FP) language are retained. The `ne-list` predicate shown below supports repeated matching of the user-supplied `elem` predicate, with the literal represented by the value of the `comma` argument matched as a list separator.[7]

```
;; ... Passing functions into predicates ...
;; Monomorphic lists (for JSON), used in section 3.1
(defn [ne-list comma elem] (pcg ⇔ s
 (s ([_ `(,v)] ⇔ [elem v])
    ([_ `(,v . ,vs)] ⇔
      [elem v] [idem comma] [s vs])
)))
```

The recursion works out of the box for predicates such as `ne-list`, which employ right-recursion. However, some grammars such as the one from the notorious expression parser of figure 1, need to use left-recursion if the associativity of operators and the naturality of the parser representation is to be maintained. We shall present a "logical" solution for this problem in section 2.4.

```
;; ... Returning functions from predicates ...
;; Left-recursion avoidance (higher-order patching)
(pcg 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])
)
```

An example of a "functional" solution would be left-recursion avoidance by returning functions from higher-order predicates [For02]. A pcg representation of the `Factor` fragment of the expression grammar illustrating this technique is shown above. Note the similarity of this to the emulation of `fold-left` by `fold-right` (see e.g., [Hut99]) and the technique introduced in [DG05]. The use of the impure `project` (π) form,[8] which requires that variables are *grounded* (i.e., instantiated), precludes the use of this predicate in reverse.

2.3 Breaking hygiene (look ma, no `gensym`)

In section 2.1 we explained *why* PURE[3] uses inference of logic variable bindings in order to promote (full) reversibility and to avoid code clutter by explicit `fresh` introductions (see section 2.4 for a convincing case). In this section we show *how* inference can be implemented in our `process-args` macro by "breaking" the weak hygiene of `syntax-rules`.

It is well known that the promise of `syntax-rules` never to cause the capturing of bindings (hygiene) can be subverted [Kis02]. The `extract` and `extract*` macros implementing the so-called Petrofsky's extraction are typically used to capture the bindings regardless of their *color* (scope information) and pass them further to the other macros. The feature of `syntax-rules` that makes this possible is the semantics of macro literals [ABB+98].

A first step towards an equational theory of name binding across disparate code fragments using a TRS consists of extracting the *free* variables from a tree of terms (`w` `syntax-rules` macro). We assume the *weak* hygiene where all bindings are intended to be local and are not redefined outside of the terms being processed. This is exactly the same assumption that `extract` macro makes (see [Kis02] for further details).

```
;; Ignoring (some) Scheme primitives
(def-syntax (scheme-bindings (k a b [s ...] . d))
   (k a b [s ... if cond begin null? list first second
                pair? car cdr + - * / ^ = ≡ : ...] . d))
```

Of course, all binders must be known to this macro (and names thus introduced must be skipped in appropriate scopes), in addition to all of the *eigen* primitives that must not be considered free in given terms. This is accomplished using the macro given above, which employs the macro-level Continuation Passing Style (CPS) [HF00] to bootstrap the `w` macro by including common Scheme primitives in a list of bindings already processed.

2.3.1 Handling attributes

In the process of generating `fresh` and `projected` predicate arguments for the inferred attribute bindings we need to make sure that the bindings given to the binder correspond to the bindings captured from the body. If there are no bindings then we default to some construct like `begin` or `all`. For `project`, we verify that all attributes are grounded and vacuously succeed otherwise.

```
;; Introducing the fresh and project binders
(def-syntax make-scopes (syntax-rules (project)
  ([_ _ _ _] #s) ;; nothing to do - just succeed
  ([_ _ () default . body] (default . body))
  ([_ project (var ...) _ . body]
    (project (var ...)
      (or (and (ground? var) ... . body) #s)
   ))
  ([_ binder vars _ . body]
   (let-syntax-rule ([K args terms]
     (binder args . terms))
      (extract* vars body (K [] body))
   ))
  ))
```

Now we're ready to complete the third step: attacking the `process-args` macro, which makes sure that the resolution of the synthesized attributes in the clause head, the clause body, logical actions, and finally - evaluation of `projected` code - are all scheduled appropriately. Correct sequencing is essential for reversibility - when run in reverse the synthesized attributes actually provide the inputs, and hence must resolve first. When running forwards, resolving them first does no harm, since clause heads can only indirectly employ constructs made available through the `process-args` implementation - unification using quasi-data and conjunction using `all` (disjunctions and recursion are *not* available inside clause heads). Projections (π) can only run in forward mode and always after resolving the clause body.

Reversible logical actions (not explained in this paper in detail, but see section 3.2 for a use-case) have to run after clause body terms when running forwards but before when run in reverse. Non-reversible actions, as implemented by escapes in the `seq` macro, are left to be explicitly scheduled by the user in clause bodies.

We ignore (inherited) attributes that are explicitly bound from outside when looking for free bindings, but do include them into the list of attributes to generate using the `make-scopes` macro. Each attribute in the clause head gets unified with the respective argument of the clause function,[9] while the final *void* attribute tail gets syntax-bound (i.e., renamed) to `'()` as is customary in Scheme variadic functions.

[7] via explicit lifting using the `idem` predicate, defined in the appendix

[8] the straightforward implementation of π is elided in this paper

[9] using a technique similar to the `seq` macro described above

Now we can bring together the full reversibility-by-default and attribute bindings inference (as implemented by the `seq` and `process-args` macros), and actually complete our TRS for the predicate clause forms as it is implemented by the `pcg/predicate` macro. Each clause is translated to a separate R5RS function which implements all aforementioned aspects of the corresponding predicate logic. Despite the seeming restriction that each clause is visible from the top-level, Scheme's `define` form actually is macro-expressible by the `letrec` binder when it precedes all other forms in a block. This is useful for implementing the `pcg` variants that hide internal predicates and expose a single starting predicate function to the top-level.[10]

Synthesized attributes Similar to the *S-attributed* grammars, where inherited attributes are not allowed, PURE^3 also is tuned for seamless expression of grammars with only synthesized attributes. In fact, all examples introduced so far used no `local:` attribute specifications, inferring the attributes in clause bodies from clause heads. Together with the ban on `project` (π), this enforces the fully reversible behavior in a "correct-by-construction" way.

