BUILDING DOMAIN-SPECIFIC LANGUAGES WITH MULTI-LANGUAGE MACROS

MICHAEL BALLANTYNE

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Khoury College of Computer Sciences
Northeastern University
Boston, Massachusetts

2025

Northeastern University
Khoury College of
Computer Sciences

PhD Thesis Approval

Thesis Title: Building Domain-Specific Languages with Multi-Language Macros

Author: Michael Ballantyne

PhD Program: ✓ Computer Science ____ Cybersecurity ____ Personal Health Informatics

PhD Thesis Approval to complete all degree requirements for the above PhD program.

Thesis Advisor	14 Aug 2025 Date
Thesis Reader	17 Aug '25 Date
Thesis Reader	17 Aug 2025 Date
Thesis Reader	August 17 2025 Date
Thesis Reader	18 August, 2025 Date
Thesis Reader	18-AUG-2025 Date

KHOURY COLLEGE APPROVAL:

Associate Dean for Graduate Programs	18 August, 2025 Date

COPY RECEIVED BY GRADUATE STUDENT SERVICES:

Pete Morency Recipient's Signature	8/20/25 Date

Distribution: Once completed, this form should be attached as page 2 immediately following the

ABSTRACT

Macros are commonly used to implement domain-specific languages (DSLs). However, macro technology was originally designed for implementing syntactic sugar, not DSLs. As a consequence, it can be difficult for macro-based DSL implementations to provide static semantics, optimizing compilation, or specialized IDE services. Protecting the invariants of the DSL where it interacts with the host also poses challenges. My dissertation confirms that a macro system specifically designed around the needs of DSLs can alleviate these problems. In particular, such a macro system can adopt the structure of a multi-language semantics, with separate host and DSL syntaxes connected via explicit boundary forms to mediate the interaction between languages. I have implemented such a multi-language-oriented macro system as a layer on top of Racket's conventional macros. The two kinds of macros work well together, resulting in DSL implementations that support a combination of optimizing compilation, protected host-language interaction, and extensibility.

ACKNOWLEDGMENTS

My chief thanks is to my advisor Matthias Felleisen, whose patience and adaptability got me through the long process of completing a PhD. I have benefitted immensely from the seriousness with which he approaches all aspects of being an academic—research and teaching with a vision, mentoring with long-term commitment, and building a research community that can sustain the work. I am grateful to my committee members Amal Ahmed, David Thrane Christiansen, Matthew Flatt, Arjun Guha, and Niko Matsakis for their extensive feedback and suggestions for relating my work to the broader context. Thanks also to my first PhD advisor Matt Might for starting me down the path by introducing me to the field and providing funding that allowed me to explore my interests.

Turning the ideas in this dissertation into reality has been so much more enjoyable with collaborators who see the same possibilities. Thank you to Mike Delmonaco, Jason Hemann, Mitch Gamberg, Cameron Moy, Alexis King, Siddhartha Kasivajhula, Dominik Pantůček, Nada Amin, Will Byrd, Ari Prakash, Zack Eisbach, Andrey Piterkin, and Luke Jianu for believing in and co-creating these ideas with me. I also had the pleasure of working with Leif Andersen, Stephen Chang, William Hatch and Greg Rosenblatt on other fascinating projects that sustained my interest in programming languages.

Finally, thanks to my partner Jeanne-Marie, my friends Corey, Di, Logan, Celeste, Hyeyoung, and Adair, and my parents for the many kayaking trips, meals, hikes, baked goods, philosophical conversations, and everything else that kept me going.

CONTENTS

Part I

DESIGN

1 INTRODUCTION

1.1 MACROS: AN IMPERFECT MATCH FOR LANGUAGE-ORIENTED PROGRAMMING

Racket (Flatt and PLT, 2010) is an extensible language designed to support language-oriented programming (LOP). Programming in a language-oriented style means solving each part of a programming task in a domain-specific language (DSL) suited to that part of the task (Felleisen et al., 2018; Ward, 1994). The overall program stitches together the various pieces via their connection to a general-purpose programming language. Racket supports this objective via a macro system that allows Racket programmers to define extensions to the syntax of the language.

To support LOP, each DSL needs to provide its special domain-specific advantages while also integrating smoothly with the host language to enable composition of the overall program. The domain-specific advantages of a DSL may come from a specialized syntax, static semantics, runtime execution model, optimizing compiler, or a synthesis of all of these. Thus, it is essential that the implementation technique for such DSLs allow DSL creators to fully realize these attributes of their intended DSL design. The degree of intermixing of DSL and host code as well as runtime protection at the boundary between languages are key issues. When DSL and host code can be intermixed arbitrarily, DSL and host-language code become inseparable, and it is not possible to provide domain-specific static semantics or optimizations. On the other hand, isolating DSL and host code in separate modules limits interoperation between the DSL and host language such that programmers cannot take advantage of their combined expressive power. Similarly, allowing host-language code full access to DSL runtime data prevents the DSL from maintaining desired invariants and equivalences, while completely isolating such data forbids useful interactions.

Previous work on multi-language semantics (Matthews and Findler, 2007) suggests that DSL designers should have precise control of the granularity of interaction. In a multi-language semantics, the grammar and semantics of each language are separately defined, and then connected at specific points in their grammars by explicit boundary forms. The boundary forms provide a hook for translating the static semantics and runtime values between languages, and for installing protective runtime contracts. The choice of locations and semantics of the boundary forms allows DSL designers to carefully control the trade-off between flexible combination of languages on the one hand and analytic and transformative power afforded to a DSL compiler on the other.

Unfortunately, existing macro systems do not align with this multi-language concept. Racket's macro system is primarily designed around the needs of defining syntactic sugar (Felleisen, 1990; Landin, 1964), not the needs of DSLs as multi-language extensions. Macros normally define individual syntactic forms, and all belong to Racket's singular "expression or definition" grammar non-terminal. For a DSL defined in this way, the compilation strategy must consider individual forms in isolation and relegate any enforcement of abstraction boundaries to run-time checks. Considering the individual forms of a language separately also makes it difficult to implement a custom static semantics for the DSL or an optimizing compiler.

Further, the key technologies in the macro system are tailored to the individual-form case. The hygienic macro expansion provided by Racket's "Binding as Sets of Scopes" (Flatt, 2016a) approach assumes that the binding structure of a macro-defined syntactic form is defined by the binding structure of the Racket code to which the macro expands. This approach works well for syntactic sugar, but not for complex optimizing compilers that need to reflect on binding structure and may eliminate bindings and references during compilation. Similarly, the `syntax-parse` pattern matching language (Culpepper, 2012) works well for defining the syntax of an individual form, but does not provide a view of the entire grammatical structure of a DSL.

It is possible to simulate multi-language structure using Racket's tools by creating macros that accept as their syntax an entire DSL fragment. However, the macro system then lacks an understanding of the internal structure of such fragments, such as binding information. This poses a problem when integrating with other host-language syntax embedded within the DSL fragment. The DSL creator must resort to a variety of complicated patterns to communicate information about the DSL syntax to the host-language macro expander (Flatt et al., 2012). Such patterns indicate a lack of expressive power or, conversely, a need for linguistic support.

1.2 THESIS

The discussion in the preceding section suggests an opportunity to explore a new *multi-language macro system* design, oriented from the beginning around the multi-language structure that seems a good fit for language-oriented programming. Thus, my thesis is:

> *Multi-language macros facilitate creating DSLs with static semantics and optimizing compilation that simultaneously integrate fluidly with their host language and admit syntactic extensions.*

I support the thesis by presenting the `syntax-spec` multi-language macro system together with DSLs built using the system and an evaluation that shows that these DSLs realize the desired properties.

My macro system design allows DSL creators to fully realize the domain-specific benefits of their DSL design while also integrating the DSL as tightly as desired with the host language. DSL designs may realize domain-specific benefits via custom syntax, static semantics, compiler optimizations, or even via execution on specialized hardware. Thus, the first primary goal of a multi-language macro system is to equip DSL creators with enough power in each of these aspects to allow them to realize their DSL's intended benefit. Creating parsers, static checkers, and optimizing compilers are complex domain-specific tasks in their own right. The area of language workbench research has identified useful metalanguages such as parser generators (Souza Amorim and Visser, 2020), binding specification languages (Keuchel, Weirich, and Schrijvers, 2016; Konat et al., 2013), and type system specification DSLs (Antwerpen et al., 2018; Chang et al., 2019). A multi-language macro system should support DSL creators by providing similar metalanguages.

The second primary goal of a multi-language macro system is to enable DSLs to benefit from integration with the host. At the most basic, this means allowing DSL code to sit inside host code and host code to sit inside DSL code in the ways specified by a multi-language design's boundary forms. To interconnect these mutually-embedded program fragments, the DSL and host implementations need to integrate at each level: parsing, static semantic checking, and runtime semantics. Of course, a DSL can only connect to the concepts that exist in the host language. Racket uses S-expression syntax, so the natural way of describing the syntactic connection would be via a tree grammar. Racket is untyped but statically scoped, so the key point of static semantic interconnection would be binding structure. Thus, in Racket, a multi-language macro system should provide declarative metalanguages for specifying the tree grammar and binding structure in order to make integration with the host simple. Finally, to connect the runtime semantics, the macro system should allow insertion of runtime checks or value conversion at every point of interaction with the host.

New language features face a long path towards widespread adoption. However, conventional macro systems are relatively widespread, and are available in industrial languages including Clojure (Hickey, 2008), Elixir[1], and Rust [2]. These existing macro systems suggest a point of leverage for adoption, prompting two secondary goals:

1. *A multi-language macro system should be implemented as a layer on top of a conventional macro system.*
2. *Multi-language DSL definitions and conventional syntactic sugar macros should coexist in a single system and work together.*

Building a multi-language macro system as a layer on top of a conventional one could provide a path towards easy adoption in existing languages. With such an

1 https://hexdocs.pm/elixir/macros.html
2 https://doc.rust-lang.org/reference/procedural-macros.html

implementation, the new macro system could be distributed as a library, decoupling its development from the host language. New users could begin creating multi-language DSLs with a simple library import. Coexisting with conventional macros may also aid in adoption by allowing DSL creators to migrate their DSL implementations from conventional macros to multi-language ones as their DSL becomes more complex. A combination of embedding techniques and conventional macros works well for quick prototyping or for discovering a DSL design via iterated abstraction over syntactic patterns. As the DSL design becomes clear, the DSL creator could migrate their implementation to use a multi-language design to enable optimizing compilation. Code from some of the conventional macros in the prototype implementation would become part of the DSL compiler, while others would remain as macros defining syntactic sugar over the DSL core language.

1.3 CONTRIBUTIONS

In response to the ideas of the preceding section, I have developed `syntax-spec`, a metalanguage that extends Racket's macro system to support multi-language DSL definitions. This metalanguage makes multi-language structure explicit, uses declarative specifications of grammar and binding structure to integrate with the host expander, and provides services to support sophisticated DSL compilers. The remainder of Part i of this dissertation demonstrates `syntax-spec` and discusses its design and related work.

Leveraging the binding rules in DSL specifications to extend Racket's hygienic macro expansion to DSL syntax posed the most substantial technical challenge. I developed a formal model of my approach to this integration in PLT Redex as a blueprint to support adoption in future language designs. The first half of Part ii presents this formal model.

While the model presents `syntax-spec` as an extension to Racket's macro expander, the real implementation is actually built as a layer on top the existing Racket system. The `syntax-spec` metalanguage compiles to DSL-specific macro expanders that use a new library I developed atop Racket's compile-time API (Ballantyne, King, and Felleisen, 2020). This architecture allows `syntax-spec` to evolve independently of Racket itself. It also suggests that `syntax-spec`-like multi-language macro systems could be layered on top of other existing procedural macro systems, requiring only limited extensions to the core expanders of those languages. The second half of Part ii explains the details of this implementation approach.

To explore the value of multi-language macros, I have developed seven DSLs in `syntax-spec` and supported students and programmers in Racket's open-source community in implementing a further six DSLs. These DSL implementations provide the basis to evaluate the `syntax-spec` design. My evaluation finds that `syntax-spec` is expressive enough to support a range of DSL designs and that it enables implementations that are substantially more concise than with prior low-

level techniques. However, it also reveals that my binding specification language places certain restrictions on DSL design and that the current implementation imposes penalties on the performance of macro expansion. Part iii presents these applications and the evaluation.

While the `syntax-spec` system is specific to Racket, the design idea of a multi-language macro system is potentially portable to other languages. However, the capabilities of any such multi-language macro system layered atop another host's conventional macro system would depend on what facilities the host exposes via its compile-time API. Some of the operations that the `syntax-spec` implementation relies on are currently only available in Racket. Implementations in other languages may also need to integrate with conventional lexical syntax and type systems. Part iv considers these and other directions for future work.

2 PROGRAMMING WITH MULTI-LANGUAGE MACROS

This chapter demonstrates how programmers declare, extend, and compile multi-language DSLs with `syntax-spec`, via the running example of an implementation of the miniKanren (Byrd, 2009) constraint logic programming DSL. The `syntax-spec` manual in Appendix A provides complete documentation for the system. The content of this chapter is adapted from collaborative work with Mitch Gamburg and Jason Hemann (Ballantyne, Gamburg, and Hemann, 2024).

Figure 2.1 shows the overall architecture of DSL implementations using `syntax-spec`. Programmers write a declarative specification of their DSL in the `syntax-spec` metalanguage. The `syntax-spec` system uses this specification to extend the host macro expander's understanding of syntax and binding structure to encompass the entire host-DSL multi-language. This extended expander checks syntax, expands macros, and provides IDE services. Macro expansion yields a program in the core multi-language syntax combining the DSL and host language. The DSL compiler is responsible for optimizing and compiling the DSL fragments of the multi-language program into host-language code.

2.1 DECLARING A MULTI-LANGUAGE DSL

A `syntax-spec` DSL declaration consists of a grammar and boundary forms connecting the DSL to Racket, both annotated with binding rules. Together with these declarations, the programmer writes a DSL compiler with entry points for each boundary form embedding the DSL in the host.

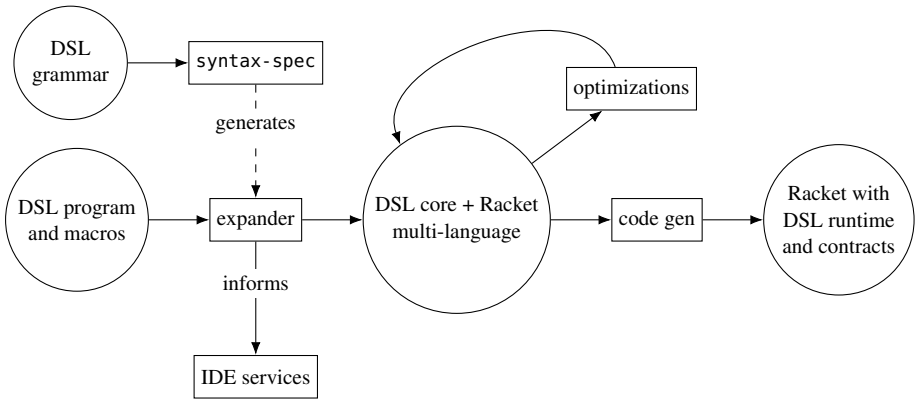

Figure 2.1: The `syntax-spec` architecture for realizing multi-language DSLs.

```
(binding-class term-variable)
(binding-class rel-name)

(nonterminal goal
  (== t1:term t2:term)
  (disj g:goal ...+)
  (conj g:goal ...+)
  (fresh1 (x:term-variable ...)
    b:goal)
  #:binding (scope (bind x) ... b)
  (r:rel-name t:term ...+)
  (GE e:racket-expr))

(nonterminal term
  x:term-variable
  (quote t:quoted)
  (cons t1:term t2:term)
  (TE e:racket-expr))

(nonterminal quoted
  n:number
  s:id
  ())
```

```
(host-interface/expression
  (run n:racket-expr (q:term-variable)
    g:goal)
  #:binding (scope (bind q) g)
  (compile-run #'n #'q #'g))

(host-interface/definition
  (defrel (r:rel-name x:term-variable ...+)
    g:goal)
  #:binding [(export r)
             (scope (bind x) ... g)]
  #:lhs [#'r]
  #:rhs [(compile-relation #'(x ...) #'g)])

(host-interface/expression
  (EG g:goal)
  (compile-expression-from-goal #'g))

(host-interface/expression
  (ET:term)
  (compile-expression-from-term #'t))
```

Figure 2.2: The declaration of the miniKanren core language in `syntax-spec`: grammar, binding structure information (underlined) and boundary forms.

2.1.1 *A DSL Specification*

Figure 2.2 shows the `syntax-spec` declaration for the miniKanren DSL. This DSL consists of goals representing logical expressions that a query should satisfy and terms that are used in those expressions. Goals include equality constraints (`==`), logical disjunction and conjunction (`disj`, `conj`), existential fresh variable binding (`fresh1`), and relation application. Terms are `cons`-trees of numbers and symbols. Thus, the declaration includes grammar `nonterminal` specifications for goal expressions, term expressions, and quoted atoms in term expressions.

The boundary with Racket is defined in two parts: `host-interface` forms embed miniKanren in Racket, and productions in the DSL grammar use `racket-expr` to embed Racket in miniKanren. The `host-interface` forms include the entry point for queries (`run`), relation definitions (`defrel`), and embeddings of goals and terms in expressions (`EG` and `ET`). The definitions of `host-interface` forms specify the position (`expression` or `definition`) in the Racket grammar where miniKanren code may be embedded, together with a grammar production for the boundary syntax. The body of each `host-interface` form includes an invocation of the DSL compiler to generate Racket code from the fragment of miniKanren code contained within the boundary.

The second part of the boundary with Racket is created by the `TE` and `GE` forms, which embed Racket expressions in miniKanren terms and goals, respectively. These multi-language boundary forms are named based on the syntax they occur in and the syntax they contain—`TE` must appear in term position and contains an

```
#lang racket
(require minikanren simple-sqlite)

(define flights (sqlite-table [flightfrom flightto]))
(for ([row (download-flights-csv)])
  (match row
    [(list from to) (insert! flights from to)]))

(defrel (direct a b)
  (GE (unify-results
        (sqlite-query flights (list (ET a) (ET b)))
        a b)))

(defrel (route origin destination path)
  (disj
    (conj (== origin destination) (== path '()))
    (fresh (hop remainder)
      (conj
        (== path (cons (list origin hop) remainder))
        (absento origin remainder)
        (direct origin hop)
        (route hop destination remainder)))))

(displayln (run 1 (q) (route "BOS" "SEA" q)))
```

Figure 2.3: A multi-language miniKanren program computing routes in an airline network database.

expression. Their grammar productions use subforms annotated with the `racket-expr` nonterminal to indicate that these positions should be processed by the standard Racket expander. The `TE` and `GE` forms are useful for multi-language programming when paired with their counterparts `ET` and `EG` to use a Racket computation to construct a term or to select a goal to execute.

Beyond the grammar and boundary forms, a `syntax-spec` declaration also specifies name binding rules. The `syntax-spec` system uses this binding structure information to control the use of DSL names in host-language code, for hygienic macro expansion, and for providing IDE services. Binding rules are indicated by the `#:binding` keyword. In `fresh1`, the `(scope (bind x) ... b)` binding rule indicates that the fresh names `x ...` are bound in a new scope and are visible in the body goal `b`. In `defrel`, the `export` portion of the binding rule indicates that the relation name is bound not in a new local scope, but in the scope surrounding the `defrel` form itself. Above the grammars are the `binding-class` definitions. These forms define the set of term variables and the set of relation names. These classes make the syntax specification precise: a reference to a name is only valid if it has the same class as the binding to which it resolves.

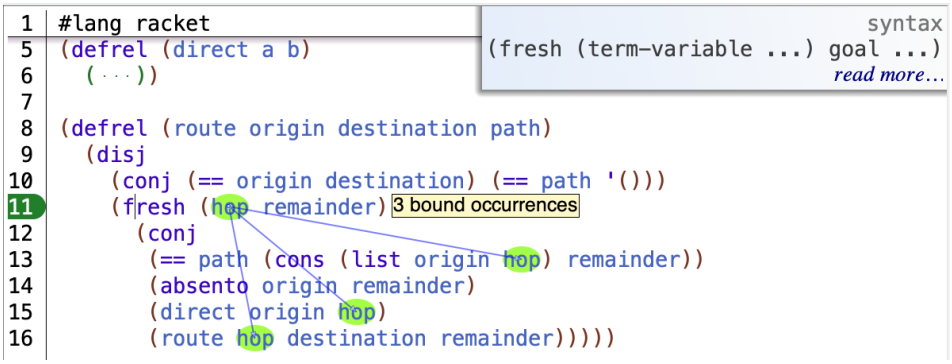

Figure 2.4: DrRacket editing the `route` relation. The edit cursor is on the `fresh` keyword, so DrRacket shows the grammar of that syntactic form in the upper-right-hand corner. It also shows the references corresponding to the binding of `hop` that is highlighted by the mouse cursor.

2.1.2 *Programming in the Resulting DSL*

Figure 2.3 shows a program written in the miniKanren-Racket multi-language implemented by the `syntax-spec` declaration of Figure 2.2. The task of the program is to query a database of airline flight paths to compute routes with layovers between an origin and a destination. The Racket portion of the program creates a SQLite database table and fills it with data loaded from a CSV data source. The miniKanren portion implements the graph search via a recursive specification of the transitive closure of direct flights. The `direct` relation should be satisfied for a pair of airports with a direct flight between them. Its implementation uses Racket code via the multi-language boundary forms to query the database. The `route` relation relates an origin airport, a destination airport, and a route between them including layovers. It is defined as a logical disjunction between a reflexive base case and a recursive case using `direct`. The `absento` constraint eliminates cyclic paths.

The flight routes program illustrates several key benefits of a multi-language-oriented macro system. First, the program uses DSL code where it is beneficial (searching for a route based on a declarative specification) and host code where needed to integrate with other components (SQLite). Second, explicit declaration of the grammar and binding structure allows the `syntax-spec` expander to catch syntax and binding errors and provide user-friendly error messages when the combination of languages is used improperly. Third, as shown in Figure 2.4, the same information allows the DrRacket IDE to understand the full program syntax. This understanding allows the IDE to provide proper syntax highlighting, binding structure information, and documentation excerpts. Finally, the checks imposed by the explicit boundaries between languages allow the miniKanren compiler to perform optimizations while ensuring that the observable behavior of the program remains unchanged.

2.2 EXTENDING A MULTI-LANGUAGE DSL

A multi-language-oriented macro system is primarily about creating new DSLs, but those DSLs are themselves objects that can benefit from extension via conventional macros. A DSL compiler benefits from a minimalist core language, while DSL users benefit from a surface syntax with additional conveniences. A hygienic macro system makes the elaboration from surface to core easy to define. Macros also allow DSL *users* to add features, and in a multi-language, macros may use host-language code in order to implement extensions that are impossible with DSL code alone. Finally, macros can add syntax that integrates separately defined multi-language DSLs.

```
(extension-class term-macro)
(extension-class goal-macro)

(nonterminal term #:allow-extension term-macro
  #| elided |#)
(nonterminal goal #:allow-extension goal-macro
  #| elided |#)
```

Figure 2.5: Additions to the `syntax-spec` declaration of miniKanren to allow macro extension.

In `syntax-spec`, the DSL creator makes the language extensible by declaring classes of extensions and indicating the grammatical positions where each class is allowed. The extended `syntax-spec` declaration for miniKanren shown in Figure 2.5 has two extension classes called `goal-macro` and `term-macro` which allow extensions to the `goal` and `term` nonterminals, respectively. The distinction between different classes of extensions enables the generated DSL-specific expander to raise precise error messages.

The DSL creator and DSL users alike define macros in much the same way as standard Racket macros. For example, the miniKanren implementor can define a convenient `fresh` form that implicitly conjoins its body elements in addition to binding logic variables with this macro:

```
(define-dsl-syntax fresh goal-macro
  (syntax-parser
    [(fresh (x:id ...+) g ...+)
     #'(fresh1 (x ...) (conj g ...))]))
```

The `define-dsl-syntax` form is the `syntax-spec` analogue to Racket's standard `define-syntax`. It registers a syntax extension belonging to a particular extension class, here `goal-macro`. The macro's transformer can be defined using any of Racket's standard macro tools. This definition uses the `syntax-parse` pattern matching and templating DSL (Culpepper, 2012).

```
#lang racket
(require minikanren minikanren/matche minikanren/facts)

(define-facts-table flights [flightfrom flightto]
  #:initial-data (download-flights-csv))

(defrel (direct a b) (query-facts flights a b))

(defrel (route origin destination path)
  (matche (origin destination path)
    [(a a '())]
    [(a b (cons (list a layover) remainder))
     (absento a remainder)
     (direct a layover)
     (route layover b remainder)]))
```

Figure 2.6: Redefinitions of the `direct` and `route` relations using the `query-facts` and `matche` extensions.

```
(define-extension query-facts goal-macro
  (syntax-parser
    [(_ table term ...)
     #'(GE (unify-results
             (sqlite-query table (list (ET term) ...))
             term ...))]))
```

Figure 2.7: Implementation of the `query-facts` extension via expansion to multi-language code.

The same mechanism allows DSL users to add non-standard features to miniKanren. Figure 2.6 shows a revised version of the flight routes program using such extensions. The cases of the `route` relation are defined using the `matche` pattern matching extension, and the `direct` relation uses the `query-facts` database query extension. The latter extension is sketched in Figure 2.7. It abstracts over the use of host-language code to access an SQLite database seen in Figure 2.3. Interaction with the host is hidden by the abstraction created by the macro, so to the miniKanren user, `query-facts` looks just like any built-in part of the DSL.

2.3 COMPILING A MULTI-LANGUAGE DSL

The `syntax-spec` system provides several services to support DSL compilers, specialized to their multi-language structure. It supports enforcing contracts at DSL-host boundaries, communicating static information between DSL fragments, and provides hygiene for DSL compilers.

2.3.1 *Inserting Boundaries*

Connecting the parts of a multi-language safely involves inserting value translations or contracts at the boundaries. Matthews and Findler (2007) describe two ways of relating the kinds of values found in the connected languages. For goals, the miniKanren DSL uses a lump embedding: the boundary forms seal miniKanren goals as opaque values that cannot be inspected by code on the Racket side of the boundary. The only valid action Racket code can perform with these values is to return them to miniKanren. For terms, miniKanren uses the natural embedding: miniKanren terms such as lists and numbers translate to the equivalent data structures in Racket, while logic variables remain opaque. Many Racket values such as vectors and structures have no translation to miniKanren term values, so passing such values from Racket to miniKanren results in a contract error.

Inserting value translations or contract checks in the compilation of explicit boundary forms such as `GE` and `EG` is a straightforward task for the DSL compiler. Cross-language name references constitute another possible channel of communication between DSL and host. The `syntax-spec` system ensures that this channel is protected by separating names belonging to different binding classes. By default, host-language code is forbidden from referring to names associated with DSL binding classes such as `term-variable` and `rel-name` declared in Figure 2.2. Thus, a Racket expression is required to use an `ET` boundary form in order to access a term variable. However, these explicit delimiters can make code overly verbose.

REFERENCE COMPILERS As an alternative, `syntax-spec` allows the DSL creator to specify a *reference compiler* procedure as part of each binding class declaration. The reference compiler is invoked for each cross-language name reference, and provides the DSL compiler the opportunity to insert value translations or contracts automatically without requiring explicit boundary syntax. For example, the full version of the miniKanren implementation specifies `compile-ET` as the reference compiler for term variables:

```
(binding-class term-variable #:reference-compiler compile-ET)
```

The `compile-ET` function is the same compiler entry point as used to implement the `ET` boundary form, and the code it generates includes the value translation from terms to Racket values.

2.3.2 *Optimizing in a Multi-Language*

The explicit boundaries between the two languages enable the optimization passes of the DSL compiler to rein in their transformations to account for the unknown behavior of the Racket code. An example of this behavior in the miniKanren compiler arises in the occurs-check elimination optimization. An occurs check is generally required to forbid cyclic terms and ensure soundness of deductions (Kowalski,

1979), but it is expensive: the cost is linear in the size of the run-time terms being unified. Unification in miniKanren always includes the occurs check, but it is unnecessary when the compiler can statically determine that the equation does not introduce a cycle.

The occurs-check elimination pass is partially limited due to the host interaction in this program:

```
(fresh (x y a b)
  (== x (cons '5 '6))
  (GE #| unknown racket code |#)
  (== a b)
  (== y x))
```

The miniKanren compiler needs to run before Racket code in boundary forms is expanded and available for analysis. Therefore, the compiler must treat the body of each such form as having potentially arbitrary behavior. The unknown Racket code could access and, via an `EG` boundary, further constrain any of the variables in scope. For example, it could assign the value `(cons '1 a)` to the variable `b`. Thus, the unification `(== a b)` still needs an occurs check to prevent a cyclic term. However, the optimizer does not lose all information below the `GE` form. Code via the host interface can only constrain logic variables by executing goals constructed via `EG` and only in ways that accord with the usual miniKanren semantics. In particular the optimizer can be sure that constraints are only extended in a monotonic way, that the assignment to the variable `x` is unchanged, and hence an occurs check for the unification `(== y x)` is unnecessary.

3 DESIGN DECISIONS

The goal of augmenting a general-purpose language with DSL syntax admits a variety of designs. This chapter explains the choices made in the design of `syntax-spec`, and how they support the desiderata introduced in Chapter 1. The miniKanren logic programming DSL of Chapter 2 illustrates why these desiderata matter. It realizes domain-specific syntax for terms and goals as well as optimizing compilation that takes advantage of domain-specific properties such as monotonicity of constraint assignments. At the same time, it allows escapes to Racket code for computations or interaction with the outside world that does not fit within the DSL. Crucially, the miniKanren implementation can be used by any Racket programmer simply by importing a library; no additional tools are required. Realizing such a combination of DSL features relies on the particular constellation of design decisions realized in `syntax-spec`.

3.1 DSL FRAGMENTS AS COMPILATION UNITS LINKED BY THE HOST

Many properties addressed by static semantics and equivalences used in optimizing compilers are nonlocal. Thus, realizing domain-specific static semantics and optimizing compilation requires allowing a DSL compiler to analyze and transform substantial chunks of syntax.

The `syntax-spec` design aims to provide the maximum flexibility to DSL compiler authors with respect to the compilation of a contiguous chunk of DSL syntax. Each fragment of DSL syntax, delimited by boundaries with the host, is provided to the DSL compiler as a complete unit. The DSL compiler is just a compile-time Racket function, so it may structure its internal compilation in arbitrary ways using any of the means of abstraction available in Racket. A common structure for the DSL compiler is a composition of functions, each of which implements a compiler pass from one intermediate representation to another.

At the same time, programmers should be able to intermix DSL code with Racket expressions and code written in other DSLs. This multi-language structure means that the program fragments processed by a DSL compiler only constitute part of a larger whole. The `syntax-spec` system provides tools to allow DSL compilers to convey compile-time information between the processing of these separate fragments. In particular, DSL compilers can use symbol tables (see Section 8.4 and Appendix A.2.2.3) to associate DSL names with compile-time data, which can then be accessed when processing references to those names, even when the bindings and references appear in different host-language modules. This information might represent a type in a static semantics or IR for a program fragment used to

implement an inlining optimization. As such, DSLs can re-use the Racket module system and its separate compilation capabilities while still allowing compilation to depend on non-local information.

