Bidirectional parsing
a functional/logic perspective

Peter Kourzanov
NXP Eindhoven/TU Delft, Netherlands
kourzanov@acm.org

Abstract
We introduce PURE\textsuperscript{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\textsuperscript{2} Report on the Algorithmic Language Scheme (R5RS). Thanks to its use of the MINI\textsc{kanren} logic programming system it supports fully reversible and extensible syntax-semantics relations. In this paper we highlight the usability and simplicity of PURE\textsuperscript{3}’s approach, address the problem of left-reursion 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/inference

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\textsuperscript{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 \textsuperscript{[DM82]} of the various properties of such inputs\textsuperscript{2}, manipulation, refactoring and optimization of programs \textsuperscript{[BD77]}, compilation to machine code \textsuperscript{[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 \textsuperscript{[ALSU06]} and \textsuperscript{[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 \textsuperscript{[Wei94]} to human understanding.

In this paper we present PURE\textsuperscript{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 MINI\textsc{kanren} \textsuperscript{[FBK]}, a logic programming system embedded in R5RS \textsuperscript{[ABB98]} 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\textsuperscript{3} (using the hygienic syntax-rules macro system and MINI\textsc{kanren}) 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 \textsuperscript{2} and then highlights the use of the formalism by specifying an admittedly simple, yet concise and flexible typing system in section \textsuperscript{3}. The problems in implementing and using extensible transformations are addressed in section \textsuperscript{4}. Related work is reviewed in section \textsuperscript{5} while the conclusions can be found in section \textsuperscript{6}. Full implementation using Bigloo and conforming to R5RS plus two relevant SRFI libraries is available at github \textsuperscript{[Kou]}.

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, \textsuperscript{3} which is often referred to as the definite clause grammar (DCG).

\textsuperscript{1}we set machine learning community aside for now
\textsuperscript{2}this is commonly known as type-checking and type-inference
\textsuperscript{3}note that our use of R5RS is flavored by macro-expressible pattern-matching as well as a few syntactic liberties for recursive (\texttt{def}) and non-recursive (\texttt{defn}) bindings, brackets and lexical syntax (viz. \texttt{reader-macros})
via Horn clauses. Logic programming languages such as Prolog and MINI\textsc{kanren} 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 this is done in MINI\textsc{kanren} 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.

\[
\begin{align*}
\langle \text{factor} \rangle & ::= \langle \text{literal} \rangle \mid \langle \text{factor} \rangle \,* \langle \text{literal} \rangle \\
\langle \text{term} \rangle & ::= \langle \text{factor} \rangle \mid \langle \text{term} \rangle \,* \langle \text{factor} \rangle \\
\langle \text{expr} \rangle & ::= \langle \text{term} \rangle \mid \langle \text{expr} \rangle \,* \langle \text{term} \rangle \\
\langle \text{expr} \rangle & ::= \langle \text{expr} \rangle \,-\langle \text{term} \rangle \\
\langle \text{expr} \rangle & ::= \langle \text{expr} \rangle \,\langle \text{term} \rangle \\
\langle \text{expr} \rangle & ::= \langle \text{expr} \rangle \,\langle \text{term} \rangle \\
\end{align*}
\]

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 MINI\textsc{kanren} 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 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.

\[
\text{peg expr} \ (\{ \exists \langle \text{term} \rangle \ [(\{\{"\*\ / \-'\} : \langle \text{term} \rangle \})])
\]

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:

\[
\text{peg} \langle \text{Factor} \rangle \text{ ~}\langle \text{Term} \rangle \text{ ~}\text{peg} \langle \text{Expr} \rangle
\]

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 MINI\textsc{kanren} predicates 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 beinterned Scheme atoms such as literals (#\text{true} and #\text{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 MINI\textsc{kanren} 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 MINI-KANREN

In this section we briefly introduce the way in which we use MINI-KANREN’s primitives [BK05] such as *success and failure* (\#a and \#u), binding of logic variables (*fresh*), unification (\(\equiv\)), disjunctions (fair choice *conde*, soft-cutting *condu*, committed choice *cond*), 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 MINI-KANREN.