Inherited attributes Having no possibility of generating and passing non-instantiated logic variables severely restricts the expressiveness of the formalism. There are examples of when such a strategy for implementing semantics is essential, e.g., predicates from section 3.2. Here we illustrate PURE^3's implementation of inherited attributes using a particular way of solving left-recursion by left-factoring that is commonly used in e.g., Prolog community.

```
;; Left-recursion elimination by left-factoring
(defn exprs (pcg ⇔ expr
(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))
))
```

Left-factoring is usually understood as the process of introducing additional predicates for matching common prefix terms of a number of clauses and then factoring them out from the original predicates. Although most often used for optimization, this technique proves helpful in converting left-recursion into right-recursion. One has to be careful, however, not to change the associativity of the operators when applying left-factoring.

Let us implement left-recursion elimination using PCG. Here, the parent predicate (e.g., `expr'`) binds a fresh variable for the inherited attribute using the `locals:` spec and then passes it to the child predicate (e.g., `term`) which resolves the attribute and communicates it to its sibling. The recursive call then unifies the semantics with a newly formed AST node upon completion. Note that this bears strong resemblance to the *L-attributed* grammars.

Although PURE^3 definitely supports it, this approach does not possess naturally declarative semantics, since the semantic actions are now interwoven with the syntax. Also, similarly to left-recursion avoidance in section 2.2.1, the attribute resolution is delayed until the end of the input, which prohibits on-line, incremental parsing. Section 2.4 presents a better approach to left-recursion.

[10] we refer to our github for further details about `select` and `pcg`

2.3.2 Handling binding in combinators

Consider the semantics of Scheme's `(-)` function: it is left-associative and accepts non-zero number of arguments. One might define both the syntax and semantics of `(-)` using the PEG's $*$ combinator as follows, assuming the variadic handling of the operator in the AST:

```
;; Minus in Scheme has arity >= 1
(pcg -' ([_ `(- ,t . ,ts)]⇔[term t] [('- : [term ts])*]))
```

Note that this avoids explicit recursion. However, now the $*$ combinator has to collect all elements of the input matching the `term ts` predicate invocation into a *list*, while each invocation of the `term` predicate still unifies with a single term only. This implies that representing each attribute as it is being synthesized with a single logical variable is not sufficient. In fact, we need 4 logical variables for each attribute: one for returning the final result, one when matching on each `term`, one for the accumulator and another one for the intermediate results as needed for *looping* in the `seq` rule that implements the Kleene-$*$ operator.

2.4 Pure, on-line left-recursion

Now we're well-equipped to attack left-recursion in PCG rules by applying the technique of logical laziness. This problem arises due to the infinite regress when a predicate recurses while not having consumed anything from the input. Still, direct-style associativity prevents us from applying left-recursion elimination or avoidance while the need to support left-recursion in a pure, on-line and fully reversible fashion precludes usage of impure techniques such as curtailment or memoing. Looks like we're stuck.

```
;; Diverging R5RS code generated for Expr (+) clause:
(define _head_422 (λ (Lin Lout . vars)
  (fresh (y x) (≡ vars (cons (list '+ x y) '()))
     (fresh (_temp_505 temp)
        (Expr Lin temp x)
        (≡ temp (cons '+ _temp_505))
        (Term _temp_505 Lout y)
      ))))
;; ... _head_424 and _head_426 elided ...
(def Expr (λ vs (conde ((apply ([extend] 'Expr) vs))
 ((apply _head_422 vs)) ;; ([_ `(+ ,x ,y)] ⇔ ...)
 ((apply _head_424 vs)) ;; ([_ `(- ,x ,y)] ⇔ ...)
 ((apply _head_426 vs)) ;; ([_ x] ⇔ ...)
)))
```

Looking at the diverging generated code for the `Expr` grammar fragment from figure 1 (see above) we immediately observe the problem: the base case of this non-well-founded recursion (`_head_426`) is never reached. Fortunately, the problem has a surprisingly simple solution which we dub "logical laziness". It comprises several steps all of which are automated using the `pcg` and the `seq` - both pure and declarative `syntax-rules` macros.

1. identifying (mutually) recursive clauses (section 2.2)
2. marking of such clauses using the *lift* form (can also be explicitly marked by the user if needed, as e.g., in section 4.3)
3. *untying* the recursive knot and insertion of replacement calls to append^0 (section 2.1) with dummies (variable `d` below)
4. delaying of the resolution and subsequent *dropping* of recursive calls at the very end (or very beginning) of a clause body
5. *tying* the knot by unifying the difference list with `[d '()]`

In essence, here we apply a *predicate transformation* where the order of predicate resolution is adapted to suit the resolution procedure. The grammar designer is not required to apply workarounds for left-recursion and can regain a high-level view as in figure 1, as long as the set of mutually recursive predicate clauses can be automatically identified by our `pcg` macro. The code below that is generated for the same grammar fragment exhibits the technique.

```
;; Reversible R5RS code generated for Expr clause:
;; ([_ `(+ ,x ,y)] ⇔ [Expr x] + [Term y])
(define _head_422 (λ (Lin Lout . vars)
 (fresh (y x) (≡ vars (cons (list '+ x y) '()))
   (fresh (_temp_505 temp d)
    (project (Lin) (if (ground? Lin) #s (Expr d '() x)))
    (append^0 d temp Lin)
    (≡ temp (cons '+ _temp_505))
    (Term _temp_505 Lout y)
    (project (Lin) (if (ground? Lin) (Expr d '() x) #s))
))))
```

This technique has the advantage of maintaining both naturality of the grammar as well as full reversibility of the resulting parser. When the input `Lin` is grounded (i.e., the parser is running forwards), the recursive call is delayed to the very end of the clause, effectively making it *tail-recursive*. When the parser is running backwards (i.e., the input `Lin` is fresh), the recursive call has to run first in the clause, because otherwise the recursion becomes non-well-founded, this time due to semantic destructuring of the `vars`.

```
;; (prefixed) infinite streams of logic variables
(def (fresh^0 x)
  (conde ([≡ x '()])
     (else (fresh (y z)
       (fresh^0 z)
       (≡ x `(,y . ,z))
      ))))
(defn (prefix^0 a b)
   (fresh (x)
      (fresh^0 x)
      (append^0 a x b)
      ))
```

```
(def fresh^0 (predicate
  ([_ '()])
  ([_ `(,y . ,z)] :-
    (fresh^0 z)
  )))
(defn prefix^0
 (predicate locals: (x)
  ([_ a b] :- (fresh^0 x)
    (append^0 a x b)
  )))
```