The fragments of Racket code generated by the DSL compiler are linked with programmer-written Racket code and code generated by other DSLs. Such linking means that DSL compilers must be careful to generate code that maintains internal invariants of the DSL implementation and the intended equivalences of the DSL semantics. The `syntax-spec` macro system ensures that DSL compilers have the opportunity to interpose at all boundaries between the DSL and host. Reference compilers (Section 2.3.1) allow the compiler to insert an explicit boundary at the implicit boundary formed by a cross-language name reference. Racket's contract system provides the tools to enforce properties at boundaries at run time.

In sum, the DSL compiler for a `syntax-spec` language gets to transform the DSL portions of a program written in the multi-language composed of the DSL, host language, and other DSLs. Using tools such as symbol tables, the DSL compiler can communicate information between the compilation of these fragments. However, the DSL compiler does not get the opportunity to transform portions of the program that are not written in the DSL. Because `syntax-spec` does not impose a structure on the internals of DSL compilers, it does not ensure that they are internally extensible. Instead, `syntax-spec` assumes that the core language for each DSL is fixed and provides extensibility on the front-end via macros (Section 3.4).

DESIGN ALTERNATIVES Rather than limit the DSL compiler to transforming only the DSL portions of the program, I could have chosen a design in which the DSL compiler can examine all parts of a module, including portions written in the host language and in other DSLs. The MPS language workbench (Pech, Shatalin, and Voelter, 2013), for example, takes this approach. MPS collects the compiler passes associated with all the DSLs used within a module and runs them all, in an order determined by ordering constraints associated with the passes. Each pass can examine and transform the entire module. This approach allows the compilation of a DSL form to depend on static analysis of surrounding host-language code, or change the meaning of host-language code contained within a DSL fragment. However, this non-local behavior also means that when using multiple DSLs together, their compilers may interfere with each other and lead to compilation failures. Even worse, one DSL compiler may break internal invariants of the other DSL. By comparison, the `syntax-spec` multi-language approach does not admit as much context-sensitive behavior, but ensures that each DSL can defend its semantic properties.

Another alternative to the `syntax-spec` approach would be to more deeply integrate the compilation processes of the host language and DSLs. For example, an object-oriented approach as in Ziggurat (Fisher and Shivers, 2008) or an attribute grammar approach as in Silver (Van Wyk et al., 2008) can allow language extensions to integrate with static analysis and optimization passes of the host

language. Such integration comes with several costs, however. Interfaces to the internal structure of the host language compiler must be exposed to allow DSLs to integrate with its analyses and optimizations. Exposing these interfaces means that they cannot be changed without breaking backwards compatibility. Furthermore, DSLs could implement these interfaces incorrectly in a way that compromises the soundness of an analysis or optimization. By contrast, the `syntax-spec` approach does not expose internals of the host language compiler and does not create opportunities to violate its invariants. The functional programming approach of a compiler being a function from syntax to syntax is also the one most familiar to Racket programmers.

3.2 A LIGHTWEIGHT METALANGUAGE FOR SYNTAX AND BINDING

The `syntax-spec` design provides a metalanguage that addresses three critical concerns: syntax, binding rules, and the interface with the host language. These portions of a DSL implementation are the most uniform across DSLs, because their structure relates as closely to the host language as to the domain of the language being defined. This connection also means that these elements require the most intricate integration with the host language implementation. The DSL implementation must interact with the host's data representations of syntax, scope, and binding environments. In an extensible language such as Racket, the processes of expansion and name resolution interleave in complex ways. Thus, a DSL implementation must invoke operations in the correct order to ensure that data is available when needed. Abstracting over these parts of DSL implementation via `syntax-spec` means that programmers need not be aware of the effectful operational details of interaction with the host. Syntax and binding structure are also the most crucial aspects of a DSL to integrate with the IDE in order to provide services such as syntax highlighting, jump-to-definition, and rename refactorings. In a typed host language, DSL type checkers would also need to be tightly coupled to the host language implementation, making metalanguage support essential.

Programmers creating DSLs with `syntax-spec` implement the back-end compilation to Racket using conventional procedural Racket code augmented by the existing `syntax-parse` pattern matching and templating DSL (Culpepper, 2012). Other technical domains in DSL implementation, such as static analysis and program transformations, could potentially benefit from metalanguage support as well. However, the specific needs of DSL implementations in these other areas vary, and they are not as tightly coupled with the host-language implementation, so they are less essential to integrate with `syntax-spec`. It is possible to use other Racket metalanguages such as the nanopass framework (Keep and Dybvig, 2013) in back-end compilers for DSLs defined in `syntax-spec`.

For those aspects of language creation that `syntax-spec` addresses, its design emphasizes simplicity and familiarity. Language specifications in `syntax-spec` take the form of tree grammars augmented with binding specifications. Grammars

are widely familiar, even to programmers without specific expertise in programming languages. Similarly, the idea of expressing scoping structure as a tree of scopes with associated bindings and references (or more generally, a graph) is relatively approachable to novices.[1] S-expressions and simple tree-structured scope are also familiar to Racket programmers because they are used in the syntax and static semantics of Racket itself.

DESIGN ALTERNATIVES An alternative design direction would have been to aim for the expressive power needed to implement the widest-possible range of DSL syntax and static semantics. The Spoofax language workbench (Kats and Visser, 2010), for example, aims to capture as much PL theory and support the widest variety of language definitions as possible. Their Statix (Antwerpen et al., 2018) static-semantics specification language also uses scope graphs, but treats scope graph nodes as first-class elements and integrates scope checking with type checking. As a consequence, Statix can express a variety of binding structures that `syntax-spec` cannot, such as the type-dependent scoping of method names in languages with statically typed classes.

The ability to express a wide variety of DSL syntaxes and binding structures is indeed valuable; however, `syntax-spec` aims to be a relatively small step up on the learning curve from conventional macros to DSLs. Its design cannot assume that all users are familiar with the theory of programming languages or formalisms such as natural-deduction style inference rules as used in Statix. Restricting expressivity to the forms of syntax and static semantics used in Racket itself also helps programmers understand specifications in `syntax-spec` via their knowledge of the host language. A further consequence is that DSLs created with `syntax-spec` share a common structure, so users of a collection of such DSLs may develop transferrable intuitions. These benefits make restricting the metalanguage's expressive power worthwhile.

3.3 A LANGUAGE WORKBENCH AS A LIBRARY

The `syntax-spec` macro system has much in common with language workbenches such as Spoofax, but with a key difference: it is packaged as a library in a general-purpose programming language. Language-oriented programming is not yet a mainstream approach to designing programs. As such, to enable adoption, language workbench tools need to be integrated into general-purpose programming languages that programmers already use. To begin using `syntax-spec`, all that Racket programmers need to do is install an additional library with the language's package manager and import the library in the module in which they are defining their DSL. Bringing in the metalanguage has no more friction than bringing in

1 I used `syntax-spec` in an undergraduate course on DSL implementation, and students were able to quickly understand the basics of binding rules.

a conventional library. Similarly, users of a DSL defined in `syntax-spec` install and import the DSL just like a conventional runtime library. Crucially, potential `syntax-spec` or DSL users do not need to change the build process for their program or introduce a new IDE—changes which might create disruption for their entire team. Avoiding these frictions means that programmers can incrementally reap the benefits of language-oriented programming without incurring new costs.

The `syntax-spec` metalanguage is implemented as a library atop Racket's procedural macro system. This architecture decouples the development of `syntax-spec` from the development of Racket itself. This is possible because Racket's procedural macro system allows macros to be implemented using arbitrary Racket code, which can itself be generated by other macros. Thus, `syntax-spec` is implemented as a macro that generates macros to implement the specified DSL. The generated DSL implementation additionally relies on a relatively small reflective API provided by Racket's macro expander that allows the DSL to integrate with Racket's expansion process (Section 5.3). This architecture means that only additions to this underlying reflective API require changes to the core Racket implementation. Other aspects of `syntax-spec` can evolve independently.

DESIGN ALTERNATIVES I could instead have worked with Racket's development team to implement `syntax-spec` as an extension to the language's core macro expander. However, I expect this approach would be difficult in language ecosystems focused on concerns other than language-oriented programming. The `syntax-spec` metalanguage is large, and it is unlikely that the development teams for most general-purpose languages would be amenable to including such a system in their core language specification and implementation. As such, the layering of `syntax-spec` as a library on top of a smaller API provided by the host language is a design choice that I anticipate to be important if the `syntax-spec` design is ported to other host languages. With this approach, only a procedural macro system and reflective API to the macro expander need to be built into the core language. Procedural macro systems are already available in languages including Rust, Scala, and Lean. Some additions to their reflective APIs would be needed, but adding such features would be a much smaller change than adding an entire metalanguage like `syntax-spec` to their core.

3.4 DSL EXTENSIBILITY VIA MACROS AND HOST INTEROPERATION

Macros atop a DSL core offer a means of extensibility that does not compromise the invariants of the optimizing compiler for the DSL core language. Macros are only capable of generating code in the same core language that programmers could write directly themselves. They have no access to the internals of the DSL implementation and thus cannot break its assumptions. However, when the DSL core language exists as part of a multi-language with the host, macros can be used to add DSL features that are not expressible by expansion to the DSL core alone. This

possibility is exemplified by the `query-facts` extension to miniKanren presented in Figure 2.7. The expressive power and safety of this approach to extensibility relies on the design of the multi-language. Interaction with the host language must be flexible enough to provide the expressive power needed to create interesting extensions, but sufficiently restricted to avoid inhibiting the optimizations that the DSL compiler should perform.

Macros atop a multi-language can also link together multiple DSLs. In the context of the `query-facts` extension to miniKanren, we can regard miniKanren and SQL as separate DSLs, each extending Racket via multi-language macros. Thanks to the multi-language structure, it is possible to use a combination of miniKanren, SQL, and Racket to perform a task. With a macro that expands to this combination, it is also possible to hide the Racket glue code between the two DSLs, making it appear as though they are directly integrated.

DESIGN ALTERNATIVES An alternative to the combination of macros and multi-language host interaction would be to implement DSLs with open compilers. Such an open DSL compiler could be extended with additional DSL syntactic forms or additional compiler passes. Open compilers for multiple DSLs could be simultaneously extended to more deeply link the DSLs. For example, a static analysis could integrate a domain-specific understanding of fragments in each DSL to more precisely predict the behavior of the complete program. At the same time, an incorrect extension to the open compilers could easily introduce bugs. This is the same design trade-off as in Section 3.1, but with respect to extensions to DSL compilers rather than regarding DSL compilers as extensions to the host. An open compiler admits more extensibility, whereas macro extensions atop a multi-language core precludes the possibility of DSLs breaking internal invariants of the compiler.

3.5 INTEGRATING CONVENTIONAL AND MULTI-LANGUAGE MACROS

Conventional and multi-language macros have complementary advantages, and the `syntax-spec` design allows them to work together to attain the benefits of both. Programmers need to learn the skill of writing compilers in the small before they can apply the skill in the large to create sophisticated DSLs. Simple macros that implement syntactic sugar are the ideal introduction, and conventional macro technology is ideal for these simple rewritings. Such macros do not need the additional tools provided by `syntax-spec`, so using `syntax-spec` for these macros incurs the additional cost of writing binding specifications without any additional benefit. Conventional hygienic macros do not require binding specifications and are thus the easiest entry-point for new programmers.

The multi-language macros provided by `syntax-spec` become valuable when a programmer is ready to implement a static check or optimizing compiler. This is the right point to transition from conventional macros to a multi-language structure

where the language grammar, boundaries, and binding rules are explicit. This transition can occur both in individual DSL projects and in the overall arc of a programmer's experience with language-oriented programming. The back-end compiler for the `syntax-spec` version of a DSL can re-use portions of the initial, conventional-macro implementation because both systems use the `syntax-parse` pattern matcher and templates for code generation.

The miniKanren DSL exemplifies the transition from conventional to multi-language macros. Most implementations of miniKanren rely on a layer of conventional macros that provide syntactic sugar over an embedding called microKanren. This implementation strategy does not realize optimizing compilation, and there are many errors possible due to language mixing that such implementations fail to not catch. However, the implementation is also simple which is a benefit as miniKanren DSL is often used as a first introduction to DSLs. For a programmer who knows Scheme or Racket, explaining miniKanren bottom-up starting from the embedding and only later adding on a syntactic sugar results in a smooth introduction. The compiler portion is also easy to introduce to novices because it consists only of basic `syntax-rules` macros that look much like the pseudocode for an expansion that one might write down in a paper. As miniKanren developed further, it became advantageous to add optimizing compilation. Multi-language macros from `syntax-spec` enabled this stage of development, supporting both an optimizing compiler (Chapter 10) and a multi-stage extension (Ballantyne et al., 2025).

DESIGN ALTERNATIVES As an alternative, a macro system following the rest of the `syntax-spec` design could omit conventional macros. All macros would require full grammar and binding-rule specifications. Such a design would not need the mechanisms of conventional macro hygiene, which could significantly simplify the implementation. Having binding rules and grammars for all extensions would also make it easier to implement reliable IDE services. However, it would come with two costs. First, programmers would need to (learn to) write binding rules even for simple syntactic-sugar macros. Second, the expressive power of syntax specifications in `syntax-spec` grammar specifications is limited as compared to the patterns of `syntax-parse`. This limitation is due to the fact that `syntax-spec` expansion must reconstruct the specified syntax as well as pattern match it. In the context of Racket where the ecosystem already has many extensions using conventional macros, integrating with them is the natural choice. The alternative design seems well worth exploring in another host language, however.

4

PRIOR WORK

Multi-language macros are one point in a large space of approaches to DSL implementation. The `syntax-spec` design synthesizes ideas from macro systems, language workbenches, binding specification languages, and multi-language semantics. The first four sections of this chapter review prior work in these areas. A major alternative to language extension is embedding, where the DSL is implemented as a library in the host language. DSL programs expressed via an embedding can also be extracted via reflection to allow for optimizing compilation. The fifth section contrasts multi-language macros with embedding and extraction.

4.1 MACROS

Multi-language macros build on several threads of work that push macros beyond simple rewritings and towards more sophisticated language extensions.

MACROS AS LANGUAGE BOUNDARIES Because procedural macros admit arbitrary transformations, they have long been used to implement substantial languages as extensions. For example, Felleisen (1985) presents a Prolog-like language hosted in Scheme.

Some macro systems are specifically designed for this kind of macro; that is, designed with the expectation that a macro is a boundary between the host and an entire DSL. Typed Literal Macros (Omar and Aldrich, 2018), Converge's DSL blocks (Tratt, 2005), and Template Haskell's quasiquoters (Mainland, 2007) are all forms of macros which process a delimited region of syntax that may be any text. The DSL within the region may thus have a syntax that radically differs from that of the host. Typed Literal Macros extend the basic idea with a mechanism called segmentation by which a macro informs the host that certain spans within the DSL syntax are host-language subexpressions that should be parsed by the host. Segmentation also ensures that the host's standard IDE services can be provided for these subexpressions.

However, in contrast to `syntax-spec` none of these systems inform the host about the structure of the DSL fragment beyond the position of host-language subexpressions. The host is not informed about the DSL's grammar or binding structure, so the host cannot automatically interpose on host-language references to DSL binders or provide IDE services.

MACROS THAT WORK TOGETHER Procedural macros also admit the possibility of using side effects for communication between macros. DSL implemen-

tations in LISPs sometimes use such techniques without an explicit support from the macro system. For example, Wand (1984) presents a DSL for executable denotational semantics, whose implementation is mentioned in Kohlbecker and Wand (1987). The DSL is implemented using a collection of procedural macros that communicate to typecheck the semantics definition. Without specialized support, however, compile-time side effects are difficult to program with. Such effects may execute in a variety of contexts: when a program is first macro-expanded, when it is loaded for compilation of another file, or when it is evaluated in a top level REPL. Some Scheme systems include an `eval-when` construct[1] to help manage these executions, but it must be used carefully to ensure information is communicated at the correct times.

Racket provides explicit support for such "macros that work together" via several mechanisms. First, its module system's phases and visits mechanisms manage the execution of module-level side effects such that they can be reliably used to communicate even in the presence of separate compilation (Flatt, 2002). Second, Racket's expander allows macros to record and retrieve additional information associated with name bindings and syntactic context (Barzilay, Culpepper, and Flatt, 2011; Flatt et al., 2012). Finally, reflective facilities expose some of the internal structures of the macro expander in order to allow macros to orchestrate custom expansion processes. Macros can invoke the expander on subexpressions and manipulate representations of scopes and bindings (Flatt, 2016a).

Macros that work together have been used to create macro-extensible DSLs, such as Racket's extensible pattern matcher (Tobin-Hochstadt, 2011), `require`, `provide`, and `syntax-parse` sub-languages. Realizing extensibility with this technique requires manually implementing a macro expander for the DSL. Prior to the API improvements I discuss in Section 7.2 such implementations were ad-hoc and could not realize macro hygiene for DSLs with internal binding structure. One exception is the "type expanders" library (Soy, 2017) for Typed Racket, which reimplements rather than reuses Racket's hygiene algorithm for its DSL expander. The McMicMac system of Krishnamurthi (2001) is a predecessor of Racket's current approach. In McMicMac, "micros" implement functionality similar to DSL expanders, but the system does not address hygiene. By comparison with all these approaches to manually implemented DSL macro expanders, `syntax-spec` provides hygienic macro-extensibility automatically for all DSLs defined in the metalanguage.

Macros that work together have also been used to implement typechecking as part of macro expansion in Chang et al. (2019) and Chang, Knauth, and Greenman (2017). The Turnstile metalanguage acts as a declarative layer on top of this implementation. This approach was even used to create a Haskell-like language that compiles to Racket (King, 2017). My work on macro expansion for DSLs was initially motivated by difficulties in the implementation of Turnstile. The new API I introduce in Section 5.3 provides a foundation that could be used in a robust

1 https://scheme.com/csug8/system.html#./system:s76

re-implementation of Turnstile. However, most DSLs in Racket do not need a type system and the `syntax-spec` DSL is much more concise for declaring untyped DSLs.

A major difficulty with the "macros that work together" approach is the degree of expertise required to use the reflective API. Using the API successfully requires DSL creators to understand the operational sequence of macro expansion steps and Racket-specific implementation concepts such as scope sets, phases, and visits in great detail. In contrast, `syntax-spec` entirely hides that API and the operational details of expansion below a declarative metalanguage. The concepts of the metalanguage are mostly familiar to working DSL implementors.

MACROS WITH DEEPER HOST-LANGUAGE INTEGRATION One of the key ideas in `syntax-spec` is that the host language should be informed of the internal structure of syntax of DSL fragments so that it can provide IDE services and support multi-language interactions. A variety of macro system designs have explored ways for macros to integrate more deeply with the syntax, static semantics, and compilation of their host languages. Fortress (Allen et al., 2009) and Lean 4 (Moura and Ullrich, 2021) allow macros to extend the parsing process of the host language. Scala macros (Burmako, 2013) can reflect on type information of subexpressions in their input. The Klister language (Barrett, Christiansen, and Gélineau, 2020) integrates macro expansion with the process of Hindley-Milner type inference, suspending type-dependent macros as needed until additional type information is available. The Ziggurat macro system (Fisher and Shivers, 2008) demonstrates how an object-oriented approach to macros can allow macros to extend the static analyses of the core language in order to yield more precise results thanks to additional structure from domain-specific languages. While currently `syntax-spec` integrates extensions with the host only in terms of information about binding structure, these other lines of work suggest avenues for future integrations as discussed in Section 16.6.

4.2 LANGUAGE WORKBENCHES

The `syntax-spec` metalanguage adapts ideas from the language workbench tradition into a new context where I anticipate an opportunity for broad adoption.

The `syntax-spec` system is particularly influenced by the Spoofax language workbench (Kats and Visser, 2010) and the "scope graphs" model (Neron et al., 2015) underlying its Statix static semantics specification language (Antwerpen et al., 2018). Spoofax aims to be general, with metalanguages expressive enough to implement the full gamut of designs compatible with well-established programming language theory. This principled, research-focused approach pushes the expressivity of language workbench metalanguages. That expressivity comes with a cost, however: programmers need to learn an expansive metalanguage and have some familiarity with the underlying theory. To reduce this cost and flatten the

learning curve I focus instead on the restricted scenario of building DSLs that fit together on top of a single host language.

The SugarJ macro system (Erdweg et al., 2011b) adapts the technologies from Spoofax to work in the context of a macro system. A SugarJ definition includes a grammar specification and a DSL compiler, much like a `syntax-spec` one. SugarJ's "editor libraries" (Erdweg et al., 2011a) allow DSL creators to extend IDE services to their DSL. However, the system does not account for binding structure, provide macro extensibility, or make multi-language structure explicit. Finally, its implementation relies on an external pre-processor, which means that potential users must accept additional dependencies and modify their build process.

Language workbenches such as Spoofax and SugarJ generate more from a DSL definition than just a compiler. They also automatically generate IDE services. These services range from basic syntax coloring and bracket matching to rich semantic services. For example, Spoofax generates code completion that takes into account the syntax, static semantics, and name binding of the language (Pelsmaeker et al., 2022). With additional effort DSL creators can also implement custom transformations such as refactorings. The IDE services provided by the current `syntax-spec` implementation are limited to those based on local binding structure, such as rename refactorings. The more sophisticated services provided by language workbenches suggest directions for future work (Section 16.6).

4.3 BINDING SPECIFICATION LANGUAGES

The research literature comes with a variety of binding-rule languages; `syntax-spec` is most inspired by Visser's Spoofax language workbench and scope graphs approach (Antwerpen et al., 2016; Antwerpen et al., 2018; Neron et al., 2015). Like `syntax-spec`, these languages augment syntax definitions with binding information. They supply binding-related services variously supporting IDEs, compilers, and proofs. My work applies this well-known idea of deriving binding-related services from declarative binding rules to the new purpose of supporting sophisticated DSL implementations in a macro-extensible language. This context poses new problems, particularly regarding integration with the host language and macro hygiene. These considerations induce constraints that dictate much of my binding language design (see Chapter 9 for details). Other examples of such binding languages include embedded DSL compilers (Weirich, Yorgey, and Sheard, 2011), proof assistants (Griffin, 1988; Keuchel, Weirich, and Schrijvers, 2016; Sewell et al., 2007), and other language workbenches (Clément, Incerpi, and Kahn, 1989; Konat et al., 2013).

Herman and Wand (2008), followed by Stansifer and Wand (2014) and Pombrio, Krishnamurthi, and Wand (2017), introduce the idea of using type-like binding specifications for declarative macros. They simultaneously aim for two objectives: (1) treating binding specifications as types for macros and (2) replacing the inference-based implementation of macro hygiene with an approach amenable

to formalization. Unfortunately, typechecking macro transformers to ensure that expansion preserves binding structure imposes serious restrictions on their expressivity. Rather than reject inference-based hygiene, I use binding rules to implement hygiene inference for macros that extend the DSLs defined in `syntax-spec`. Most importantly, the binding language of `syntax-spec` is *not* a restrictive type system, meaning it accepts all macro transformers.

4.4 MULTI-LANGUAGE SEMANTICS AND LANGUAGE BOUNDARIES

The structure of `syntax-spec` declarations takes inspiration from the structure of multi-language semantics (Matthews and Findler, 2007). The grammars of the component parts of a multi-language are joined by boundary forms. In a multi-language operational semantics, reduction rules for the component languages are defined with different evaluation contexts to keep them from applying to the wrong language. Then, additional reduction rules for boundary forms define the ways that the component languages can interact. As values cross the boundary, they may be transformed into values belonging to the other language. Matthews and Findler propose several value translation strategies In the lump embedding, values are opaque in the opposing language. In the natural embedding, values of one language translate to corresponding values on the other side. Such a transformation may also wrap and monitor the values to ensure that they behave according to an expected type in the destination language. My implementations of DSLs in `syntax-spec` use such embedding strategies. For example, miniKanren employs lump embedding for goals and natural embedding for term values.

Higher-order contracts (Findler and Felleisen, 2002) correspond to the checks at boundaries in a multi-language that are necessary to ensure evaluation does not reach stuck states. Whereas first-order properties of values crossing the boundary can be checked immediately, enforcing higher-order properties requires wrapping values with proxies that check subsequent uses of the values. Beyond simply raising an error, contracts also track blame in order to correctly indicate which party violated a contract. Blame assignment accomplishes this task even when the original contract boundary and the eventual failing check are separated due to higher-order control flow. Trace contracts (Moy and Felleisen, 2023) are a recent addition which allow contracts to monitor temporal properties as well. Racket includes a sophisticated higher-order contract system which can be used in the compilation of `syntax-spec` DSLs.

DSLs implemented with `syntax-spec` insert contracts as part of the compilation of boundary forms to protect internal invariants of the DSL implementation. Generated code within a DSL fragment may use low-level representations that do not locally enforce invariants. To make this approach safe, it is essential that the contract system ensure that all parts of the contracts between DSL and host are actually monitored. Proper enforcement is challenging in the context of higher-order data flows and mutable data. For example, consider a DSL data structure

that uses mutable cells, where the value within satisfies some invariant. It is not enough to check the invariant when the cell crosses between DSL and host. The contract system must also interpose on subsequent mutations to ensure that host code cannot later assign an improper value that breaks the invariant.

The formal property of *complete monitoring* (Dimoulas, Tobin-Hochstadt, and Felleisen, 2012) from the higher-order contracts literature and the related property of *vigilance* (Gierczak et al., 2024) from the world of gradual typing address this problem. Complete monitoring requires that contracts be capable of interposing on all (dynamic) channels of interaction between DSL code and host-language code. Vigilance further requires that the dynamic checks together ensure that values reaching positions annotated by types (or equivalently contracts) in fact have the behavior denoted by those annotations. In conjunction with these dynamic properties of the host-language contract system, it is critical that `syntax-spec` allow DSL compilers to insert contracts at all syntactic (static) boundaries between DSL and host. In particular, the reference compilers feature (Section 2.3.1) allows the DSL compiler to insert a contract at the implicit boundary created by a cross-language variable reference.

4.5 EMBEDDING AND EXTRACTION

The task of extending a host language with a DSL suggests re-using some facets of the host in the DSL that is being built, and there is a widely explored space of trade-offs (Mernik, Heering, and Sloane, 2005) surrounding such linguistic re-use. This design space is anchored at two ends by the *deep* and *shallow* embeddings. Both re-use host-language syntax, but they differ to what degree they re-use host language semantics. Shallow embeddings make it trivial for the implementer, or an ordinary DSL user, to extend the DSL and to integrate host-language code, because DSL code is itself simply host-language code. In shallow embeddings of miniKanren (Ballantyne, 2024; Friedman et al., 2018; Kosarev and Boulytchev, 2018), for example, each syntactic form is realized in the host language as either a call to a function or via an individual host-language macro that expands to runtime functions and to host-language binding forms such as `lambda`. However, there is no DSL compiler that gets an overall view of the program, so optimizing compilation is impossible.

By contrast, deep embeddings can offer whole-program compilation. A DSL compiler executing at the host language's runtime takes the abstract syntax as input and produces host-language code as output. In a deep embedding of miniKanren for example, host-language code constructs a datatype representing miniKanren abstract syntax (Lozov and Boulytchev, 2021; Verbitskaia, Berezun, and Boulytchev, 2020; Verbitskaia, Engel, and Berezun, 2023).

Some approaches to embedding attempt to reconcile extensibility and optimizing DSL compilers. One fruitful approach is to layer a shallow embedding on top of a deep embedding (Elliott, Finne, and Moor, 2003; Gibbons and Wu, 2014;

Svenningsson and Axelsson, 2015). The shallow embedding provides integration with the host language syntax and static semantics. It also achieves extensibility. Extensions are host language function definitions, whose applications run in the host language. The semantic domain of the shallow embedding is the datatype of the deep embedding. A DSL optimizing compiler processes the data representation of the deep embedding. DSL implementations in `syntax-spec` have parallels to this layering. DSL extensions via macros are analogous to functions defining extensions in a shallow embedding, but the process that generates DSL core language code is macro expansion rather than host-language evaluation.

Another approach relies on reflection to extract code from a shallow embedding for compilation (Atkey, Lindley, and Yallop, 2009; Najd et al., 2016; Rompf et al., 2012; Scherr and Chiba, 2014; Shaikhha, Jovanovic, and Koch, 2018; Šinkarovs and Cockx, 2021). DSL programs are written as host-language code calling functions that represent DSL forms, just as in a shallow embedding. These functions may be defined as normally in a shallow embedding, or they may instead be mere stubs. This design works because the host-language code is not actually executed in the normal way. Instead, a quotation or compile-time reflection mechanism is used to extract a representation of the host-language code. A DSL compiler provides an alternate interpretation of the code, recognizing applications of the stub functions as DSL constructs to compile. This DSL compiler thus has control over the complete DSL program, so it may perform analyses and optimizations.

This reflection-based strategy implicitly supports a multi-language architecture. Extensibility may be achieved by employing a normalizer for the host language, either built-in to the reflection mechanism or separately defined. The normalizer is configured to avoid reducing calls to the stub functions representing the DSL core language, but it reduces other calls. Just as function calls representing uses of extensions evaluate in the host language in a shallow embedding, they reduce during normalization and leave behind only calls that belong to the DSL core language.

Both of these approaches rely on encoding the DSL in the host language syntax and static semantics. Indeed, re-use of these components is a key, separate motive for embedding-based implementation techniques, beyond extensibility. This is particularly true when the motive is to reuse the host language type system, as in the case of re-using Haskell's type system to create a typed logic programming language (Claessen and Ljunglöf, 2001; Hinze, 1998), or embedding a DSL in a theorem prover in order to support verification (Šinkarovs and Cockx, 2021). However, reuse of the host syntax and static semantics can come at a serious cost when the DSL has significant differences with the host. If the DSL's type system cannot be expressed in the host type system, there is no benefit to re-use. If a DSL's re-interpretation of the host language only supports a small subset of its syntax, DSL users may have trouble understanding what constructs they can use in DSL code. Finally, DSL compiler authors must contend with the complexity of the

host language and its representation in the reflection system, including portions irrelevant to their DSL.

The `syntax-spec` approach to DSLs and extension has particular advantage over embedding in cases where the DSL and host differ in important ways. DSLs may come with their own new syntax, including binding structure, and new static semantics. Macros, too, define new syntax and binding structure rather than reuse that of the host. This approach avoids any impedance when either using or implementing the DSL.

Part II

IMPLEMENTATION

IMPLEMENTATION

This part discusses the implementation of the `syntax-spec` system in Racket, including hygienic macro expansion, the binding specification language, and integration with Racket's conventional macro system.

In Racket, the macro expander realizes the entire front-end of the language implementation. Beyond expanding macros, it also checks syntax to raise syntax errors, ensures that names are bound, and captures information for IDEs. This process requires knowledge of the grammar and binding structure of the core language that is the target of expansion. From this perspective, the job of `syntax-spec` is essentially to extend Racket's expansion process to handle DSL core languages in addition to Racket's own core language. Given a language declaration, the `syntax-spec` macro system extends Racket's expansion process to handle the DSL core syntax as well as Racket code.