\[\text{; ; the swiss army knife of logic programming}
\]

\[
\begin{align*}
\text{(def append0 (predicate)} & \text{ } \\
\text{ (\text{'(' ) b b) )} & \text{ } \\
\text{ (\text{'(x . ,a1) b '(%x . ,c1) :- [append0 a1 b c1])})}
\end{align*}
\]

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.

\[\text{; ; e is somewhere in t ;; using explicit disjunction}
\]

\[
\begin{align*}
\text{(def member0 (predicate)} & \text{ } \\
\text{ (\text{'(e ) :- #x) )} & \text{ } \\
\text{ (\text{'(e '(% ,t)) :- [(\text{\equiv e h} / [\text{member0 e t}])(\text{ member0 e t))])})}
\end{align*}
\]

By design shared with MINI-KANREN, 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` predicate above, *explicit* PEG-style disjunction in clause bodies is often essential\(^4\) avoiding duplication of clause bodies and heads.

In contrast to MINI-KANREN, 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` 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` 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 \(\equiv\), \(2\) natural representation of semantics, i.e., an AST - to the left of \(\equiv\), \(3\) direct-style operator *associativity and precedence* and \(4\) inverse for free (note that we separate the clause head from the clause body by \(\equiv\) to indicate full reversibility. Our final version of a reversible syntax-semantics relation for expressions is given in figure [\[\].

\[
\begin{align*}
\text{(pcg)} & \text{ } \\
\text{(Factor)} & \text{ } \\
\text{(\text{'(c,x,y)) :- [Factor x] * [literal y])} & \text{ } \\
\text{(\text{'(x) :- [literal x])}) & \text{ } \\
\text{(Term)} & \text{ } \\
\text{(\text{'(\text{\equiv ,x,y}) :- [Term x] * [Factor y])} & \text{ } \\
\text{(\text{'(\text{\equiv ,x,y}) :- [Term x] / [Factor y])}} & \text{ } \\
\text{(\text{'(x) :- [Factor x])}) & \text{ } \\
\text{(Expr)} & \text{ } \\
\text{(\text{'(+ ,x,y)) :- [Expr x] + [Term y])} & \text{ } \\
\text{(\text{'(- ,x,y)) :- [Expr x] - [Term y])} & \text{ } \\
\text{(\text{'(x) :- [Term x])}) & \text{ }
\end{align*}
\]

---

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

---

![Figure 1: Pure, declarative PCG parser analyzer](image-url)

The design of PCG (see figure [\[\]) 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.

![Figure 2: PURE\(^3\) DSL architecture](image-url)

The `seq` macro (see figure [\[\]) 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, \(\epsilon\), 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 \(2.3\) for an example), an expanded\(^6\) `seq` 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 \(\beta\)-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 \(2.2\) as well as
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 compensation recognizer from section 3.4) 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.

;; ... Passing functions into predicates ... ;; Monomorphic lists (for JSON), used in section 3.1
(defun [ne-list comma elem] (pcg ≜ s
(a ([ ,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
([,λ z x (if (null? z) x ‘[ ,z ,x)])])
⇒ [literal x] [Factor y])
([,λ z x (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 [DG02]. 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 start predicate (π) form, 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 uses inference of logic variable bindings in order to promote (full) reversibility and to avoid code clutter by explicit fresh introductions (see section 3.4 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 (\texttt{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).

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 \texttt{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) [HP03] to bootstrap the \texttt{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 \texttt{begin} or all. For project, we verify that all attributes are grounded and vacuously succeed otherwise.

;; Introducing the fresh and project binders (def-syntx make-scopes (syntax-rules (project)
([ ; ; nothing to do - just succeed
( ; ; default . body) (default . body))
( ; ; project (var ...) . body)
(project (var ...) . body)
(or (and (and? ... var) ... body) \#s))
))

([; ; let-syntx-rule (K \texttt{args} terms)
( \texttt{binders . terms})
(extract* \texttt{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 3.4 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 \texttt{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\footnote{using a technique similar to the seq macro described above}, while the final void attribute tail gets syntax-bound (i.e., renamed to \texttt{()} as is customary in Scheme variadic functions).
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 peg/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 peg variants that hide internal predicates and expose a single starting predicate function to the top-level.