Does this work as promised, in an on-line fashion? Lets generate an infinite input using predicates above (where both MINIKANREN and PURE[3] versions are given) and `verify` by `running`!

```
;; Parsing prefixed infinite input stream
(verify Expr (run 1 (q)
  (fresh (l)
     (prefix^0 '(1 * 2 + 3 * 5) l)
     (Expr l '() q)))
 ===> (+ (* 1 2) (* 3 5)))
```

One disadvantage of this technique is the *non-determinism* (and thus a quadratic-time slowdown) introduced by the `append`^{0}. In practice, however, this is not problematic because non-determinism often is and/or can be easily constrained by putting the limit to the *lookahead* by tokens that immediately follow the recursive call.

3. Type systems a la carte

Having introduced all the tools necessary to attack a more practical problem of adding types to a fully reversible JSON parser, we now turn to figure 3, which depicts a type-free implementation of the JSON syntax.[11] This PCG is our starting point for this section.

The semantics is represented by the AST where JSON literals remain strings, symbols and numbers. Name-value pairs become List Processing (LISP) pairs, while arrays and objects turn into

```
(pcg
 (json-symbol ([_ x] ⇔ [strings x]))
 (json-key = json-symbol)
 (json-number = number)
 (json-bool ([_] ⇔ ('true / 'false)))
 (json-value ([_] ⇔ 'null)
  ([_ x] ⇔ [json-bool x])
  ([_ x] ⇔ ([json-symbol x] / [json-number x]))
  ([_ x] ⇔ ([json-object x] / [json-array x])))
 (json-pair ([_ `(,n . ,v)] ⇔
  [json-key n] |:| [json-value v]))
 (json-value-list ([_ '()] ⇔ ϵ)
  ([_ l] ⇔ [(ne-list |,| json-value) l]))
 (json-pair-list ([_ '()] ⇔ ϵ)
  ([_ l] ⇔ [(ne-list |,| json-pair) l]))
 (json-array
  ([_ `(arr . ,es)] ⇔ |[| [json-value-list es] |]|))
 (json-object
  ([_ `(obj . ,es)] ⇔ |{| [json-pair-list es] |}|))
)
```

Figure 3. Type-free JSON

explicitly tagged lists of JSON values. This grammar is interesting because of this recursion and the presence of various data-types.

3.1 Type checking

Base Types		Pairs, Lists	
$\dfrac{}{\Gamma\vdash\mathbf{null}{:}U}$	(UNIT)	$\dfrac{\mathrm{T}^0{=}\{U,B,S,N\},S{\times}\mathrm{T}^1}{\mathrm{T}^1{=}\mathrm{T}^0{\cup}\{A(\mathrm{T}^1),O(S,\mathrm{T}^1)\}}$	
$\dfrac{x{\in}\{\mathbf{true},\mathbf{false}\}}{\Gamma\vdash x{:}B}$	(BOOL)	$\dfrac{\forall t{:}\mathrm{T}^1,\Gamma\vdash v_1,\ldots v_n{:}t}{\Gamma\vdash[v_1,\ldots v_n]{:}A(t)}$	(ARRI)
$\dfrac{x{\in}\mathrm{Strings}}{\Gamma\vdash x{:}S}$	(STR)	$\dfrac{\forall t{:}\mathrm{T}^1,\Gamma\vdash k{:}S,\Gamma\vdash v{:}t}{\Gamma\vdash(k,v){:}S{\times}t}$	(PAIRI)
$\dfrac{x{\in}\mathbb{R}}{\Gamma\vdash x{:}N}$	(NUM)	$\dfrac{\forall t{:}\mathrm{T}^1,\Gamma\vdash v_1,\ldots v_n{:}S{\times}t}{\Gamma\vdash\{v_1,\ldots v_n\}{:}O(S,t)}$	(OBJI)

Table 1. Typing rules for monomorphic JSON

Thanks to the availability of unification, addition of types to the grammars using PURE[3] is easy. In figure 4 on the left-hand side (i.e., the semantics), each clause is extended with an additional attribute, while the right-hand side (i.e., the syntax) is essentially not modified. The higher-order `ne-list` predicate (section 2.2.1) is not touched for the *monomorphic* lists, a variant for introducing types in JSON that ensures homogeneity of objects and arrays through *static* typing.

```
;; Monomorphic JSON lists (i.e., arrays and objects)
(pcg            ;; replace these in Figure 4. below
(json-value-list
  ([_ '() `(List ,t)] ⇔ ϵ)
  ([_ l `(List ,t)] ⇔
    [(ne-list |,| (_ json-value t)) l]))
(json-pair-list
  ([_ '() `(PList (,t1 ,t2))] ⇔ ϵ)
  ([_ l `(PList (,t1 ,t2))] ⇔
    [(ne-list |,| (_ json-pair `[Pair ,t1 ,t2])) l])
))
```

Conventional typing rules are given in table 1 while their translation to PURE[3] is a straightforward extension of rules from figure 3 with typed versions of `json-value-list` and `json-pair-list` predicates. The code above implements the ARRI, PAIRI and OBJI typing rules (the rest of the rules can be found in figure 4). We rely on *sectioning* to partially apply the `json-value` and `json-pair` predicates to known types, and to introduce type schemes (containing "fresh" types) for empty containers. The logical unification ensures type correctness by applying the same types throughout the lists once they are instantiated, leading to monomorphic containers.