As background for the remainder of the part, Chapter 5 reviews Racket's approach to hygienic expansion, known as "sets of scopes." Using these preliminaries, Chapter 6 presents a model of hygienic expansion for DSL syntaxes declared in `syntax-spec`. The model shows how `syntax-spec` interprets binding specifications to drive the expansion of DSL syntax.

While the model takes the form of an interpreter, the actual implementation uses a mix of strategies to allow `syntax-spec` to be built as a layer on top of Racket's existing conventional macro system. Chapter 7 explains how `syntax-spec` is implemented via macros that consume a DSL specification and generate other macros and compile-time code that implement the DSL. This compile-time code is effectively a DSL-specific macro expander, specialized to the DSL's core language. To implement the multi-language that combines the host and DSL, the Racket expander and this specialized DSL expander must be integrated. Chapter 8 presents this integration, including how the expanders invoke each other and share the static information they glean about the syntax they expand.

Finally, Chapter 9 reflects on the constraints that the goal of implementing hygienic macro expansion imposes on the design of `syntax-spec`'s language of binding specifications. Some desirable binding structures cannot be realized with Racket's "sets of scopes" hygienic expansion process, and furthermore some ways of specifying binding structure do not provide enough information to derive this process.

5

A REVIEW OF HYGIENIC EXPANSION WITH BINDING AS SETS OF SCOPES

Before we dive in to the implementation of `syntax-spec`, this chapter reviews the problem of macro hygiene and the solution from Flatt, 2016a used in Racket. The `syntax-spec` macro system extends this solution to apply to expansion targeting DSL core languages as declared in the metalanguage. This summary does not aim to be exhaustive, and a complete understanding of the model of Chapter 6 is likely to require additional familiarity with the details of this prior work.

5.1 BINDING AS SETS OF SCOPES BY EXAMPLE

The top of Figure 5.1 defines a Racket macro for list comprehensions that resembles an ordinary for-loop. The left column below shows a use of this macro in context. The right column below displays the naive interpretation of the macro use; that is, a copy of the concrete template syntax from the macro definition replaces the macro use and the pattern variables in the template are replaced with the concrete syntax elements of the macro use. It is naive because neither of these substitutions is sensitive to the lexical scope of Racket or the bindings of any names.

```
(define-syntax-rule
  (for/list ([element init]) body)
  ;; rewrites to
  (let loop ([lst init] [acc '()])
    (if (null? lst)
        (reverse acc)
        (let ([element (first lst)])
          (loop (rest lst)
                (cons body acc))))))
```

```
;; a use of `for/list`
(let ([reverse ...]
      [acc 5])
  (for/list ([cons l])
    (+ cons acc)))
```

```
;; ... expands naively to
(let ([reverse ...]
      [acc 5])
  (let loop ([lst init] [acc '()])
    (if (null? lst)
        (reverse acc)
        (let ([cons (first lst)])
          (loop (rest lst)
                (cons (+ cons acc) acc))))))
```

Figure 5.1: A macro expansion illustrating the three kinds of hygiene violations in naive expansion

As a result, the binding structure of the naive expansion differs from the intended one, and the program thus exhibits the wrong behavior. In this case, the references to `reverse` and `cons` from the macro template are captured by unrelated bindings in the context of the macro use. Similarly, the meaning of the reference `acc` in the macro use is changed in an unexpected manner.

In other words, naive interpretation of macros causes three kinds of problems:

1. A use-site *context* may capture free references in macro templates. Syntax is moved from the template into the context of the use site. Thus, the `reverse` reference from the template moves into the scope of the let-binding of `reverse` that surrounds the use of `for/list`.

2. Use-site bindings may capture free references in macro templates. The syntax of the macro use itself is mutually-substituted into the template. Binders originating in the use such as `cons` are placed into binding positions in the template via pattern variables. Here, the binding of `cons` captures the template reference to the standard prelude function.

3. Template-introduced bindings may capture use-site references. When expansion replaces pattern variables in the template with variable references, binders in the template may capture these references. Here, `acc` is captured by the loop accumulator in the template.

The decades-old solution to this problem is to make macro expansion hygienic. Racket uses Flatt's approach to macro hygiene, which attaches scope tokens to syntax in order to impose constraints on name resolutions in expanded code. Concretely, Flatt's algorithm uses three kinds of scope tokens that prevent the three categories of capture mentioned above:

LEXICAL SCOPE TOKENS represent what we normally think of as Algol-like scopes. They prevent use-site context captures of free references in macro templates.

USE-SITE SCOPE TOKENS restrict bindings in the macro use to capture only references also in the use site and thus prevent use-site captures of free references in macro templates.

MACRO-INTRODUCTION SCOPE TOKENS restrict bindings in the syntax introduced by the macro to capture only references that are introduced by the same macro expansion step, thus preventing template-introduced captures of use-site references.

Each scope token represents a constraint that the scoped identifier may serve as a binder only for other identifiers marked with the same scope token. Thus, a set of scope tokens constrains a binder to capture only references that have a superset of those scope tokens. Resolution for a reference works by finding binders with the

appropriate subset. Normal lexical shadowing is accommodated by looking for the binder whose set of tokens is a superset of all other candidate binders.

The expander attaches these scope tokens to each region of syntax before expansion moves and combines syntax. The following is the scope-sensitive expansion of the above macro with each identifier annotated with the expansion-introduced scope tokens:

```
(let ([reverse{a_lex} ...]
      [acc{a_lex} 5])
  (let loop ([lst init] [acc{b_mac, c_lex} '()])
    (if (null? lst)
      (reverse{b_mac, d_lex} acc{b_mac, d_lex})
      (let ([cons{a_lex, c_use, d_lex, e_lex} (first lst)])
        (loop (rest lst)
              (cons{b_mac, d_lex, e_lex} (+ cons{a_lex, d_use, d_lex, e_lex}
                                            acc{a_lex, d_use, d_lex, e_lex})
                                         acc{b_mac, d_lex, e_lex})))))))
```

The subscripts identify the kind of scope token, and the underlined scopes are those specifically that prevent the three forms of undesired capture in this example.

The remaining question is how precisely the expander decides to attach these scope tokens to identifiers. Flatt, 2016a explains this process in detail for the core language of Racket. Chapter 6 shows how I extend his idea to macro expansion for any DSL defined in `syntax-spec`.

5.2 A RECAP OF THE CORE MODEL OF BINDING AS SETS OF SCOPES

My model in Chapter 6 builds on the data definitions from the formalization of Flatt, 2016a. This section explains these definitions, which are shown in Figure 5.2.[1] Syntax is represented by *syntax objects*, which represent a tree structure augmented with a set of scopes associated with each node. An *identifier* is a syntax object containing a name as its datum. The binding store is a global memory that maps names with scopes to unique names representing binding identities. The expander environment maps unique binding identities to values representing the meaning of syntax. Flatt's environment values represent Racket syntax meanings. My model needs different kinds of environment values in order to generalize to the multiple grammatical non-terminals, binding classes, and extension classes of hosted DSL syntaxes. One kind of environment value in both models is the definition of a macro. Part of such a value is a representation of the macro transformer. A transformer value pairs a function from syntax to syntax with a scope identity used for managing hygienic expansion. Finally, the expander state consists of the store, environment, and information for tracking details of hygiene to be discussed later.

Using the evaluation syntax, Flatt's key operation resolves a reference to a unique binding:

1 I made a slight adjustment to ease the transition to my model.

$$
\begin{aligned}
\mathit{stx} &= \mathbf{Syntax}(\mathit{atom}, \mathit{scps}) \mid \mathbf{Syntax}(\mathbf{List}(\mathit{stx}, \ldots), \mathit{scps}) \\
\mathit{id} &= \mathbf{Syntax}(\mathit{name}, \mathit{scps}) \\
\mathit{scps} &= \{\mathit{scp}, \ldots\} \\
\mathit{scp} &= \text{a token that represents a scope} \\
\\
\mathit{store} &= \text{finite mapping from (name, } \mathit{scps}\text{) to name} \\
\mathit{exp\text{-}env} &= \text{finite mapping from name to } \mathit{env\text{-}val} \\
\mathit{env\text{-}val} &= \ldots \\
\mathit{transformer} &= \mathbf{T}(\mathit{fn}, \mathit{scope\text{-}id}) \\
\mathit{exp\text{-}st} &= \mathbf{St}(\mathit{store}, \mathit{exp\text{-}env}, \mathit{scps}, \mathit{scp})
\end{aligned}
$$

Figure 5.2: Evaluation syntax from Flatt

$\texttt{resolve} : \mathit{id}, \mathit{store} \rightarrow \mathit{name}$ Find candidate binders whose scope sets are a subset of that of the reference, and return the one whose set is the superset of all other candidates.

Otherwise, Flatt's model intermixes operations on the expander state with the handling of Racket's particular syntactic forms.

5.3 AN API FOR HYGIENIC EXPANSION

Ballantyne, King, and Felleisen (2020) introduce an API layer that hides the details of this manipulation of the expander state. The internal details of these operations are not important to understanding the implementation of `syntax-spec`, but their interfaces are:

$\texttt{enter-scope} : \mathit{exp\text{-}st} \rightarrow \langle \mathit{scp}, \mathit{exp\text{-}st} \rangle$ Adjusts the expander state to expand within a new scope and produces a token representing that scope plus the new expander state.

$\texttt{add-scope} : \mathit{stx}, \mathit{scp} \rightarrow \mathit{stx}$ Adds the scope token from `enter-scope` to all the scope sets of a syntax object.

$\texttt{bind} : \mathit{exp\text{-}st}, \mathit{id}, \mathit{env\text{-}val} \rightarrow \langle \mathit{id}, \mathit{exp\text{-}st} \rangle$ Creates a new binding in the store and a corresponding entry in the expander environment, associating the identifier with the environment value. It returns an identifier that may have additional scopes and a correspondingly updated state.

$\texttt{lookup} : \mathit{exp\text{-}st}, \mathit{id} \rightarrow \mathit{env\text{-}val} \cup \texttt{False}$ Uses the binding store and expander environment to resolve an identifier and, if it exists, to find the corresponding environment value.

eval-transformer : $exp\text{-}st, stx \rightarrow transformer$ Evaluates a given syntax object as Racket code representing a compile-time tranformer procedure. Constructs a transformer object encapsulating the procedure plus information for maintaining hygiene.

apply-transformer : $exp\text{-}st, transformer, stx \rightarrow \langle stx, exp\text{-}st \rangle$ Applies an encapsulated transformer object to a syntax object, yielding hygienically expanded syntax and a modified state.

6 HYGIENIC EXPANSION FOR `syntax-spec` DSLS

Equipped with the preliminaries from Chapter 5, let us turn to the formal model of hygienic expansion for DSLs declared in `syntax-spec`. In order to focus specifically on hygienic expansion, the model does not address the way the host and DSL connect to expand multi-language code. See Chapters 7 and 8 for this connection.

The grammar in Figure 6.1 presents an abstract syntax for the core language of DSL specifications in `syntax-spec`. Latter sections of this chapter extend the grammar to address more sophisticated binding specifications. Conceptually, a `syntax-spec` language specification is a finite map from nonterminal names *nt-name* to nonterminal definitions *nt*. A nonterminal definition combines an optional extension class *eclass* with a list of tree-grammar productions *prod*. If an extension class name is provided, then `syntax-spec` will apply transformers of that category. A production consists of the ordinary grammar production plus a binding specification.

spec	=	finite mapping from *ntname* to *nt*
nt	=	**NT**(ϵ-*eclass*, (*prod*, ...))
ϵ-*eclass*	=	ϵ | *eclass*
prod	=	**Prod**((*formname*, *pvar*, ...), *bspec*)
bspec	=	**Scope**(*bspec*)
	|	**Bind**(*pval*, *bclass*)
	|	**Ref**(*pval*, *bclass*)
	|	**Subexp**(*pval*, *ntname*)
	|	**Group**(*bspec*, ...)
	|	**BindSyntax**(*pval*, *eclass*, *pval*)
pval	=	*pvar* (to be extended below)
ntname, *formname*, *pvar*, *bclass*, *eclass*	=	name

Figure 6.1: Model `syntax-spec` definition syntax

The core model includes six forms of binding specifications. Elaboration from the concrete syntax introduced in Section 2.1.1 and documented in Appendix A.2.1.5 to the abstract syntax is straightforward. For example, while

```
(let v:var e:expr b:expr) #:binding [e (scope (bind v) b)]
```

is the concrete version of the syntax and binding specification for the simple `let`-binding form from Racket, its abstract counterpart is

Prod((`let`, `v`, `e`, `b`),
  Group(**Subexp**(`e`, `expr`), **Scope**(**Group**(**Bind**(`v`, `var`), **Subexp**(`b`, `expr`)))))

As the example shows, the concrete syntax for bindings, references, and subexpressions places the binding class and nonterminal names as qualifications on the pattern variables in the grammar. By contrast, the abstract syntax places this information in the *bspec* forms. Finally, the additional **BindSyntax** form allows for macro binding positions within DSL syntax specifications.

Specifications are subject to some simple static semantic constraints. Pattern variables in binding specifications must be used linearly. The nonterminal name *ntname* referenced in a **Subexp** must match some entry in the *spec*. Finally, **Bind** may appear only within a **Scope**.

EVALUATION SYNTAX The model needs additional syntax to represent intermediate data and the outcome of interpretation. The first pieces are new constructors for environment values:

env-val = **Var**(*bclass*) | **Ext**(*eclass*, *transformer*)

They comprise variable bindings **Var** and syntax extension bindings **Ext**.

The remaining extensions to the core syntax represent intermediate states and outputs in the `syntax-spec` expander. Matching syntax against a production pattern yields a substitution, which is then applied to the production's binding specification. The binding specification is interpreted into a second substitution with expanded syntax. This second substitution is applied to a template derived from the original pattern to produce the completely expanded form. Each of these steps requires new syntax:

pval = ... | *pvar* ↦ *stx*
tsubst = finite mapping from *pvar* to *stx*
template = **Tmpl**(*scopes*, *id*, *tsubst*)

The extension to *pval* represents the association of pattern variables with syntax inside a *bspec* after the first substitution. Template substitutions *tsubst* are the second substitution mentioned above; they contain the expanded syntax that results from interpreting the binding specification. Finally, the *template* captures the remaining information needed to construct the final result from *tsubst*. It includes the scope set from the parentheses of the original syntactic form[1], the tagging identifier from the head of the form, and an additional *tsubst* recording the original, unexpanded syntax for each subform. The latter is used if the binding specification does not provide an expansion for that subform.

1 The template needs only one scope set corresponding to parentheses because the restricted shape of patterns in the model does not allow internal tree structure within a production.

EXPANSION My model defines the meaning of a `syntax-spec` specification *spec* as a function that expands a concrete syntax object *stx* relative to a nonterminal name *ntname* and state *exp-st*:

$$\text{expand} : \textit{bspec-eval-f} \to \textit{spec} \to \textit{stx}, \textit{ntname} \to \textit{exp-st} \to \langle \textit{stx}, \textit{exp-st} \rangle$$
$$\text{where } \textit{bspec-eval-f} = \textit{spec} \to \textit{bspec} \to \textit{exp-st} \to \langle \textit{tsubst}, \textit{exp-st} \rangle$$

It relies on two helper functions, eval-bspec, which is supplied as a parameter, and apply-spec:

$$\text{eval-bspec} : \textit{bspec-eval-f}$$
$$\text{apply-spec} : \textit{spec} \to \textit{stx}, \textit{ntname} \to \textit{exp-st} \to \langle \textit{bspec}, \textit{template}, \textit{exp-st} \rangle$$

Parameterizing the core model over eval-bspec enables the straightforward extension of the core model with support for exporting (Section 6.1) and nesting (Section 6.2) binding forms.

The expand function uses apply-spec directly:

$$\text{expand}(\textit{bspec-eval-f})(\textit{spec})[\![\textit{stx}, \textit{ntname}]\!](\textit{exp-st}) = \langle \textit{stx}_1, \textit{exp-st}_2 \rangle$$
$$\begin{aligned} \text{where} \quad & \langle \textit{bspec}, \textit{template}, \textit{exp-st}_1 \rangle = \text{apply-spec}(\textit{spec})[\![\textit{stx}, \textit{ntname}]\!](\textit{exp-st}) \\ & \langle \textit{tsubst}, \textit{exp-st}_2 \rangle = \textit{bspec-eval-f}\ [\![\textit{bspec}]\!](\textit{exp-st}_1) \\ & \textit{stx}_1 = \text{fill-template}(\textit{template}, \textit{tsubst}) \end{aligned}$$

The apply-spec function matches the syntax against the specification to obtain a binding specification to drive expansion and a template to reconstruct the expanded form. The matched portions of the syntax are embedded into the resulting binding specification, within those forms that describe how they should expand. The given *bspec-eval-f* function expands the concrete syntax embedded within the binding specification to produce a template substitution. The template substitution represents the expanded subforms of the original syntax, and is used together with the template to complete the expansion. Both helper functions also accept an expander state and produce an updated state.

The initial call to expand requires an initial state:

$$\text{exp-st}_0 = \mathbf{St}(\text{store}_0, (), \varnothing, \text{scope}_0)$$

To make the overall expansion process concrete, consider the expansion of the following `let` expression similar to those found in Racket:

```
(let x (+ y 3) (sqr x))
```

Here the `(sqr x)` form is a macro that should expand to `(expt x 2)`. Expansion relies on the previously-presented binding specification, here in abstract syntax:

$$\mathbf{Prod}((\texttt{let}, \texttt{v}, \texttt{e}, \texttt{b}),$$
$$\mathbf{Group}(\mathbf{Subexp}(\texttt{e}, \texttt{expr}), \mathbf{Scope}(\mathbf{Group}(\mathbf{Bind}(\texttt{v}, \texttt{var}), \mathbf{Subexp}(\texttt{b}, \texttt{expr})))))$$

Following the process described in the preceding paragraph, the apply-spec(*spec*) function substitutes concrete subforms from the given let expression into the binding specification:

$$\mathbf{Group}(\mathbf{Subexp}(\texttt{e} \mapsto \texttt{(+}^{\{\}}\ \texttt{y}^{\{\}}\ \texttt{3}^{\{\}}\texttt{)}^{\{\}}, \texttt{expr}),$$
$$\mathbf{Scope}(\mathbf{Group}(\mathbf{Bind}(\texttt{v} \mapsto \texttt{x}_{\{\}}, \texttt{var}), \mathbf{Subexp}(\texttt{b} \mapsto \texttt{(sqr}^{\{\}}\ \texttt{x}^{\{\}}\texttt{)}^{\{\}}, \texttt{expr}), \texttt{s1})))$$

To make this example easier to read we can write $(\texttt{sqr}^{\{\}}\ \texttt{x}^{\{\}})^{\{\}}$ to represent the scoped syntax tree **Syntax**((**Syntax**(sqr, {}), **Syntax**(x, {})), {}). Note also that the substitution retains the pattern variable name. Using this concretized binding specification, the eval-bspec(*spec*) function interprets the specification to create a scope, bind the variable, expand subforms and return the template substitution:

$$(\texttt{e} \mapsto (\texttt{+}^{\{\}}\ \texttt{y}^{\{\}}\ \texttt{3}^{\{\}})^{\{\}}, \texttt{v} \mapsto \texttt{x}^{\{\texttt{s1}\}}, \texttt{b} \mapsto (\texttt{sqr}^{\{\texttt{s1}\}}\ \texttt{x}^{\{\texttt{s1}\}})^{\{\texttt{s1}\}})$$

Note that the syntax in the template substitution is annotated with the new `s1` lexical scope token. By combining this template substitution with the template also produced by apply-spec(*spec*), expand(*spec*) may now construct the final expansion of the original `let` expression.

The rest of this subsection explains these helpers in detail, starting with apply-spec(*spec*). This function is responsible for either selecting a matching production from the named nonterminal in the grammar, or applying macros according to the extension class specified in said nonterminal. Consider the first case, for matching a production:

$$\begin{aligned}
&\text{apply-spec}(\mathit{spec})[\![\mathit{stx}, \mathit{ntname}]\!](\mathit{exp\text{-}st}) = \langle \mathit{bspec}_1, \mathit{template}, \mathit{exp\text{-}st}\rangle \\
&\quad \text{where} \quad \mathbf{Syntax}(\mathbf{List}(\mathit{id}, \mathit{stx}_{\mathit{arg}}, \ldots), \mathit{scps}_{\mathit{form}}) = \mathit{stx} \\
&\qquad\qquad \mathbf{Syntax}(\mathit{formname}, \mathit{scps}_{\mathit{id}}) = \mathit{id} \\
&\qquad\qquad \mathbf{NT}(\epsilon\text{-}\mathit{eclass}, (\mathit{prod}, \ldots)) = \text{select-nonterminal}(\mathit{spec}, \mathit{ntname}) \\
&\qquad\qquad \langle(\mathit{formname}, \mathit{pvar}, \ldots), \mathit{bspec}\rangle = \text{select-production}(\mathit{formname}, (\mathit{prod}, \ldots)) \\
&\qquad\qquad \mathit{bspec}_1 = \text{substitute}(\mathit{bspec}, (\mathit{pvar} \mapsto \langle \mathit{pvar}, \mathit{stx}_{\mathit{arg}}\rangle, \ldots)) \\
&\qquad\qquad \mathit{template} = \mathbf{Tmpl}(\mathit{scps}_{\mathit{form}}, \mathit{id}, (\mathit{pvar} \mapsto \mathit{stx}_{\mathit{arg}}, \ldots))
\end{aligned}$$

The first two lines destructure the syntax to find the name of the form, *formname*, so that the next two lines can extract the appropriate production from the *spec*. The production consists of a pattern (*formname*, *pvar*, ...) and a binding specification *bspec* that contains reference to the variables *pvar* of the pattern. Using these pieces, apply-spec(*spec*) creates a new binding specification by substituting the concrete syntax of subforms for their corresponding pattern variables. Finally, apply-spec(*spec*) constructs the desired template.

The second case expands a macro and recurs:

$$\begin{aligned}
&\text{apply-spec}(\mathit{spec})[\![\mathit{stx}, \mathit{ntname}]\!](\mathit{exp\text{-}st}) = \text{apply-spec}(\mathit{spec})[\![\mathit{stx}_1, \mathit{ntname}]\!](\mathit{exp\text{-}st}_1) \\
&\quad \text{where} \quad \mathbf{Syntax}(\mathbf{List}(\mathit{id}, \mathit{stx}_{\mathit{arg}}, \ldots), \mathit{scps}) = \mathit{stx} \\
&\qquad\qquad \mathbf{NT}(\mathit{eclass}, (\mathit{prod}, \ldots)) = \text{select-nonterminal}(\mathit{spec}, \mathit{ntname}) \\
&\qquad\qquad \mathbf{Ext}(\mathit{eclass}, \mathit{transformer}) = \text{lookup}(\mathit{exp\text{-}st}, \mathit{id}) \\
&\qquad\qquad \langle \mathit{stx}_1, \mathit{exp\text{-}st}_1\rangle = \text{apply-transformer}(\mathit{exp\text{-}st}, \mathit{transformer}, \mathit{stx})
\end{aligned}$$

It resolves the tagging identifier in the given syntax to an extension binding whose *eclass* matches the one specified in the nonterminal. Then, it uses apply-transformer to invoke the extension's transformer and recurs with the result. The apply-transformer operation is responsible for painting syntax with two of the three kinds of scope tokens needed to achieve hygiene. It applies use-site scope tokens to

$$\text{eval-bspec}(spec)[\![\mathbf{Scope}(bspec)]\!](exp\text{-}st) = \langle tsubst, exp\text{-}st_2\rangle$$
$$\text{where}\quad \langle scp, exp\text{-}st_1\rangle = \text{enter-scope}(exp\text{-}st)$$
$$bspec_1 = \text{bspec-add-scope}(bspec, scp)$$
$$\langle tsubst, exp\text{-}st_2\rangle = \text{eval-bspec}(spec)[\![bspec_1]\!](exp\text{-}st_1)$$
$$\text{eval-bspec}(spec)[\![\mathbf{Bind}(pvar \mapsto id, bclass)]\!](exp\text{-}st) = \langle (pvar \mapsto id_1), exp\text{-}st_1\rangle$$
$$\text{where}\quad \langle id_1, exp\text{-}st_1\rangle = \text{bind}(exp\text{-}st, id, \mathbf{Var}(bclass))$$
$$\text{eval-bspec}(spec)[\![\mathbf{Ref}(pvar \mapsto id, bclass)]\!](exp\text{-}st) = \langle (pvar \mapsto id), exp\text{-}st\rangle$$
$$\text{where}\quad \mathbf{Var}(bclass) = \text{lookup}(exp\text{-}st, id)$$
$$\text{eval-bspec}(spec)[\![\mathbf{Subexp}(pvar \mapsto stx, ntname)]\!](exp\text{-}st) = \langle (pvar \mapsto stx_1), exp\text{-}st_1\rangle$$
$$\text{where}\quad \langle stx_1, exp\text{-}st_1\rangle = \text{expand}(\text{eval-bspec})(spec)[\![stx, ntname]\!](exp\text{-}st)$$
$$\text{eval-bspec}(spec)[\![\mathbf{Group}()]\!](exp\text{-}st) = \langle \mathbf{Empty}(), exp\text{-}st\rangle$$
$$\text{eval-bspec}(spec)[\![\mathbf{Group}(bspec_1, bspec_{rest}\ldots)]\!](exp\text{-}st) = \langle \text{tsubst-union}(tsubst, tsubst_1), exp\text{-}st_2\rangle$$
$$\text{where}\quad \langle tsubst, exp\text{-}st_1\rangle = \text{eval-bspec}(spec)[\![bspec_1]\!](exp\text{-}st)$$
$$\langle tsubst_1, exp\text{-}st_2\rangle = \text{eval-bspec}(spec)[\![\mathbf{Group}(bspec_{rest}, \ldots)]\!](exp\text{-}st_1)$$
$$\text{eval-bspec}(spec)[\![\mathbf{BindSyntax}(pvar \mapsto id, eclass, pvar_1 \mapsto stx)]\!](exp\text{-}st) = \langle tsubst, exp\text{-}st_1\rangle$$
$$\text{where}\quad tsubst = (pvar \mapsto id_1, pvar_1 \mapsto stx)$$
$$transformer = \text{eval-transformer}(exp\text{-}st, stx)$$
$$\langle id_1, exp\text{-}st_1\rangle = \text{bind}(exp\text{-}st, id, \mathbf{Ext}(eclass, transformer))$$

Figure 6.2: The eval-bspec(*spec*) function for interpreting binding specifications.

the input *stx* and macro-introduction scope tokens to pieces of syntax in the output stx_1 that did not appear in the input *stx*.

Continuing with eval-bspec, shown in Figure 6.2, we get to the second part of the process. The eval-bspec function is a straightforward syntax-directed interpreter for binding specifications that dispatches operations to Ballantyne et. al's API introduced in Section 5.2. The only subtle case is that for **Scope(***bspec***)**, whose interpretation is responsible for painting the third kind of scope tokens that realize hygiene: lexical ones. After allocating a new scope token in the expander state, eval-bspec paints this token on all pieces of concrete syntax inside the **Scope** binding specification form. Using the running `let` example, this step looks as follows:

$$\text{bspec-add-scope}(\mathbf{Group}(\mathbf{Bind}(\texttt{v} \mapsto \texttt{x}^{\{\}}, \texttt{var}), \mathbf{Subexp}(\texttt{b} \mapsto \texttt{(sqr}^{\{\}}\ \texttt{x}^{\{\}}\texttt{)}^{\{\}}, \texttt{expr}), \texttt{s1}))$$
$$= \mathbf{Group}(\mathbf{Bind}(\texttt{v} \mapsto \texttt{x}^{\{\texttt{s1}\}}, \texttt{var}), \mathbf{Subexp}(\texttt{b} \mapsto \texttt{(sqr}^{\{\texttt{s1}\}}\ \texttt{x}^{\{\texttt{s1}\}}\texttt{)}^{\{\texttt{s1}\}}, \texttt{expr}))$$

Critically, the lexical scope token `s1` is painted on the `let` body containing the macro application `(sqr x)` *before* the macro is expanded. The interpretation of this binding specification continues via a recursive call to produce a template substitution.

6.1 SEPARATE SCOPE AND BINDING

Having presented the core model, we can now extend it with a semantics for the `syntax-spec` binding rule forms that enable the specification of separate scoping and binding syntaxes. Figure 6.3 presents a small language specification using these binding rule forms. The `block` syntax creates a scope to which contained `define`

```
(syntax-spec
  (nonterminal expr
    #| other expression forms elided |#
    (block d:def-or-expr ...)    #:binding (scope (import d) ...))
  (nonterminal/exporting def-or-expr
    (define v:racket-var e:expr) #:binding (export v)
    e:expr))
```

Figure 6.3: Declaration of a `block` syntax with mutually-recursive definitions and `begin`, similar to `racket/block`.

syntaxes contribute (mutually-recursive) bindings. The `def-or-expr` nonterminal uses *exporting binding rules* that can contribute bindings to a surrounding context using the `export` form. The binding rule for `block` uses the `import` form to attach these bindings to a scope. A full description of exporting binding rules is available in the `syntax-spec` documentation (Appendix A.2.1.3).

The extension of the model for separate scope and binding forms consists of an additional nonterminal form and three kinds of binding specifications:

$$nt = \ldots \mid \textbf{ExportingNT}(\epsilon\text{-}eclass, (prod, \ldots))$$
$$bspec = \ldots \mid \textbf{Export}(pval, bclass) \mid \textbf{Reexport}(pval, ntname) \mid \textbf{Import}(pval, ntname)$$

The **ExportingNT** corresponds to the surface-syntax `nonterminal/exporting` definition, with its abstract syntax directly resembling the concrete syntax. The **Export** and **Reexport** forms may appear only in the top-level of a binding specification in a **ExportingNT** nonterminal. The **Import** form may appear only within a **Scope** form.

The expansion process for exporting nonterminals is split into two passes in order to support mutual recursion. Different forms in a shared scope may each export bindings to which the others refer. Such DSL names and macro names share a namespace in Racket, so an exported name may shadow a macro. Thus it is critical that bindings are established before the forms in their scope are expanded. The expansion process must first traverse all such exporting forms to register the bindings and then traverse the forms again to expand macros in subforms and resolve references.

