A nanopass approach to a modular RDF implementation

Duncan Guthrie
University of Glasgow
Glasgow, UK
d.guthrie.1@research.gla.ac.uk

Paul Harvey
University of Glasgow
Glasgow, UK
Paul.Harvey@glasgow.ac.uk

Michele Sevegnani
University of Glasgow
Glasgow, UK
Michele.Sevegnani@glasgow.ac.uk

Abstract

Resource Description Framework (RDF) is a widespread W3C standard defining a domain-specific language for knowledge representation, providing a platform to build higher-level languages such as OWL or SHACL. RDF *concrete syntaxes* define an external representation of its abstract model for data interchange. Despite sharing the same core semantics, RDF concrete syntaxes differ in how well they support common modelling idioms at the surface level. Many RDF libraries have been implemented in *Scheme*, however, they support few concrete syntaxes and/or do not exhibit compliance with the W3C test suites.

Contributions. We develop a new open-source RDF 1.1 implementation for R^{6}RS Scheme, leveraging a *nanopass* architecture. The improved flexibility of the approach in a Scheme-based environment is shown by the ability to reuse transformations across three RDF concrete syntaxes. Its practicality towards real-world interchange is demonstrated through experimental compliance with the W3C standards. Finally, we achieve better performance than the most complete existing library for Scheme.

CCS Concepts: • **Software and its engineering → Source code generation**; • **Information systems → Resource Description Framework (RDF)**.

Keywords: RDF 1.1, Scheme, Lisp, nanopass framework, W3C standards compliance, semantic web, compiler transformation, data interchange

ACM Reference Format:
Duncan Guthrie, Paul Harvey, and Michele Sevegnani. 2026. A nanopass approach to a modular RDF implementation. In *19th ACM SIGPLAN International Conference on Software Language Engineering (SLE '26), July 2–3, 2026, Rennes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3806383.3815514

SLE '26, Rennes, France

ACM ISBN 979-8-4007-2639-2/26/07
https://doi.org/10.1145/3806383.3815514

1 Introduction

Resource Description Format (RDF) is a domain-specific language standardised by the W3C since 1999, for knowledge representation and metadata interchange on the world-wide web [49]. RDF has found widespread use in metadata modelling and as a database-agnostic graph model [32], and its formal model and specification [20] mean that RDF data can be readily consumed by compliant implementations, supporting interoperability.

Scheme, a dialect of Lisp originating in 1975, has found significant use developing domain-specific languages, including its descendent Racket [41]. Standardised in a series of reports by the Scheme community, the language is distinguished by clear semantics based on the lambda calculus formalism [54], highly uniform syntax (s-expressions), and an expressive, high-level metaprogramming framework (`syntax-case`) [44].

Both RDF and Scheme originate in the field of symbolic computation. As a meta-language on which higher-level languages like OWL are built [26], RDF has been used to create DSLs, just like Scheme has. Indeed, there has been persistent interest from the mid-2000s onwards in RDF and Scheme: to date, at least five RDF libraries have been developed for various Scheme dialects (Scheme48 [14], Guile [43], and Racket [27, 36, 42]). However, we found that existing RDF libraries exhibited limited compliance with the W3C standards, and/or supported few concrete syntaxes. We argue that RDF's formal specification and extensive compliance suites have been key to its support for interchange of metamodels and their semantics, meaning that poor library support impedes these libraries' utility.

Current implementations' processing of RDF documents consists of very few transformations, similar to older compilers, meaning that common processing actions cannot be extracted and reused. Yet, as RDF concrete syntaxes inherit core semantics from the abstract model (e.g. blank nodes, collections) [31], intuitively, processing these in one syntax should be transferable across all others. The **nanopass**

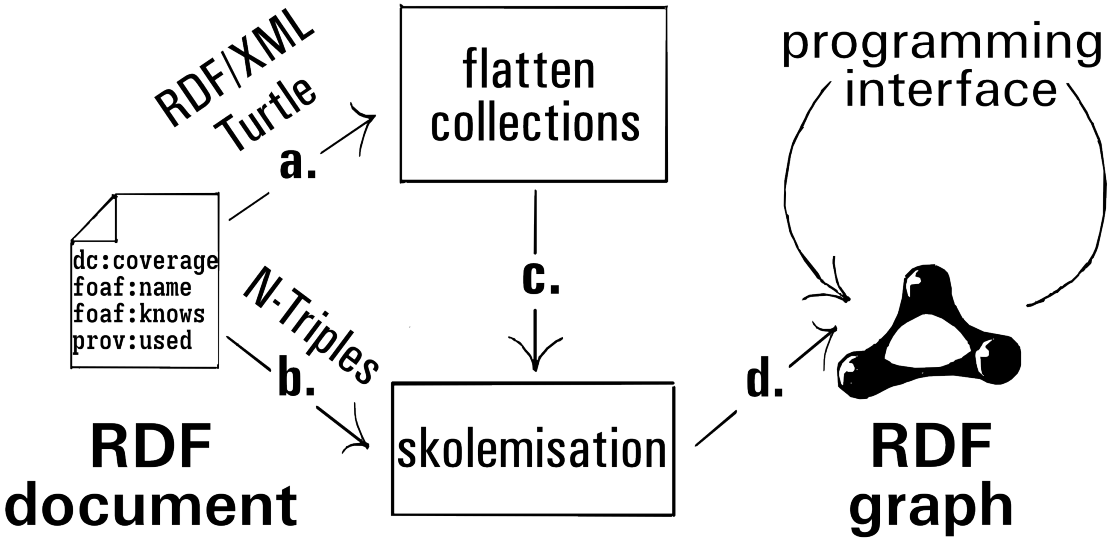

Figure 1. Commonalities in processing different RDF syntaxes: a) Syntax-specific processing and intermediate representations for RDF/XML and Turtle. b) Minimal processing of N-Triples. c) Minimal processing of RDF/XML and Turtle. d) Extract RDF triples.

architecture [39] in contrast consists of numerous small, relatively self-contained transformation passes and intermediate representations, better supporting reuse in isolation.

Contributions. We describe a new, nanopass-based library for R^{6}RS Scheme, in which we modularise and reuse common processing stages. We demonstrate that it is more comprehensive than, and unifies existing work, supporting three RDF concrete syntaxes as well as the RDF model theory semantics (see sections 4–6). We further show that it strictly conforms with the W3C standard test suites, and that it demonstrates adequate performance in processing real-world data (see sections 6–8). To our knowledge, this is the first RDF implementation to take a nanopass approach.

2 Background

This section introduces key concepts underlying this paper, namely the RDF model, RDF concrete syntaxes, and the nanopass architecture.