[11] we rely on BIGLOO reader for symbol, string and number parsing

```
(defn tjson-value (pcg ⇔ json-value
 (json-symbol extend: extend ([_ x 'Str] ⇔
  [strings x]))
 (json-key extend: extend ([_ x t] ⇔
  [json-symbol x t]))
 (json-number ([_ x 'Num] ⇔ [number x]))
 (json-value ([_ 'null 'Unit] ⇔ 'null)
  ([_ x 'Bool] ⇔ [json-bool x])
  ([_ x t] ⇔ ([json-symbol x t] / [json-number x t]))
  ([_ x t] ⇔ ([json-object x t] / [json-array x t])))
 (json-pair ([_ `(,n . ,v) `(Pair ,tn ,tv)] ⇔
  [json-key n tn] |:| [json-value v tv]))
 (json-value-list ([_ '() `(List ,t)] ⇔ ϵ)
  ([_ l `(List ,ts)] ⇔
   [(poly-ne-list |,| json-value) l ts]))
 (json-pair-list ([_ '() `(PList (,t1 ,t2))] ⇔ ϵ)
  ([_ l `(PList . ,ts)] ⇔
   [(poly-ne-list |,| json-pair) l ts]))
 (json-array ;; promote List to an Array
  ([_ `(arr . ,es) `(Array ,t)] ⇔
   |[| [json-value-list es `(List ,t)] |]|))
 (json-object ;; promote List to an Object
  ([_ `(obj . ,es) `(Object ,ts)] ⇔
   |{| [json-pair-list es `(PList . ,ts)] |}|))
))
```

Figure 4. Typed, extensible JSON

Note that we are making the `json-symbol` and `json-key` predicates extensible via `extend:` spec. This shall prove its usefulness in section 4 where we extend the set of JSON values by proper symbols and JSON keys by numbers and booleans while preserving the modularity and compositionality of the grammar above.

3.2 Type inference

The previous section has introduced type-checking to JSON for monomorphic containers denoting objects such as arrays of numbers or objects of pairs of strings. Such types can be either checked (provided they are given as instantiated attributes to the `tjson-value` predicate), or inferred (provided they are given as free logic variables). However, for homogeneous arrays and objects, inference in this context will typically mean that the type shall be derived using the first element (pair), with the rest of the elements simply checked against the already instantiated type.

$\mathrm{T}^2{=}\mathrm{T}^1\cup\{\sum(\mathrm{T}^2,\mathrm{T}^2)\}$		$\dfrac{\exists i{:}t{=}t_i}{\sum(t,\sum t_i){=}\sum t_i}$	(SUM1)
$\mathrm{T}^3{=}\mathrm{T}^1\cup\{A(\mathrm{T}^2),O(S,\mathrm{T}^2)\}$		$\dfrac{\forall i{:}t{\neq}t_i}{\sum(t,\sum t_i){=}\sum t_{i+1}}$	(SUM2)
$\mathrm{T}^2{\cong}\sum(\mathrm{T}^2)$	(VACU)		
$\dfrac{\forall t{\in}\mathrm{T}^2}{\Gamma{\vdash}[]{:}A(t)}$	(ARR0)	$\dfrac{\forall t{\in}\mathrm{T}^3,\Gamma{\vdash}x{:}t,y{:}A(\sum t_i)}{\Gamma{\vdash}[x]{\oplus}y{:}A(\sum(t,\sum t_i))}$	(ARR*)
$\dfrac{\forall t{\in}\mathrm{T}^2}{\Gamma{\vdash}\{\}{:}O(S,t)}$	(OBJ0)	$\dfrac{\forall t{\in}\mathrm{T}^3,\Gamma{\vdash}x{:}t,y{:}O(S,\sum t_i)}{\Gamma{\vdash}(s,x){\otimes}y{:}O(S,\sum(t,\sum t_i))}$	(OBJ*)

Table 2. Typing rules for polymorphic JSON

In this section we develop a more general notion of type inference for JSON. Rather than insisting on monomorphic lists, we allow polymorphism. The mechanisms included in PURE^3 support the declarative specification of a larger class of *polymorphic* typing rules. With such rules, a *sum-type* can appear in type terms, expressing the set of possible value-types that might appear in a given JSON list. The polymorphic typing rules are given in table 2.

To implement the SUM1 and SUM2 rules we introduce the following predicate, which handles injection and insertion of types into a set of types. This is easily accomplished using the *soft-cut*, or logical "if-then-else" construction (also available in many Prolog implementations). In the `sigma` predicate below, the type is first checked for membership in a given union representing a set of types. If found, the set is unified with the result. Otherwise, a new set that is formed by insertion of the new type is returned.

```
;; Working with types
(defn sigma (predicate locals: (ts')
 ([_ t `(Union . ,ts') `(Union . ,ts)] :-
  ([member^0 t ts'] *-> [≡ ts' ts]
   / [insert^0 t ts' ts]))
 ([_ t t' ts] :-
  ([≡ t t'] *-> [≡ t' ts]
   / ([! car^0 t' 'Union]
     :[≡ ts `(Union . ,ts')]
     :[insert^0 t `(,t') ts'])))
))
```

This predicate also keeps unions non-vacuous by collapsing singleton sets to their isomorphic member (VACU rule) using "soft-cuts" (`*->`) and enforces *flat* unions via "negation-as-failure" (`!`).

```
;; Polymorphic lists
(defn [poly-ne-list comma elem] (pcg ⇔ s
 (s locals: (t ts')
  ([_ `(,v) t] ⇔ [elem v t])
  ([_ `(,v . ,vs) ts] <=[(sigma t ts' ts)]=>
   [elem v t] [idem comma] [s vs ts'])
 )))
```

Because the PCG predicate that matches a list of JSON values is required to compute new types as it parses the input terms, it can not remain *homomorphic* in the presence of polymorphic types, as the `ne-list` predicate. A straightforward extension that uses the `sigma` predicate above, and which implements the ARR* and OBJ* rules from table 2 is shown above (`poly-ne-list` predicate). It utilizes a reversible logical action (using the `<=[actions ...]=>` syntax) to make sure that the type inference of the JSON values is performed in a fully reversible way (see section 2.3.1 above).

3.3 Term generation

We make no distinction between the parsing and the typing phases - both kinds of semantic attributes are computed together. Looking at the figure 4, we observe that types are just like other semantic attributes, obtained by removing some details from semantic terms (the AST), in a way that parallels *abstract interpretation* [CC77]. Using PURE^3, one might even compute a number of different types or kinds of semantics via unification of PCG attributes in parallel.

```
;; Constrained generation of syntax-semantics pairs
(verify enum1 (run* (q) (fresh (x) (tjson-value
          x '() q `(Object (Pair Bool Num)))))
 =>) ;; there are no terms of this type (yet)
;; there are infinitely many terms of this type
(verify enum2 (run 5 (q) (fresh (x) (tjson-value
          x '() q `(Object (Pair Str Num)))))
---> (obj ("a" . 0)) (obj ("b" . 0)) (obj ("a" . 1))
     (obj ("a" . 0) ("a" . 0)) (obj ("a" . 2)))
```

Because of this, the constrained generation of syntax-semantics pairs based on types is just a mode of running PCG predicates.