In the model, the two passes of expansion are kicked off by the eval-bspec case for an **Import** from an exporting nonterminal:

$$\text{eval-bspec}(\mathit{spec})[\![\mathbf{Import}(\mathit{pvar} \mapsto \mathit{stx}, \mathit{ntname})]\!](\mathit{exp\text{-}st}) = \langle(\mathit{pvar} \mapsto \mathit{stx}_2), \mathit{exp\text{-}st}_2\rangle$$
$$\text{where} \quad \langle \mathit{stx}_1, \mathit{exp\text{-}st}_1\rangle = \text{expand}(\text{eval-bspec-pass1})(\mathit{spec})[\![\mathit{stx}, \mathit{ntname}]\!](\mathit{exp\text{-}st})$$
$$\langle \mathit{stx}_2, \mathit{exp\text{-}st}_2\rangle = \text{expand}(\text{eval-bspec-pass2})(\mathit{spec})[\![\mathit{stx}_1, \mathit{ntname}]\!](\mathit{exp\text{-}st}_1)$$

The two passes of expansion interpret the binding specifications in the exporting nonterminal with different evaluation functions:

$$\text{eval-bspec-pass1} : \mathit{spec} \to \mathit{bspec} \to \mathit{exp\text{-}st} \to \langle \mathit{tsubst}, \mathit{exp\text{-}st}\rangle$$
$$\text{eval-bspec-pass2} : \mathit{spec} \to \mathit{bspec} \to \mathit{exp\text{-}st} \to \langle \mathit{tsubst}, \mathit{exp\text{-}st}\rangle$$

The first pass is responsible for creating bindings for **Export** occurrences within the binding specification and within subforms indicated by **Reexport**. It leaves syntax corresponding to other portions of the binding specification unchanged. The second pass does not touch **Export** occurrences but handles all other binding specification forms just like eval-bspec from the preceding section. Macros expand in whichever pass first triggers processing of the subform in which they occur. Because the functions are just minor variants of eval-bspec, I omit their definitions.

Macro abstractions over definitions require special treatment of use-site scopes, as explained by Flatt (2016a, section 3.4). This treatment is implemented by a collaboration of the enter-scope, apply-transformer, and bind API operations I re-use from Ballantyne, King, and Felleisen (2020).

6.2 NESTING BINDING

Finally, we can extend the model with an interpretation for specifications of nesting binding structure. Nesting binding structures are those where bindings earlier in a sequence of syntactic elements are visible in the following but not the preceding elements. Examples include Racket's `let*` syntax, local variable bindings in C, and telescopes in dependently typed languages (de Bruijn, 1991). Figure 6.4 shows how a version of Racket's `let*` syntax can be specified in `syntax-spec`.[2] The `binding-pair` nonterminal is defined with a *nesting binding rule*. Binding

```
(syntax-spec
  (nonterminal expr
    #| other expression forms elided |#
    (let* (b:binding-pair ...) e:expr) #:binding (nest b ... e))
  (nonterminal/nesting binding-pair (nested)
    [v:var e:expr] #:binding (scope (bind v) nested)))
```

Figure 6.4: Declaration of a Racket-like `let*` syntax with nesting binding structure.

structures created by nesting binding rules include a hole which must be filled

2 This definition omits support for internal definitions in the `let*` body.

with additional scoping structure, indicated by the `nested` keyword. When a subexpression with nesting binding rules is referenced in another binding rule, the hole must be filled using the `nest` keyword. In the binding rule for `let*`, `(nest b ... e)` indicates that the hole in the binding structure associated with each binding pair `b` should be filled with the binding structure for the subsequent binding pair, and the final hole should be filled with the binding structure of the body expression `e`. The `syntax-spec` documentation describes nesting binding in more depth (Appendix A.2.1.3).

The extension to the model for nested binding needs another augmentation of the abstract syntax:

$$nt = \ldots \mid \textbf{NestingNT}(\epsilon\text{-}eclass, (prod, \ldots))$$
$$bspec = \ldots \mid \textbf{Nested} \mid \textbf{NestOne}(pval, ntname, bspec)$$

Nesting nonterminals defined with `nonterminal/nesting` are represented by the **NestingNT** abstract syntax form. **Nested** straightforwardly corresponds to the concrete syntax, while **NestOne** models a restricted, binary form of `nest`. The model requires that **Nested** appears exactly once in the binding specification for each production of a nesting nonterminal and does not appear in other kinds of nonterminals.

$$\text{eval-bspec}(spec)[\![\textbf{NestOne}(pvar \mapsto stx, ntname, bspec_n)]\!](exp\text{-}st) = \langle tsubst_2, exp\text{-}st_3\rangle$$
$$\begin{aligned}
\text{where}\quad & \langle bspec_{pre}, template_{pre}, exp\text{-}st_1\rangle = \text{apply-spec}(spec)[\![stx, ntname]\!](exp\text{-}st)\\
& \langle bspec, template\rangle = \alpha\text{-rename-template-vars}(template_{pre}, bspec_{pre}, bspec_n)\\
& \underline{bspec_{subst} = \text{substitute}(bspec, (\textbf{Nested} \mapsto bspec_n))}\\
& \langle scp, exp\text{-}st_2\rangle = \text{enter-scope}(exp\text{-}st_1)\\
& bspec_{scope} = \text{bspec-add-scope}(bspec_{subst}, scp)\\
& \langle tsubst, exp\text{-}st_3\rangle = \text{eval-bspec}(spec)[\![bspec_{scope}]\!](exp\text{-}st_2)\\
& stx_1 = \text{fill-template}(template, tsubst)\\
& tsubst_1 = \text{remove-template-vars}(tsubst, template)\\
& tsubst_2 = tsubst_1[pvar \mapsto stx_1]
\end{aligned}$$

Figure 6.5: The interpretation of nesting binding specifications

The semantic function for interpreting binding specifications, eval-bspec, is extended by one clause; see Figure 6.5. While the clause resembles the expand(*spec*)-based implementation of **Subexp**, it differs in that it also substitutes the provided $bspec_n$ for **Nested**. This substitution step is underlined in the figure. As a result $bspec_n$ never contains **Nested**, because it is always eliminated via substitution.

Ultimately, the expanded syntax for portions of the form represented by *stx* and for portions of original syntax contained in $bspec_n$ are both returned in *tsubst* from recursive evaluation. Elements of the first category are used to fill the template and

produce the expanded stx_1, while the latter are returned to allow the surrounding context to construct expanded syntax including the nested portion. This strategy of combining and then re-separating portions of two binding specifications creates the potential for conflation of the pattern variables belonging to each. The eval-bspec(*spec*) rule for **NestOne** avoids this conflation first by renaming to ensure that the pattern variable names do not conflict locally and then by removing the entries only needed for the local template so they do not conflict with names used in the surrounding template. The first task is accomplished by α-rename-template-vars and the second by remove-template-vars.

Use-site scopes also require special treatment for nesting, because they would prevent intended bindings. My solution is to apply an extra lexical scope token to all syntax in a nesting binding rule after filling the hole, which obviates the need for the use-site scope token. This approach is inspired by Flatt (2016b, section 2.3)'s treatment of `letrec-syntax`.

7

LAYERING SYNTAX-SPEC ATOP RACKET'S CONVENTIONAL MACRO SYSTEM

The `syntax-spec` macro system is built as a layer on top of Racket's conventional macro system. This implementation strategy supports the design goals outlined in Sections 3.3 and 3.5. That is, it decouples the development of `syntax-spec` from the core of Racket and enables the re-use of and integration with elements of the conventional macro system. Specifically, the `syntax-spec` metalanguage is implemented as a macro that accepts as input a DSL specification and generates as output compile-time code and macros that implement the DSL. The generated compile-time code connects to the host via an API that provides reflective access to the host expander. The generated code also re-uses existing abstractions built for conventional macros, including the `syntax-parse` pattern matching and templating language (Culpepper, 2012).

7.1 THE GENERATED DSL EXPANDER

The code generated from a `syntax-spec` DSL declaration implements a macro expander specialized to the core DSL syntax. The generated macros act as entry points from host-language expansion into the DSL-specific expander and DSL compiler. As an example, Figure 7.1 outlines the structure of the code generated for the specification of miniKanren's syntax from Figure 2.2. The generated code includes analogues of each element of the miniKanren `syntax-spec` declaration. All the code within `begin-for-syntax` blocks or appearing on the right-hand-side of a `define-syntax` macro definition is code that executes at Racket compile time, triggered by macro expansions (Flatt, 2002).

Binding classes and extension classes are each realized as compile-time structure type declarations (such as `term-variable-rep` and `goal-macro-rep`). Instances of these structure types are used to represent bindings of names and macros in the expander environment. They are analogous to *env-val* values from the model (Page 44) instantiated with *bclass* and *eclass* names. The new structure types inherit from parent types `binding-class-rep` or `extension-class-rep`. These parent types implement the `prop:set!-transformer` structure-type property protocol[1] in order to raise an error when the DSL name or macro is improperly used in a Racket rather than DSL context. Any instance of a structure implementing this protocol is recognized as a macro by the Racket expander, and the procedure associated with the property is invoked to define its expansion. In this case, the

1 https://docs.racket-lang.org/reference/stxtrans.html#%28def._%28%28quote._~23~25kernel%29._prop~3aset%21-transformer%29%29

```
;; Representations for DSL bindings in the expander environment
(begin-for-syntax
  (struct term-variable-rep binding-class-rep   ["term variable"])
  (struct rel-name-rep      binding-class-rep   ["relation name"])
  (struct goal-macro-rep    extension-class-rep ["goal macro"])
  (struct term-macro-rep    extension-class-rep ["term macro"]))

;; Syntax bindings for each syntactic form of the DSL
(define-syntax == (literal-rep '==))
;; similar literal definitions for disj, conj, fresh1, GE, quote,
;; cons, and TE elided.

;; Expand functions for each nonterminal in the DSL grammar
(begin-for-syntax
  (define goal-expander   (syntax-parser #| elided |#))
  (define term-expander   (syntax-parser #| elided |#))
  (define quoted-expander (syntax-parser #| elided|#)))

;; Macros for each host-interface form
(define-syntax run (expression-interface (syntax-parser #| elided |#)))
(define-syntax defrel (definition-interface (syntax-parser #| elided |#)))
(define-syntax EG (expression-interface (syntax-parser #| elided |#)))
(define-syntax ET (expression-interface (syntax-parser #| elided |#)))
```

Figure 7.1: An outline of the compile-time code and macros generated by `syntax-spec` for the specification of miniKanren's syntax from Figure 2.2

expansion raises a syntax error indicating that the form is used in an inappropriate position.

Each syntactic form name occurring as part of a nonterminal in a `syntax-spec` declaration creates a new name binding identifying the form. Identifying forms via these name bindings means that they can be shadowed by macros and imported and exported from modules, allowing for local tailorings of DSL syntax. In the generated code, the name bindings are realized by `define-syntax` macro definitions with transformers generated by the `literal-rep` helper, such as the `==` binding in Figure 7.1. DSL expanders use the name bindings to identify the forms, but do not invoke the associated transformer. Thus the macro transformer is called only when the form is used in a Racket context, and it raises a syntax error indicating that the form was improperly used from Racket.

Each nonterminal from a `syntax-spec` declaration is realized as a compile-time function that implements a macro expander specialized to the grammar of the nonterminal, such as `goal-expander` in the figure. These DSL expanders implement the expand and apply-spec(*spec*) metafunctions from the model (Page 46), specialized to the *spec* and a given *ntname*. Figure 7.2 provides a detailed view for the case of the `goal-expander` function that expands the miniKanren `fresh1` form. The task of selecting a production matching the input syntax (select-nonterminal in the model) is accomplished via pattern matching with `syntax-parse`. Like

```
(define goal-expander
  (syntax-parser
    [((~literal fresh1) (x:id ...) b ...)
     (eval-bspec
      (group (scope (bind 'x term-variable-rep)
                    (subexp b goal-expander)))
      (hash 'x (attribute x) 'b (attribute b))
      (lambda (tsubst)
        (with-syntax ([(x ...) (hash-ref tsubst 'x)]
                      [(b ...) (hash-ref tsubst 'b)])
          #'(fresh1 (x ...) b ...)))))]
    #| elided cases |#))
```

Figure 7.2: A part of the compiled expander for miniKanren goals, showing the case for the `fresh1` form.

in the model, scoping, binding, and expansion of subexpressions is orchestrated via interpretation of the binding specification. The eval-bspec(*spec*) metafunction from the model (Figure 6.2) is realized as a compile-time Racket function that is defined in the `syntax-spec` library. Each case of the DSL expand functions calls this interpreter, providing data representing the binding specification, a pattern environment with syntax values captured from the pattern match, and a continuation. The continuation realizes the application of the fill-template metafunction from the model to the template substitution that results from evaluating the binding specification. The continuation's job is to reconstruct the syntax of the form around the expanded subforms; it is implemented with a `syntax-parse` template.

Finally, each host-interface form in a `syntax-spec` declaration is compiled to a Racket macro, such as `run` in Figure 7.1. The Racket expander calls the macro when encountering a use of the boundary form, and the macro calls the DSL expander (such as `goal-expander`) and then the DSL compiler provided as part of the host-interface declaration. Ultimately the macro expands to the Racket code generated by the DSL compiler. The `expression-interface` and `definition-interface` decorator functions add checks to the transformers to ensure the host-interface form is invoked in the expected expression or definition context.

One way to understand `syntax-spec` is as an abstraction over a set of design patterns used in conventional Racket macros. For example, Ballantyne, King, and Felleisen (2020, Section 6) provide a design recipe for manually writing code like that in Figure 7.1 to implement a DSL-specific macro expander. Compared to `syntax-spec`, the low-level approach has several drawbacks. The low-level code is much more verbose. More importantly, writing a DSL expander by hand requires a sophisticated understanding of the "sets of scopes" hygiene model and the operational order in which macro expansion proceeds. Finally, a declarative specification in `syntax-spec` is much more useful to a human reader as documentation of the

structure of the language, or as a notation in which to think about and iterate on language designs.

7.2 INTEGRATING WITH RACKET VIA A REFLECTIVE API

The DSL expanders generated by `syntax-spec` reuse the syntax representation, expander environment, and hygiene mechanism of Racket macro expander. The `eval-bspec` interpreter for binding specifications accesses these elements via a reflective API exposed by the host expander that includes the operations described in Section 5.3. Reusing rather than re-implementing these parts of the host expander is essential in order to properly integrate conventional host and DSL macros.

This integration is particularly important when a macro abstracts over a combination of host and DSL syntax. Consider the `query-facts` macro from Figure 2.7. The macro is defined as a `goal-macro`, and will be expanded by the DSL expander. However, the code it generates within the `GE` boundary form includes Racket code, such as a reference to the `unify-results` function. To realize hygiene for this Racket identifier, the scopes applied by the DSL expander must also influence the resolution of Racket names.

Sharing the expander environment also enables cross-language name references. Because the DSL expander creates name bindings in an environment shared with the host expander, the host expander can recognize references to those names in host-language code and transform them with a reference compiler (Section 2.3.1). The shared environment also allows the host and DSL to recognize forms intended for use in the opposite context and raise helpful error messages.

At the time I began work on macro-extensible DSLs and `syntax-spec`, Racket did not expose a reflective API suitable for implementing DSL expanders. The operations described by Flatt et al. (2012) support binding and lookup in the expander environment and expansion of Racket subexpressions. The latter operation includes a facility to stop expansion on certain literal form names to allow the calling macro to provide an alternate interpretation of these forms. This is adequate for creating extensions to Racket's grammar that allow additional productions. For example, the body of Racket's `class` form (Flatt, Findler, and Felleisen, 2006) accepts Racket's usual expressions and definitions as well as class-specific forms such as field visibility and inheritance declarations. Class body elements are parsed by a macro that uses the above-mentioned operations. The essential thing missing from this API is a way to expand syntax for a context that does not allow standard Racket expression and definition forms—that is, a context where *only* DSL forms are accepted.

The `local-apply-transformer`[2] operation added by Alexis King provides the initial fix for this shortcoming. This operation hygienically expands a macro,

2 https://docs.racket-lang.org/syntax/transformer-helpers.html#(part._syntax/apply-transformer)

but unlike `local-expand` it does not carry on to expand the output of the macro under the expectation that it is Racket syntax. I wrapped this operation and those of Flatt et al. into a convenient form for creating DSL expanders, with the signature described in Section 5.3, and Ballantyne, King, and Felleisen (2020) show how to use this API to create DSL expanders. I subsequently discovered that the way Racket realized "sets of scopes" hygiene (Flatt, 2016a) for these operations was subtly incorrect, and contributed a repair.[3]

3 https://github.com/racket/racket/pull/3927

8 MULTI-LANGUAGE EXPANSION: INTEGRATING HOST AND DSL

Chapters 6 and 7 explain how `syntax-spec` implements macro expansion and is layered on top of Racket. However, this explanation covers only standalone DSLs that sit within Racket but do not integrate with it. This chapter explains how `syntax-spec` realizes safe multi-language interactions between a DSL and surrounding host-language code, host-language subexpressions, the host module system, and host IDE.

The elements that realize this integration include the following:

- A form of hygiene for DSL compilers that ensures that DSL names are isolated from host-language code, except as explicitly specified.
- Support for host-language subexpressions via an expansion process that suspends when they are reached and resumes their expansion when the code generated by the DSL compiler is expanded.
- Integration with the multiple passes of expansion in Racket definition contexts, ensuring that DSL definitions may participate in a common mutual recursion with host-language definitions.
- A mechanism for persisting static information about DSL code in host-language modules to communicate across separate compilations.
- A mechanism for communicating binding structure to IDE services.

As with the basic implementation of `syntax-spec` described in Chapter 7, each of these elements can be seen as an abstraction over a design pattern for programming with conventional Racket macros. A reader may wish to skip this rather technical, Racket-focused chapter on a first pass.

8.1 HYGIENE FOR DSL COMPILERS

In addition to *macro hygiene*, discussed in Chapters 5 and 6, `syntax-spec` also provides *compiler hygiene* for DSL compilers. Compiler hygiene ensures that code generated by the DSL compiler is properly isolated from the surrounding host-language code. It also assigns unique names to each binding occurrence to allow DSL compilers to perform transformations without worrying about shadowing.

Consider the compilation process for the `drop-2` relation:

```
(defrel (drop-2 lst tail)
  (fresh (el rest)
    (== lst (cons el rest))
    (fresh (el)
      (== rest (cons el tail)))))
```

Before invoking the DSL compiler, `syntax-spec` expands DSL macros and records syntax bindings for names. For example, it creates a syntax binding for the name `drop-2` in the containing module, recording that it is a relation name so that it is possible to check that references to this relation name are valid. Once invoked, the miniKanren optimizing compiler (Chapter 10) performs optimizations at the level of miniKanren syntax before generating Racket code. In this case, it combines the fresh variable allocations into a single `fresh`, inlines the unification with `rest` into the first unification, and drops the `rest` variable allocation:

```
(defrel (drop-2 lst tail)
  (fresh (el¹ el²)
    (== lst (cons el¹ (cons el² tail)))))
```

Notice that the optimization places the two distinct `el` variables in the same scope, making it necessary to distinguish them in order to avoid conflation. After optimizing in the intermediate representation, the DSL compiler generates a Racket procedure definition that is used to implement the relation:

```
;; Term Term -> State -> State
(define (drop-2ᶠⁿ lst tail)
  (lambda (st)
    (let ([el¹ (var)] [el² (var)])
      (unify lst (cons el¹ (cons el² tail)) st))))
```

The Racket procedure is curried in order to accept a state object which is invisible at the level of the surface syntax. It allocates fresh logic variable objects and then performs the unification, returning a new state. The name of this procedure definition is derived from the name of the miniKanren relation, but needs to be distinct because the binding will co-exist in the compiled Racket module with the syntax definition of the relation. In order to maintain the abstraction boundary of the DSL, it should be possible to reference the generated procedure only by compiling a reference to the corresponding surface-syntax relation name.

Compiler hygiene automates name management with respect to those these issues. After macro expansion, `syntax-spec` *alphatises* (Steele, 1978) DSL bindings and references before passing syntax to the DSL compiler. That is, it selects a unique fresh name for each binding. Because each binding has a unique name, the compiler can safely move bindings and references without the possibility of unintended capture due to shadowing, so long as it does not duplicate syntax. When a DSL compiler duplicates syntax, it must explicitly apply a renaming operation to

1. DSL expansion applies scope tokens (`mod`, `rel`, `f1`, `f2`) for macro hygiene:

```
(defrel (drop-2{mod} lst{mod,rel} tail{mod,rel})
  (fresh (el{mod,rel,f1} rest{mod,rel,f1})
    (== lst{mod,rel,f1} (cons el{mod,rel,f1} rest{mod,rel,f1}))
    (fresh (el{mod,rel,f1,f2})
      (== rest{mod,rel,f1,f2} (cons el{mod,rel,f1,f2} tail{mod,rel,f1,f2})))))
```

2. Alphatisation renames, using an identifier with a unique scope token for each binding:

```
(defrel (drop-2{1} lst{2} tail{3})
  (fresh (el{4} rest{5})
    (== lst{2} (cons el{4} rest{5}))
    (fresh (el{6})
      (== rest{5} (cons el{6} tail{3})))))
```

3. The miniKanren DSL compiler performs optimizations in the IR:

```
(defrel (drop-2{1} lst{2} tail{3})
  (fresh (el{4} el{6})
    (== lst{2} (cons el{4} (cons el{6} tail{3})))))
```

4. The miniKanren DSL compiler generates Racket code:

```
(define (drop-2{1} lst{2} tail{3})
  (lambda (st{i})
    (let ([el{4} (var)] [el{6} (var)])
      (unify lst{2}
             (cons el{4} (cons el{6} tail{3}))
             st{i}))))
```

5. Racket expansion of this code applies scope tokens for Racket macro hygiene:

```
(define (drop-2{1,mod} lst{2,mod,def} tail{3,mod,def})
  (lambda (st{i,mod,def,lam})
    (let ([el{4,mod,def,lam,let} (var)] [el{6,mod,def,lam,let} (var)])
      (unify lst{2,mod,def,lam,let}
             (cons el{4} (cons el{6,mod,def,lam,let} tail{3,mod,def,lam,let}))
             st{i,mod,def,lam,let}))))
```

Figure 8.1: Expansion and compilation process illustrating compiler hygiene. Superscripts show the set of scope tokens attached to each identifier.

maintain alphatisation. The names given to the DSL compiler are fresh and cannot conflict with names used in the surface program. As a consequence, the only way to refer to a Racket binding created with one of these names generated for a DSL binding is to use a name generated for the corresponding DSL reference. This property ensures that the code generated by the DSL compiler is properly isolated from surrounding Racket code, except as explicitly allowed by boundary forms and reference compilers.

Figure 8.1 steps through the expansion process for the `drop-2` definition to show how compiler hygiene works. As is conventional in Racket, unique fresh names are generated by creating an identifier that has the same symbol as the original name, but a unique fresh scope attached to it. This approach ensures that operations that extract just the symbol part for display to a user such as error message generation produce the original name. Internally, `syntax-spec` uses a persistent table (see Section 8.4) to map from surface syntax names to generated names. Persistent tables work across separate compilation boundaries, ensuring that even when a DSL name is bound in one module and referenced in another, compiler hygiene renames the binding and reference consistently.

Compiler hygiene is distinct from macro hygiene because DSL compilers manipulate names in different ways from syntactic sugar macros. A macro defines the scoping structure of a new form by the way it translates to existing scoping structures. For example, the following two implementations of the miniKanren `fresh` form using conventional macros yield two different scoping structures:

```
(define-syntax-rule (fresh (x ...) b) (let ([x (var)] ...) b))
(define-syntax-rule (fresh (x ...) b) (let* ([x (var)] ...) b))
```

With the first definition, only one occurrence of a given name is allowed in the binding group. With the second definition, there may be multiple occurrences, with the last one shadowing the others.

In contrast, the scoping structure for the DSL core syntax processed by a DSL compiler is explicitly declared by `syntax-spec`'s binding specifications. Consider the following `syntax-spec` production and binding specification for the `fresh` form:

```
(fresh (x:term-variable ...) b:goal)
#:binding (scope (bind x) ... b)
```

Given this declaration, the miniKanren DSL compiler could compile `fresh` forms to either Racket `let` or `let*` forms and yield the same behavior. This is because the names are checked for uniqueness during DSL expansion and assigned fresh names by compiler hygiene, so in this case the difference in scoping structures in the generated code is not observable.

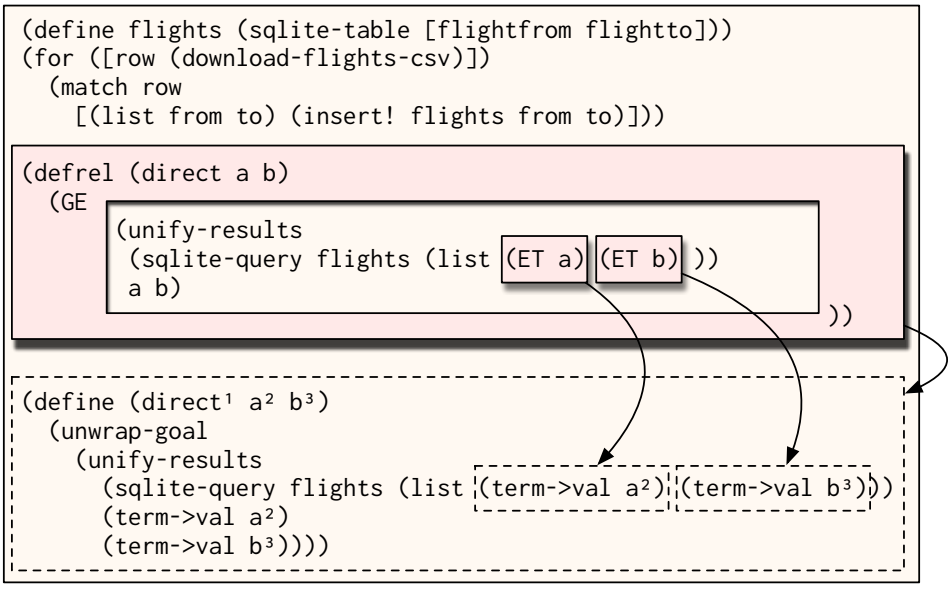

Figure 8.2: The compilation ordering problem for nested boundaries. The pink layer is DSL code, while the tan layer is host-language code.

8.2 HOST SUBEXPRESSIONS AND CROSS-LANGUAGE REFERENCES

When DSL code contains host-language subexpressions which in turn contain more DSL code, the question arises in what order macro-expansion and compilation of the various fragments should proceed. My design delays the expansion of host-language subexpressions until they are integrated into the compilation of the surrounding DSL code. This ordering ensures that bindings for generated names are available in the expander environment when expanding corresponding references generated from nested DSL code. Figure 8.2 illustrates this scenario. Notice that the compilation of the outer fragment of DSL code produces the alphatised $\texttt{a}^2$ and $\texttt{b}^3$ name bindings that are referenced in the inner DSL compilations. Because the `ET` host-interface form is implemented as a macro that expands to Racket, macro-expansion of the host subexpression within the `GE` boundary (displayed inset) necessarily compiles the nested DSL code and expands the result. As such, the outer DSL code must compile to establish the bindings before the contents of the `GE` boundary can be fully expanded.

Delaying expansion of the host subexpressions is almost as simple as copying them to the expanded output unchanged. However, when the subexpressions eventually expand they will need to be processed in an expander environment that contains bindings both from the DSL surface syntax and from the code generated by the DSL compiler. Thus, `syntax-spec` wraps the suspended expansions with a macro call that pairs the unexpanded code with a copy of the expander environment at the point where expansion suspended. For example, the body of the `GE` expression of Figure 8.2 in the result of DSL expansion will be:

```
(#%host-expression
  (unify-results
    (sqlite-query flights (list (ET a) (ET b)))
    a b))
```

The `#%host-expression` form is a macro whose transformer resumes the suspended expansion after adding the captured environment to the current expander environment. The syntax to resume expanding is its body. The closed-over expander environment is attached to the `#%host-expression` form. Racket supports attaching such additional data via *syntax properties*, which are key-value associations stored within syntax objects. When Racket expansion invokes the `#%host-expression` macro, the macro uses Racket's `local-expand` API to add the closed-over entries to the expander environment and continue expansion. At this point the bindings for surface-syntax names and their counterparts in the DSL compilation are both available in the environment.

Delaying expansion of host-language subexpressions also means delaying the alphatisation of names in nested DSL code that is performed by compiler hygiene (Section 8.1). This complicates the hygienic expansion of the compiled code. When alphatisation generates a fresh name for the name `a` in Figure 8.2, for example, it produces the identifier $\texttt{a}^{\{2\}}$ with a single fresh scope. However, expansion of the generated code will add scopes for the module and the `define` procedure body to yield the identifier $\texttt{a}^{\{2,\texttt{mod},\texttt{body}\}}$ before it is bound in the expander environment. Because expansion of the host subexpression inside the `GE` form is delayed, compilation of the `(ET a)` reference to the alphatised Racket name $\texttt{a}^{\{2\}}$ is also delayed until after the `mod` and `body` scopes have already been painted on the syntax. Without additional special behavior, this would make it impossible for the reference to resolve to the binding. To fix this problem, alphatisation of references uses Racket's `syntax-local-get-shadower` operation to apply extra scopes. The Racket expander tracks the lexical scopes corresponding to surrounding scoping forms at the current point of expansion, and the `syntax-local-get-shadower` operation adds these scopes to an identifier.[1] Thus while $\texttt{a}^{\{2\}}$ is recorded as the alphatisation of `a`, the additional scopes are also added to the reference to yield $\texttt{a}^{\{2,\texttt{mod},\texttt{body}\}}$.

REFERENCE COMPILERS As introduced in Section 2.3.1, binding class declarations can provide a reference compiler that defines the meaning of a DSL name when it is referred to directly from Racket code. For example, the binding class for miniKanren term variables declares that such references are compiled with an implicit `ET` value-translation boundary:

```
(binding-class term-variable #:reference-compiler compile-ET)
```

1 https://docs.racket-lang.org/reference/stxtrans.html#%28def._%28%28quote._~23~25kernel%29._syntax-local-get-shadower%29%29

The implementation of reference compilers elaborates on the structure types inheriting from `binding-class-rep` in Figure 7.1 of Chapter 7. For binding classes that implement a reference compiler, the underlying structure type's implementation of the `prop:set!-transformer` protocol alphatises the name and invokes the reference compiler rather than raising a syntax error.

DRAWBACKS OF DELAYED EXPANSION The design choice to delay expansion of host-language subexpressions poses problems for some DSL compilers. The host-language code is not yet in a form that can be analyzed or manipulated. Computing the free variables of host expressions is only possible by eagerly forcing their expansion, which may fail if there is further DSL code inside. It is also not possible to perform renamings within host expressions using the operations described in Appendix A.2.2.4. While none of these features are needed for the case studies examined in Chapter 14, I anticipate exploring an alternate, eager expansion order for host subexpressions in the future. This approach will require deeper integration with the Racket expander in order to macro-expand DSL code nested within host-language subexpressions without also immediately compiling this DSL code to Racket.

8.3 INTEGRATING WITH HOST DEFINITION CONTEXTS

The implementation of definition host-interface forms requires care to integrate the passes of `syntax-spec` expansion and compilation with the two-pass expansion process for Racket definition contexts. Like expansion for `syntax-spec`'s exporting nonterminals (Section 6.1), expansion for Racket definition contexts proceeds in two passes. The first pass registers bindings for defined names. The second pass expands definition right-hand sides. When DSL definition forms contribute bindings to host definition contexts, both DSL expansion and compilation to Racket code need to be split between the two passes to work properly. DSL expansion must be split between the passes to support mutual recursion at the level of the DSL surface syntax. Compilation must be split in order to support mutual recursion in the generated code.