2.1 Resource Description Format (RDF)

RDF consists of a) an abstract model of logical assertions of facts about *resources* (entities denoted by unique identifiers) and b) a series of concrete syntaxes used to interchange representations of the abstract model. A number of formal semantics for the abstract model have been developed, but support for these is at the discretion of the data modeller or software developer, and different languages built on top of RDF (like OWL) extend or modify the W3C's suggested semantics (the RDF Schema) [16]. Indeed, RDF software tooling often only supports concrete syntax(es).

RDF asserts facts about resources, which are abstract entities identified by IRIs, a generalisation of URIs to unicode [12, 21]. Resources may also be denoted by *blank nodes*, which are often equivalent to IRIs, but they are anonymous: labels allocated to blank nodes are an artefact of the serialisation [20]. Syntactic forms in RDF concrete syntaxes (see section 2.2) often allow modellers to omit a blank node entirely when using it to denote a resource.

These assertions take the form of triples, composed of a subject, predicate and object, each of which are resources. Additionally, an object may consist of a *literal* (corresponding to datatypes like numbers and text). A set of triples is known as an *RDF graph*, but although RDF is often interpreted as an edge-labelled graph, as in the *ShEx* structural schema formalism [53], work is ongoing to formalise mappings of RDF graphs to current attributed graph databases [5].

RDF concrete syntaxes are largely declarative, but processing them correctly requires some degree of care. As well as skolemisation (see section 4.4), syntaxes often emphasise *terseness* for expressing common data modeling idioms. For example, these special forms include expressing collections (ordered objects equivalent to linked lists), avoiding repetition of subjects in blocks of triples, and omission of generated blank nodes.

2.2 RDF concrete syntaxes

RDF concrete syntaxes are the means of interchange of (meta)data models based on RDF. We selected three of the most important concrete syntaxes for implementation.

RDF/XML [25] is an XML-based syntax for RDF. This is one of the most widely-used RDF syntaxes, with the Web Ontology Language (OWL) standardised by the W3C, requiring compliant implementations to support it [30], and applications like RSS[1] depending on it.

Turtle [9] is a syntax designed to be readily readable and writable by people, and consequently it captures common modelling idioms used by data modellers. For example, a namespace common to many IRIs can be expressed as a prefix, repeated subjects and objects can be condensed into blocks, and RDF collections (like linked lists; ordered first/rest statements) can be expressed succinctly.

N-Triples [8] is a line-oriented subset of Turtle [9], reducing the amount of syntactic sugar into lists of triples separated by full stops. As well as getting this for free from supporting Turtle, N-Triples is used in all of the W3C RDF test suites, so it is essential to support this. Some tooling also supports the related N-Quads syntax [15], which annotates each triple with an additional field typically identifying the source RDF graph.

2.3 Nanopass framework

Keep's Nanopass is a *meta-framework* for developing nanopass compilers, developed for R[6]RS Scheme [38]. Meta-frameworks provide optimised code-generation for common idioms in writing passes and intermediate representations. The framework consists of:

1. **Language, or intermediate representation definitions**: context-free grammars defining abstract syntax trees
2. **Pass definitions**: a series of transformation clauses mapping input to output language terms
3. **Parsers** and serialisers from s-expressions to syntax trees, and vice versa

The framework takes significant advantage of R[6]RS Scheme's *metaprogramming* support. Language definitions deviate significantly from standard Scheme forms, providing syntactic support for extensions to existing languages (only changes given), which supports the common nanopass architectural pattern of incremental changes to intermediate representations. Transformer passes, and clauses within for language terms, become standard Scheme procedures, which in our experience means that the DSL integrates well with the rest of the Scheme environment, especially when picking and choosing passes relevant to a particular RDF concrete syntax (see fig. 7).

The framework is at minimum capable of generating transformers from one language's terms to another if there is no change. It defines a custom pattern-matcher, rebinding Scheme's quasiquote to emit language terms when in a transformer. This pattern of intermediate representations, and transformation passes between which, appears to be fundamental to the nanopass approach, is used in almost every example in Keep's Nanopass manual [38], and was used pervasively throughout this work.

[1] RSS 0.90, 1999: https://www.rssboard.org/rss-0-9-0

3 Related work

We now examine nanopass approaches to domain-specific languages, and existing RDF implementations for Scheme. See table 1 for a high-level comparison.

3.1 Nanopass DSLs

Two aspects of **nanopass** approaches to processing computer languages can be distinguished. The general approach consists of a succession of small transformation passes, contrasted with single or few passes in traditional compilation approaches. Nanopass *meta-frameworks* originate from compiler education [48] and were further developed as a domain-specific language embedded in Scheme by Keep [39] (see previous section 2.3). Nanopass involves numerous intermediate representations and passes where relatively little of the syntax tree changes between passes. This often means that boilerplate mapping unchanged portions builds up quickly, hence meta-frameworks have value in providing efficient code generation.

Recent examples of nanopass approaches in compiler engineering include Ebresafe et al. [22], who emphasise two advantages of a nanopass approach as modularisation of compiler extensions and internal, intermediate representations. Hsu completed several pieces of work on compilation of APL to LLVM (explicitly using Keep's Nanopass framework) [34, 35]. Other relevant work includes compilation of the parallel programming language TOCK [13, 47]. Another perspective is provided by Ringo et al. [45] who combine a nanopass approach with attribute grammars, enabling propagation of typing information up a syntax-tree and passing inherited information back down it, capturing a common pattern in writing transformation passes, even for languages which are largely declarative, like many DSLs.

There are fewer examples of a nanopass approach to domain-specific languages: Ballantyne et al. [6] show a nanopass approach to compiling miniKanren [24], a popular DSL for logic programming. This application of a nanopass approach to *declarative*, embedded languages is also found in Kanor[33]. P4 is a DSL for programming packet-processing devices [3]. The DSL uses a nanopass approach in translating the DSL into multiple compiler backends, with individual passes subject of analysis to discover bugs [46]. Finally, a meta-framework similar to Keep's Scheme nanopass framework is the Ruby-Write DSL [17].

A common element across these works is the emphasis on demonstrating overall program *correctness* by demonstrating correctness in the individual sub-passes.

3.2 RDF libraries for Scheme

Racket [23] (originally PLT Scheme) descends from Scheme, developed as a platform for a number of languages, including a compliant implementation of the R⁶RS standard (to which one of the main authors contributed [51]). The most recent RDF work for Racket is **Racket-RDF**, packages implementing a API for the RDF data model [36] and auxiliary parsers and serialisers for concrete syntaxes [37]. Only N-Quads/N-Triples are fully implemented, but regular expressions are used instead of a parser, which failed language benchmarks for documents embedding quotes in literals and comments, see table 1. The package is not tested for compliance, nor does it support RDF semantics. The **`linkeddata`** [42] package implements N-Quads and JSON-LD. JSON-LD [52] is related to RDF but differs from most RDF syntaxes by omitting blank nodes. This package is more complete than `racket-rdf` and employs a true parsing approach, but compliance is tested only for JSON-LD, bundling the JSON-LD test-suite. **`Racket-librdf`** is the oldest Racket library, developing FFI bindings to Redland Raptor [27], as well as programming interfaces for manipulating RDF graphs, e.g. folds over a graph. While the underlying Redland `librdf` is compliant with the W3C test suites [7], this Racket library is not explicitly tested for conformance, and the API does not expose RDF semantics. This library could not be run on a modern Racket release, possibly due to changes in the foreign function interface made to recent versions of Racket.