**Synthesized attributes** Similar to the S-attributed grammars, where inherited attributes are not allowed, PURE 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 (\tau), 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’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
(define exprs (peg ⇔ expr
 (factor locals: (x))
 ([\l y] ⇔ [literal x] [factor' x y]))
 (factor locals: (y))
 ([\l x z] ⇔ * [literal y] [factor' (* x , y) z])
 ([\l x] ⇔ e))
 (term locals: (x))
 ([\l y] ⇔ [factor x] [term' x y]))
 (term locals: (y))
 ([\l x z] ⇔ * [factor y] [term' (* x , y) z])
 ([\l x z] ⇔ / [factor y] [term' (/ x , y) z])
 ([\l x] ⇔ e))
 (expr locals: (x))
 ([\l y] ⇔ [factor x] [expr' x y]))
 (expr locals: (y))
 ([\l x z] ⇔ / [term y] [expr' (+ x , y) z])
 ([\l x] ⇔ [term y] [expr' (+ x , y) z])
 (term locals: (x))
 ([\l x] ⇔ e))
))
```

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 difference list with the knot by unifying the child 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 difference list with the knot by unifying the difference list with the knot by unifying the difference list with the knot by unifying the difference list with the knot.

Looking at the diverging generated code for the Expr (+) clause:
```
(define _head_424 (lambda (Lin Lout . vars)
  (fresh \y x) (\equiv vars \(\cons \(+ x y\) \))\))
```

2.3.2 Handling binding in combinators

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

```
;; Minus in Scheme has arity \geq 1
(peg \%\% \((\cons \((- , t . , ts)\) \equiv [\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_424 (lambda (Lin Lout . vars)
  (fresh \y x) (\equiv vars \(\cons \(+ x y\) \))\))
```