The design of `host-interface/definition` declarations makes it possible to split out the portions that must happen in each pass. Figure 8.4 reproduces the definition of the `defrel` host-interface form from Figure 2.2, annotating it to show which portions relate to each pass of expansion and compilation. Definition host-interface forms use exporting binding rules (Section 6.1), which segment out portions that occur in the first pass of expansion with the `export` and `re-export` forms. Here, the binding of the relation name `r` is declared with an export rule, so it is created during the first pass of expansion of the surrounding definition context. Binding of the relation arguments `x ...` and expansion of the body goal `g` do not use the exporting binding rule forms, so these steps are deferred to the second pass. The compilation task is segmented into the two passes via the `#:lhs`

```
(host-interface/definition
  (defrel (r:rel-name x:term-variable ...+)
    g:goal)
  #:binding [(export r)
             (scope (bind x) ... g)]
  #:lhs [#'r]
  #:rhs [(compile-relation #'(x ...) #'g)])
```

Figure 8.3: The host-interface form declaration for miniKanren relation definitions. Portions that relate to pass 1 expansion and compilation are underlined. Portions that relate to pass 2 expansion and compilation are doubly underlined.

and `#:rhs` clauses of the declaration. The `#:lhs` clause generates the name for the Racket definition that a DSL definition compiles to. It executes during the first pass of expansion for the surrounding definition context. The `#:rhs` clause generates code for the right-hand side of the Racket definition. It executes during the second pass, when the Racket expander returns to expand the right-hand side of each definition. Here the generated right-hand side is a `lambda` expression implementing the relation.

Integrating with the host's two-pass expansion process requires yielding control to the host expander at two points. In each case, `syntax-spec` yields code for the host expander to process and arranges to continue its own expansion process by including in this code a call to a continuation macro. Figure 8.4 illustrates these two patterns of control flow.

YIELDING DEFINITIONS FOR EXPORTED NAMES The first case in which `syntax-spec` yields to the host expander is to create a DSL name binding following an `export` binding rule. The relation name `all-same` in Figure 8.4 is bound in this way. This relation name is intended to be visible in the surrounding definition context, which may be a module body. The only way to register a binding in a module is to expand to a `define` or `define-syntax` form in the module body. This restriction ensures that the binding will also be available when the module is loaded for the separate compilation of a dependent module (Flatt, 2002). Thus, the `defrel` macro in Figure 8.3 must expand to a `define-syntax` form that records `all-same` as a relation name. Furthermore, `syntax-spec` must wait to continue further DSL expansion until this definition is processed by the host expander, because the new binding may shadow the name of a syntactic form and thus change how subsequent syntax is parsed.

The first step of expansion in Figure 8.4 shows how `syntax-spec` yields to the host expander to allow it to process such a definition. The `defrel` macro generated by `syntax-spec` expands to a `begin` form containing the `define-syntax` to be processed by the host expander and a `continue-pass1` macro invocation that resumes `syntax-spec` expansion. The computation that must suspend is

```
(defrel (all-same a b c) (conj (== a b) (b c)))
```

1. In host pass 1, `syntax-spec` begins pass 1 expansion of the `defrel` and generates a syntax definition for the relation name. The `continue-pass1` macro call contains a continuation that resumes `syntax-spec` expansion once the `define-syntax` has been processed.

```
(begin
  (define-syntax all-same (rel-name-rep))
  (continue-pass1 #<procedure>))
```

2. Continuing host pass 1, `syntax-spec` completes pass 1 expansion of the `defrel` and generates the corresponding Racket `define`, with a macro call on the right-hand side to continue expansion in host pass 2.

```
(begin
  (define-syntax all-same (rel-name-rep))
  (define all-same¹
    (defrel-pass2
      (defrel (all-same¹ a b c) (conj (== a b) (== b c))))))
```

3. In host pass 2, expansion continues into the `define` right-hand side. The `defrel-pass2` macro triggers `syntax-spec` pass 2 expansion which eliminates the `conj` DSL macro and alphatises. Finally, `defrel-pass2` emits a call to the DSL compiler macro for the right-hand side, `compile-relation`.

```
(begin
  (define-syntax all-same (rel-name-rep))
  (define all-same¹
    (compile-relation (a² b³ c⁴) (conj2 (== a² b³) (== b³ c⁴)))))
```

4. Continuing host pass 2, the DSL compiler generates Racket code for the RHS.

```
(begin
  (define-syntax all-same (rel-name-rep))
  (define all-same¹
    (lambda (a² b³ c⁴)
      (lambda (st)
        (bind (unify a² b³ st)
              (lambda (st) (unify b³ c⁴ st)))))))
```

Figure 8.4: Annotated trace of the expansion process for a use of the `defrel` host-interface form defined in Figure 2.2 in a Racket definition context.

the interpretation of the binding specification, performed by the eval-bspec interpreter (Chapter 6). This interpreter is structured in direct style. To suspend it, `syntax-spec` uses Racket's support for delimited continuations. When the interpreter invokes the operation to bind a name, this operation captures the state of the interpreter in a continuation, and attaches this continuation value to the `continue-pass1` macro call. When this macro is invoked, it invokes the continuation to resume `syntax-spec` expansion.

YIELDING CONTROL UNTIL THE SECOND PASS The second case in which `syntax-spec` yields to the host expander is when it completes pass 1 expansion of the definition form and is ready to generate a Racket `define`. At this point, pass 2 of DSL expansion and the compilation needed to generate the right-hand side of the `define` is not yet complete. But before this further expansion and compilation can occur, the host expander must register the name of the compiled `define` and complete pass 1 expansion for other definitions in the module. Only once the names of all definitions have been registered can DSL compilation for the right-hand side continue.

Steps 2–4 of Figure 8.4 show how `syntax-spec` orchestrates this process. After completing pass 1 expansion, `syntax-spec` expands to a `define` form with a continuation macro call on the right-hand side. This continuation macro is invoked when the host expander reaches the definition in pass 2 expansion.

Unlike the case above that requires delimited control, here it is easy to separate out the pass 1 and pass 2 expansion processes into separate macros as part of `syntax-spec` compilation. Thus, pass 1 expansion for a `defrel` is accomplished by the `defrel` macro, while pass 2 expansion is accomplished by a hidden `defrel-pass2` macro generated by `syntax-spec`. This second macro receives as input DSL syntax that has been expanded and reconstructed by pass 1 expansion. In step 2 of Figure 8.4, this is the complete syntax of the `defrel` form, where the `all-same`[1] name has been alphatised by pass 1 expansion. After completing pass 2 expansion and alphatisation, the `defrel-pass2` macro invokes the DSL compiler provided in the `#:rhs` clause of the host-interface definition.

8.4 PERSISTING STATIC DATA IN HOST MODULES

In multi-language semantics, typing judgments track static information for names from both parts of the multi-language (Matthews and Findler, 2007). This allows fluid interactions between the parts. For example, if DSL code establishes name bindings, then those names may be referenced from DSL code nested within a host subexpression, across two language boundaries. When `syntax-spec` DSLs are used within Racket modules, definitions of DSL entities, such as a miniKanren relation, often occur in one host-language module but are used in another. Checking static semantics for such a reference requires accessing static information across the boundaries connecting the DSL with host modules. The `syntax-spec`

system provides a new abstraction for such communication: persistent symbol tables (Appendix A.2.2.3). These tables allow a DSL compiler to record compile-time information associated with DSL name bindings. These entries are globally visible, so they can be accessed across language boundaries. Table extension is restricted to be monotonic to avoid pitfalls due to ordering of effects. Entries are keyed by name bindings rather than just names, so the tables also integrate smoothly with hygiene.

Crucially, persistent symbol table entries persist across separate compilations. For example, the compilation of a `defrel` defining the `all-same` relation in module A may create a symbol table entry recording that the relation has three parameters. The expander writes the compiled module A to a file, and the Racket virtual machine shuts down. Suppose module B depends on module A. Then the Racket expander processing module B in a new instance of the Racket virtual machine loads module A in order to access its syntax definitions and other static information. Persistent symbol tables ensure that entries recorded during the expansion of module A will be available during the expansion of the dependent module B. Thus, compilation of a call to the `all-same` relation can consult the symbol table to check that the call has the correct number of arguments.

The implementation of persistent symbol tables abstracts over a pattern used by conventional Racket macros. Racket's module system (Flatt, 2002) allows macros to generate code that performs side effects each time a module is *visited*—that is, loaded to support the expansion of another module. For example, the macro that implements `defrel` could include in its expansion code like the following:

```
(begin-for-syntax
  (free-id-table-set! relation-arity #'all-same 3))
```

A `begin-for-syntax` block at module level denotes code to execute each visit. In this case, the code mutates a global `relation-arity` table.

Unfortunately this pattern of expanding to visit-time code is awkward to use in complex DSL compilers. A compiler pass that traverses DSL syntax to perform a static check may need to create and consult many table entries as it performs the traversal. Entries created via the pattern of expanding to visit-time code are only accessible once that code executes after the DSL compiler macro expands, so they cannot be consulted during the local compilation. Furthermore, some entries do not need to persist across separate compilations because they relate to local names, whereas other entries need to persist. Such a static check would need to maintain a separate, transient table as local state and then manually reflect the selection of entries that need to persist into a `begin-for-syntax` block as above.

Persistent symbol tables provide a simpler interface. Adding an entry to a persistent symbol table with `symbol-table-set!` has immediate global effect, and entries automatically persist across separate compilations (only) if the key is a module-level name. Under the hood, a persistent symbol table is implemented with a combination of two free-identifier tables, which are tables keyed by identifier's

hygienic binding identities.[2] One holds entries transiently, while the other holds entries that persist via the `begin-for-syntax` pattern discussed above. Entries are initially placed in the transient table. At the end of the execution of the `#:lhs` DSL compiler clause of a definition host-interface form, `syntax-spec` automatically reflects entries into a `begin-for-syntax` block. This moves the entries from the transient portion of the table to the persistent portion. Lookup in the table consults each of the underlying tables in turn.

8.5 RECORDING INFORMATION FOR THE IDE

Properly integrating `syntax-spec` DSLs with Racket also means integrating with the Racket IDE (Findler et al., 2002). If programmers are to experience Racket and the DSL together as a multi-language system, then the Racket IDE needs to understand DSL code as well as cross-language interactions. In particular, Racket's IDE needs information about bindings and references in order to provide services such as jump-to-definition and arrows that point from references to their bindings.

It is straightforward to analyze code in Racket's core syntax to identify bindings and references. However, syntax defined by macros poses a challenge. Racket's approach is to augment syntax objects with source location information to track the textual origin of syntax during expansion. Then, it analyzes the fully-expanded core syntax and uses the attached source location information to correlate to positions in the surface syntax. This technique works well for macros that implement simple syntactic sugar, but is inadequate in the presence of sophisticated DSL compilers. The Racket code output by a DSL compiler may not include bindings and references corresponding to every surface-syntax binding and reference. For example, some may be eliminated by a dead-code removal optimization. When this happens, the source-location correlation approach will not recognize these surface-syntax bindings and references. Macros can work around the problem by manually annotating syntax with `'disappeared-use` and `'disappeared-binding` syntax properties.[3] Identifiers included in these properties are treated as references and bindings, respectively, by the IDE's analysis.

Because `syntax-spec` requires binding structure declarations, it avoids the pitfalls of source-location correlation for DSL compilers. The DSL expanders generated by `syntax-spec` capture information for the IDE when they process bindings and references following the DSL's binding specification. They ultimately convey this information to the IDE by attaching `'disappeared-use` and `'disappeared-binding` syntax properties on the output of host-interface form expansions. Unlike with conventional macros this requires no extra work on the part of the DSL creator. Because binding information is captured by the DSL

2 https://docs.racket-lang.org/syntax/syntax-helpers.html#%28part._.Dictionaries_for_free-identifier~3d_%29

3 https://docs.racket-lang.org/tools/Check_Syntax.html#%28idx._%28gentag._47._%28lib._scribblings%2Ftools%2Ftools..scrbl%29%29%29

expander, DSL compilers may perform optimizations that eliminate bindings and references without impacting IDE services.

9 REFLECTION: HYGIENIC EXPANSION AND BINDING SPECIFICATIONS

Having explained how the `syntax-spec` expander uses binding specifications to hygienically expand DSL code in Chapter 6, we can now explore how this requirement constrains the design of the binding specification language. Two major issues stand out.

9.1 MACRO EXTENSIBILITY AND EXPANSION ORDER

To equip DSL macros with the same power as Racket macros, `syntax-spec` allows DSL macro definitions to shadow value definitions and vice versa. This affordance demands that the processes of understanding binding and expanding macros be interleaved. As a consequence, the order of expansion in scopes with mutually-recursive definitions is observable, and `syntax-spec` must allow its explicit specification.

```
(define-syntax-rule
  (def v e)
  (define v e))
(define-syntax-rule
  (res)
  "outer")
(block
  (begin
    (def res (lambda () "inner")))
  (res))
```

```
(define-syntax-rule
  (def v e)
  (define v e))
(define-syntax-rule
  (res)
  "outer")
(block
  (define (f) (res))
  (def res (lambda () "inner"))
  (f))
```

Figure 9.1: The significance of expansion order in mutually-recursive scopes

To make this point concrete, consider the Racket programs in Figure 9.1, and assume that Racket's binding structure has been specified with `syntax-spec`. Both programs should evaluate to `"inner"`. In both programs, the expansion of the use of the `def` macro must happen first. The macro introduces a new variable definition of the name `res` that shadows the original macro definition `res`, so it is essential that it expand before the macro use `(res)`. Subforms of Racket's `begin` are intended to include definitions that contribute bindings to the surrounding context. So it makes sense to expand the subform first and then the sibling form. However, that order does not work for `define`, which supports mutually recursive definitions. Supporting mutual recursion requires discovering all the `define`s in

the block in a first pass to determine bound names and only in the second pass expanding the `define` right-hand-sides, which cannot contribute bindings.

To decide in which pass a macro should expand, `syntax-spec` needs to know whether the subforms mentioned in a binding rule can contribute new bindings to the current context or are guaranteed not to. The type distinction between exporting nonterminals (`nonterminal/exporting`) versus standard nonterminals provides `syntax-spec` with this information. An alternative design could infer this exporting information rather than rely on an explicit, syntactic distinction, but because its behavior observably differs from the standard one, I prefer the explicit design.

9.2 HYGIENE

"Sets of scopes" hygiene demands a certain order of scope-painting operations within the expander and thus imposes a restriction on the binding language design.

```
(define x "outer")
(define-syntax-rule (m a) (list x a))
(block
  (define x "inner")
  (m x))
```

```
;; expands to (with scope tokens)
(define x{a_lex} "outer")
(block
  (define x{a_lex, b_lex} "inner")
  (list x{a_lex, c_mac} x{a_lex, b_lex, d_use}))
```

Figure 9.2: The expander must attach scope before expanding the block body

Figure 9.2 shows an example of a situation that motivates the restriction. Recall from Section 5.1 that scope sets uses lexical scope tokens to prevent use-site context captures of free references in macro templates. The key scope token in this example is the underlined b_{lex}. In order for b_{lex} to appear on the `x` originally from the macro use `(m x)` but not on the `x` from the macro template, the expander must paint this b_{lex} token on the body of the `block` *before* expanding the macro use.

However, before the macro use `(m x)` is expanded, the role of the identifiers contained in the use is unknown. They may be binders, references, or names of syntactic forms; they may belong to the current scope or a more nested one. It is only possible to justify painting the scope b_{lex} because the scoping structure can be conservatively approximated: regardless of the roles of the identifiers within the block, they definitely belong inside the `block` scope.

Because `syntax-spec` is designed with the goal of implementing hygienic expansion via scope sets, it can express only binding structures where this approximation is valid. Python's `nonlocal` and `global` keywords[1] are examples of features with binding structure for which the approximation would be invalid. The restriction is reflected in the syntax of the binding specification language in that the **Scope** rule works by lexical containment: all contained subforms are unconditionally part of the scope. By contrast, other binding specification languages such

1 See https://docs.python.org/3/reference/simple_stmts.html#the-nonlocal-statement.

as Statix (Antwerpen et al., 2018) allow scopes to be passed as values but do not realize macro extensibility. In such languages, binding rules may associate such scopes with syntax conditionally based on other factors such as type information.

Part III

APPLICATIONS AND EVALUATION

APPLICATIONS AND EVALUATION

This part examines a number of DSL implementations built with `syntax-spec` and uses these implementations to evaluate the effectiveness of its design.

DSL IMPLEMENTATIONS Chapters 10 and 11 present the largest DSL implementations I built using `syntax-spec`: a miniKanren with an optimizing compiler and a Parsing Expression Grammars DSL. These implementations each come with a static semantics and optimizing compiler that take advantage of the `syntax-spec` macro system's features.

Chapter 12 presents a variety of other `syntax-spec` DSL implementations I created. This collection of languages helps test the expressivity of `syntax-spec`'s declaration language. These DSLs include state machines, a reimplementation of a subset of Racket's class system, a command-line argument parser, a reimplementation of a small hardware-description language, and a multi-stage variant of miniKanren (Ballantyne et al., 2025).

Chapter 13 presents DSLs created by other programmers since the public release of `syntax-spec`. Cameron Moy leveraged `syntax-spec` in his research on "domain-specific contract languages" (Moy, Jung, and Felleisen, 2025) for specifying properties of Racket programs. Each of these DSLs implements a different logic for validating properties of traces of events that are captured by a trace contracts system. Additionally, students from the "hack your own language" course I taught in Spring 2025 created new implementations of a teaching language for logical specifications and a finite-choice logic programming language. Finally, developers in the Racket community created DSLs for point-free programming and pattern matching using lenses.

EVALUATING `syntax-spec` These DSL implementations provide an opportunity to evaluate the `syntax-spec` design. The unique design choice of `syntax-spec` is to conceptualize DSLs as extensions to a multi-language system. Thus, a first question is whether DSLs in practice benefit from the multi-language structure. That is, whether they can leverage fine-grained interaction with the host while simultaneously realizing domain-specific static semantics and optimizing compilation in a sound manner.

Beyond this basic question I evaluate additional issues of expressive power and utility. The `syntax-spec` system relies on declarative specifications of syntax and binding structure. Thus, a second question is whether the specification language is expressive enough to capture the desired syntax and binding rules of all the DSLs. Given a specification, a third question is whether the services that `syntax-spec` provides are useful to support the implementation of DSL static

semantics and optimizing compilation. The system also automatically provides hygienic macro extensibility, so a fourth question is whether this extensibility is useful for implementing desugaring and end-user extensions in practice.

A fifth question is to what degree the `syntax-spec` metalanguage yields shorter implementations and requires less programmer expertise as compared to low-level macro programming. It is after all possible to implement multi-language and macro-extensible DSLs in Racket directly using the same reflective APIs that `syntax-spec` uses under the hood. The `syntax-spec` metalanguage is designed to provide a concise and accessible abstraction over these techniques. Finally, a sixth question is whether the expansion-time performance of `syntax-spec` is acceptable.

The chapters in this part covering DSL implementations discuss how each DSL takes advantage of `syntax-spec` and where they run into difficulties. Chapter 14 summarizes these results into a holistic evaluation and addresses the further issues of comparative concision and expansion-time performance.

10 THE MINIKANREN OPTIMIZING COMPILER

This chapter explores the `syntax-spec`-based miniKanren implementation from Chapter 2 in more depth. In particular, it focuses on the benefits of the multi-language architecture. Programs gain expressive power from the combination of DSL and host-language code. Macros can abstract over this combination to extend the DSL. At the same time, the multi-language boundaries protect the abstractions of the DSL, making optimization compilation safe. The content of this chapter is the result of collaborative work with Mitch Gamburg and Jason Hemann (Ballantyne, Gamburg, and Hemann, 2024). As such, this chapter uses the plural pronoun "we" to refer to the authors.

10.1 EXTENSION AND MIXING LIKE A SHALLOW EMBEDDING

Using `syntax-spec` imbues our miniKanren implementation with the same powers of extension and intermixing of DSL and host-language code one gets with a shallow embedding. Specifically, miniKanren programmers can extend the syntax of the DSL with macros, and they can commingle host and DSL code in a disciplined manner with boundary forms.

Programmers working in shallow embeddings of miniKanren have dreamed up many language extensions and developed applications that take advantage of the ability to intermix miniKanren and Racket. Together these programs make the miniKanren ecosystem a kind of natural experimental environment.

In this section, we reproduce some of these examples. By doing so, we show how `syntax-spec` endows an optimizing implementation with the same freedom of language extension and intermixing as a shallow embedding—with little effort.

10.1.1 *Extensibility*

Many of those programmer-designed extensions to shallow miniKanren embeddings are simple host-language macros. Hence it is perhaps unsurprising that we can similarly implement them as DSL macros. The DSL programmer uses the same extension mechanism as the DSL designer does for syntactic sugar. The only difference between an end-programmer language extension and built-in syntactic sugar is who designs it. Since DSL extensions are macros that expand to DSL core-language code, the compiler's expectations about the source code continue to hold.

Concretely, some shallow miniKanren implementations include a `matche` (Keep et al., 2009) pattern-matching form, like that used in `route` of Figure 2.6. The

first form in each `matche` clause is a pattern. The process of pattern matching introduces the necessary logic variables and unifies a term corresponding to the pattern against the relation parameters.

The `matche` implementation is a fairly pedestrian `goal-macro`, which we show in Figure 10.1. The implementation compiles each pattern group to a term expression `pats^` and a list of pattern variables `xs`. This process relies on the compile time helper function `compile-pats`, whose definition we omit. The macro then constructs a goal that introduces a name for the arguments list followed by a `conde`. Each `conde` clause introduces the pattern variables with `fresh`, unifies the term expression `pats^` with the arguments list, and executes the sequence of goals `g ...` as a conjunction.

Since the host and DSL use the same extension system—an explicit design goal of `syntax-spec`—we can even write macros that generate mixed-language code. For instance, a programmer might want to express a relation definition like `route` even more concisely. In `route`, we list the parameters in the header, and then match against that same parameter list. We can abstract over that duplicated syntax with `defrel/matche`, an ordinary Racket macro that generates a `defrel` with a `matche` as its body. The top of Figure 10.2 shows the definition, and the bottom contains a re-implementation of `route` using this new abstraction.

10.1.2 *Mixing DSL and Host-Language Code*

Sometimes miniKanren programmers want just a touch of Racket *inside* their miniKanren programs. The miniKanren language is a distillation of purely relational constraint logic programming that deliberately eschews features such as side effects and higher-order control flow. However, sometimes you just need to do a little `printf` debugging.

Figure 10.3 demonstrates how a programmer can write ordinary Racket code within a miniKanren program. This interaction relies on the `GE`, `EG`, and `ET` boundary forms declared in Figure 2.2. Recall that the `GE` boundary form allows a miniKanren programmer to include some Racket code inside a goal. In this example, the Racket code constructs a string containing information about the variable `q` and its value. This string construction relies on yet another cross-language boundary to access a miniKanren term variable from the Racket context. The `ET` form admits a miniKanren term in a Racket context; the cross-language translation of the value is trivial except for fresh miniKanren logic variables, which are opaque to the Racket context. The final line in the `GE` form contains an ordinary Racket function call to `succeed/print`, which consumes and prints a string.

Beyond allowing Racket code to access term variables and perform side effects in the context of a goal, `GE` also allows Racket code to define the behavior of the goal—whether it succeeds or fails, and whether it constrains any variables. The `succeed/print` body returns a goal value created by the `EG` boundary form to represent this behavior. In this example the goal is simply `succeed`—Section 10.1.3

```
(define-dsl-syntax matche goal-macro
  (syntax-parser
    [(matche (arg ...+) [pats g ...] ...+)
     #:with ([pats^ xs] ...) (map compile-pats (attribute pats))
     #'(fresh (ls)
         (== ls (list arg ...))
         (conde [(fresh xs (== pats^ ls) g ...)] ...))]))

;; (Listof Pattern) -> (Pair Term (Listof TermVar))
(define (compile-pats pats) #| elided |# )
```

Figure 10.1: The implementation of the `matche` pattern matching extension.

```
(define-syntax defrel/matche
  (syntax-parser
    [(defrel/matche (name:id arg:id ...+)
       clause ...+)
     #'(defrel (name arg ...)
         (matche (arg ...) clause ...))]))

(defrel/matche (route origin end path)
  [(a a '())]
  [(a b (cons (list a layover) remainder))
   (absento a remainder)
   (direct a layover)
   (route layover b remainder)])
```

Figure 10.2: The `defrel/matche` macro and an even more concise re-definition of the `route` relation.

```
> (define (succeed/print str)
    (printf str)
    (EG succeed))
> (run 1 (q)
    (fresh (x)
      (== q (list x 'cat))
      (GE
        (let ([str (format "value of ~a: ~a\n" 'q (ET q))])
          (succeed/print str)))))
value of q: (#<mk-lvar> cat)
'((_.0 cat))
```

Figure 10.3: Using Racket to construct and print a string through the host FFI.

```
(GE (let ([str (format "~a: ~a\n" 'q (ET q))])
      (printf str)
      (EG succeed)))

;; -Compiles to->

(lambda (state)
  (define goal-val
    (let ([str (format "~a: ~a\n" 'q (translate-term
                                       (substitute q-rt state)))])
      (printf str)
      (seal-goal succeed-rt)))
  (apply-goal (unseal-goal goal-val) state))
```

Figure 10.4: The above `GE` sub-form from the example in Figure 10.3 compiles to the below Racket implementation code. We have in-lined here the body of `succeed/print` in both examples for clarity.

shows a more sophisticated use. An additional boundary form `TE` form allows Racket code to compute a term; it is not used in this example.

As mentioned in Section 2.3.1, connecting the parts of a multi-language safely involves inserting value translations or contracts at the boundaries. To show how these translations are inserted, Figure 10.4 illustrates the compilation of the example in Figure 10.3. The compilation of each cross-language boundary inserts a call to a value translation function. Together, `seal-goal` and `unseal-goal` implement the lump embedding for goal values. The `translate-term` operation is responsible for implementing the natural embedding of term values, translating term data to Racket values while sealing logic variables. This Racket-safe version of a miniKanren term can find its way back to a miniKanren context through `TE`, which unseals the logic variables.

Some of the boundary forms need to access information from the miniKanren state. In our runtime, the current value of a logic variable depends on the received state, which contains information on the history of unifications during the program's execution. We implement goals as functions from a state to a stream of states representing the nondeterministic result of the goal. In Figure 10.4, the `GE` form compiles to such a function, making the `state` variable accessible to the generated code within. The compilation of `ET` uses this state with `substitute` to fill in known logic variable values before passing the term to Racket. The compilation of `GE` uses the state via `apply-goal` to continue execution with the goal produced by the Racket code.

10.1.3 *Host Code in DSL Extensions*

The synthesis of extensibility and host-language interoperation produces some surprisingly powerful behaviors. For instance, the PL enthusiast will quickly tire of writing and rewriting that same kind of `printf` logic from Section 10.1.2. Instead, the informed enthusiast will instinctively reach for the means to abstract over syntactic boilerplate: macros.

```
(define-syntax trace-defrel
  (syntax-parser
    [(_ (name:id a*:id ...) b:goal/c)
     #'(defrel (name a* ...)
         (GE (begin (printf "~s\n" (list 'name a* ...))
                    (EG b))))]))
```

Figure 10.5: The `trace-defrel` form combining extension with host FFI interaction

For example, the `trace-defrel` macro in Figure 10.5 provides a quick way for the programmer to see the values of a relation's arguments at every entry to that relation. Users of `trace-defrel` need not even be aware that it uses cross-language code, because it is all is hidden behind the abstraction. This particular macro is defined with Racket's standard `define-syntax` rather than `syntax-spec`'s `define-dsl-syntax` because relation definitions sit at the boundary between miniKanren and Racket.

As previewed in Section 2.2, database access provides a substantial application of the combination of syntactic extension and host interoperation. Recall that Figure 2.6 re-implements the `direct` flights relation using an extension that connects miniKanren to an SQLite database. We now have all the pieces we need to explain the extension's implementation in detail. The `define-facts-table` form is a standard Racket macro. It expands to code that uses the Racket `database` library to create and populate an SQLite database table; we elide its implementation. More interesting is the `query-facts` goal macro defined in Figure 10.6, which straddles the boundary between Racket and miniKanren.

```
(define-dsl-syntax query-facts goal-macro
  (syntax-parser
    [(_ table term ...)
     #'(GE (query-facts-rt table (list (ET term) ...)))]))
```

Figure 10.6: The `query-facts` goal macro combining extension with host FFI interaction.

The `query-facts` macro consumes a reference to a facts table and a sequence of miniKanren term expressions that should evaluate to either atomic miniKanren values or logic variables. The macro expands to a goal formed from a Racket expression that, at runtime, actually executes a database query. The expansion

```
;; Table (Listof TermVal) -> GoalVal
(define (query-facts-rt table terms)
  (define matching-rows (do-query table (map wildcardify terms)))
  (unify-query-results matching-rows terms))

;; TermVal -> (Or Atom Wildcard)
;; THROWS when term is instantiated to a non-atom
(define (wildcardify term) #| elided |# )

;; Table (Listof (Or Atom Wildcard)) -> (Listof (Listof Atom))
(define (do-query table args) #| elided |# )

;; (Listof (Listof Atom)) (Listof TermVal) -> GoalVal
(define (unify-query-results query-res args)
  (match query-res
    ['() (EG fail)]
    [(cons fst rst)
     (EG (conde
           [(== fst args)]
           [(GE (unify-query-results rst args))]))]))
```

Figure 10.7: The runtime portion of the `query-facts` extension.

leverages the `ET` multi-language boundary form to check that the argument syntax `term` is valid term syntax and to convert the term value to a Racket value for use in the runtime helper.

The implementation of the runtime support for the extension is presented in Figure 10.7. The entry point `query-facts-rt` relies on three helper functions. First, it uses the `wildcardify` function to transform sealed logic variables into a "select all" wildcard that the database understands. Then, it executes the query with `do-query` to produce a list of matching table rows. Finally, it uses `unify-query-results` to non-deterministically unify the original term arguments with each possible option returned from the database. The `wildcardify` and `do-query` implementations are straightforward Racket functions using parts of the miniKanren runtime and Racket database library. The `unify-query-results` function has the most interesting multi-language interaction.

The `unify-query-results` function takes an arbitrarily long list of matching rows from the database lookup and produces a goal that non-deterministically unifies `args` with each of these values in turn. These unifications can fail because of delayed constraints (like `absento` from Figure 2.3) that cannot map directly to restrictions in the query. When the list of results is non-empty, the function returns a `conde` goal to implement that nondeterministic choice. In the second disjunct we make the recursive call to `unify-query-results`. This mixing of recursive Racket computation with goal construction relies on nested language boundaries.

To the miniKanren programmer using it in Figure 2.6, the `query-facts` extension looks like any other miniKanren form. Creating a DSL extension lets us hide the implementation details of complex cross-language operations behind simple, familiar looking syntax.