4. Extensibility

For JSON, one of common objections is the lack of human-friendly surface syntax. CSON [Lup], for example, dispenses with the need to quote all symbols as JSON strings. It is therefore useful to allow *contained* extensibility for the specification of DSL *families*.

The PCG formalism as introduced so far supports a rather rigid specification of syntax and semantics. Referential transparency of the pure predicates implies that any *local* change done to a grammar requires redefinition of all dependent predicates. Relying on implicit reference cells as used by Scheme's `define` primitive would not work with pcg rules that hide internals. Also, there are difficulties with this approach when running it on the BIGLOO interpreter as well as with BIGLOO's native, and Java Virtual Machine (JVM) back-ends, which disallow redefinition of procedures [Se].

In PURE[3] we would like to be able to introduce orthogonal (i.e., homomorphic) extensions to both syntax and semantics, in a compositional and modular fashion. For example, a natural extension of the `json-symbol` predicate to include proper symbols as JSON values should not require a reiteration of the full grammar from figure 4. Also, some DSLs would benefit from an external JSON-like representation of *sparse* arrays, i.e., maps where the object key (`json-key` clause) can be numeric rather than always only a string.

```
;; Allow symbols as values
(defn [tjson-ext-sym] (let ([extend' (extend)])
  (fn-with [apply extend'] | 'json-symbol =>
     (pcg ([_ x 'Sym] ⇔ [symbol x]))
  )))
;; Allow numbers and booleans as keys
(defn [tjson-ext-key] (let ([extend' (extend)])
  (fn-with [apply extend'] | 'json-key =>
     (pcg ([_ x 'Num] ⇔ [number x])
          ([_ x 'Bool] ⇔ [json-bool x]))
  )))
```

Using the anonymous functions with pattern-matching (as described in [KS13]), we can define extensions in a straightforward fashion, as shown above. Here we make use of *dynamic* binding mechanism implemented by the SRFI#39: "Parameter objects" [Fee03]. The current value of the `extend` parameter object is saved, and the tag passed to the extension call is checked. If it matches the extended predicate's name, then new clause(s) generated by the `pcg` macro are returned as the predicate extension. Otherwise, saved extension is invoked (via the `[apply extend']` handler).

This way, the predicates can be row-extended seamlessly. The `Expr` predicate function in the generated code from section 2.4 exposes the internals. As a first choice point introduced with MINIKANREN's `conde`, we invoke the `extend` parameter. By default, predicates returned by parameter objects `fail`, as below:

```
;; Implementing extensibility
(defn *def-extend* (λ (head) (λ (in out . results) #u)))
(defn extend [make-parameter *def-extend*])
```

In addition to the destructive updates expressed by calling the parameter object with a single argument, the SRFI#39 supports modifying dynamically scoped bindings with new values in a statically defined scope, specified via the `parameterize` form. This is useful for expressing local, contained changes to PCG grammars.

```
;; Using extensible syntax
(parameterize ([extend (tjson-ext-sym)])
 ;; forwards
 (verify test15.1 (run 1 (q) (fresh (x t) (tjson-value
           '#h: [{foo:quux},{bar:snarf},1] ^L '() x t)
            [≡ q `(,t ,x)]))
 ===> [(Array (Union (Object (Pair Sym Sym)) Num))
       (arr (obj (foo . quux)) (obj (bar . snarf)) 1)])
 ;; backwards
 (verify test15.2 (run* (q) (tjson-value
            q '() '[obj (a . (arr "b" "2.3"))]
            `(Object (Pair Sym (Array Str)))))
 ===> (|{| a |:| |[| "b" |,| "2.3" |]| |}|)
))
(parameterize ([extend (tjson-ext-key)])
 (verify test15.3 (run* (q) (fresh (x t) (tjson-value
           '#h: [{12: "quux"},{42: "snarf"}] ^L '() x t)
            [≡ q `(,t ,x)]))
 ===> [(Array (Object (Pair Num Str)))
       (arr (obj (12 . "quux")) (obj (42 . "snarf")))]
))
```

4.1 Chaining extensions

Often, extensions make sense only when applied together, as a group. PURE[3] supports expression of extension *dependencies* by static chaining of corresponding extension functions. This can be achieved by simply capturing the dependent extension rather than the current value of the parameter object in dependee's definition:

```
;; A contrived example of chaining ext-key to ext-sym
(defn [ext-key] (let ([extend' (ext-sym)]) ...))
```

4.2 Composing extensions

Because each extension hooks onto the current value of the `extend` parameter object, we can also compose such extensions in a natural way - simply by nesting appropriate `parameterize` scopes.

```
;; Composing JSON extensions
(parameterize ([extend (tjson-ext-sym)])
(parameterize ([extend (tjson-ext-key)])
 (verify test16 (run* (q) (fresh (x t) (tjson-value
           '#h: [{12:quux},{42:snarf}] ^L '() x t)
            [≡ q `(,t ,x)]))
 ===> [(Array (Object (Pair Num Sym)))
       (arr (obj (12 . quux)) (obj (42 . snarf)))])
 ))
```

While in section 4.1 the composition is static, here we apply dynamic resolution and chaining of extensions referenced by the `extend` parameter object. This improves the modularity for library-based implementation of DSL families that support localization, such as the JSON extensible parser grammar described here.

4.3 Power and danger

Combining extensions with committed choice (which can be forced by the `condo:` PCG modifier), one can use extensions to subvert existing grammar in a non-monotonic fashion. Although PCGs only seem to support row-extensibility, addition of new clauses that may take precedence over previous clauses is definitely possible. The ability to recursively refer to previously defined clauses enables extensions to the rows themselves. For example, the expression grammar can be extended with new operators as follows:[12]

```
;; Extending Term predicate
(defn Term+ (let ([extend' (extend)][T' Term])
 (fn-with [apply extend'] | 'Term =>
  (pcg ([_ π(@ x y)] ⇔ [lift T' x] @ [Factor y]))
)))
```

Because we allow impure logical code here, and because the escape to the Scheme level is made possible via MINIKANREN's `run` and `project` (π) primitives, the PCG predicates can be *extended while parsing*. For example, the `@` in the clause head above can be referring to any Scheme function or procedure, and that may perform any (effectful) computation. Since the SRFI#39 allows destructive/imperative update to the `extend` reference cell, this alone effectively makes the formalism *Turing-complete* (i.e., Chomsky Type-0). This can be easily seen by translation to vW-grammars, or 2-level grammars whereby infinite context-free grammars can be generated from a finite set of (Type-3, even) meta-rules [vW74].