GNU Guile [1] is one of the oldest Scheme implementations, originally designed to be embedded as an extension language in C programmes, and compliant with the recent R⁶RS and R⁷RS standards. **Guile-RDF** [43] is a library with support for the N-Triples/N-Quads and Turtle concrete syntaxes. This library is distinguished from other libraries in its completeness. First, it tests for compliance with the W3C test suites directly, fetching test cases with the Guile HTTP client. Second, in order to use the W3C test suites directly, it implements isomorphism between RDF graphs. Finally, it complies with the suggested RDF 1.1 Model Theory semantics [31], and entailment regimes show whether an RDF graph is consistent with each of the three suggested semantics (simple, datatype and RDF Schema semantics).

Other relevant Scheme systems reviewed included the R⁷RS-small package repository Snow-Fort[2]; the `akku.scm` repository[3]; Chicken Scheme's listing of packages[4]; libraries included with the Scheme systems listed as compliant with the R⁶RS standard [51]; and systems listed as compliant with R⁷RS-small [50]. Of these, the **`schemantic-web`** library [14] was the only relevant result. This library is tested for compliance with the Turtle and N-Triples concrete syntaxes, albeit the older RDF 1.0 standard from 2004.

Table 1. Summary of Scheme RDF systems

Library	Syntaxes			Conformance			Semantics
	RDF/XML	Turtle	N-Triples	RDF/XML	Turtle	N-Triples	
Racket-RDF	✗	✗	✔	N/A	N/A	fail	N/A
`linkeddata`	✗	✗	✔	N/A	N/A	✗	N/A
`Racket-librdf`[5]	✔	✔	✔	✔	✔	✔	N/A
`schemantic-web`	✗	✔	✔	N/A	✔	✔	N/A
Guile-RDF	✗	✔	✔	N/A	✔	✔	✔
This work	✔	✔	✔	✔	✔	✔	✔

With the exception of the much older `schemantic-web` library, the RDF libraries covered here heavily employed features specific to the target Scheme dialect, or depended on non-portable foreign function interfaces. Racket libraries were non-portable mainly due to higher-order software contracts at the library module boundary, or optional keyword arguments to procedures. While some libraries provided additional programming interfaces for RDF graphs, a recurring barrier to reuse was non-portable assumptions of structural equivalence between record types (such that `equal?` could compare records where `equal?` held between fields).

4 Methodology

We now present our RDF library, as presented in fig. 1. The library is a pure R⁶RS Scheme library which develops programming interfaces for a) working with RDF triples; b) transforming RDF documents into RDF graphs; and c) checking RDF semantics and isomorphism between RDF graphs.

The main RDF graph programming interface defines core R⁶RS Scheme record types for an RDF graph. For IRIs, there is a complete implementation of RFC 3986/3987 [28] defining parsers for URIs and IRIs, with setters and getters for IRI record type components such as path and hostname. There is a further implementation of the RFC 3986 relative reference resolution and normalisation algorithms. Record types are defined for blank nodes and RDF literal values. RDF triples are defined as a record type with subject/predicate/object setters and getters.

Procedures which process RDF documents into RDF graphs (lists of triples) are implemented for the RDF/XML, Turtle and N-Triples concrete syntaxes. The syntaxes are separated into R⁶RS Scheme library modules, each exposing a

[2]Snow: https://snow-fort.org/
[3]Akku: Scheme Package Manager: https://akkuscm.org/
[4]CHICKEN eggs: https://eggs.call-cc.org/
[5]Conformance is assumed to be inherited from Redland `librdf`, but did not appear to be explicitly tested

`string->triples` procedure consuming a string and a default base IRI (against which to resolve relative IRIs).

A summary of the processing stages for the Turtle syntax, and where they were reused by RDF/XML and N-Triples, is given in table 2. These primarily intersected in processing collections (which are like linked lists, see section 4.3; and skolemisation, see section 4.4).

Separate library modules expose implementations of the simple, RDF datatype and RDF Schema semantics (entailment regimes), as well as isomorphism between RDF graphs.

4.1 A pure-R⁶RS library

In order to unify existing duplication of work, and to be portable across existing Scheme systems, we developed a pure R⁶RS Scheme library, a standard supported by the most widespread Scheme dialects (Guile [2], Racket [4], Chez Scheme [18]). An approach using a foreign function interface to a library like Redland `librdf` may have been more straightforward, but the R^{N}RS standard reports have never defined nor required an FFI[6]. Developing a pure-Scheme library is also advantageous as a key development in recent years has been compilation of Scheme to WebAssembly[7], where external dependencies are cumbersome and native implementations are favoured [55]. Chez Scheme was selected because as well as supporting the R⁶RS standard, it is considered one of the fastest Scheme implementations[8]. The library is released as open source, under an LGPLv3 license, which allowed for direct reuse of the isomorphism and entailment components of Guile-RDF [28, 29].

4.2 Pre-processing for concrete syntaxes

Input to Keep's Nanopass framework is a parser from s-expressions to language terms. This made it necessary to process a given concrete syntax into an s-expression corresponding to an abstract syntax tree. The Turtle concrete syntax is not built on top of an existing interchange format, so it was sufficient to recognise a Turtle document in terms of its grammar using a parser combinator approach alone. In contrast, for RDF/XML, existing XML parsers were used, but additional work was required to *recognise* the AST in terms of RDF/XML. In fact, for RDF/XML, the "reader" was based on SXML/SSAX [40], followed by a number of transformations using Nanopass. For N-Triples (a subset of Turtle), the Turtle reader stage was used during initial testing for compliance, with a more efficient reader implemented later using many of the same parser combinators.

4.3 Processing of collections

RDF collections are ordered lists of objects, very similar to linked lists. Both RDF/XML and Turtle have explicit support for linked lists within the syntax, and processing was very similar. The collection form is generative of blank nodes: each list element has a blank node allocated, with an `rdf:first` statement made between the blank node and the object, and an `rdf:rest` statement made between the blank node and the next blank node in the sequence, or a terminal `rdf:nil` statement. Processing these interacts with other syntax-specific forms, especially those which may also generate blank nodes after processing. RDF/XML had few of these forms, whereas Turtle had additional syntax for embedding a list of predicate/object pairs corresponding to a generated blank node.