10.2 OPTIMIZING LIKE A DEEP EMBEDDING

Like in a deep embedding, our miniKanren compiler has access to a syntactic representation of DSL program fragments, and it can thus realize all kinds of optimizations. It uses a traditional multi-pass compiler architecture with a number of standard optimizations. The generated code uses the runtime system from `faster-miniKanren` (Ballantyne, 2024). The overall architecture provided by `syntax-spec` ensures that code generated by extensions benefits from optimizations, too. Most notably, our compiler works carefully around host-language code contained in miniKanren goals to optimize where possible while accounting for the host-language code's unknown behavior. In some sense, the details of the optimizations and the performance they yield are straightforward, but they do demonstrate that a `syntax-spec` DSL can be equipped with a standard compiler back-end. To underline how effective the compiler is, we show at the end of this section that our compiler produces substantial and even asymptotic performance improvements.

10.2.1 *Optimizations for miniKanren*

Following the nanopass approach (Keep and Dybvig, 2013; Sarkar, Waddell, and Dybvig, 2005), our compiler back-end consists of many small passes. We group them into four major steps to discuss their effects.

CONSTANT FOLDING The first major pass implements constant folding. It tracks statically-known equational information with a compile-time substitution data structure. The compiler uses information gained from earlier conjuncts to simplify subsequent ones. When unifications are statically guaranteed to succeed trivially or fail, the compiler simplifies them to just `succeed` or `fail` respectively. This pass also decomposes complex equations into conjunctions of simple "variable on the left" ones.

DEAD-CODE ELIMINATION Dead-code elimination requires several small passes. The first one removes any code dominated in control flow by `fail` (usually introduced by constant folding). The second pass finds equations that are statically known to succeed and whose execution does not further constrain the domains of external variables, and it replaces all such equations with `succeed`. The remaining passes simplify conjunctions with trivial `succeed`s and remove the bindings of unused logic variables from `fresh` goals.

UNIFICATION ANALYSIS Two further passes annotate unifications with information that allows the code generator to specialize code (Van Roy, 1994). The first of these two employs abstract interpretation to mark unifications for which it is safe to skip an occurs check (Søndergaard, 1986). An occurs check is generally required to forbid cyclic terms and ensure soundness of deductions (Marriott and Søndergaard, 1989), but it is expensive: the cost is linear in the size of the run-time terms being unified. Unification in miniKanren always includes the occurs check, but it is unnecessary when the compiler can statically determine that the equation does not introduce a cycle. The analysis correspondingly uses an abstract domain that records whether each variable is fresh, is known to refer to a limited set of other variables, or has a wholly unknown value. The second unification analysis pass marks the first reference to a newly introduced logic variable. This reference can be compiled efficiently because the specializer knows the variable is fresh.

SPECIALIZATION Normally, miniKanren performs unification via a runtime operation that recursively inspects the structure of two terms. Our code generator specializes unification to any syntactically evident structure. For example, in

```
(== x (cons first (cons second rest)))
```

the term on the right-hand-side always has at least two pairs. The unification procedure's dispatch can thus be unfolded and simplified for this portion of the match. The code generator also employs the annotations from unification analysis to generate calls to a version of unification without the occurs check when possible.

10.2.2 *Extensions Get Optimized Too*

DSL users who add syntactic extensions to this miniKanren implementation also benefit from the compiler pipeline, following Dybvig's "macro writer's bill of rights" (Dybvig, 2004). Dybvig proposes that compilers should guarantee to perform certain optimizations such as constant folding and dead-code elimination, allowing macro authors to write simple transformations that may sometimes introduce unnecessary indirections or add macro-generated dead code—without sacrificing performance. Through `syntax-spec`, our DSL's architecture allows us to offer DSL programmers these rights.

To show how extensions automatically benefit from our compiler pipeline, we step through an example. Below is `leo`, an inequality relation on Peano numerals adapted from the work of Rozplokhas and Boulytchev (2021):

```
(defrel/matche (leo x y)
  [('Z y)]
  [((cons 'S x1) (cons 'S y1)) (leo x1 y1)])
```

Our `leo` definition uses `defrel/matche`, which in turn uses `matche`. The simplistic implementation of `matche` from Section 10.1.1 introduces inefficiencies.

Figure 10.8 shows `leo` program after each of three major steps, using plain source code for readability.

```
(defrel (leo x y)
  (fresh (ls)
    (conj (== ls (cons x (cons y '())))
          (disj (fresh (y^) (== (cons 'Z (cons y^ '())) ls))
                (fresh (x1 y1)
                  (conj (== (cons (cons 'S x1) (cons (cons 'S y1) '())) ls)
                        (leo x1 y1)))))))

(defrel (leo x y)
  (fresh (ls)
    (conj (== ls (cons x (cons y '())))
          (disj (fresh (y^) (conj (== x 'Z) succeed))
                (fresh (x1 y1)
                  (conj (conj (== x (cons 'S x1))
                              (conj (== y (cons 'S y1)) succeed))
                        (leo x1 y1)))))))

(defrel (leo x y)
  (fresh ()
    (disj (fresh () (== x 'Z))
          (fresh (x1 y1)
            (conj (conj (== x (cons 'S x1)) (== y (cons 'S y1)))
                  (leo x1 y1))))))
```

Figure 10.8: The `leo` program after each of: expansion (top); constant folding (middle); and dead-code elimination (bottom). Underlines indicate the parts of the program that are changed in the next frame. In the first frame, they highlight the unifications that are simplified by constant folding. In the second, they show which portions of the program are removed by dead code elimination.

The macro expander desugars the Racket `defrel/matche` macro into a `defrel` with a `matche` in the body. The expander further desugars `matche` and all the miniKanren surface syntax forms together into our core language. These expansions result in a program with a number of unnecessary indirections, shown in the left-hand frame of Figure 10.8. The expansion introduces the intermediate variable `ls`, a list of all the relation's arguments. Each disjunct unifies `ls` with the term compiled from the pattern. Through this indirection, the generated code naively unifies lists of all arguments against entire patterns; this is wasteful when the terms are statically known to share structure.

Constant folding and dead code elimination address these inefficiencies. The middle frame of Figure 10.8 shows the result of constant folding. The pass eliminates two references to `ls` and also simplifies the remaining equations. Dead code elimination cleans up after the constant folding pass by removing trivial pieces. By the end of dead code elimination, shown in the right-hand frame, the compiler optimizes away all uses of the unnecessary variable `ls`. It retains `(fresh () ...)` nodes in the final frame of Figure 10.8 even after dead code elimination because

the `faster-minikanren` run-time system establishes interleaving points for its search at `fresh`. The compiler therefore keeps these nodes to achieve answer order equivalence with the existing implementation.

Keep et al., who first introduced `matche`, write that one of their primary aims is "to generate code that will perform at least as well as if the generated code had been written by a human" (Keep et al., 2009). Our simplistic implementation of `matche` from Section 10.1.1 initially seems to fall short of that aim, but macro-expanding to the input language of an optimizing compiler solves the problem with no extra effort on the behalf of the macro author. Our compiler removes `matche`'s unnecessary indirections whether the `matche` comes from the source program or through the expansion of another macro like `defrel/matche`. Without these optimization guarantees, the DSL extension programmer would have to inspect the entire macro stack to ensure that a new macro will generate performant code. An optimizing compiler for the core language relieves the extension programmer of this burden. In short, the benefits stack up as the layers of languages and extensions do.

10.2.3 *Optimizing at the Boundary with Racket*

Compiler correctness and performance also entail preserving certain properties of mixed miniKanren and Racket language programs. As mentioned in Section 10.1.2, we can understand the DSL-host language combination as a Matthews-Findler multi-language. When adding extensibility and a host-language interface, the key is to hit a "sweet spot" of adding the desired expressive power (Felleisen, 1991) without losing the ability to reason about the DSL program as something more than mere host language code.

Our multi-language hits such a sweet spot. It increases the expressive power of extensions without exposing internal implementation details that would prevent semantics-preserving optimization. Consider programs of the following shape:

```
(conj
  (fresh (x y)
    (== x y))
  (GE #| ... unknown racket code ... |#))
```

Our optimizer's constant propagation and dead code elimination transform the first conjunct into:

```
(fresh () succeed)
```

After all, no matter what Racket code is in the following goal, it cannot observe the fact that the optimizer has removed the allocation of those logic variables. Alternative host-interface designs could render such optimizations impossible. For example, if the Racket code were able to access the data structure storing the current values of all logic variables and enumerate them all, its behavior could change when our compiler removes otherwise-dead variables.

At the same time, the explicit boundaries between the two languages enable the optimization passes to rein in their transformations to account for the unknown behavior of the Racket code, as discussed in Section 2.3.2.

The `syntax-spec` framework facilitates hitting the aforementioned expressive power sweet spot by structuring the DSL definition as a multi-language where the DSL and host interact only at the specified boundary. The DSL has a separate grammar only connecting to the host at specified `host-interface` and `racket-expr` positions. Name bindings are similarly protected. DSL names like `term-variables` belong to separate binding classes. They may be used in Racket code only if a value translation is defined using `#:reference-compiler` (see Section 2.3.1). These choices ensure we can have the interactions that we want and prevent unexpected interaction. The multi-language structure could also provide the basis for formal reasoning about the correctness of our miniKanren compiler, along the lines of Perconti and Ahmed (2014).

10.2.4 *Benchmarks and Results*

To evaluate our optimizations' effectiveness, we assembled a benchmark suite and measured the speedup produced by each important category of compiler optimization. The point though is not to prove the effectiveness of our particular optimizations, but to show that the architecture enabled by `syntax-spec` accommodates an optimizing compiler.

10.2.4.1 *Benchmark Suite*

We assembled a benchmark suite from examples in several papers on pure relational programming in miniKanren.

OCCURS CHECK Rozplokhas and Boulytchev (2021) analyzed the asymptotic complexity of a variety of miniKanren programs. They note that the occurs check sometimes contributes substantially to the asymptotic cost. Because we expect that our optimizer can remove some of these checks, we adopt two of these programs as benchmarks. One uses the `leo` program introduced in Section 10.2.1 and searches for a Peano numeral greater than 8000. The second program uses a two-place `append` relation that connects a pair of lists with the result of appending the first to the second. This test runs the relation with a pair of two ground lists of 10,000 elements each.

ARITHMETIC The second set of benchmark programs exercise the miniKanren relational arithmetic suite introduced by Kiselyov et al. (2008). The first of these solves a difficult logarithm. The second program solves the `four-fours` puzzle (Ball, 1914) for 256, and the third searches for a number with a factorial of 720.

Table 10.1: Our compiler's performance results on a selection of miniKanren tests. The faster-mK benchmark column reports time in milliseconds; subsequent columns report speedup ratios over that column. Larger numbers report better speedups. Tests were run on a M1 Max Macbook Pro with 64GB RAM.

Benchmark	faster-mK	no opts	prop only	dead code	occurs check	overall
Occurs check						
`leo` 8000	209	1	1	1	52.25	69.67
`appendo` w/2 lists	327	0.99	1	1.01	81.75	109
Arithmetic						
`logo`	437	0.99	1	0.98	1.08	1.44
`four fours` of 256	77	0.95	0.97	0.94	1.01	1.26
`fact` x = 720	122	1.04	1.03	1.01	1.18	1.56
Interpreters						
one quine	962	0.99	0.96	0.97	1.01	1.01
9,900 `(I love you)`s	1378	0.96	0.95	0.98	1.01	1.04
`append` synthesis	252	1	1	0.98	1.33	1.81
dynamic and lexical	18	1	1.06	1	1.2	1.5
four thrines	683	0.98	1	1	1.03	1.06
countdown from 2	64	0.97	1	1	6.4	7.11

INTERPRETERS The third and largest suite of programs use relational interpreters to synthesize programs with specified behaviors. The first example replicates the inaugural application of relational interpreters presented by Byrd, Holk, and Friedman (2012): generating a Scheme quine. The next several are by Byrd et al. (2017). They include deriving 9900 expressions that evaluate to `(I love you)`, example-based synthesis of part of the standard `append` function, and synthesizing programs that evaluate differently under different semantics. One more program by Byrd et al. (2017) computes a thrine, a 3-cycle of different programs that evaluate to one another. The last program in this suite uses a relational interpreter to evaluate a large known program.

10.2.4.2 *Results*

Table 10.1 shows the performance improvements on our benchmark suite. To bring across what benefits each optimization realizes, we show the time to compute each program with the baseline `faster-miniKanren` implementation, and the speedup from our compiler with different configurations of optimization passes enabled. The header of each column indicates the additional optimization added in that configuration; each configuration includes the optimizations enabled in the

columns to its left. The final "overall" column thus includes the speedup provided by the final optimization pass, unification specialization.

The occur-check removal is the most helpful optimization. In addition to the examples where we expected to see improvement, we found it was also important for the relational interpreter benchmark with a large ground program ("countdown from 2 in λ-calc"). Unification specialization is usually helpful or harmless, but occasionally produces slowdowns by increasing the size of generated code. Constant propagation and dead-code elimination do not confer much benefit. Their performance is artificially limited by our desire to maintain answer order equivalence with `faster-miniKanren`, which prevents us from removing dead code that impacts search order. We validate the correctness of our compiler against a larger test suite, and on these tests we do achieve answer-order equivalence with `faster-miniKanren`.

11 PARSING EXPRESSION GRAMMARS

This chapter presents a second substantial DSL implementation in `syntax-spec`, of Parsing Expression Grammars (Ford, 2004). Parsing is a natural application domain for multi-language macros. Many programs need to embed a parser for a custom input format, and parsers typically need to embed host-language code that performs semantic actions. With multi-language macros, a parsing DSL can be fluidly integrated with the host language while also benefitting from static semantic checks and optimizations.

My PEG DSL implementation also illustrates particularly powerful forms of macro extensibility. The `syntax-spec` system lifts the Racket macro system's notions of local expansion and interposition points to apply to DSL expansion as well. Extensions to the PEG DSL take advantage of these features to layer a higher-level abstraction for simple parsing tasks atop the core DSL and customize the DSL to integrate it with a separately-developed lexer. Parts of this chapter are adapted from collaborative work with Alexis King (Ballantyne, King, and Felleisen, 2020).

11.1 PEGS AS A MULTI-LANGUAGE DSL

Ford (2004) introduced PEGs as an alternative to context free grammars that avoids ambiguity by relying on prioritized choice between alternatives. This section shows how my implementation of PEGs acts as a multi-language DSL integrated with Racket. It also discusses the static semantics and optimizations offered by its DSL compiler. Finally, it extends the PEG DSL via macros.

11.1.1 *PEG Syntax as an Extension to Racket*

Figure 11.1 specifies the grammar of the PEG DSL and its interface to Racket. Figure 11.2 demonstrates its use with a fragment of a PEG parser for Python.[1] The DSL can parse text from a string or tokens from a list. Here the example parses a list of tokens represented as Racket string and number values. The first line of the example imports the DSL library, making its syntax and runtime support available within the module.

Parsing requires recognizing elements from a token stream and constructing a syntax tree. In my PEG DSL the basic expressions recognize patterns of to-

1 This example is motivated by projects that implement Python atop Racket in order to take advantage of Racket's pedagogical tools (Ramos and Leitão, 2014), explore cross-language interoperability (Meunier and Silva, 2003), and investigate Python's semantics (Politz et al., 2013).

```
<racket-def> := ... | (define-peg [<nonterminal-id> <peg>] ...)
<racket-exp> := ... | (parse <nonterminal-id> <racket-exp>)
<terminal-literal> := <string> | (token <racket-exp>) | ...
<nonterminal-id> := <identifier>
<peg> := <nonterminal-id>                    nonterminal reference
       | ϵ                                   empty match
       | <terminal-literal>                  terminal
       | (seq <peg> ...+)                    sequence
       | (alt <peg> ...+)                    ordered choice
       | (* <peg>)                           zero or more
       | (! <peg>)                           negative lookahead
       | (: <racket-id> <peg>)               parse variable binding
       | (=> <peg> <racket-exp>)             semantic action
```

Figure 11.1: PEG DSL syntax

kens such as sequences (`seq`), alternatives (`alt`), and repetition (`*`). Semantic actions (`=>`) use Racket expressions in order to construct abstract syntax as Racket data. As illustrated in Figure 11.2, PEG non-terminals such as `arith-expr` are defined in Racket modules alongside Racket structures and functions such as `left-associate-binops`. Thus the syntax of Racket and the PEG DSL integrate in two ways: PEG non-terminal definitions live in Racket modules, and Racket expressions are embedded in PEG's semantic actions.

Name bindings are key to the interaction between the two languages. PEG non-terminal bindings may be used with `require` and `provide` of Racket's modules to import and export PEG non-terminals alongside other Racket bindings such as those for functions. *Parse variables* mediate between the parts of semantic actions: PEG binding expressions (`:`) in the parser part bind parse variables that can be referred to in the Racket action expression. These parse variables contain the result of the semantic action for the corresponding PEG subexpression, or if the binding occurs nested within repetition expressions (`*`), the variable contains lists of results nested to the same depth. For example, the parse variable `e*` in Figure 11.2 contains a list of semantic-action results from parsing all but the first subexpression of the arithmetic expression, a `term`.

Although the lexical syntax of the PEG DSL uses S-expressions, the DSL is not embedded in Racket's syntactic forms. Instead it adds the new syntactic category of PEG expressions, and extends Racket's definition and expression categories via the `define-peg` and `parse` interface macros, respectively. The two languages cannot be freely intermixed—writing a Racket expression where a PEG expression is expected leads to a compile-time error message describing the mistake. Similarly, PEG non-terminal bindings belong to a separate category from Racket variable bindings, and referring to a Racket variable in a PEG expression or vice versa leads to a compile-time error.

```
#lang racket
(require peg)

(struct binop-ast [lhs op rhs])

(define-peg
  [term #| elided #|]
  [arith-expr (=> (seq (: e1 term) (* (seq (: op* (alt "+" "-"))
                                           (: e* term))))
                  (left-associate-binops e1 op* e*))])

(define (left-associate-binops e1 op* e*)
  (foldl (lambda (op e base) (binop-ast base op e))
         e1 op* e*))

(parse arith-expr '(1 "+" 2 "-" 3))
;; evaluates to:
;; (binop-ast (binop-ast 1 "+" 2) "-" 3)
```

Figure 11.2: Fragment of a PEG parser for Python arithmetic expressions

11.1.2 *PEG Static Semantics*

The PEG DSL has a custom static semantics. Its DSL compiler rejects left-recursive non-terminals, because parsing with left-recursive PEGs may fail to terminate. Consider this alternate expression of the `arith-expr` non-terminal:

```
(define-peg
  [arith-expr-leftrec
   (alt term
        (=> (seq (: e1 arith-expr-leftrec)
                 (: op (alt "+" "-"))
                 (: e2 term))
            (binop-ast e1 op e2)))])
```

Parsing with this definition correctly accepts arithmetic expressions because the `term` alternative is attempted before the left recursion, but loops forever if the `term` alternative never matches. Without the static check it would be easy to write buggy code. Non-terminal definitions may be mutually recursive within a single `define-peg`, but not between separate definition groups. Checking for left recursion involves a fixed point computation across the mutually-recursive definitions. Because a parser may be composed of non-terminals defined in different modules, the left-recursion check must communicate information between separate module compilations.

Static checks are most useful to programmers when their IDE provides feedback as they type. Racket's macro system integrates tightly with its IDE, DrRacket, and this extends to multi-language DSLs. Here DrRacket uses the DSL's static

semantics to highlight the specific non-terminal references which create the left recursion, as soon as the definitions are complete.

11.1.3 *PEG Compilation and Optimization*

The PEG DSL's compiler performs optimizations. For example, scannerless parsers often include non-terminals that consist of a choice among fixed character sequences, as in this definition of Python comparison operators:

```
(define-peg
  [comp-op (alt "==" ">=" "<=" "<" ">" "!=" "in" "not" "is")])
```

Naive execution must check every alternative and backtrack as needed. A proper compiler for the PEG DSL can use binary search and a single backtrack point. Note that this optimization applies for my DSL only when parsing text, not tokens. As illustrated in Section 11.3.2, the DSL is designed to integrate with arbitrary token representations. These may lack the operations needed to support the binary search optimization. A variety of other optimizations have been proposed in previous work on PEGs (Grimm, 2004), and my approach can accommodate many of them. These optimizations rely on analyzing and transforming DSL syntax, so it is important that `syntax-spec` allows DSL compilers to assume a fixed core language.

11.1.4 *PEG Macros*

Because it is built with `syntax-spec`, the PEG DSL is naturally macro-extensible. Internally, the implementation uses macros to realize features that are abbreviations over terms of the core language. For example, the DSL includes a `(? e)` expression indicating an optional element of a sequence, and it expands to `(alt e` ϵ`)`.

PEG DSL users can also define macros to create syntactic sugar for commonly-seen patterns in their parser definitions. For example, the Python grammar includes many non-terminals such as `arith-expr` for binary and prefix operators, structured to encode operator precedence. Thiemann and Neubauer (2008) propose using grammar macros to simplify such definitions. The `binops` macro captures the pattern for parsing binary operators:

```
(define-dsl-syntax binops peg-macro
  (syntax-parser stx
    [(_ op-e subexpr-e)
     #'(=> (seq (: e1 subexpr-e) (* (seq (: op* op-e)
                                         (: e* subexpr-e))))
           (left-associate-binops e1 op* e*))]))
```

Using the `peg-macro` extension class in `define-dsl-syntax` indicates that the macro is intended to extend the PEG language. Using the macro in another context, such as a Racket expression position, results in a syntax error. Otherwise the

macro is defined in just the same way as a Racket macro: as compile-time Racket code using the `syntax-parse` DSL. PEG expressions generated by PEG macros can contain Racket subexpressions, and macro hygiene works for cross-language bindings such as `e1` and `e*`.

Together with a similar macro for prefix operators, the `binops` macro allows for concise specification of precedence hierarchies:

```
(define-peg
  [or-test
   (binops "or"
     (binops "and"
       (prefix "not"
         (binops comp-op
           expr))))])
```

The PEG DSL also provides a `local-expand-peg` procedure much like Racket's `local-expand` procedure, which allows PEG macros to reflectively expand and then operate on PEG syntax. Sophisticated extensions to the PEG DSL appear in Section 11.3.

11.2 IMPLEMENTING THE PEG DSL WITH syntax-spec

Figure 11.3 shows the essential pieces of the `syntax-spec` declaration for the PEG DSL. In Section 11.3.2 I elaborate on the declaration slightly to admit additional forms of macro extensions.

The declaration includes binding rules for parse variable and PEG nonterminal bindings. To allow the Racket code in semantic actions to refer to the parse variables but not mutate them, the declaration of the `parse-var` binding class uses `immutable-reference-compiler`. This reference compiler succeeds for references but raises a syntax error when the name is used in a `set!`. Unlike parse variables, PEG nonterminal names may not be directly referenced from Racket and are not declared with a reference compiler. This restriction allows the compilation of the DSL to include contract checks at only the `parse` entry point and not in the code generated for each nonterminal.

11.2.1 *Nesting Binding for Parse Variables*

The PEG DSL has an interesting binding structure: parse variables are visible only after their binding in sequences, and splice out of inner sequences and repetitions. For example, consider a PEG of the following shape, where `<P1>` and `<P2>` are some PEG expressions and `<E>` is some Racket expression:

```
(=> (seq (* (seq (: v1 <P1>) sp)) (: v2 <P2>))
    <E>)
```

```
(syntax-spec
  (binding-class parse-var
    #:reference-compiler immutable-reference-compiler)
  (binding-class peg-nt)
  (extension-class peg-macro)

  (nonterminal simple-peg #:allow-extension peg-macro
    ϵ (token e:racket-expr) (text t:text-expr)
    n:peg-nt (! p:peg) (alt p:peg ...+)
    (=> bp:binding-peg e:racket-expr) #:binding (nest bp e))

  (nonterminal text-expr
    c:char s:string e:racket-expr)

  (nonterminal/nesting binding-peg (nested)
    #:allow-extension peg-macro
    (: v:parse-var bp:binding-peg) #:binding (scope (bind v) nested)
    (seq bp:binding-peg ...+)      #:binding (nest bp ... nested)
    (* bp:binding-peg)             #:binding (nest bp nested)
    p:simple-peg)

  (nonterminal peg
    bp:binding-peg #:binding (nest bp []))

  (host-interface/definitions
   (define-peg [name:peg-nt p:peg] ...) #:binding [(export name) ...]
   (leftrec-check! (attribute name) (attribute p))
   #'(begin (define name (lambda (in) (compile-peg p in))) ...))

  (host-interface/expression
   (parse name:peg-nt in-e:racket-expr)
   #'(compile-parse name in-e)))
```

Figure 11.3: The declaration of the PEG core language in `syntax-spec`.

The parse variables `v1` and `v2` are both visible in the semantic action `<E>`. The value of `v1` contains a list of parsed values, one for each repetition. The parse variable `v1` is also in scope for any semantic actions nested in `<P2>`. On the other hand, `v2` is not in scope for nested semantic actions in `<P1>`, which run before the value for `v2` is available.

This binding structure is achieved using `syntax-spec`'s support for nesting binding structures. In `syntax-spec`, nonterminals declared via `nonterminal/nesting` have *nesting binding rules*. A nesting binding rule represents a fragment of scoping structure that has a hole. The hole is indicated with the `nested` keyword. When a nonterminal with nesting binding rules is referenced from another binding rule, the hole must be filled with another fragment of scoping structure. The `nest` binding rule syntax describes the hole filling, with the last subform given to `nest` filling the hole in the binding structure of the preceding subform. A nest rule with an ellipsis like `(nest pv ... bs)` indicates multiple hole fillings. The hole in the binding structure of each $\texttt{pv}_i$ is filled with the binding structure of $\texttt{pv}_{i+1}$, with the binding structure described by `bs` filling the most nested hole.

In Figure 11.3, the `(:)`, `(seq)` and `(*)` forms have nesting binding rules: each of these rules has a `nested` hole to fill. This nesting is what allows parse variable bindings to be visible later in a sequence and splice outside containing `seq` and $*$ forms. The scoping structures for subsequent parts of the sequence fill in the `nested` hole in earlier parts. Because nesting binding rules have a different shape than normal binding rules—they have a hole that must be filled—forms with nesting binding must appear in separate nonterminals. As such, the declaration in Figure 11.3 has a nesting nonterminal `binding-peg` for PEG forms that bind parse variables in scope of their `nested` hole, and a `simple-peg` nonterminal for PEG forms that do not allow bindings to escape in this way. The `peg` nonterminal wraps a `binding-peg`, filling its hole with an empty scoping structure to allow it to be used in non-nesting contexts.

11.2.2 *The Left-Recursion Check*

The implementation of the left-recursion check exercises two interesting aspects of `syntax-spec`: persistent symbol tables (Section 8.4), and an alternative host-interface declaration form, `host-interface/definitions`. These features are necessary in order to realize the check across module boundaries and for mutually-recursive PEGs.

Recall from Section 11.1.2 that a PEG nonterminal is left-recursive if a parse beginning with the nonterminal may re-enter the nonterminal without consuming any input. When this happens, parsing can follow this loop forever and fail to terminate. The formulation of the left recursion check relies on the additional concept of nullability. A nullable PEG expression is one that may succeed without consuming any input. The simplest such PEG is the primitive ϵ form, which consumes no input and succeeds. A PEG expression which parses a potentially

```
(define expanded-defs (local-symbol-table))
(define entered (local-symbol-table))
(define-persistent-symbol-table def-nullable?)

;; (ListOf Identifier), (ListOf PEGSyntax) -> Void
;; THROWS a syntax error when left recursion is detected.
;; EFFECT records nullability in the persistent def-nullable? table.
(define (leftrec-check! names pegs)
  (for ([name names] [rhs pegs])
    (symbol-table-set! expanded-defs name rhs))
  (for ([name names])
    (nullable-nonterminal? name)))

;; PEGNonterminalID -> Boolean
(define (nullable-nonterminal? id)
  (case (or (symbol-table-ref def-nullable? id #f)
            (symbol-table-ref entered id #f)
            'unvisited)
    [(nullable) #t]
    [(not-nullable) #f]
    [(entered)
     (raise-syntax-error #f "left recursion through nonterminal" id)]
    [(unvisited)
     (symbol-table-set! entered id 'entered)
     (define rhs (symbol-table-ref expanded-defs id))
     (define res (nullable? rhs))
     (symbol-table-set! def-nullable? id
                        (if res 'nullable 'not-nullable))
     res]))

;; PEGSyntax -> Boolean
(define (nullable? stx) #| elided |#)
```

Figure 11.4: The implementation of the left-recursion check using symbol tables.

empty sequence of any number of digits is also nullable because it can succeed while parsing zero digits. Left recursion depends on nullability in the case of sequences. Consider checking for left recursion in the following definition:

```
(define-peg
  [nt1 #| elided |#]
  [nt2 (seq nt1 nt2)])
```

The PEG nonterminal `nt2` is left recursive if and only if the PEG nonterminal `nt1` is nullable.

Figure 11.4 presents the compile-time code that implements the left-recursion check, which is invoked in the definition of `define-peg` from Figure 11.3. The `leftrec-check!` entry point checks for left recursion in the mutually-recursive

group of PEG nonterminals defined by a single `define-peg` form. In the case where these definitions refer to other PEG nonterminals outside of the mutually recursive group, the check relies on nullability information already being recorded in the global `def-nullable?` table. As the check is performed, it also records such nullability information for use when checking later definitions. The check uses two other tables that do not need to persist across separate compilations. The `expanded-defs` table maps PEG nonterminal names to PEG expression syntax to allow the check to continue when it reaches a nonterminal reference. The `entered` table records which nonterminals have already been traversed, in order to detect left-recursion when traversing a cycle.

The left recursion check leverages two important features of `syntax-spec`'s persistent symbol tables with respect to the `def-nullable?` table. First, entries in the table persist across separate compilation boundaries. Thus, when a nonterminal is defined in one module and referenced in another, the left-recursion check for the referencing nonterminal can access the nullability information stored in the previous module expansion. Second, extending the table is accomplished via a side effect. This means that the persistent entries can be easily created within the same computation that traverses the mutually-recursive definitions. This stands in contrast to the more complex pattern of programming required with Racket's lower-level tools; see Section 8.4.

Because the left-recursion check needs to traverse the graph implied by the references in a mutually-recursive group, it is possible to perform this check only once the entire group has been fully macro-expanded. This raises a difficulty with respect to `syntax-spec`'s integration with the passes of expansion in Racket's definition contexts. Given that Racket's definition contexts feature only two passes of expansion, it is not possible for `syntax-spec` to both:

1. Allow mutual-recursion between separate host-interface forms.

2. Allow the DSL compiler to simultaneously examine all the fully-expanded definition right-hand sides.

Supporting both features would require three passes of expansion overall: one to bind the left-hand-sides, one to expand the right-hand-sides, and one to run the DSL compiler. Instead, `syntax-spec` provides two different host-interface declaration forms to allow programmers to make the trade-off. The `syntax-spec` for miniKanren in Figure 2.2 uses the `host-interface/definition` form to allow mutual recursion between separate definitions. This comes with the trade-off that the miniKanren DSL compiler must process each `defrel` body in isolation. The resulting expansion process is described in Section 8.3. The PEG declaration in Figure 11.3 instead uses the `host-interface/definitions` form. With this kind of host-interface, the two passes of `syntax-spec` expansion both occur within the first pass of Racket definition-context expansion. This allows the PEG DSL compiler to simultaneously access the fully-expanded right-hand-sides of all the

definitions in the group to perform the left-recursion check. However, references to PEG nonterminals defined by a separate `define-peg` further down in the module result in an error.