5. Related work

A traditional approach to the problem of left-recursion is its effective *elimination* [HMU03]. The work that formalized PEGs *avoids* left-recursion by putting it outside of the set of well-formed grammars [For04]. With Prolog DCGs the problem is typically solved via *ad-hoc* methods such as *cancellation tokens* [NK93], *memoing*

[12] note that we need to explicitly *lift* the left-recursive call in this case

[BS08] or through elimination (see section 2.3.1 for a PCG rendering of this). Parser combinators are often applying *curtailment* [FH06] and [FHC08], an old idea to limit matches by the input length [Kun65]. The work on OMETA has also advocated *memoing* for solving left-recursion [WDM08].

Unlike prior art where programming was constrained to use only reversible compositions of bijections [RO10] or where the reverse transformations are derived from the forward transformations [Vis01] and OMETA [WP07], we maintain a *single* grammar/parser that can be used in different modes: forwards, backwards, sideways etc. Unlike the more restrictive formalism of *lenses* [FGM+05] and [BFP+08] we do not rely on carrying the original sources along but allow structural changes within the limits of the information theory.

Mode analysis, inference and scheduling of predicate resolutions has been addressed, e.g., in Prolog [DW88] and in Mercury [OSS02]. Our approach differs from those as it seamlessly integrates with an existing FP language rather than relies on an abstract interpretation framework implemented inside a dedicated compiler.

Of all proposals to improve the syntax of LISP going back to "M-expressions", "I-expressions" [MÖ5], *sweet* "t-expressions" [DAW13] and CGOL [Pra73] the method of "enforestation" [RF12] seems to be the closest to our approach. This work utilizes the Pratt operator precedence parsing (which is less general than PCG) that avoids rather than addresses issue of left-recursion.

6. Conclusions

Fully reversible syntax-semantics relations are enforced by a "correct-by-construction" inference of logical variable bindings from clause heads and equation of those with the bindings from clause bodies. The information is not *dissipated* by default, so the transformations remain reversible and in fact, might become very inexpensive to run [Lan00] in the future. Parts of the information may be *hidden*, which is useful for e.g., implementing updatable views or for keeping programs invisibly statically typed.

The novel technique of logical laziness allows us to retain the purely declarative style of on-line, left-recursive grammar specifications, without sacrificing either the direct-style associativity or the naturality of the syntax, semantics and typing specifications.

We offer one possible answer to the question about what hygiene of `syntax-rules` actually means [Kis02]: it implements the access to the name and binding in Scheme (i.e., scoping rules), whereby hygiene is maintained by default while still supporting equational theory of bindings across disparate code fragments. In effect, (weak) hygiene breaking `syntax-rules` should be seen as *specifications* of such theories [Her10] and not just as cool hacks.

PCGs are "macros no more": we see no need to use arcane, albeit elegant rewriting systems such as `syntax-rules` for programming in a homoiconic language such as Scheme. The syntax and the semantics are better specified using pure declarative methods of PURE[3], naturally expressing reversible (i.e., inferrable) types, providing declarative disambiguation operators, enabling on-line/incremental processing as well as providing support for essential error reporting and debugging interfaces for practical DSLs.

Acknowledgments

We would like to thank Oleg Kiselyov for noting the problem of left-recursion in on-line parsers. This paper has benefited from the discussions with Willam Byrd, Ralf Lämmel and Vadim Zaytsev.

References

[ABB+98] 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, 1998. URL: http://doi.acm.org/10.1145/290229.290234.

[ALSU06] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[App06] Andrew W. Appel. *Compiling with Continuations (corr. version)*. Cambridge University Press, 2006.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977. URL: http://doi.acm.org/10.1145/321992.321996.

[BFP+08] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *POPL*, pages 407–419, 2008. URL: http://doi.acm.org/10.1145/1328438.1328487.

[BS08] Ralph Becket and Zoltan Somogyi. Dcgs + memoing = packrat parsing but is it worth it? In *PADL*, pages 182–196, 2008. URL: http://dx.doi.org/10.1007/978-3-540-77442-6_13.

[Byr10] William E Byrd. *Relational programming in miniKanren: techniques, applications, and implementations*. PhD thesis, Department of Computer Science, Indiana University, 2010.

[Cam05] Taylor Campbell. Srfi 46: Basic syntax-rules extensions. Internet, 2005. URL: http://srfi.schemers.org/srfi-46.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252. ACM, 1977. URL: http://doi.acm.org/10.1145/512950.512973.

[DAW13] Alan Manuel K. Gloria David A. Wheeler. Srfi 49: Sweet-expressions (t-expressions). Internet, 2013. URL: http://srfi.schemers.org/srfi-110.

[DG05] Olivier Danvy and Mayer Goldberg. There and back again. *Fundam. Inform.*, 66(4):397–413, 2005.

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982. URL: http://doi.acm.org/10.1145/582153.582176.

[DW88] Saumya K. Debray and David Scott Warren. Automatic mode inference for logic programs. *J. Log. Program.*, 5(3):207–229, 1988.

[FBK] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. minikanren homepage. URL: http://minikanren.org.

[FBK05] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The reasoned schemer*. MIT Press, 2005.

[Fee03] Marc Feeley. Srfi 39: Parameter objects. Internet, 2003. URL: http://srfi.schemers.org/srfi-39.

[FGM+05] 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 *POPL*, pages 233–246, 2005. URL: http://doi.acm.org/10.1145/1040305.1040325.

[FH06] 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. URL: http://doi.acm.org/10.1145/1149982.1149988.

[FHC08] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *PADL*, pages 167–181, 2008. URL: http://dx.doi.org/10.1007/978-3-540-77442-6_12.