Table 2. Processing stages for Turtle shared with N-Triples and RDF/XML

Processing stage	Turtle	N-Triple	RDF/XML
Reader to s-expressions	✔	✗	✗
Substitute core RDF datatypes	✔	✗	✗
Flatten blank node lists	✔	✗	✗
Flatten collections (linked lists)	✔	✗	✔
Generative subject blocks	✔	✗	✗
Generative object blocks	✔	✗	✗
Skolemise blank nodes	✔	✔	✔
Flatten triple blocks	✔	✗	✔
Extract RDF graph (triples)	✔	✔	✔

4.4 Skolemisation

Skolemisation is the process of allocating a new, globally-unique label to every blank node, to ensure that they remain locally scoped to the document [20]. There is relatively little to say about this transformation stage *in isolation*, because the pass itself is compact, having been separated from other passes which *generate* blank nodes from concrete syntax-specific syntactic forms. Separating this stage from forms which generated blank nodes was advantageous because it was immediately reusable across the three concrete syntaxes, regardless of whether they had many generative components (Turtle), few (RDF/XML), or none at all (N-Triples). The nanopass approach led to more terse transformation passes for Turtle's generative forms, with less pattern-matching. That is, it is conjectured that separating skolemisation supported correctness, as it reduced the complexity of the other passes.

4.5 Joining up the RDF/XML and Turtle pipelines

The library was initially developed around the Turtle concrete syntax, with the more efficient N-Triples implementation added later (smaller reader stage, unneeded Turtle processing passes omitted). In contrast, RDF/XML was implemented in two stages. First, passes were implemented

[6] R⁶RS PFFI: https://github.com/ktakashi/r6rs-pffi
[7] Hoot: Scheme on WebAssembly: https://spritely.institute/hoot/
[8] R⁷RS benchmarks: https://ecraven.github.io/r7rs-benchmarks/

```
(define-language TTLinitial
  (terminals
    (literal (lit)) (blank-node (bn)) (iri (ident))
    (reference (ref)) (local-reference (refL)))
  (entry TurtleDoc)
  (Ident/PN (ident/pn)
    ident (prefixed-name ref refL))
  (Literal (dl)
    (declare-literal ref ident/pn))
  (Subject (s)    ident/pn bn bns lst)
  (Predicate (p)  ident/pn)
  (Object (o)     ident/pn bn lit dl bns lst)
  (BlankNodeList (bns)
    (declare-anonymous-list (p os) ...))
  (ObjectList (os)   o (o ...))
  (Collection (lst)  (collection o ...))
  (Directive (dp)
    (declare-prefix ref ident)
    (declare-base-prefix ident))
  (Triples (trs)
    (declare-triples s (p os) ...))
  (Statement (stmt)  dp trs)
  (TurtleDoc (stmts) (stmt* ...)))
```

Figure 2. Example 1: Initial Turtle document language

which recognised an XML document in terms of the RDF/XML grammar [25]. Each test case in the W3C test suite was tested against this to reveal programming errors as each pass was implemented. Second, passes were developed which transformed the resulting syntax tree into one of the Turtle intermediate representations. This intermediate representation was equivalent to triple blocks with collections (see row 4 of table 2), i.e. collections to be expanded, followed by skolemisation of the entire document. A complete summary of the processing steps implemented for Turtle, and where they were reused by N-Triples and RDF/XML, is given in table 2.

It was observed that without additional work, around 3/4 of the RDF/XML cases with a testable result passed, which supports the viability of this approach. Beyond bug-fixes in treatment of relative IRIs, the main modification necessary was to support RDF/XML's reification of collections.

5 Nanopass usage examples

Example 1 (fig. 2) is an initial language corresponding closely to the Turtle grammar ([9]). The reader module for Turtle produces s-expressions which are then parsed by the Nanopass framework into this language form. There is a distinct terminal for IRIs, and a production `prefixed-name` which is expanded to an IRI as the transformation proceeds. Similarly, literals may include unexpanded prefixed names, there is a terminal `declare-literal` indicating expansion.

Example 2 (fig. 3) defines a language which eliminates prefix declarations and use of prefixed names within the document body. The associated transformation pass (omitted for brevity) expands these prefixed names into the full IRI.

```
(define-language TTLdatatypes
  (extends TTLinitial)
  (terminals
    (- (reference (ref)))
    (- (local-reference (refL))))
  (Ident/PN (ident/pn)
    (- ident)
    (- (prefixed-name ref refL)))
  (Literal (dl)
    (- (declare-literal ref ident/pn)))
  (Subject (s)
    (- ident/pn)
    (+ ident))
  (Predicate (p)
    (- ident/pn)
    (+ ident))
  (Object (o)
    (- ident/pn)
    (- dl)
    (+ ident))
  (Directive (dp)
    (- (declare-prefix ref ident))
    (- (declare-base-prefix ident)))
  (Statement (stmt)
    (- dp)))
```

Figure 3. Example 2: Language eliminating Turtle prefixes

```
(define-language TTLunwrapped
  (extends TTLdatatypes)
  (Statement (stmt)
    (- trs))
  (TurtleDoc (stmts)
    (- (stmt* ...))
    (+ (trs ...))))

(define-pass unwrap
  : TTLdatatypes (stmts) -> TTLunwrapped ()
  (Statement : Statement (stmt) -> Triples ())
  (TurtleDoc : TurtleDoc (stmts) -> TurtleDoc ()))
```

Figure 4. Example 3: Code generation to eliminate redundant statements

Prefixed names can also be used within literal datatype IRIs, and these are transformed into a literal record type proper.

The intermediate representation in example 2 (fig. 3) eliminated prefix declarations, meaning that the `Statement` production is redundant, now that it includes only triple blocks. This example defines an intermediate representation where all statements are replaced by triple blocks, and a pass which leverages the meta-framework's code generation to generate transformers.

Example 4 (fig. 5) shows how skolemisation occurs. A mutable hashtable is set up which records mappings of old labels to newly-allocated nodes, with a current count recorded as a boxed integer (integer with single mutable state).