11.2.3 *Limitations of `syntax-spec`*

Development of the PEG DSL exposed two limitations of `syntax-spec`. First, `syntax-spec`'s binding rules cannot express one portion of my intended design for PEG. It would be easier to construct semantic actions for alternatives if the pattern variables from both alternative branches in an `(alt p ...)` PEG were bound in the semantic action. The values for variables coming from the branch of the alternative that was not evaluated in a given parse would be `#f`. With this scoping rule, the scope for a semantic action would have multiple parents, one for each alternative. Such DAG-shaped rather than tree-shaped scoping structures cannot yet be expressed with my binding-rule language.

Second, the PEG DSL's compiler would benefit from an additional name analysis that `syntax-spec` does not yet provide. The compilation of the zero-or-more PEG expression `(* p)` requires computing a set of parse-variable names that should be bound to lists of results for the repeated match. While `syntax-spec` provides services for computing the free- and bound- identifiers of a term, the set of names needed here is difficult to define. The set should include the identifiers that are bound as pattern variables within the $*$ form, but not those whose scope is limited by further-nested semantic actions `(=>)`. This set can also be described as those names that are bound in the term and placed in scope for the syntax that fills the `nested` hole in the binding structure of the term. The syntax-spec implementation does not yet provide a service for computing this set of names, so the PEG compiler computes it manually.

11.3 THE POWER THAT COMES WITH EXTENSIBLE DSLS

Macros enable powerful extensions to DSLs that go beyond syntactic sugar. They enable DSL programmers to create towers of DSLs to allow each program component to be written at the right level of abstraction. They also allow programmers to customize the syntax of a DSL to smoothly integrate with other components of their software system (Felleisen et al., 2018). This section presents extensions to the PEG DSL that exemplify these ideas.

11.3.1 *Layering DSLs*

When designing a DSL, language authors must make a choice along a spectrum of specialization. A specialized DSL allows the most direct expression of programs within its narrow purview. A general DSL demands more programming effort in

exchange for flexibility. Luckily there is a way to escape the trade-off through towers of multiple DSLs, each at a different degree of specialization. The most specialized languages are at the top of the tower, and they are implemented by translation to the layers below. This arrangement allows specialized DSLs to take advantage of the shared languages at the lower layers (Andersen, Chang, and Felleisen, 2017; Ward, 1994).

Extensible DSLs enable another way of using towers of DSLs: programs can combine components written in DSLs belonging to different layers of the tower to achieve a mixture of concision and custom behavior. Higher-level DSLs are implemented as macro extensions of the base DSLs, just as the base DSLs are implemented as extensions to Racket, as illustrated in Section 11.1. The result is that the syntax and static semantics of DSLs at different layers of the tower are tightly integrated and programs may mix languages easily.

Parsing tools reflect a spectrum of specialization regarding abstract syntax tree construction. Some tools automatically construct syntax trees based on the structure of the grammar, whereas others leave the parser programmer to build syntax trees using semantic actions. My PEG DSL uses the second approach so that parsers can construct syntax trees corresponding to the structure of the grammar even when left factoring requires a reorganization. For example, the `arith-expr` semantic action constructs a left-associated syntax tree.

In other cases, however, the PEG closely matches the structure of the parsed language. Consider the task of parsing Python's `raise` form. With the PEG DSL design, we separately define a structure to represent the abstract syntax and a PEG to parse the concrete syntax:

```
;; a structure with a super type, `ast`
(struct raise-ast ast [exn from])
(define-peg
  [raise (=> (:src-span srcloc
               (seq "raise" (? (seq (: exn test)
                                    (? (seq "from" (: from test)))))))
             (raise-ast srcloc exn from))])
```

This implementation also captures a source location and records it in the abstract syntax, relying on an additional `:src-span` PEG form omitted from in Figure 11.3. The `:src-span` PEG form captures a source location description spanning the first and last tokens parsed by the interior PEG. Notice that for the `raise` parser and structure definition, the parse variable bindings in the PEG correspond exactly to the fields of the structure used for the abstract syntax tree: `exn` and `from`. Other statements in the Python grammar such as `return` and `assert` exhibit a similar correspondence. This pattern suggests an opportunity to abstract.

Parsing these forms is more convenient with a DSL that automatically constructs syntax trees based on the parse variable bindings in the grammar. Such a specialized DSL is easily realized via a `peg-macro` atop the core PEG DSL. With the extension, the `raise` parser becomes:

```
(define-peg-ast raise raise-ast
  (seq "raise" (? (seq (: exn test) (? (seq "from" (: from test)))))))
```

This defines both the `raise-ast` structure with `exn` and `from` fields, and the `raise` PEG non-terminal. Source locations propagate into the AST automatically. A complete Python parser can use a mixture of `define-peg-ast` for syntactically simple non-terminals and `define-peg` for non-terminals that require specialized processing.

Implementing the Extension

Figure 11.5 shows the definition of the `define-peg-ast` macro. It simultaneously defines a structure type to represent abstract syntax (`ast-name`) and a PEG parser non-terminal for parsing the concrete syntax (`peg-name`). The field names for the structure type are inferred from parse variable bindings in the PEG expression, such as `exn` and `from` in the example above.

```
(struct ast [srcloc])
(define-syntax define-peg-ast
  (lambda (stx)
    (syntax-parse stx
      [(_ peg-name:id ast-name:id p:peg)
       (define/syntax-parse
         (var ...)
         (find-parse-var-bindings (local-expand-peg #'p)))
       #'(begin
           (struct ast-name ast [var ...])
           (define-peg
             [peg-name (=> (:src-span srcloc p)
                           (ast-name srcloc var ...))]))])))
```

Figure 11.5: Implementation of `define-peg-ast`

To implement this behavior, the macro transformer must discover the variable bindings within the given PEG expression `p`. It cannot analyze the syntax directly because `p` may contain macro uses, so it invokes `local-expend-peg` to obtain expanded syntax that uses only the core language. It can then use a procedure `find-parse-var-bindings` (not shown) to walk the expanded core AST and build a list of binding variable names. In the template, the names are used as the field names of a `struct` definition and as parse variable references in the semantic action.

The `local-expand-peg` procedure is implemented by the core PEG DSL. Its implementation is trivial: it simply dispatches the task to `syntax-spec`:

```
(define (local-expand-peg stx)
  ((nonterminal-expander peg) stx))
```

The `nonterminal-expander` form accesses the macro expander for a given `syntax-spec` nonterminal.

11.3.2 *Integrating with Other Components*

Any given DSL addresses only one domain of the many involved in creating a large, complex software system. It is thus essential that each DSL provide programmers with tools to integrate it with other program components. Using macros, programmers can customize the syntaxes of DSLs to integrate with other parts of their program.

Consider the integration of the PEG-based Python parser with a Python lexer written in plain Racket. To allow integration with any lexer, the PEG DSL does not fix a specific token representation. Instead, programmers provide a Racket predicate which the parser uses to recognize tokens. For example, my Python lexer represents keyword tokens as structures:

```
(struct keyword-token [name])
```

To integrate the PEG DSL with this token representation, we first write a function that generates predicates matching `keyword-token` structs whose `name` field is equal to a given value:

```
;; (-> string? (-> any/c boolean?))
(define (keyword expected-name)
  (lambda (t)
    (and (keyword-token? t)
         (equal? (keyword-token-name t) expected-name))))
```

The PEG DSL provides a `(token <racket-exp>)` syntax for parsing tokens. The Racket expression argument provides the predicate the parser uses to recognize tokens. Thus the following PEG expression matches Python `return` statements:

```
(seq (token (keyword "return")) (? (: exp testlist-star-expr)))
```

This integration via Racket procedures is effective, but also syntactically verbose.

Macros permit customization of the syntax of the PEG DSL to improve integration with the Python lexer in two ways. First, we can repurpose string literal syntax to match keywords concisely:

```
(seq "return" (? (: exp testlist-star-expr)))
```

Second, we can raise compile-time syntax errors for misspelled keywords by checking against a list of Python keyword names.

Implementing the Extension

Figure 11.6 shows the implementation of a PEG macro that adds these features to the DSL. The macro takes advantage of an *interposition point* (Culpepper et al.,

2019) to change the meaning of string literal syntax. These special extensibility points allow macros to change the meaning of elements of DSL syntax that lack explicit names, such as function application or literal syntax for datums such as strings and numbers. The PEG DSL offers the `#%peg-datum` interposition point to allow users to change how literal datums are interpreted.

```
(require (for-syntax python-lexer))

(define-dsl-syntax #%peg-datum peg-macro
  (syntax-parser stx
    [(_ s:string)
     (unless (member (syntax-e #'s) python-keyword-list)
       (raise-syntax-error #f "Invalid keyword token" #'s))
     #'(token (keyword 's))]))
```

Figure 11.6: Changing the meaning of string literals in PEG expressions to match keyword tokens

Providing a custom behavior for an interposition point is accomplished by defining a macro using the interposition point's name. The new implementation of the behavior is used wherever the macro is in scope. The new `#%peg-datum` first checks that the string corresponds to an actual Python keyword name. Then it expands to a use of the `token` PEG expression syntax with a token-matching function constructed by the `keyword` function. In order to identify valid Python keywords, the macro uses a `python-keyword-list` provided by the lexer.

```
(syntax-spec
  #| elided declarations from the original spec |#

  (nonterminal simple-peg #:allow-extension peg-macro
    #| elided cases from the original nonterminal |#

    (~> (~or lit:string lit:char lit:number lit:regexp)
        #:with #%peg-datum (datum->syntax #'lit '#%peg-datum)
        #'(#%peg-datum lit))))

(define-dsl-syntax #%peg-datum peg-macro
  (syntax-parser
    [(_ (~or* lit:char lit:string)) #'(text lit)]))
```

Figure 11.7: The implementation of the `#%peg-datum` interposition point in the PEG DSL's `syntax-spec` declaration, modifying Figure 11.3.

This extension is possible because the design of the core PEG DSL anticipates the need for this form of extension. Figure 11.7 shows how the interposition point is implemented via a modification of the `syntax-spec` declaration from Figure 11.3. The implementation consists of a rewrite production in the `simple-peg` nonter-

minal declaration as well as a `#%peg-datum` macro defining the default behavior for the interposition point. A rewrite production, signaled by a `(~>)`, augments the parsing process for the nonterminal with a syntax rewriting. The rewriting can use all the features of `syntax-parse` to match and reconstruct syntax. Here, the rewriting matches on string, character, number, and regular expression literals and wraps them with a call to a `#%peg-datum` macro.

To make `#%peg-datum` act as an interposition point, the rewriting bends macro hygiene with `datum->syntax`. Normally, hygiene would fix the reference to refer to the default definition of `#%peg-datum` present at the definition-site of the rewriting. The use of `datum->syntax` copies the lexical context from the syntax of the character, string, number, or regexp literal in order to refer to whatever definition of `#%peg-datum` is in scope in the lexical context where the literal is written. This may be the default `#%peg-datum` if the user program imports this definition from the core PEG implementation. Alternately it may be a local, overriding definition such as that provided in Figure 11.6 to integrate with the Python lexer. The default `#%peg-datum` implementation interprets character and string literals as string fragments to match textually with the `text` peg form.

12 A PLETHORA OF DSLS

While miniKanren and PEG parsers represent my most substantial DSL implementations in `syntax-spec`, I have also created a collection of smaller DSL implementations. These include re-implementations of (subsets of) well-known DSLs such as `racket/class` and `racket/cmdline` from Racket's standard library, as well as DSLs from the broader literature. I also used `syntax-spec` in my work on multi-stage miniKanren. The `syntax-spec` design makes expressivity tradeoffs in order to keep declarations simple and enable hygienic macro extensibility (discussed in Section 3.2 and Chapter 9). The array of implementations presented in this chapter together with those by other authors in Chapter 13 help establish that `syntax-spec` is expressive enough to conveniently implement many valuable DSLs. At the same time, the DSLs also illustrate some limitations imposed by the binding specification language.

12.1 STATE MACHINES

Taking inspiration from statecharts (Harel, 1987), the state machine DSL enables programmers to express state-machine patterns directly. Here is a simple program in the DSL, modeling a subway turnstile:

```
(define (turnstile fare)
  (machine #:initial locked
    (data accumulated-value_e 0)
    (state unlocked_e
      (on (person-passes-through)
        (emit 'lock) (goto locked)))
    (state locked_e
      (on (coin value_s) #:when (>= (+ value accumulated-fare) fare)
        (set accumulated-value 0) (emit 'unlock) (goto unlocked))
      (on (coin value_s) #:when (< (+ value accumulated-fare) fare)
        (set accumulated-value (+ value accumulated-value))
        (goto locked)))))
```

The program consists of states, event-triggered transitions, and computations on data performed via Racket subexpressions.

BINDING STRUCTURE Data and state names use exporting binding structure (underlined with a subscript `e`) within the `machine` body. Transition arguments have a simple binding structure (underlined with subscript `s`). Racket expressions in guards, setters, and emit expressions can refer to data variables and event arguments.

USE OF `syntax-spec` SERVICES The DSL protects state names from use by Racket expressions via a binding class, ensuring that all state transitions can be easily found by static analysis. This restriction is useful because the DSL compiler performs a static check to ensure that all states are reachable from the initial state (ignoring `#:when` clause restrictions). It uses symbol tables to connect the definition of state names to their references in `goto`s.

12.2 CLASSES

The class DSL reimplements the essential subset of Racket's `class` macros (Flatt, Findler, and Felleisen, 2006), which add object-oriented capabilities to the language's functional core.[1] The following example is a tiny excerpt from a Racket GUI library:

```
(define canvas%
  (class object% (super-new)
    (public edit-sequence)
    (define/match (edit-sequence_e . expr_s)
      ['() (send this end-edit-sequence)]
      [(cons first-edit-step other-steps)
       (send this step-edit-sequence first-edit-step)
       (send this edit-sequence other-steps)])))
```

Class features are expressed as Racket forms that are reinterpreted in the `class` context. In particular, function definitions are reinterpreted as method definitions. The class language is thus extensible with standard Racket macros such as `define/match`, a pattern-matching function definition facility; its expansion results in a reinterpreted function definition.

The class DSL represents precisely the form of extension that Racket's "macros that work together" features (Flatt et al., 2012) are designed for. That is, the DSL extends the syntax of Racket definitions and expressions within the `class` context, rather than creating an entirely new sub-language. I found that `syntax-spec` also works for this kind of extension and results in a concise implementation thanks to its declarative nature.

BINDING STRUCTURE The class DSL uses exporting binding structure for method and field names (no fields shown) and simple binding for method arguments. Macro definitions are allowed in all Racket definition contexts, including in the body of classes. The declaration of the class DSL uses the `export-syntaxes` `syntax-spec` binding rule form to implement bindings of macro definitions (documented in Appendix A.2.1.5).

USE OF `syntax-spec` SERVICES Via `syntax-spec`'s macro extensibility features, the class DSL is able to re-use many extensions to Racket's defi-

1 Thanks to Michael Delmonaco for his collaboration in this re-implementation.

nition syntax, such as `define/match`. The definition of the class DSL uses the `racket-macro` extension class, built-in to `syntax-spec`, to indicate that macros intended for Racket contexts are usable in the class context as well.

12.3 COMMAND-LINE ARGUMENT PARSING

The `cmdline` DSL (Ballantyne, King, and Felleisen, 2020) provides a concise syntax for parsing command-line arguments to programs and scripts. It is inspired by the `racket/cmdline` DSL in the Racket standard library (Flatt and PLT, 2010), For example, here is the parser for a compiler executable that supports optimization and output flags:

```
(define/command-line-options
  #:options
  [optimize-level_e
    (choice/o #:default 0
      (numbered-flags/f "--o" [0 3] "optimization level"))]
  [output_e (required/o "-o" outfile_s "the output filename" outfile)]
  #:arguments [file-to-compile_e existing-file/p])
```

The definition above parses these flags and defines module-level names with the provided values. The DSL's operation as a definition form deviates from the original `racket/cmdline`, which requires imperative programming to capture the values of arguments. In addition to the argument parsing behavior, the specification is also used to automatically generate a usage message that displays when the `--help` flag is supplied.

BINDING STRUCTURE The command-line option values are bound in the surrounding definition context thanks to exporting binding and a definition host-interface form. Flag arguments are simple local bindings.

USE OF `syntax-spec` SERVICES The cmdline language is macro extensible. For example, the `numbered-flags/f` macro expands into a set of flag definitions, one for each number in a range.

12.4 TINYHDL

The TinyHDL DSL (Savaton, 2021) is a hardware description language for experimenting with some concepts from Verilog and VHDL. Here is a half-adder in my `syntax-spec` re-implementation:

```
(begin-tiny-hdl
  (entity half-adder_e
    ([input a] [input b] [output sum] [output carry]))
  (architecture half-adder-arch_e half-adder
    (assign sum   (xor a b))
    (assign carry (and a b))))
```

Entity definitions describe the interface of a hardware component in terms of input and output ports; an architecture implements an entity.

BINDING STRUCTURE Unfortunately the binding rules for port names are not expressible in the `syntax-spec` binding language. The port names that are in scope in an architecture depend on the entity that the architecture implements. Checking these references would require first resolving the entity reference in the architecture and then consulting some associated scope to find the valid port names. The "scopes as types" approach of Statix (Antwerpen et al., 2018), for example, supports modeling such references. The `syntax-spec` binding language cannot yet express such dependent references. Instead, my implementation of the DSL treats the port names as uninterpreted identifiers in the binding language and uses a separate static checking pass to ensure they are valid.

USE OF `syntax-spec` SERVICES TinyHDL names are protected from use in Racket expressions. The DSL compiler uses persistent symbol tables to check the dependent binding structure for ports.

12.5 MULTI-STAGE MINIKANREN

Multi-stage miniKanren (Ballantyne et al., 2025) extends miniKanren with ideas from multi-stage languages like MetaML (Taha, 1999). This makes it possible to write relations whose execution is separated into staging-time and run-time parts. The result of staging-time execution is a specialized program retaining the run-time elements. When some arguments to a relation (or parts thereof) are statically known, specializing to those arguments with staging can substantially improve run-time performance. This is especially effective when the relation is an interpreter and staging removes interpretive overhead.

Figure 12.1 presents a small interpreter written as a multi-stage miniKanren program together with a query that is specialized by staged execution. The `evalo` relation interprets boolean literals and `or` expressions. The query fills a hole in a boolean expression sketch. Staging specializes the interpreter to the sketch. Each query result includes a synthesized expression filling the hole and the value to which the overall expression evaluates with the completion.

BINDING STRUCTURE Multi-stage miniKanren has the same kinds of binding structures as the miniKanren `syntax-spec` declaration of Figure 2.2.

```
(defrel/staged (evalo_e e_s v_s)
  (fallback
   (conde
     ((booleano e) (== e v))
     ((fresh (e1_s e2_s v1_s)
        (== e `(or ,e1 ,e2))
        (evalo e1 v1)
        (gather
         (conde
           [(== v1 #f)
            (evalo e2 v)]
           [(=/= v1 #f)
            (== v1 v)])))))))
```

```
(run 4 (e_s v_s)
  (staged
    (evalo `(or ,e #f) v)))
⇓ staging specializes the query to:
(run 4 (e v)
  (fresh (e1 e2 v1)
    (evalo/fallback1 e1 v1)
    (conde
      [(== v1 #f) (== v #f)]
      [(=/= v1 #f) (== v v1)])))
↪ the query evaluates to the results:
'((#t #t)
  (#f #f)
  ((or #t _.0) #t)
  ((or #f #t) #t))
```

Figure 12.1: A multi-stage miniKanren relation (left), and query with specialization and execution (right).

USE OF `syntax-spec` SERVICES Using `syntax-spec` makes it easy for the multi-stage miniKanren implementation to provide two interpretations of the same core language. Every staged relation is compiled both as staging-time code and as fallback, runtime code. Because miniKanren allows holes anywhere, even values that are expected to be available at staging time may in fact be missing data. In such cases, the specialized code calls the fallback version of the relation to compute that portion.

Staging introduces additional static semantics because there is a distinction between staged relations and relations that are only available at run time. The multi-stage miniKanren implementation uses persistent symbol tables to implement the check to ensure relations are called in the appropriate stage.

13 INITIAL COMMUNITY ADOPTION

The `syntax-spec` system is publically available, and a number of DSLs using the system have been created by other researchers and members of the Racket open-source community. I also taught a course on DSL design and implementation in Spring 2025 at Northeastern University titled "Hack Your Own Language", and some students created DSLs using `syntax-spec`. I provided technical support for each of these projects' use of `syntax-spec`, but the designs and implementations of these DSLs are the product of their authors. As such, they demonstrate that `syntax-spec` can support DSLs beyond those that I as the creator of `syntax-spec` anticipated as ideal fits for the metalanguage. These developments have also led to improvements to `syntax-spec` as well as identified shortcomings that would be worth considering in the design of future similar metalanguages.

13.1 DOMAIN-SPECIFIC CONTRACT LANGUAGES

Moy, Jung, and Felleisen (2025) present a collection of domain-specific contract languages that implement logics used as part of software contracts. Trace contracts (Moy and Felleisen, 2023) capture a history of events, often tied to specific objects, and signal a contract violation if a property of the trace is refuted. Different kinds of properties over traces are best expressed using different formalisms, so Moy, Jung, and Felleisen provide a collection of logic DSLs for use with trace contracts. Some of these implementations take advantage of `syntax-spec` in order to implement static semantics, compilation, and macro extensibility.

For example, the following specification describes the expected behavior of a component that manages TCP connections:

```
(define manager-qea
  (qea
    (∀ port)
    (start ready)
    [-> ready '(listen ,port) listening]
    [-> listening '(close-listener ,port) ready]))
```

The specification uses a Quantified Event Automata, or QEA (Barringer et al., 2012). A QEA specifies a family of state machines, quantified over variables that may appear in transition events. In this case, the quantifier ensures that a separate machine tracks the state of each TCP port.

A second domain-specific contract language that makes interesting use of `syntax-spec` implements Past-time Linear Temporal Logic, or PLTL (Lichten-

stein, Pnueli, and Zuck, 1985). The following formula, adapted from Havelund et al. (2018), concerns a property of Java map, collection and iterator objects:

```
(define-pltl unsafe-map-iterator-violation
  (∃ (m c i)
     (∧ (next i)
        (♦ (S (update m)
              (∧ (iterator c i)
                 (♦ (create m c))))))))
```

It is formulated to describe the situation when the property is violated: When a collection (`c`) is created from a map (`m`) and an iterator (`i`) is created from that collection, and the underlying map is subsequently modified, it is an error to continue to use the iterator. The statement describes this situation using PLTL's "once" (♦) and "since" (S) modalities. The existential quantification ensures a violation is identified only when the recorded events concern related objects.

BINDING STRUCTURE The QEA and PLTL DSLs both use bindings for logical quantifiers. The `define-pltl` form used in the example above is implemented with a separate PLTL formula binding class and a host-interface form to allow abstraction within PLTL formulas.

USE OF `syntax-spec` SERVICES Only a subset of syntactically valid PLTL formulas can be handled by the DSL's monitoring algorithm, so it implements a static check to restrict programs to this subset. This static check uses `syntax-spec`'s operation for computing the free variables of formulas. Similarly, the compiler for QEAs computes the variables mentioned in patterns. Some forms of PLTL are defined as syntactic sugar with macros.

13.2 QI

Kasivajhula (2021)'s Qi DSL equips Racket with a point-free sublanguage. For example, the following code computes the root mean square of a list of numbers:

```
;; (-> (listof number?) number?)
(define-flow root-mean-square
  (-< (~> length (as l_n))
      (~> Δ (>< sqr) + (/ l) sqrt)))
```

Each expression in the language implicitly accepts and returns some number of values, and the control-flow forms determine how those values flow between expressions. The initial list is split along two paths by the `-<` "tee" junction form. The threading form `~>` passes a value from each flow to the next.

Thus the first path computes the length of the list, while the second path squares each element, sums them, divides the sum by the previously computed length, and finally takes the square root. The `as` form binds a variable to the value that flows

into it. Such bindings are useful to convey information that does not follow the linear flow of threading.

Qi was initially implemented using conventional Racket macros, without support for macro-extensibility or binding structure. Kasivajhula worked with me to port Qi to `syntax-spec` and add these features. Qi now leverages the macro system to desugar the DSL's large surface syntax into a smaller core language. Further work with Kasivajhula together with Dominik Pantůček and other Qi contributors has taken advantage of the new architecture with `syntax-spec` to augment the implementation with compiler optimizations including deforestation.

BINDING STRUCTURE Bindings introduced by the `as` form use nested binding structure in order to match static scope to the runtime availability of values: only references later in the flow may refer to the binding. The nested binding position in the example above is annotated with a subscript `n`.

Certain forms in Qi that express branching and joining computations DSL could benefit from DAG-structured rather than tree-structured scoping structure. This would allow variables bound in both branches to be visible in the computation that follows the join. Unfortunately, as discussed in the context of the PEG DSL in Section 11.2.3, `syntax-spec` cannot express DAG-structured scope.

USE OF `syntax-spec` SERVICES The Qi DSL compiler lifts all `as` bindings to the top-level of the flow and inserts assignment expressions in their original positions. This strategy allows the compiler to otherwise remain a simple recursive traversal of the syntax. The alphatisation provided by `syntax-spec` automatically ensures that this transformation does not make the bindings visible in earlier portions of the flow. Qi also leverages macro extensibility. Many of the forms in the language are defined as macros on top of a core language.

13.3 LENS MATCH

Delmonaco (2022) introduces a lens-match DSL that supplements an implementation of functional lenses (Foster et al., 2007) with a pattern-matching language. Pattern variables are bound to lenses, which may be used to modify the corresponding path in the matched data. For example, a function that shifts the x-coordinate of a circle's Cartesian center point to the left is defined as follows:

```
;; (-> Circle Circle)
(define (shift-left circle)
  (update circle
    [(list color_e radius_e (cons x_e y_e))
     (modify! x sub1)]))

(shift-left (circle 'red 5 (cons 1 2)))
;; => (circle 'red 5 (cons 0 2))
```

The body of a match clause is conceptually a monadic context, where `modify!` updates the contextual value in a functional manner.

BINDING STRUCTURE The pattern variable production must come with a binding rule that uses definition context binding structure. For example, the `x` binding inside the `cons` pattern form must escape the form in order to be visible in the `modify!` expression.

Unfortunately, exporting binding rules produce a surprising behavior in `update` patterns. If a pattern variable's name `fn` appears as the function name in a guard pattern `(? fn)`, like this:

```
(update (list 1 2)
  [(list number? (? number?)) (modify! number? add1)])
```

then the first occurrence shadows the second one. Ideally, the binding of `number?` as a pattern variable should be in scope only in the body of the `update` clause. Using exporting binding rules, however, the pattern variable binding also captures the reference in the guard pattern, which is really intended to refer to Racket's `number?` predicate. The `syntax-spec` binding language cannot express the expected binding structure. The problem is the result of my design decision to map AST nodes one-to-one to scope graph fragments.

USE OF `syntax-spec` SERVICES Macros are used to expand the language of lens patterns down to only three core pattern forms: guard checking, conjunction, and lens binding. For example, the `cons` pattern is defined by the following macro:

```
(define-dsl-syntax cons pattern-macro
  (syntax-parser
    [(cons a d) (and (? cons?) (lens car-lens a) (lens cdr-lens d))]))
```

The expanded pattern uses `and` to have the pattern match only when the `(? cons?)` guard pattern matches and subsequently binds the `a` and `d` pattern variables to lenses. Pattern variable bindings use a binding class to allow references appearing in the body of an update clause to be transformed by reference compilers. A reference to a pattern variable compiles to code that accesses part of the contextual value using the lens associated with the pattern variable. A Racket `set!` expression referring to a pattern variable compiles to code that modifies the portion of the contextual value accessed by the lens.

13.4 LOGICAL STUDENT LANGUAGE

Piterkin and Jianu (2025) developed a `syntax-spec` re-implementation of Moy and Patterson (2025)'s Logical Student Language (LSL) as a project in my "Hack Your Own Language" course. The original implementation of LSL used the reflective expander API I created prior to `syntax-spec` (Section 5.3) to manually define a

DSL-specific macro expander. The re-implementation aims to take advantage of `syntax-spec` to simplify the implementation.

LSL is a teaching language designed to allow students to express and tests software specifications using sophisticated forms of software contracts. For example the following program expresses a dependent contract stating that the result of the `longest-string` function must not be shorter than any string in the input:

```
(define (longer-than-in?e los)
  (λ (x) (andmap (λ (s) (>= (string-length x) (string-length s)))
                 los)))

(: longest-string
   (Function (arguments [loss (NonemptyList string?)])
             (result (AllOf string? (longer-than-in? los)))))
(define (longest-string loss) #| elided |#)
```

The example is due to Moy and Patterson, modified to use the `syntax-spec` re-implementation's syntax.

BINDING STRUCTURE The binding forms of LSL mirror those in Racket. Beyond those binding forms in common, dependent contracts bind the names of the function arguments for use in the result contract.

USE OF `syntax-spec` SERVICES Like Racket itself, LSL has a small core language with other forms being defined as syntactic sugar via macros, enabled by `syntax-spec`. Because LSL is targeted at students, it emphasizes providing informative error messages. The syntax errors provided automatically by `syntax-spec` are acceptable, but better suited to professional programmers than learners. An additional problem arises for contract failure messages. These messages contain fragments of program syntax to indicate portions of the program involved in the error. Macro expansion complicates this error reporting, because the syntax handled by the compiler is the expanded core language, whereas users think in terms of the surface syntax. The `syntax-spec` expander currently does not provide good tools for accessing the surface syntax that corresponds to a portion of expanded syntax. The LSL implementation has to work around these `syntax-spec` limitations, which points to an important direction for future improvements to `syntax-spec`.

13.5 MINI-DUSA

Prakash and Eisbach (2025) developed miniDusa, a re-implementation of the Dusa finite-choice logic programming language (Martens, Simmons, and Arntzenius, 2025), as a project in my "Hack Your Own Language" course. This miniDusa program queries for a 3-coloring of a graph:

```
(logic
  (graph edge_e
    ('a ['b 'c 'e])
    ('c ['b 'd]))
  (((color_e X_e) is 1 2 3) :- (edge X _))
  (forbid (edge X_e Y_e)
          ((color X) is C)
          ((color Y) is C)))
```

Here, `graph` is a application-specific extension (created via a macro) that encodes a graph as a symmetric `edge` relation. Finite-choice logic programming extends Datalog with functional dependencies, indicated by rules using the `is` keyword. In this program, `color` is constrained to globally act as a function, mapping each input to just one of 1, 2, or 3. This ensures that nodes are assigned a unique color in the graph coloring.