[For02] Bryan Ford. Packrat parsing: : simple, powerful, lazy, linear time, functional pearl. In *ICFP*, pages 36–47, 2002. URL: http://doi.acm.org/10.1145/581478.581483.

[For04] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL*, pages 111–122, 2004. URL: http://doi.acm.org/10.1145/964001.964011.

[GBJL02] Dick Grune, Henri E. Bal, Ceriel J. H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. John Wiley, 2002.

[Her10] David Herman. *A theory of typed hygienic macros*. PhD thesis, Northeastern University Boston, 2010.

[HF00] Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. In *Scheme Workshop*, 2000.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *J. Funct. Program.*, 9(4):355–372, 1999. URL: http://journals.cambridge.org/action/displayAbstract?aid=44275.

[Kis02] Oleg Kiselyov. How to write seemingly unhygienic and referentially opaque macros with syntax-rules. In *Scheme Workshop*, 2002.

[Kou] Peter Kourzanov. purecube. Internet. URL: https://github.com/kourzanov/purecube.

[KS13] Peter Kourzanov and Henk Sips. Lingua franca of functional programming (fp). In Hans-Wolfgang Loidl and Ricardo Pena, 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.

[Kun65] Susumu Kuno. The predictive analyzer and a path elimination technique. *Commun. ACM*, 8(7):453–462, 1965. URL: http://doi.acm.org/10.1145/364995.365689.

[Lan00] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 44(1):261–269, 2000. URL: http://dx.doi.org/10.1147/rd.441.0261.

[Lup] Benjamin Lupton. Coffeescript-object-notation parser. Internet. URL: https://github.com/bevry/cson.

[MÖ5] Egil Möller. Srfi 49: Indentation-sensitive syntax. Internet, 2005. URL: http://srfi.schemers.org/srfi-49.

[NK93] M.-J. Nederhof and Cornelis H.A. Koster. Top-down parsing for left-recursive grammars. Technical Report 93-10, University of Nijmegen, Department of Computer Science, 1993.

[OSS02] David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of mercury. In *PPDP*, pages 109–120, 2002.

[Pra73] Vaughan R. Pratt. Top down operator precedence. In *POPL*, pages 41–51, 1973. URL: http://doi.acm.org/10.1145/512927.512931.

[PW80] 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. URL: http://dx.doi.org/10.1016/0004-3702(80)90003-X.

[RF12] Jon Rafkind and Matthew Flatt. Honu: syntactic extension for algebraic notation through enforestation. In *GPCE*, pages 122–131, 2012. URL: http://doi.acm.org/10.1145/2371401.2371420.

[RO10] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *Haskell*, pages 1–12, 2010. URL: http://doi.acm.org/10.1145/1863523.1863525.

[Sch] Chicken Scheme. Unit library. Internet. URL: http://wiki.call-cc.org/man/4/Unit%20library#set-sharp-read-syntax.

[Se] Manuel Serrano and et.al. Bigloo homepage. URL: http://www-sop.inria.fr/indes/fp/Bigloo.

[SW95] Manuel Serrano and Pierre Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *SAS*, pages 366–381, 1995. URL: http://dx.doi.org/10.1007/3-540-60360-3_50.

[Vis01] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In *RTA*, pages 357–362, 2001. URL: http://dx.doi.org/10.1007/3-540-45127-7_27.

[vW74] Adriaan van Wijngaarden. The generative power of two-level grammars. In *ICALP*, pages 9–16, 1974. URL: http://dx.doi.org/10.1007/3-540-06841-4_48.

[WDM08] Alessandro Warth, James R. Douglass, and Todd D. Millstein. Packrat parsers can support left recursion. In *PEPM*, pages 103–110, 2008. URL: http://doi.acm.org/10.1145/1328408.1328424.

[Wel94] J. B. Wells. Typability and type-checking in the second-order lambda-calculus are equivalent and undecidable. In *LICS*, pages 176–185, 1994. URL: http://dx.doi.org/10.1109/LICS.1994.316068.

[WP07] Alessandro Warth and Ian Piumarta. Ometa: an object-oriented language for pattern matching. In *DLS*, pages 11–19, 2007. URL: http://doi.acm.org/10.1145/1297081.1297086.

A. PCG syntax-rules

```
(def-syntax seq (syntax-rules (qq skip quote unquote
      quasiquote unquote-splicing do ϵ when unless
                              ? + * / : lift unlift)
;; handling escapes
([_ in out c acc ts hs do[acts ...] . rest]
  (seq in out c (acts ... . acc) ts hs . rest))
;; handling ϵ
([_ in out c acc tmps hs ϵ . rest]
  (seq in out c ([≡ in out] . acc) tmps hs . rest))
;; handling sequencing (recursively)
([_ in out c acc temps [h(ac...)] (: . goals) . rest]
  (let ([temp #false]);; generate a new temporary
    (seq temp out c ((all
        (seq in temp c () () [h(ac... . acc)]
           . goals)) . acc)
       (temp . temps) [h (ac ...)] . rest)
   ))
;; handling quasi-data (one-level only)
([_ in out c acc tmps heads (qq d) . rest]
  (seq in out c acc tmps heads `d . rest))
([_ in out c acc tmps [h(ac ...)] `d . rest]
  (let ([temp #false][data #false]);; new temporaries
    (seq temp out c ((qs [] ;; remove quasi-quotation
      (seq data '() c () () [h(ac ... . acc)]) `d)
        [≡ in `(,data . ,temp)] . acc)
       (temp data . tmps) [h (ac ...)] . rest)
   ))
;; handling non-terminals
([_ in out c acc temps heads [goal . args] . rest]
  (let ([temp #false]);; generate a new temporary
    (seq temp out c ([goal in temp . args] . acc)
        (temp . temps) heads . rest)
   ))
;; handling atoms (this rule has to be the last one)
([_ in out c acc tmps heads datum . rest]
 (let ([temp #false]);; generate a new temporary
   (seq temp out c ([≡ in `(datum . ,temp)] . acc)
       (temp . tmps) heads . rest)
  ))
))
```