Example 5 (fig. 6) shows how two very similar intermediate languages are joined up. In this case, the main difference between these was that RDF/XML documents had a root node which consisted of semi-final collections of triples

```
(define-language TTLskolemised
    (extends TTLexplicit-objects)
    (terminals (- (blank-node (bn))))
  (Subject (s) (- bn))
  (Object (o)  (- bn)))

(define (skolem-increment! sk)
  (let ([val (unbox sk)])
    (set-box! sk (+ val 1))
    (make-new-blank-node
    (format "genid~a" val))))

(define (compare-skolem! bn sk mappings)
  (let ([old-lab (blank-node-label bn)])
   (or (hashtable-ref mappings old-lab #f)
       (let ([new-bn (skolem-increment! sk)])
         (hashtable-set! mappings old-lab new-bn)
         new-bn))))

(define-pass skolemise-explicit
  : TTLexplicit-objects (stmts sk mapping)
    -> TTLskolemised ()
  (Subject : Subject (s) -> Subject ()
    [,bn (compare-skolem! bn sk mapping)]
    [else `,s])
  (Object : Object (o) -> Object ()
    [,bn (compare-skolem! bn sk mapping)]
    [else `,o]))
```

Figure 5. Example 4: Skolemisation

```
(define-pass emit-common
  : RDF+XMLprop-triples (top) -> TTLflat-objects ()
  (split-p+o : Pred+Obj (p+o) -> * (obj)
    [(,p ,o) (values p o)])
  (Subject : Subject (s) -> Subject ())
  (Predicate : Predicate (p) -> Predicate ())
  (Object : Object (o) -> Object ())
  (Collection : Collection (lst) -> Collection ())
  (Triples : Triples (trs) -> Triples ()
    [(declare-triples ,s ,p+o* ...)
     (let loop ([pool p+o*] [ps '()] [os '()])
       (match pool
         ['()
          `(declare-triples ,(Subject s)
                            (,ps ,os) ...)]
         [(cons head tail)
          (call-with-values
              (lambda () (split-p+o head))
            (lambda (p o)
              (loop tail
                    (cons (Predicate p) ps)
                    (cons (Object o) os))))]))])
  (Top : Top (top) -> TurtleDoc ()
    [(*TRIPLES* ,trs ...)
     `(,(map Triples trs) ...)]))
```

Figure 6. Example 5: Joining up RDF/XML and Turtle

grouped by subject. This pass likely could be made more terse: note in the `Triples` transformer, the input language term represent predicate/object pairs by a dedicated production (corresponding to `(p o)` in the grammar), whereas the output language term has `(p o)` explicitly, with no dedicated

```
(define (process-TTL xs default-base)
  (let ([skolem-count (box 0)])
    (chain xs
      (substitute-datatypes _ default-base)
      (unwrap-statements _)
      (flatten-objects _)
      (flatten-blank-node-lists _)
      (flatten-linked-lists _ skolem-count)
      (flatten-collections _ skolem-count)
      (flatten-implicit-subjects _ skolem-count)
      (flatten-implicit-objects _ skolem-count)
      (flatten-explicit _ skolem-count)
      (flatten-triple-blocks _))))

(define (process-XML-as-TTL xs)
  (let ([skolem-count (box 0)])
    (chain xs
      (flatten-linked-lists _ skolem-count)
      (skolemise-implicit-subjects _ skolem-count)
      (skolemise-implicit-objects _ skolem-count)
      (skolemise-explicit _ skolem-count)
      (flatten-triple-blocks _))))
```

Figure 7. Example 6: Turtle and RDF/XML modular pipelines of passes

production. Note how this pass is otherwise automatically generating transformers from identical terms like `Subject`.

Example 6 (fig. 7) shows how passes are reused in different processing pipelines. The first procedure processes a Turtle document after parsing to a Nanopass intermediate representation. The second procedure processes an RDF/XML document immediately after transformation in the previous example (fig. 6). For RDF/XML, the main portions reused are the flattening of collections, and skolemisation of the whole document (first and fourth stages). The transformations in the remaining stages largely do not trigger, and simpler passes could be defined instead. Construction of these pipelines[9] does not use the Nanopass DSL at all, as transformation passes become Scheme procedures after definition.

6 Compliance with W3C standards

To ensure that this library generates conformant RDF graphs from input documents, procedures for each concrete syntax were tested directly against public W3C test suites.

The W3C standards define for each concrete syntax test suites which every compliant implementation must pass. These suites test that processing some RDF document produces an RDF graph which is isomorphic to a testable result[10]. Isomorphism with respect to blank nodes is necessary because the label verbatim on blank nodes is not meaningful and is an artefact of the serialisation. The result is given as an N-Triples document, which of the standard RDF concrete syntaxes requires the least processing (an RDF graph being a set of triples).

[9] SRFI 197 `chain` is like Elixir, F#, R's `|>` pipeline operator.
[10] RDF Test Suites: https://www.w3.org/2011/rdf-wg/wiki/RDF_Test_Suites

Additionally, test cases for model theory semantics (namely entailment) have a testable result or indicate non-entailment. Testing is similar to that of concrete syntaxes, but entailment procedures are used instead, and entailment may be with respect to a set of allowed datatypes.

Testing was bootstrapped from the Guile-RDF library, using a script which processed a given test suite's RDF metadata into Scheme syntax. This syntax was loaded and run directly by the new tooling. Later, this script was rewritten for Chez, and the test syntax regenerated.

The new tooling is compliant with the following test suites:

- RDF 1.1 RDF/XML syntax tests[11]
- RDF 1.1 Turtle tests[12]
- RDF 1.1 N-Triples tests[13]
- RDF 1.1 Schema and semantics tests[14]

7 Strengths and limitations of the nanopass approach

One of the strengths of the meta-framework embedded in a modern *Scheme* system was leveraging meta-programming to hide (extensive) code generation behind a domain-specific language, while integrating with the wider Scheme environment. This DSL interface let us incrementally, and tersely, define languages as extensions of one another. As described in section 4.5, RDF/XML support was added by employing nanopass to recognise an XML document in terms of RDF/XML, followed by processing steps which eventually joined up with those of Turtle, eliciting an RDF graph. Similar to how RDF/XML is built on top of XML, it is not difficult to imagine a similar approach to our RDF/XML implementation going further, letting us define additional layers of languages based on the RDF standards, like ShEx or RDF-Star.

While the meta-framework allowed us to omit a lot of boilerplate we would have had to write manually, certain patterns repeatedly arose which were beyond the meta-framework. Most commonly, these were procedures which transformed syntax belonging to a single intermediate representation, but were used within passes which transformed from one to another. This meant we could not rely on the DSL within that pass, and had to manually invoke the internal macros `nanopass-case` and `with-output-language`. Another pattern was keeping track of state (e.g. expansion of RDF/XML `rdf:li` collection elements). Although the meta-framework does support passing arguments around implicitly termed catamorphism syntax), we did not employ it, and typically used boxed values with shared mutable state.

Speaking more generally about the applicability of the approach, we found that the transformations corresponding to the most declarative parts of different RDF syntaxes tended to be the most terse. For example, in Turtle, certain syntactic forms depend on positional information and state, such as skolemisation of blank nodes and expansion of prefixed names. Passes focused on a single syntactic transformation, between two distinct intermediate representations were a) the most terse; and b) had the fewest programming errors. In our experience, intermediate representations which made the fewest, most specific changes tended to require less complex business logic: we were less likely to write non-exhaustive pattern-matching clauses and we relied on mutable state the least.