```
;; ..._head_424 and _head_426 elided ...
(define _head_426 (lambda (Lin Lout . vars)
  (fresh \y x) (\equiv vars \(\cons \(+ x y\) \))\))
```

10We refer to our github for further details about select and peg
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 3 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: (define _head.622 (λ (Lin Lout . vars) (fresh (y x) (= (cons (list * x y) '())) (fresh (_temp.505 temp d) (project (Lin) (if (ground? Lin) #s (Expr d '() x))) (append0 d temp Lin) (≡ temp (cons * _temp.505)) (_temp.505 temp 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 fresh0 x) (def fresh1 (predicate (conde (λ (false (true (''))))) (else (fresh y z) (conde (λ (false (true (((y ,z) . ,x) ,x) ))) (fresh1 z) (≡ x '()) ))) (append0 a x b)) (define prefix0 a b) (define prefix1 (fresh (x) (predicate locals: (x) (fresh1 x) (append0 a x b) )))))

Does this work as promised, in an on-line fashion? Let's generate an infinite input using predicates above (where both MINIKANREN and PUREβ versions are given) and verify by running!

;;;; Parsing prefixed infinite input stream (verify Expr (run 1 q) (fresh 1) (prefix0 '1 (* 2 + 3 * 5) 1) (Expr 1 '() q))

One disadvantage of this technique is the non-determinism (and thus a quadratic-time slowdown) introduced by the append0. 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. 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 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

Table 1. Typing rules for monomorphic JSON

<table>
<thead>
<tr>
<th>Base Types</th>
<th>Pairs, Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNI</td>
<td>T×D</td>
</tr>
<tr>
<td>BOO</td>
<td>T×B</td>
</tr>
<tr>
<td>STR</td>
<td>T×STR</td>
</tr>
<tr>
<td>NUM</td>
<td>T×NUM</td>
</tr>
<tr>
<td>ARR</td>
<td>T×ARR</td>
</tr>
<tr>
<td>PAR</td>
<td>T×PAR</td>
</tr>
<tr>
<td>OBJ</td>
<td>T×OBJ</td>
</tr>
</tbody>
</table>

| (def json-symbol (list x) ⇒ [strings x]) | (json-key = json-symbol) |
| (json-number = number) | |
| (json-bool ([] ⇒ (true / 'false)) | (json-value ([] ⇒ 'null) |
| ([x] ⇒ [json-symbol x] / [json-number x]) | ([x] ⇒ ([json-object x] / [json-array x])) |
| (json-pair ([, (, (x . ,y)) ⇒ [json-key x] : [json-value y]) | |
| (json-value-list (L 'O) ⇒ e) | (json-pair-list ([, 'O]) ⇒ e) |
| (json-array ([, 'arr . ,es]) ⇒ ([, [json-value-list es] [)]) | (json-object ([, 'obj . ,es]) ⇒ ([, [json-pair-list es] []]) |

Thanks to the availability of unification, addition of types to the grammars using PUREβ 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 2.2.3) 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 (json-value-list (L '('List ,t) ) [)]) (json-pair-list (L '('Pair ,t1 ,t2) ) [)])

Conventional typing rules are given in table 1 while their translation to PUREβ 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 ARRJ, PAR1 and OIH typing rules (the rest of the rules can be found in figure 3). 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.
Note that we are making the json-sym 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 json-value predicate), or inferred (provided they are given as free logic variables). However, for homogeneous arrays and objects of pairs of strings. Such types can be either monomorphic containers denoting objects such as arrays of numbers and JSON keys by numbers and booleans while preserving the modularity and compositionality of the grammar above.

It turns out that the type system we are using is still more powerful than necessary, because it is possible to check 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.

(Note that we are making the json-sym 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 json-value predicate), or inferred (provided they are given as free logic variables). However, for homogeneous arrays and objects of pairs of strings. Such types can be either monomorphic containers denoting objects such as arrays of numbers and JSON keys by numbers and booleans while preserving the modularity and compositionality of the grammar above.

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 \textsuperscript{2} 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.

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 \textsuperscript{2} 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.

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 \textsuperscript{2} 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.

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 \textsuperscript{2} 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.
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 \[56\]. In PURE\[56\] we would like to be able to introduce orthogonal (i.e., homomorphically) 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\[56\]. 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.

---

**4.1 Chaining extensions**

Often, extensions make sense only when applied together, as a group. PURE\[6\] 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 dependable’s definition:

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

**4.2 Composing extensions**

Because each extension hooks onto the current value of the \texttt{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 'sym: \{[12:quux],[42:snarf]\} 'L' x t) 

\[===> \(\{\text{Array} \\text{Object} \\text{Pair} \\text{Num} \text{Sym}\}\) 

\(\{\text{arr (obj (12 . quux)) (obj (42 . snarf))}\}\))

))
```

While in section \[4.1\] the composition is static, here we apply dynamic resolution and chaining of extensions referenced by the \texttt{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 \texttt{cond}: 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\[7\]:

```
;; Extending Term predicate

(defn Term+ (let ([extend' (extend)] T' Term))

(fn-with [apply extend'] | 'Term =>

\(\{\text{pcg ([\(\pi (\Theta x y)\) \(\text{[lift T' x]} \Theta \text{[Factor y]}\)])[}]\))
```

Because we allow impure logical code here, and because the escape to the Scheme level is made possible via MINI\textsc{kanren}\textsc{'s} run and project (\texttt{\pi}) primitives, the PCG predicates can be extended while parsing. For example, the \(\Theta\) in the clause head above can be referring to any Scheme function or procedure, and that may perform any (effectful) computation. Since the SRFI39 allows destructive/imperative update to the extend reference cell, this alone effectively makes the formalism \textsc{Turing-complete} (i.e., Chomsky Type-0). This can be easily seen by translation to \(\text{\pi}\)-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 \[HM03\]. 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 \[NK97\]. memoing