Figure 5. TRS for threading PCG monadic state (abridged)

```
;; A snippet from the process-args macro implementation
;; base-case (generate code for args, terms and project)
([process-args k acc [] goals rest (e es ...)
                                aa res ps (locals ...)]
 (let-syntax-rule ([K . vars]   ;; collect the free vars
 (let-syntax-rule ([K wv wp wt ws] ;; use extracted vars
 (let-syntax ([K (syntax-rules ()  ;; use extracted pvars
   ;; ... other special cases elided ...
   ([_ pvars (ee . bis) pats (terms []) hots]
    (make-scopes fresh bis all ;; schedule clause terms
       (let-syntax ([ee '()]);; last ee=e must be empty
          (all . pats))     ;; when resolving arguments,
       terms                ;; clause body in the middle,
       (make-scopes project  ;; projected terms delayed
          pvars all . hots) ;; to the end of the clause
   ))
   ;; ... handlers for clauses with actions elided ...
   )])
    (extract* vars (wp wt) ;; extract all bindings
       (K () wv wp wt ws)) ;; in projected terms
    ))
    (extract* (e es ... locals ... . vars) ;; extract all
       (res goals) (K () res goals ps)) ;; fresh bindings
    ))
    (scheme-bindings (w [] (K) (locals ...) aa))
 ))
;; recursive case: collect unifiers and attributes (e's)
([process-args k acc [v . vs] goals rest (ee . es)
                               aa (res ...) ps locals]
 (let ([e #false]) ;; generate a new temporary
    (process-args k acc vs goals rest (e ee . es)
       (v . aa)
       (res ... [≡ ee '(,v . ,e)])
       ps
       locals)
))
```

Figure 6. TRS equational theory for name binding

```
;; A snippet from the predicate (pred) macro
;; ... top-level predicate generation elided ...
([_ (begin ks ..) params ([_ . args] ⇔ . body) . rest]
  (let-syntax ([head #false])      ;; generate a new head
  (let-syntax-rule ([K heads condo locals]
   (pred (begin ks ..   ;; collect all clause heads and
    (define head (λ (Lin Lout . vars) ;; deliver to the
       (process-args condo (ks ..) args   ;; top-level
         ([seq Lin Lout condo () () [heads (ks ..)]
           . body] [])
         vars
         locals)
       ))
    ) params . rest))
     (select (K) (0 0 1 1 1) . params)
  )))
;; ... elided combing params to order specifiers
;; ... such as condo:, locals: and extend: ...
```

Figure 7. Implementing PCG predicates

```
(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))
 ;; ... other binders such as let, do and project elided
 ([_ 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))
))
```

Figure 8. Extracting free variables from Scheme terms

```
...
([_ in out condo acc temps
                   [(r . heads) (ac ...)] (goals ... *)]
 (let-syntax ([K (syntax-rules .. ()
  ([_ in out vars ..]
   (let loop ([lin in][lout out] [vars '()] ..)
     (let-syntax ([K (syntax-rules ... ()
       ([_ res ...]
        (let ([res #false] ...)
          (make-scopes (res ...) begin
           (letrec-syntax ([K (syntax-rules .... ()
            ([_ gls (v v1 v2 v3) ....]
            (condo ([≡ lin lout]
                    [≡ v1 v] ....)
               ([let ([temp #false][v3 #false] ....)
                (fresh (temp v3 ....)
                   (let-syntax ([v v3] ....)
                     (seq lin temp condo () ()
                      [(r . heads) (ac ... . acc)] . gls
                    ))
                   (append⁰ v1 '(,v3) v2) ....
                   (loop temp lout v2 ....))]))))]
           [K1 (syntax-rules ()
            ([_ var gls . args]
             (zip4 (K gls) var . args)
            ))]
           [K0 (syntax-rules ()
            ([_ . vs]
             (extract* vs (goals ...)
               (K1 [] (goals ...) (vars ..) (res ...) vs)
           )))])
          (scheme-bindings (w [] (K0) heads (goals ...)))
          )))))])
       (scheme-bindings (w [] (K) heads (goals ...)))
       ))))])
      (seq in out condo acc temps [(r . heads) (ac ...)]
        do[(scheme-bindings (w [] (K in out) heads
            (goals ...)))])
      ))
...
```

Figure 9. Implementing the Kleene-* combinator

B. PCG standard library

```
;; miniKanren examples
(def *digits* [make-parameter (list-tabulate 10 values)])
(def (range start end)
   (unfold [_ char>? end]
     values
     (∘ integer->char
        [_ + 1]
        char->integer)
     start))
(def *letters* [make-parameter (range #a #z)])
(def [lifto pred stream] (λ (x)
   (conda ([project (x)
      (or (and (ground? x) (pred x) #s) #u)])
      ([take-from (stream) x])
      )))
(def numbers? [lifto number? *digits*])
(def symbols? [lifto symbol? (λ ()
     (map (∘ string->symbol list->string list) [*letters*]))])
(def strings? [lifto string? (λ ()
     (map (∘ list->string list) [*letters*]))])
(def (! p . args)
   (condu ((apply p args) #u)
      (else #s)))
(def (null⁰? x) [≡ x '()])
(def (pair⁰? x) (fresh (x0 x1) [≡ x `(,x0 . ,x1)]))
(def (car⁰ x y) (fresh (t) [≡ x `(,y . ,t)]))
(def (cdr⁰ x y) (fresh (h) [≡ x `(,h . ,y)]))
(def (cons⁰ h t l) [≡ l `(,h . ,t)])
(def (number Lin Lout x) (all (cons⁰ x Lout Lin) (numbers? x)))
(def (symbol Lin Lout x) (all (cons⁰ x Lout Lin) (symbols? x)))
(def (strings Lin Lout x) (all (cons⁰ x Lout Lin) (strings? x)))
(def (literal Lin Lout x) (conde ([symbol Lin Lout x])
                                 ([number Lin Lout x])))
(def (idem Lin Lout v) (cons⁰ v Lout Lin))
```

C. Acronyms

AST	Abstract Syntax Tree
BNF	Backus-Naur Formalism
CPS	Continuation Passing Style
DCG	Definite Clause Grammar
DSL	Domain-Specific Language
FP	Functional Programming
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LISP	List Processing
NXP	Next Experience Semiconductors
PCG	Parsing Clause Grammar
PEG	Parsing Expression Grammar
R5RS	Revised^5 Report on the Algorithmic Language Scheme
SRFI	Scheme Request for Implementation
TRS	Term-Rewriting System
TU	Technical University