More specifically, we found that transformations working within a single intermediate representation were most error-prone, because they were more likely to preserve syntactic forms which we were aiming to eliminate. In contrast, transformations to a different representation, which eliminated a particular syntactic form, forced the meta-framework to raise an error at runtime if obsolete syntax were still present. For example, we originally wrote a pass which eliminated collections, which transformed from a representation `TTLflat-objects` to the same `TTLflat-objects`. We rewrote the pass to output `TTLno-collections`, which modified `TTLflat-objects` to eliminate collections entirely. While we have just described working within a meta-framework, we expect that a similar pattern would appear in other meta-frameworks and nanopass approaches, because it seems to arise from the general nanopass approach of numerous intermediaries and transformers. Indeed, after implementing the RDF/XML concrete syntax, and testing with more complex Turtle documents, we rewrote many of the Turtle transformation passes, eliminating the errors we found during testing.

8 Performance

8.1 Test setup

Processing of RDF documents is a pipeline made up a number of transformation passes (see table 2). The aim was to measure the time for each component pass, as well as processing the entire document. Hardware used was an Apple M4 Macbook Air (MacOS 26.3.2) with 16GB of LPDDR5X-7500 memory, and 4 performance cores and 6 efficiency cores, and 512GB of storage. Chez Scheme's `statistics` procedure was used to get the current CPU time. For a given transformation pass, a wrapper procedure sampled using `statistics` information about the system immediately before, and after, running. This produced a series of pre-/post-stage times for each stage in a processing pipeline. Processing of each RDF document was performed ten times, for both an RDF/XML representation and an equivalent RDF/Turtle representation,

[11] RDF/XML Syntax tests: https://w3c.github.io/rdf-tests/rdf/rdf11/rdf-xml/index.html
[12] Turtle tests: https://w3c.github.io/rdf-tests/rdf/rdf11/rdf-turtle/index.html
[13] N-Triples tests: https://w3c.github.io/rdf-tests/rdf/rdf11/rdf-n-triples/index.html
[14] RDF Schema and Semantics tests: https://w3c.github.io/rdf-tests/rdf/rdf11/rdf-mt/index.html

Table 3. RDF/XML processing time (mean CPU time, milliseconds)

Stage	MarineTLO	MarineTLO (SD)	PROV	PROV (SD)	KM4City	KM4City (SD)
reader to AST	3.0854	0.1019	1.9179	0.2282	20.7688	1.5726
eliminate XML PI/comments	0.1025	0.0836	0.0488	0.0044	0.7813	0.5338
recognise RDF/XML AST top	0.0059	0.001	0.0014	4e-04	0.0328	0.0068
recognise attributes (1)	0.0585	0.0033	0.0482	0.0039	0.8275	0.67
recognise attributes (2)	5.6672	0.2783	2.2963	0.3936	32.713	4.1771
recognise properties	0.0533	0.0051	0.0457	0.0055	0.7178	0.4465
recognise nodes	0.1105	0.1116	0.0663	0.0048	0.7374	0.5376
expand attributes	0.0781	0.0063	0.074	0.0343	0.9123	0.2934
expand nodes	4.1922	3.4207	1.0589	0.0928	22.4232	4.0819
count rdf:li elements	0.0306	0.0034	0.0257	0.0027	0.2417	0.0769
process properties	10.9024	0.6899	9.5455	3.5997	239.4893	22.5401
Turtle-like representation	0.0293	0.002	0.0377	0.0064	0.2522	0.0851
flatten collections	0.5602	0.2285	0.3945	0.0361	3.5415	0.3317
skolemise anon. lists (subject; skipped)	0.4774	0.0202	0.3824	0.0228	3.3875	0.1346
skolemise anon. lists (object; skipped)	0.5029	0.0706	0.3983	0.0773	3.7126	0.8006
skolemise whole document	0.0597	0.0038	0.0838	0.0249	0.5608	0.1562
flatten triple blocks (skipped)	0.8043	0.0723	0.8058	0.1475	5.2146	0.1228
total	26.7204	3.8435	17.2313	4.1704	336.3142	20.4351

Table 4. Turtle processing (mean CPU time, milliseconds)

Stage	MarineTLO	MarineTLO (SD)	PROV	PROV (SD)	KM4City	KM4City (SD)
reader to AST	48.1319	4.7299	30.4262	4.4713	171.7223	10.4283
substitute datatypes	13.091	0.2488	13.0545	0.2073	58.8438	1.6068
eliminate prefixes	0.0381	0.0053	0.0665	0.0095	0.268	0.0857
flatten comma-separated objects (1)	0.014	8e-04	0.036	0.0298	0.0977	0.0035
flatten comma-separated objects (2)	0.0158	8e-04	0.0346	0.024	0.1059	0.0421
flatten collections	0.0469	0.0018	0.096	0.0248	0.3641	0.0885
skolemise anon. lists (subject)	0.0664	0.06	0.0834	0.0021	0.4218	0.2894
skolemise anon. lists (object)	0.0455	0.0022	0.0812	0.0027	0.3817	0.0995
skolemise whole document	0.0514	0.0377	0.0448	0.0027	0.1391	0.0957
flatten grouped triple blocks	0.285	0.0258	0.4075	0.0022	2.2833	0.8163
total	61.7862	4.8356	44.3307	4.4267	234.6277	9.3964

Table 5. Guile-RDF Turtle processing (mean CPU time, milliseconds)

MarineTLO	MarineTLO (SD)	PROV	PROV (SD)	KM4City	KM4City (SD)
95.4806	11.541	75.8937	10.0559	257.4466	15.3873

conversion between which was using the Redland `rapper` command-line tool. A pause of 500ms between each run was taken. The *total* run was the aggregated time for each stage to run. Mean values were taken, then rounded to four decimal places and summarised in table 4 and table 3. Table 5 additionally summarises the equivalent of this test setup for Guile-RDF, albeit limited to the total time for a given document transformation.

Test data selected are from the following sources:

- The MarineTLO ontology [56]: 1243 triples
- The PROV-O ontology [10]: 1817 triples
- The KM4City ontology [11]: 8015 triples

8.2 Test data

Test data were selected because OWL's RDF/XML serialisation makes use of RDF collections[15]. The `rapper` conversion to the Turtle syntax makes use of blank node lists, and collections are expressed using the collection syntax (rather than explicit `first`/`rest` statements), so that the Turtle file is readable and terse. The assumption is that this is a realistic test of the various Nanopass-based transformations, and that these are comparable in size to many real-world RDF documents.

[15]MarineTLO and KM4City were OWL/XML documents, and had to be processed with the `librdf` command-line `rapper` tool, because the SSAX-based parser to an XML AST tripped up on these files, which may be a programming error in the SSAX implementation.