---

\[12\] note that we need to explicitly lift the left-recursive call in this case.
length \[Kun65\]. The work on \textsc{OMeta} \[FH06\] and \[FHC08\], an old idea to limit matches by the input ordering of this). Parser combinators are often applying curtailment \[BS08\] or through elimination (see section \[2.3.1\] for a PCG rendering of this). Unlike prior art where programming was constrained to use only reversible formalism of bijections \[RO10\] or where the reverse transformations are derived from the forward transformations \[Vis01\] and \textsc{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 \textsc{lenses} \[FGM^*08\] 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 \textsc{mix}-expressions\[\textsc{mix}^*-\textsc{expressions}\[\textsc{mix}^*05\], sweet \textsc{t}-expressions\[\textsc{t}-expressions\] \[DAW13\] 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-reursive 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 \textsc{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 \textsc{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 \textsc{syntax-rules} for programming in a homoiconic language such as Scheme. The syntax and the semantics are better specified using pure declarative methods of \textsc{pure} \[FHM06\], naturally expressing reversible (i.e., inferrable) types, providing declarative disambiguation operators, enabling online/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 William Byrd, Ralf Lämmel and Vadim Zaytsev.

### References


\[\text{BFP}^*08\] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In \textsc{POPL}, pages 407–419, 2008. \url{http://doi.acm.org/10.1145/1326438.1328487}

\[\text{BS08}\] Ralph Becket and Zoltan Somogyi. Dggs + memoing = packrat parsing but is it worth it? In \textsc{PDL}, pages 182–196, 2008. \url{http://dx.doi.org/10.1007/978-3-540-77442-6_13}


\[\text{Cam05}\] Taylor Campbell. \textsc{Srfi} 46: Basic syntax-rules extensions. Internet, 2005. \url{http://srfi.schemers.org/srfi-46}


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


\[\text{FBK}\] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. minikanren homepage. \url{http://minikanren.org}


\[\text{Fee03}\] Marc Feeley, \textsc{Srfi} 39: Parameter objects. Internet, 2003. \url{http://srfi.schemers.org/srfi-39}


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

A. PCG syntax-rules

(def-syntax seq (syntax-rules (qq skip quote unquote (do ϵ (when unless (+ * (/ lift unlift))))
  (\[; in out c acc ts hs . rest\])
  (\[seq in out c (\[acts ... \]. acc) ts hs . rest\])
  (\[seq in out c acc tmps heads 'd . rest\])
  (\[seq in out c acc temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c acc tmps heads datum . rest\])
  (\[seq in out c acc temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c acc tmps heads datum . rest\])
  (\[seq in out c acc temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c acc temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])
  (\[seq in out c temps (\[h(ac ... . acc)\] \[h (ac ...\]) . rest\])

Figure 5. TRS for threading PCG monadic state (abridged)
A snippet from the process-args macro implementation

```
(let-syntax-rule ([K . vars] ;; collect the free vars
    (let-syntax ([K wv wp wt ws] ;; use extracted vars
        (let-syntax ([K (]) ;; when resolving arguments, terms
            ;; clause body in the middle,
            (make-scopes project ;; projected terms delayed
                pvars all . hots) ;; to the end of the clause
        )))
    (extract* vars (wp wt) ;; extract all bindings
        (K () wv wp wt ws))
    (extract* (e es ... locals ... . vars) ;; extract all
        res goals) (K () res goals ps))
(scheme-bindings (w [] (K) (locals ...) aa)))
```
B. PCG standard library

;; miniKanren examples
(def *digits* [make-parameter (list-tabulate 10 values)])
(def (range start end)
  (unfold [_. char? end]
    values
    (o 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)
  (cond [(apply p args) #u]
    (else #a)))
(def (null? x) (≡ x '()))
(def (pair? x) (fresh (x0 x1) (≡ x '(',x0 ,x1)))
(def (car? x y) (fresh (t) (≡ x '(',y ,t)))
(def (cdr? x y) (fresh (h) (≡ x '(',h ,y)))
(def (cons? h t l) (≡ l '(',h ,t))
(def (number Lin Lout x) (all (cons? x Lout Lin) (numbers? x)))
(def (symbol Lin Lout x) (all (cons? x Lout Lin) (symbols? x)))
(def (strings Lin Lout x) (all (cons? x Lout Lin) (strings? x)))
(def (literal Lin Lout x) (conde ([symbol Lin Lout x])
    ([number Lin Lout x])))
(def (idem Lin Lout v) (cons? 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
   Revised5 Report on the Algorithmic Language Scheme
SRFI
   Scheme Request for Implementation
TRS
   Term-Rewriting System
TU
   Technical University