8.3 Test results

Mean timings, rounded to four decimal places, as well as associated standard deviation, are summarised in table 3 and table 4. Table 5 additionally gives the equivalent totals for the Guile-RDF library, for Turtle (as Guile-RDF does not implement RDF/XML). Of the totals, the new library was around 1/3 faster than Guile-RDF for the two smaller test files, and at least as fast for KM4City.

For the Turtle syntax, the majority of processing was in the parser combinator based reader to a syntax tree, followed by substituting datatypes, whereas for RDF/XML, the majority of processing was in recognising attributes on XML nodes in terms of RDF/XML, followed by expanding node elements, and property elements. This was consistent across the three documents.

We emphasise that a more direct, granular comparison between our work and Guile-RDF cannot be made, because a) Guile-RDF's parsing and transformation steps are tightly integrated, effectively forming a single step; and b) adaptation of Guile-RDF to a nanopass approach was infeasible, partly impeded by lack of meta-frameworks (no port of Keep's Nanopass framework to Guile) at the time of writing.

This speedup may be unintuitive, as nanopass approaches incur overhead during transformations over each intermediate representation. As discussed in section 7, the nanonpass approach helped us write more maintainable transformation passes, which we were able to optimise individually (modularisation). There is independent evidence that the nanopass approach does support writing performant code: the adaption of Chez Scheme to Keep's Nanopass framework did increase overall speed [39] by up to 27%. Finally, it could be attributed to the underlying Scheme implementation: in existing benchmarks of Scheme systems, Chez is consistently one of the fastest Scheme systems, including faster than Guile[16]. However, we also refer readers to our discussion immediately above of factors impeding direct comparison.

9 Conclusion and future work

To overcome the challenges of an inconsistent and fragmented landscape of RDF library implementations, this work demonstrated that a *nanopass* architecture is well-suited to processing RDF concrete syntaxes. To the best of our knowledge, this work contributes the first implementation of RDF 1.1 based on a nanopass architecture, and the first single compliant implementation of the RDF/XML, Turtle and N-Triples concrete syntaxes for Scheme. This work showed that despite having a wildly different external, concrete representation, the RDF/XML processor could reuse significant portions (namely flattening of collections, and skolemisation) of that developed for Turtle. This was facilitated directly by the nanopass approach, which effectively modularised these, allowing them to be reused in isolation. Furthermore, the library demonstrates adequate performance for real-world ontologies, demonstrating an extensible and performant approach.

The library is open source software[28, 29] and it is anticipated that both the RFC 3986/3987 implementation and the RDF library itself would be a good fit for the community *SRFI* process[17], given the persistent interest from the Scheme community in RDF to date. This would also be an opportunity to review existing work's application programming interface, for example by using a data structure representing a true ordered set of RDF triples [19], or a language tag programming interface[18].

There appear to be similarities between certain processing stages and attribute grammars (as suggested by Ringo et al. [45]), e.g. recognition of RDF/XML grammar elements; passing up blank node counts during skolemisation. Attribute grammars might reduce boilerplate and reliance on mutable state. Another underlying pattern was that RDF/XML transformation passes mirrored the XML events in its specification, whereas the RDF/XML reader recognised RDF/XML *grammar* elements alone. It is feasible that XML event-based parsers could be expressible using a nanopass approach.

This library is compliant with the W3C standards but more exhaustive, deeply nested test data, or a fuzzing approach, would be productive. An alternative approach to finding or generating larger test data might be to show correctness of passes, both individually or in aggregate. This is a theme of some of the other work around nanopass compilers [22]. Finally, future work could support additional concrete syntaxes, such as JSON-LD and Notation3, as well as the upcoming RDF 1.2 standard, or non-standard extensions like generalised RDF triples, or RDF-Star.

Acknowledgments

Work was supported by EU Horizon NEMECYS (101094323), with later work supported by EPSRC grant TransiT (EP/Z533221/1).

References

[1] GNU's programming and extension language, . URL https://www.gnu.org/software/guile/.
[2] R⁶RS support (Guile reference manual), . URL https://www.gnu.org/software/guile/manual/html_node/R6RS-Support.html.
[3] *P4 Open Source Programming Language*. URL https://p4.org/.
[4] Racket R⁶RS: Scheme. URL https://docs.racket-lang.org/r6rs/index.html.
[5] R. Angles, H. Thakkar, and D. Tomaszuk. Mapping RDF databases to property graph databases. *IEEE Access*, 8:86091–86110, April 2020.
[6] M. Ballantyne, M. Gamburg, and J. Hemann. Compiled, extensible, multi-language DSLs (Functional Pearl). *Proceedings of the ACM on Programming Languages: ICFP*, 8(238):64–87, August 2024.
[7] D. Beckett. Redland RDF libraries, 2000. URL https://librdf.org/.

[16] R⁷RS benchmarks: https://ecraven.github.io/r7rs-benchmarks/

[17] Scheme Requests for Implementation (SRFI): https://srfi.schemers.org/

[18] RFC 5646: Tags for Identifying Languages: https://www.rfc-editor.org/rfc/rfc5646.txt

[8] D. Beckett. RDF 1.1 N-Triples: A line-based syntax for an RDF graph. Technical report, W3C, February 2014. URL https://www.w3.org/TR/rdf11-n-triples.

[9] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, and G. Carothers. RDF 1.1 Turtle: Terse RDF triple language. Technical report, W3C, February 2014. URL https://www.w3.org/TR/rdf11-turtle.

[10] K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao. PROV-O: The PROV Ontology. Technical report, W3C, April 2013. URL https://www.w3.org/TR/prov-o.

[11] P. Bellini, M. Fanfani, P. Nesi, and M. Soderi. km4city, the DISIT Knowledge Model for City and Mobility. Technical report, DISIT Lab, July 2024. URL http://www.disit.org/km4city/schema.

[12] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic syntax, January 2005. URL https://www.rfc-editor.org/rfc/rfc3986.

[13] N. C. Brown and A. T. Sampson. Alloy: fast generic transformations for Haskell. *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 105–116, September 2009. doi: https://doi.org/10.1145/1596638.1596652.

[14] T. R. Campbell. *Schemantic Web version 0.0 (beta)*, 2007. URL https://mumble.net/~campbell/darcs/schemantic-web/.

[15] G. Carothers. RDF 1.1 N-Quads: A line-based syntax for RDF datasets. Technical report, W3C, February 2014. URL https://www.w3.org/TR/n-quads.

[16] J. Carroll, I. Herman, and P. F. Patel-Schneider. OWL 2 Web Ontology Language RDF-based semantics (second edition). Technical report, W3C, December 2012. URL https://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/.

[17] A. Chauhan, A. Keep, C.-Y. Shei, and P. Ratnalikar. RubyWrite: A Ruby-embedded domain-specific language for high-level transformations. Technical Report TR704, Indiana University Computer Science, January 2013.

[18] *Chez Scheme Version 10.0.0 User's Guide*. Cisco Systems, Inc., February 2024.

[19] J. Cowan. *SRFI 153: Ordered Sets*, May 2023. URL https://srfi.schemers.org/srfi-153/srfi-153.html.

[20] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 concepts and abstract syntax. Technical report, W3C, February 2014. URL https://www.w3.org/TR/rdf11-concepts/.

[21] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs), January 2005. URL https://www.rfc-editor.org/rfc/rfc3987.

[22] O. Ebersafe, I. Zhao, E. Jin, A. Bright, C. Jian, and Y. Zhang. Certified compilers à la carte. *Proceedings of the ACM on Programming Languages: PLDI*, 9(162):372–395, June 2025.

[23] M. Flatt and R. B. Findler. The Racket Guide. URL https://docs.racket-lang.org/guide/index.html.

[24] D. P. Friedman, W. E. Byrd, O. Kiselyov, and J. Hemann. *The Reasoned Schemer*. The MIT Press, second edition, 2018.

[25] F. Gandon and G. Schreiber. RDF 1.1 XML syntax. Technical report, W3C, February 2014. URL https://www.w3.org/TR/rdf11-xml.

[26] B. C. Grau, I. Horrocks, B. Parsia, A. Ruttenberg, and M. Schneider. OWL 2 Web Ontology Language mapping to RDF graphs. Technical report, W3C, December 2012. URL https://www.w3.org/TR/2012/REC-owl2-mapping-to-rdf-20121211/.

[27] N. Gray. *The RDF library, wrapping the Redland RDF library*, September 2010. URL https://nxg.me.uk/dist/racket-librdf/.

[28] D. Guthrie. *URI and IRI utilities for R⁶RS Scheme (RFC 3986 and RFC 3987)*, February 2026. URL https://codeberg.org/dguthrie/scheme-iri.

[29] D. Guthrie. *Modular RDF implementation for R⁶RS Scheme*, February 2026. URL https://codeberg.org/dguthrie/scheme-rdf.

[30] S. Hawke, M. Horridge, B. Parsia, and M. Schneider. OWL 2 Web Ontology Language conformance (second edition). Technical report, W3C, December 2012. URL https://www.w3.org/TR/2012/REC-owl2-conformance-20121211.

[31] P. J. Hayes and P. F. Patel-Schneider. RDF 1.1 semantics. Technical report, W3C, February 2014. URL https://www.w3.org/TR/rdf11-mt.

[32] A. Hogan. The semantic web: Two decades on. *Semantic Web*, 11(1):169–185, 2020. doi: 10.3233/SW-190387. URL https://journals.sagepub.com/doi/abs/10.3233/SW-190387.

[33] E. Holk, W. E. Byrd, J. Willcock, T. Hoefler, A. Chauhan, and A. Lumsdaine. Kanor: A declarative language for explicit communication. *Practical Aspects of Declarative Languages (PADL 2011)*, pages 190–204, 2011.

[34] A. W. Hsu. Co-dfns: Ancient language, modern compiler. *ARRAY'14: Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pages 62–67, June 2014.

[35] A. W. Hsu. A data parallel compiler hosted on the GPU. Indiana University, 2019.

[36] S. Johnston. *Racket Package* `rdf-core`, March 2024. URL https://github.com/johnstonskj/racket-rdf-core.

[37] S. Johnston. *Racket Package* `rdf-io`, May 2024. URL https://github.com/johnstonskj/racket-rdf-io.

[38] A. W. Keep. Nanopass framework users guide. 2013.

[39] A. W. Keep and R. K. Dybvig. A nanopass framework for commercial compiler development. *ACM SIGPLAN Notices*, 48(9):343–350, September 2013. doi: https://doi.org/10.1145/2544174.2500618.

[40] O. Kiselyov. SXML specification. *ACM SIGPLAN Notices*, 37(6):52–58, June 2002. doi: https://doi.org/10.1145/571727.571736.

[41] Y. Lee, K. Gopinathan, Z. Yang, M. Flatt, and I. Sergey. DSLs in Racket: You want it how, now? In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*, SLE '24, pages 84–103, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400711800. doi: 10.1145/3687997.3695645. URL https://doi.org/10.1145/3687997.3695645.

[42] C. Lemmer-Webber. Linked data tooling for the Racket programming language, December 2017. URL https://github.com/cwebber/racket-linkeddata/.

[43] J. Lepiller. *Implementation of RDF formats and algorithms in GNU Guile*, April 2020. URL https://framagit.org/tyreunom/guile-rdf.

[44] M. Nieper-Wißkirchen. Extending a language — writing powerful macros in Scheme. URL https://mnieper.github.io/scheme-macros/README.html.

[45] N. Ringo, L. Kramer, and E. R. V. Wyk. Nanopass attribute grammars. *SLE 2023: Preceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*, pages 70–83, October 2023.

[46] F. Ruffy, T. Wang, and A. Sivaraman. Gauntlet: Finding bugs in compilers for programmable packet processing. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, November 2020.

[47] A. T. Sampson and N. C. Brown. Generics in small doses: Nanopass compilation with Haskell. 2008.

[48] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. *ICFP '04: ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, (19):201–212, September 2004. doi: https://doi.org/10.1145/1016850.1016878.

[49] G. Schreiber and Y. Raimond. RDF 1.1 primer. Technical report, W3C, June 2014. URL https://www.w3.org/TR/rdf-primer/.

[50] A. Shinn, J. Cowan, and A. A. Gleckler. *Revised⁷ Report on the Algorithmic Language Scheme*. 2013.

[51] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten. *Revised⁶ Report on the Algorithmic Language Scheme*. Cambridge University Press, 2009.

[52] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, P.-A. Champin, and N. Lindström. JSON-LD 1.1: A JSON-based serialization for linked data. Technical report, W3C, July 2020.

[53] S. Staworko, I. Boneva, J. E. L. Gayo, S. Hym, E. G. Prud'hommeaux, and H. Solbrig. Complexity and expressiveness of ShEx for RDF. *International Conference on Database Theory*, 18th International Conference on Database Theory (ICDT 2015), March 2015.

[54] G. J. Sussman and G. L. S. Jr. Scheme: An interpreter for extended lambda calculus. MIT AI Memo 349, Massachusetts Institute of Technology Artificial Intelligence Laboratory, December 1975.

[55] D. Thompson. Functional hash tables explained, May 2025. URL https://spritely.institute/news/functional-hash-tables-explained.html.

[56] Y. Tzitzikas, C. Alloca, C. Bekiari, Y. Marketakis, P. Fafalios, M. Doerr, N. Minadakis, T. Patkos, and L. Candela. Integrating heterogeneous and distributed information about marine species through a top level ontology. In *Proceedings of the 7th Metadata and Semantic Research Conference (MTSR'13)*, Thessaloniki, Greece, November 2013.