BINDING STRUCTURE Relation and logic variable names in miniDusa both use exporting binding structure and are bound by the first occurrence in a given scope. Relation names are scoped to an entire `logic` block while logic variable names are scoped to individual rules. Realizing this scoping structure is possible but not straightforward in `syntax-spec`.

The first challenge is that `syntax-spec` expects distinguished binding and reference positions. However, miniDusa uses the convention common in logic programming languages that variables occurring anywhere in a rule are implicitly understood to be quantified. No particular occurrence of a logic variable is distinguished as the binding occurrence. To work around this difference, the miniDusa implementation considers the first occurrence the binder and expands other occurrences as references. This is accomplished using a rewrite production that checks during expansion whether a name is already bound and conditionally expands to a binding or reference form:

```
(~> v:id
    (if (lookup #'v (binding-class-predicate logic-var))
        #'(#%ref v)     ; if v has been bound as a logic-var already
        #'(#%bind v)))
```

The same trick is applied for relation names.

The second challenge is that relation and logic variable binding occurrences both occur within rules, but should have different scope. Unfortunately, `syntax-spec` can assign only a single scoping structure to a given syntax. To work around this, the expansion process for miniDusa uses another rewrite production to look into rules and extract relation names to bind in an outer position. This is possible because the positions where miniDusa is macro extensible do not overlap with the portion that needs to be eagerly analyzed for this extraction.

Although it is possible to work around these scoping structure issues with rewrite productions, they point to important limitations of the `syntax-spec` binding-rule language.

USE OF `syntax-spec` SERVICES The miniDusa compiler checks several simple static semantic properties, including that relations are used with the correct number of arguments and that variables and functional relations may appear only in certain positions. After static checking and elaboration, the compiler expands to quoted syntax that is interpreted at runtime.

The miniDusa implementation takes advantage of the macro-extensibility offered by `syntax-spec` to implement certain forms such as `forbid` by expansion to lower-level rules. Macros can also be useful to miniDusa programmers for encoding domain-specific data and computations into the restricted language of miniDusa rules, as in the example of the `graph` macro.

Taking advantage of `syntax-spec`'s approach to multi-language structure, miniDusa also allows programmers to easily import Racket functions for use in functional dependencies. This feature goes beyond what is offered in the original Dusa implementation, which only includes a limited set of built-in relations.

14 EVALUATION

The DSLs in the preceding chapters illustrate that `syntax-spec` can express a wide range of DSLs but also imposes certain limitations. This chapter begins with an evaluation that summarizes what these DSL implementations tell us about the expressivity of `syntax-spec` and the services it provides DSL creators. Next, I explore the advantages of `syntax-spec` over existing, low-level macro implementation approaches. Finally, I examine the performance cost of the system.

14.1 EXPRESSIVENESS AND UTILITY

This section summarizes the way that the 13 DSL implementations discussed in this part use `syntax-spec`. The use of `syntax-spec` is evaluated across 12 dimensions forming three categories. These categories address how each DSL benefits from `syntax-spec`'s multi-language approach, whether it easily fits within the declarative language, and whether it benefits from the specific tools offered by `syntax-spec`.

Evaluation dimensions

The first category of dimensions concerns how the DSLs take advantage of multi-language structure, implementing sophisticated compilation while still fluently integrating with Racket:

STATIC SEMANTICS Does the DSL enforce a static semantics beyond the grammar and binding checks provided by `syntax-spec`?

NON-LOCAL COMPILATION Does the DSL compile in a way that is not syntax-directed, such that the compilation would be difficult to accomplish with a composition of conventional macros?

HOST EXPRESSIONS Does the DSL include `racket-expr` host subexpression positions?

DEFINITION INTERFACE Does the DSL use definition host-interface forms to integrate with Racket definition contexts?

PROTECTED NAMES Does the DSL use binding classes to restrict or transform uses of DSL names in host expressions?

The second category describes the DSLs' use of `syntax-spec`'s declarative binding language:

EXPORTS Does the DSL implement separate scope and binding forms with `export` (beyond the exports in any definition host-interface forms)?

NESTING Does the DSL use nested binding rules?

FAITHFULLY EXPRESSED Can `syntax-spec` faithfully express all of the binding structure desired by the DSL designer?

The final category describes the DSLs' use of specific services offered by `syntax-spec`, which are derived from the declared syntax and binding structure:

SYMBOL TABLES Does the DSL use symbol tables to implement static checks or optimizations?

BINDING ANALYSES Does the DSL use operations like `free-variables` that expose `syntax-spec`'s analysis of binding structure?

COMPILER HYGIENE Does the DSL implementation rely on alphatisation to make certain transformations performed by the compiler hygienic?

MACRO EXTENSIBILITY Does the DSL take advantage of macros for sugar or user extensions?

Table 14.1: Summary of DSLs' use of `syntax-spec`

	mK (10)	PEG (11)	State (12.1)	Class (12.2)	cmdline (12.3)	TinyHDL (12.4)	ms-mK (12.5)	QEA (13.1)	PLTL (13.1)	Qi (13.2)	Lens (13.3)	LSL (13.4)	miniDusa (13.5)
Multi-language structure													
Static semantics	✓	✓	✓	—	—	✓	✓	—	✓	—	—	—	✓
Non-local compilation	✓	✓	✓	✓	✓	—	✓	—	—	✓	—	—	—
Host expressions	✓	✓	✓	✓	✓	—	✓	✓	✓	✓	✓	✓	✓
Definition interface	✓	✓	✓	—	✓	✓	✓	—	✓	✓	—	✓	—
Protected names	✓	✓	✓	✓	—	✓	✓	✓	✓	—	✓	✓	✓
Binding structure													
Exports	—	—	✓	✓	—	✓	—	✓	—	—	✓	✓	✓
Nesting	—	✓	✓	—	—	—	—	—	—	✓	—	—	✓
Faithfully expressed	✓	X	✓	✓	✓	X	✓	✓	✓	X	X	✓	✓
Use of `syntax-spec` services													
Symbol tables	✓	✓	✓	✓	—	✓	✓	—	—	—	—	✓	—
Binding analyses	—	—	—	—	—	—	—	✓	✓	—	—	—	—
Compiler hygiene	✓	✓	—	—	—	—	—	—	—	✓	—	—	—
Macro extensibility	✓	✓	—	✓	✓	—	✓	—	✓	✓	✓	✓	✓
DSL size													
`syntax-spec` LOC	59	59	41	25	58	44	107	34	43	154	23	107	63
Total LOC	4548	641	169	126	257	144	812	313	377	2442	159	1395	742

Results

Table 14.1 presents the results. The bottom two rows record the lines of code for the complete DSL implementation and how many of these lines are `syntax-spec` definitions.

The four Xs in the "faithfully expressed" row indicate shortcomings of the expressive power the current `syntax-spec` binding rules, that is, cases where `syntax-spec` cannot express the binding structure from the original DSL design. The PEG and Qi designs feature DAG-structured binding for branching-and-rejoining computation structures. The TinyHDL design includes dependent binding structure, where one reference points a composite structure (in this case a TinyHDL "entity"), and the names of the components of that structure should also be in scope. Finally, the Lens-match design intends that pattern variables splice out of pattern forms and bind the body of a match, without capturing references elsewhere in the pattern. The simple tree-structured and syntax-directed binding structure accommodated by `syntax-spec`'s binding language cannot express these designs. The corresponding sections for each of these DSLs describe the problems in more detail.

Most of the `syntax-spec` features are widely used across the selection of DSLs. The binding analysis features for computing, e.g., the free identifiers of a term are less widely used. This is in part because these features were added more recently to `syntax-spec`, so some DSLs implement these analyses manually instead. The binding analyses also cannot yet handle certain situations with nested binding or host-language subexpressions as discussed in Sections 8.2 and 11.2.3. Only the miniKanren, PEG, and Qi DSL implementations rely on compiler hygiene because these are the only implementations that perform complex program transformations to implement compiler optimizations.

14.2 CONCISION

Five of the DSLs discussed in this part have previous, comparable implementations using lower-level macro programming techniques. Specifically, all of these implementations employ a manually-written DSL-specific macro expander.

Table 14.2 provides a quantitative assessment of the relative effort required for these implementations. The first row labeled "`syntax-spec` LOC" reports the numer of lines of code required to declare the grammar and binding rules in `syntax-spec`. The second row labeled "Procedural LOC" reports the number of lines in the alternative implementation's manually-written macro expander.

Beyond the obvious quantitative advantage, a `syntax-spec` implementation comes with two important qualitative advantages. The first one concerns the elimination of design patterns in macro programming. Like OO design patterns, macro design patterns indicate weaknesses in the macro system. Further, the patterns obscure the actual specification of the DSL. By contrast, the `syntax-spec` sys-

Table 14.2: Comparison of case study implementations based on `syntax-spec` and manually-written macros

	Class	TinyHDL	PEG	miniKanren	cmdline
`syntax-spec` LOC	25	44	59	59	58
Procedural LOC	78	87	109	97	82

tem eliminates those patterns and allows programmers to focus on the essence of specifying grammars and binding structures.

The second advantage is the elimination of the need to understand Racket's complex macro system APIs, concepts such as scope sets (Flatt, 2016a), and the need to think in low-level imperative terms. A manually written DSL expander traverses syntax objects, annotates scope, and manipulates a compile-time environment. These operations are achieved through either Racket's low-level expander API (Flatt et al., 2012) or the library discussed in Section 5.3. Both APIs require advanced knowledge of Racket's expander, scoping, and syntax models to use.

```
(define/hygienic (expand-peg stx) #:definition
  (syntax-parse stx #:literal-sets (peg-literals)
    [nonterm-name:id #:when (lookup #'nonterm-name peg-non-terminal?)
     #'nonterm-name]
    [(: var-name:id subexp:peg)
     (define/syntax-parse subexp^ (expand-peg #'subexp))
     (define/syntax-parse var-name^ (bind! #'var-name (racket-var)))
     #'(: var-name^ subexp^)]
    [(=> subexp:peg action:expr)
     (with-scope sc
       (define/syntax-parse subexp^ (expand-peg (add-scope #'subexp sc)))
       (define/syntax-parse action^ (local-expand (add-scope #'action sc)))
       #'(=> subexp^ action^))]

    ;; elided cases for eps, seq, alt, *, +, token, peg-datum

    [(macro-name:id rest ...) #:when (lookup #'macro-name peg-macro?)
     (define transformer
       (peg-macro-transformer (lookup #'macro-name peg-macro?)))
     (expand-peg (transformer stx))]))
```

Figure 14.1: Ballantyne, King, and Felleisen (2020)'s manually-written expander for the PEG DSL

To give a sense of what these manually-written expanders look like, Figure 14.1 shows a fragment of the expander for the `PEG` DSL. It recurs through DSL syntax; it attaches scoping information to syntax objects and reflects bindings into the Racket expander's compile-time environment; and at macro uses, the DSL-specific expander retrieves the macro transformer and uses an API call to hygienically apply it. To provide IDEs with static binding information, it must also propagate syntax

properties from the surface syntax to the expanded code. All of this is accomplished with the manual invocation of the Racket expander (`local-expand`), scope manipulation (`with-scope`, `add-scope`), management of the expander environment (`bind!`, `racket-var`, `lookup`).

By comparison the declarative approach of `syntax-spec` hides all of these details and requires only an understanding of tree grammars and the binding language. As a result, the language implementor can focus on the essence of DSL design and implementation.

14.3 EXPANSION PERFORMANCE

In my experience with `syntax-spec` so far I have not encountered performance problems that inhibited practical use, but most of the programs I have developed are relatively small. This section presents the results of a preliminary evaluation of the expansion-time performance of `syntax-spec` to identify the current limits of the system's scalability. I have yet to invest any effort in improving expansion-time performance, so it is likely that these results can be improved with future work.

The miniKanren implementation presented in Chapter 10, `hosted-minikanren`, provides an ideal opportunity for comparison because its surface syntax is mostly compatible with `faster-minikanren` (Ballantyne, 2024). In fact, after performing optimizations `hosted-minikanren` generates code that uses a mixture of the `faster-minikanren` surface syntax and internal runtime system components. The implementation of `faster-minikanren` is a shallow embedding with a layer of conventional macros on top.

Table 14.3: Time required to expand and compile the `hosted-minikanren` benchmark suite. All experiments ran on a MacBook Pro with an M2 Pro CPU and 16GB of RAM with Racket 8.17.

`hosted-minikanren` with all optimizations	3743 milliseconds
`hosted-minikanren` with no optimizations	3012 milliseconds
`faster-minikanren`	981 milliseconds

As a first point of comparison, Table 14.3 reports the time required to expand and compile the 3013 lines of code of the `hosted-minikanren` benchmark suite using each system. In the configuration where optimizations are disabled, `hosted-minikanren` does little more than expand and check syntax with `syntax-spec` and then delegate further compilation to `faster-minikanren`. Given this layering, the expansion cost for `hosted-minikanren` must include the cost for the further expansion of the generated `faster-minikanren` code. Nonetheless, the approximately 3x higher expansion time for `hosted-minikanren` without optimizations over `faster-minikanren` suggests that the `syntax-spec` portion of expansion is especially slow. A likely cause is interpretive overhead.

Table 14.4: Expansion time scalability of hosted-miniKanren as compared to faster-miniKanren. The reported values indicate the largest n for which expansion and compilation of a program following each schema completes in less than 10 seconds, in hosted-miniKanren and faster-miniKanren respectively. All experiments ran on a MacBook Pro with an M2 Pro CPU and 16GB of RAM with Racket 8.17.

Benchmark	n for hosted-mK	n for faster-mK
Relation with n conjoined blocks	270	1072
Relation with `fresh` bindings nested n deep	239	521
Module with n trivial relation definitions	549	4983
Relation with n boundaries back-and-forth	18	636

Whereas the Racket expander has specialized code for each core language form, the current implementation of `syntax-spec` interprets binding specifications to drive expansion. It may be possible to improve expansion performance via an implementation that statically specializes expansion to each binding specification.

Certain shapes of program can be expected to incur specific expansion costs. Table 14.4 presents synthetic benchmark that test expansion time for programs that are large in particular dimensions. The table reports the largest size n for each given dimension in which a program can expand and compile in less than 10 seconds. Nested bindings are interesting because Racket's "binding as sets of scope" hygiene incurs a non-linear cost as nesting depth increases, as each identifier is annotated with scopes for all the surrounding binding forms. However, this cost is incurred similarly by both systems. The scaling limits for `hosted-minikanren` as compared to `faster-minikanren` are especially severe in the case of many relation definitions and of many boundary crossings. The overhead for relation definitions is likely due to the trampolining expansion process for definition-context host-interface forms (Section 8.3) and the expansion of persisted symbol table entries (Section 8.4). The cost of nested boundary crossings is likely due to re-expansion incurred by the delayed expansion implementation for host subexpressions (Section 8.2). The Turnstile system (Chang et al., 2019) encountered a similar problem, though there the re-expansions occur at every subexpression rather than at language boundaries. Critically, the `faster-minikanren` version of the boundary-crossing benchmark is not directly comparable because its implementation of the boundaries does not enforce the abstraction boundary of the DSL.

Part IV

REFLECTION

15

LOOKING BACK

This dissertation introduces multi-language macros and shows how they make it possible to implement DSLs with a confluence of desirable properties. Multi-language macros extend a host language with additional fragments of a multi-language rather than individual syntactic forms. The host language's syntax checking, binding analysis, and macro expander are augmented to handle the added parts, creating a unified front-end that understands the DSL, the host, and their interactions.

Multi-language structure facilitates the implementation of a static semantics and optimizing compiler for a DSL in several ways. The DSL compiler receives each contiguous fragment of DSL code as a compilation unit, allowing it to analyze and transform the entire chunk. The macro system provides tools for conveying compile-time information between fragments separated by host-language code to enable a global static semantics. And finally, it is straightforward to protect all boundaries between the DSL and host language with contracts to maintain the internal invariants of the DSL.

At the same time, interaction between the DSL and host language remains fine-grained. DSLs can contribute definitions to host language modules, and DSL code can embed host-language subexpressions. Code in each language can refer to names bound in the other, to the degree allowed by the DSL specification. Syntactic extensions can abstract over multi-language code to implement features that require the expressive power of both languages and to integrate separately developed DSLs.

To allow for the declarative definition of multi-language extensions, my approach marries language workbench technology with macro systems. The `syntax-spec` system brings a grammar and binding structure specification metalanguage into a general-purpose language as a library. Further, it builds on top of the conventional macro system already present in Racket. This positioning allows for a gentle learning curve from general-purpose programming to language extension with conventional macros, and finally to DSL creation with multi-language macros. None of these steps require programmers to abandon their existing tools, build processes, or programming environment.

Several novel technical contributions are required to weave together the separate strands from multi-language semantics, language workbenches, and macro systems. First, I extend Racket's "sets of scopes" hygienic macro expansion to handle any DSL core language declared in `syntax-spec`. The constraints of hygienic expansion require a new binding language design, yielding `syntax-spec`'s lightweight binding specifications. Second, I integrate the DSL expansion and host expansion processes, without requiring significant extensions to the host-language

expander. This integrated expansion process also addresses cross-language name references via reference compilers, ensuring that DSL compilers can interpose on every boundary between languages. Finally, I layer compiler hygiene and persistent symbol tables atop Racket's lower-level primitives to make it easy to work with names in DSL compilers.

The 13 DSLs explored in Part iii demonstrate how `syntax-spec` lowers the barrier to creating complex domain-specific languages. Small DSLs like my implementation of state machines can check syntax, integrate with Racket definitions and expressions, enforce a static semantics, and implement a non-local compilation strategy with fewer than 200 lines of code. Larger DSLs like the miniKanren optimizing compiler can realize optimizing compilation with substantial performance speedups while safely interacting with Racket. The declarative nature of `syntax-spec` makes these implementations concise and automatically provides features including macro extensibility and IDE services to every DSL. Beyond the immediate technical benefits, working with `syntax-spec` declarations allows DSL creators to think and communicate at the level of their DSL design rather than in terms of low-level details.

16 LOOKING FORWARD

The `syntax-spec` system represents a leap forwards for macros in the Lisp family of programming languages as a tool for building multi-language DSLs. At the same time, many directions for future work remain open. These include improving IDE services, simplifying macro hygiene, integrating with visual and interactive syntax, taking advantage of multi-language structure for formal reasoning, and using multi-language structure to integrate execution across disparate hardware targets. The `syntax-spec` design could also be brought to other languages, which would require further research into the integration of the metalanguage with conventional concrete syntax and type systems.

16.1 A RICHER CONNECTION TO THE IDE

DSLs created with `syntax-spec` integrate with the DrRacket IDE in the same manner as other macro-based DSLs in Racket (Feltey et al., 2016). However, the provided services are limited to error highlighting, jump-to-definition, and rename refactorings. Macro-extensible languages have long faced challenges providing advanced IDE services. Syntax defined by a macro is specified only via its procedural expansion into host-language syntax. This lack of structure makes it difficult to implement parsing, name analysis, or typechecking processes that recover from errors. Autocompletion faces a similar problem taking into account the grammar and static semantics of a macro-defined DSL.

Looking ahead, the syntax and binding rule declarations in `syntax-spec` provide the structure that is missing from conventional macro definitions. Thus it should be possible for `syntax-spec` to derive rich IDE services from DSLs specifications, just as language workbenches such as Spoofax do (Pelsmaeker et al., 2022). The open challenge lies in connecting the editor-service code that would be generated by `syntax-spec` to the IDE. One possible approach is to modify the interface for macros to separate out analysis and compilation procedures. The analysis procedure would return the information needed to implement IDE services. The host language could then expose these services via the Language Server Protocol (Bünder and Kuchen, 2020; Microsoft Corporation, 2022). While the task of implementing such an analysis interface for each language extension would be a burden for a traditional macro author, `syntax-spec` could generate it automatically.

The macro system of Lean 4 (Ullrich, 2023) takes a step in this direction by separating each macro definition into a parser extension and a transformer. To the extent that the parser can recover from errors, the IDE can understand

syntactic structure even when expansion fails. However, binding information and name completion candidates are discovered by elaborating the core Lean syntax that results from macro expansion. The IDE thus cannot provide services based on this information for regions of program text that are defined by macros that fail to expand. With the benefit of binding structure declarations in addition to grammatical information, `syntax-spec` could in principle recover from errors encountered during elaboration to provide binding-based IDE services even when DSL compilation fails.

The ability to analyze incomplete programs is potentially useful in contexts beyond IDEs for human programmers. Blinn et al. (2024) demonstrate that program synthesis with large language models can benefit from semantic context derived from the expected type and environment at the hole to synthesize. They propose a language server protocol extension for exposing such information. A multi-language macro system that also accommodates type system declarations could provide such a service.

16.2 REPLACING CONVENTIONAL MACROS ALTOGETHER

While `syntax-spec` provides a combination of multi-language macros for the DSL core and conventional macros for sugar, it would be worth exploring a system in which *all* macros come with grammar and binding specifications. Such a system could provide more reliable IDE services and implement macro hygiene more simply.

As discussed above, grammar and binding declarations can help IDE analyses handle incomplete and erroneous programs. Providing quality IDE services in all situations would require having this information for the entire surface syntax. Fully separating analysis and expansion would also simplify the task of making analysis incremental. As a programmer edits a program in an IDE, every keystroke yields a new program that needs analysis. Re-using portions of previous analysis results that are still valid allows for more responsive IDE services (Pacak, Erdweg, and Szabó, 2020). Racket's procedural macros allow arbitrary computations including side effects, so in general expansion cannot be made incremental. Separating out an analysis pass with a restricted structure while confining general computation and effects to the DSL compilation pass could solve this problem. The analysis process could nonetheless have a procedural foundation so that programmers can manually implement analysis for syntaxes that are not expressible in the declarative specification language.

Having binding rules for every form could also enable a simpler approach to hygiene. Conventional macro hygiene is complex because the binding structure of the use-site and template syntax is not apparent until after they are combined by expansion. With binding structure defined up-front, hygienic resolution could be less lazy. Conventional hygiene is particularly difficult to implement for certain patterns of expansion that arise in multi-language DSLs. For example, in Racket's

`class` DSL, macros may abstract over forms in the class body. The `class` macro uses `local-expand` to eliminate such macros uses before compiling the class core language (Flatt et al., 2012). What complicates hygiene is that the initial expansion process is not complete—it does not process method bodies, for example. The expansion of method bodies only resumes when the compiled code is further expanded. Expansion in `syntax-spec` follows this same structure, delaying expansion for `racket-expr` host subexpression positions. Hygienic expansion for such partial local-expansions could be simpler if all the syntactic forms involved come with explicit binding rules. In particular, resolving hygiene for macro expansion steps targeting the DSL core language would not need information derived from the subsequent compilation from DSL to Racket. A simpler hygiene algorithm could in turn relax the constraints that "sets of scopes" hygiene imposes on the design of the binding specification language, which are discussed in Chapter 9.

16.3 DSLS FOR DOMAIN EXPERTS

Some DSLs are designed with non-programmer domain experts in mind, rather than software engineers. Implementing a DSL for non-programmers demands different design considerations than those addressed by `syntax-spec`. Fluid integration with general-purpose code and standard programming tools is less critical. However, providing structured editing to reduce errors, guidance via autocompletion or form-like interfaces, and live reaction to edits to show their effects become more important. Visual representations of domain concepts can also make their manipulation more tangible. Language workbenches such as JetBrains MPS (Voelter and Lisson, 2014) that produce external DSLs equipped with projectional editors provide advantages for these situations.

Nonetheless, `syntax-spec` may sometimes be a good way to create DSLs for domain experts. Especially for initial prototyping, the ease with which a library-based DSL integrates into an existing software project may present an attractive trade-off against the advantages of a more sophisticated implementation in a standalone workbench. As I improve the IDE services provided by `syntax-spec`, this trade-off becomes more advantageous.

Several directions of prior work suggest ways to integrate the custom editor services and visual and interactive elements that are important for domain experts. "Editor libraries" allow DSLs to provide custom editor services such as refactorings (Erdweg et al., 2011a). "Visual syntax" provides a mechanism for using interactive visualizations in place of textual syntax for macro-defined language extensions (Andersen, Ballantyne, and Felleisen, 2020). Along similar lines, "livelits" provide interactive syntactic elements that take on a more limited linguistic role but integrate with live evaluation (Omar et al., 2021). Integrating visual, interactive elements with the multi-language approach of `syntax-spec` could yield a system that works well for domain experts and software engineers alike.

16.4 REASONING AND VERIFICATION

Given that `syntax-spec` gives DSLs a multi-language structure, techniques employing multi-language semantics would be appropriate for reasoning about DSLs defined in the system. Because each `syntax-spec` DSL compiles to Racket and the overall system is open to extension, compositional compiler correctness (Perconti and Ahmed, 2014) or semantic type soundness via realizability models (Patterson, 2022) would be most applicable. Such techniques would only establish the correctness of a formal model of a DSL rather than the actual implementation, because the complete semantics of Racket and `syntax-spec` are not themselves formalized and their implementations are not verified.

More speculatively, one could imagine building a multi-language macro system for a host language that is itself verified and which supports mechanized proofs. In such a setting, a multi-language macro could require that extensions provide a compiler correctness proof. Each DSL might specify a multi-language operational semantics extending that of the host. Then, a compiler correctness proof in the style of Perconti and Ahmed (2014) could show that compilation preserves the DSL semantics in all host contexts.

Or, for a weaker guarantee, the system could require that the DSL specify a realizability model defining the meaning of DSL types. Then the required proof would show that the DSL compiler produces host terms that are in the realizability relation indexed at the appropriate DSL type. A declaration that defines a boundary between multiple DSLs would prove the type soundness of this FFI via the technique of Patterson et al. (2022).

16.5 BEYOND THE RACKET VM

Many DSLs exist to take advantage of special-purpose hardware like GPUs. Of course, computations performed on such special-purpose hardware need to connect to the general-purpose code that provides access to data and orchestrates tasks. BraidGL (Sampson, McKinley, and Mytkowicz, 2017) combines multi-language programming and staging to make such interactions convenient in the context of graphics programming. Similarly, multitier programming languages (Weisenburger, Wirth, and Salvaneschi, 2020) allow a single program to express a computation whose execution is distributed across different platforms, like a web browser, backend server, and database.

Multi-language macros would be a natural fit for implementing these kinds of multi-target DSLs. The DSL core language declared in `syntax-spec` need not have any relationship to Racket's core language. While the DSLs discussed in Part iii all eventually compile to Racket code, a DSL compiler could equally well generate code for another platform. Explicit language boundaries and the reference compilers associated with `syntax-spec` binding classes provide an opportunity to insert code to communicate data between execution targets.

For GPU programs, compilation could write GLSL (Kessenich, Baldwin, and Roost, 2023) or CUDA (NVIDIA, 2007) code to an external file, compile it, and dynamically link with it via the Racket FFI. For multitier programs, compilation could generate JavaScript code for a Racket program to serve in response to HTTP requests. As a small experiment in this direction, I have implemented an S-expression syntax for a subset of JavaScript in `syntax-spec`.[1] The DSL compiler emits an AST in JSON format and generates JavaScript source using the `escodegen` library.[2]

Previous work in the Racket ecosystem has demonstrated such multi-target interactions. In the absence of a tool like `syntax-spec`, these systems have required complicated architectures and boilerplate to define their core language, check syntax, and expand macros. Magnolisp (Hasu and Flatt, 2016) is a language built atop Racket that compiles to C++, managing the code for external execution using Racket's submodules. To accomplish DSL expansion, MagnoLisp encodes its DSL syntax into a subset of Racket syntax, expands with the standard Racket expander, and later extracts a DSL AST from this expanded code. Sham (Walia, Shan, and Tobin-Hochstadt, 2021) provides a toolkit for building DSLs hosted in Racket's syntax that compile via LLVM and link with Racket. The Sham front-end uses Racket macros that expand to a shallow embedding in Racket to handle DSL name bindings. The shallow embedding code then evaluates to a deep embedding. Urlang[3] provides a Racket front-end for JavaScript syntax. It uses a manually written DSL expander, with drawbacks similar to those discussed in Section 14.2.

16.6 SCALING UP TO MAINSTREAM HOST LANGUAGES

The real value of multi-language macros will only be realized when they become available in mainstream programming languages. Languages such as Clojure, Elixir, Haskell, Julia, Lean, Rust, and Scala already include procedural macro systems that a multi-language macro system could be layered on top of. However, several challenges present themselves: limitations of the available reflective APIs, integration with conventional syntax, and integration with typechecking.

Although the basics of a metalanguage like `syntax-spec` can be hosted atop most procedural macro systems, its support for multi-language interaction between DSL and host requires a reflective compile-time API. Most essentially, the analysis function for a DSL must be able to (1) determine whether a name has a meaning established in the surrounding host context; (2) create an extended binding environment with entries for new names; and (3) analyze a host language subexpression in such an extended environment. In Racket, definition contexts, `local-expand`, and manipulation of scope sets provide these capabilities. Additional reflective operations are needed for implementing reference compilers, conveying static

1 https://github.com/michaelballantyne/syntax-spec/blob/v3/tests/dsls/js/js.rkt
2 https://github.com/estools/escodegen
3 https://github.com/soegaard/urlang

information across separate compilation, and recording information for the IDE, as explored in Chapter 8. Outside of Racket, macro systems in dependently typed languages that build on top of elaborator reflection (Christiansen and Brady, 2016) provide some of the needed operations. Lean 4 (Ullrich, 2023) and Agda[4] are two examples of such languages.

A multi-language macro system hosted by a language with conventional syntax would need to integrate with the host's parser. Two approaches are represented in modern macro systems. The first option is to use an extensible parser, as for example in Lean. The second option is to use a reader syntax that is flexible enough to represent extensions but structured enough to delimit regions defined by macros, as in Rust and a variety of other systems (Bachrach and Playford, 1999; Disney et al., 2014; Flatt et al., 2023; Rafkind and Flatt, 2012). In either case, the grammar provided in a language declaration would describe strings rather than S-expressions.

Many popular languages are statically typed. In a typed language programmers would benefit substantially from integration between the type systems of each DSL and the host language. Ideally, a multi-language macro system in a typed host would feature a metalanguage like Statix (Antwerpen et al., 2018) for specifying DSL type systems, suitably modified to integrate with the host's type system. A macro API that provides the right kind of interaction with the host language type system is an open research problem. The Turnstile metalanguage for Racket (Chang et al., 2019; Chang, Knauth, and Greenman, 2017), the Klister language (Barrett, Christiansen, and Gélineau, 2020), recent work on implementing typed languages with micros (Bocirnea and Bowman, 2025), and type tailoring (Wiersdorf et al., 2024) suggest potential directions. Once again, Lean's elaboration system has some of the needed facilities, providing reflection on host types but lacking certain features needed to connect host and DSL typechecking.

Managing the type environment for certain kinds of type system extensions poses a particular challenge. Multi-language static semantics typically require that the typechecker for each component language propagate the type environment for the other components (Matthews and Findler, 2007; Patterson et al., 2017). While Lean's metaprogramming system provides a mechanism for augmenting its global typing context with additional kinds of entries, this capacity does not extend to the local typing context. The problem is especially challenging in the case of a substructural DSL type systems. For example, Patterson et al. (2022) present a multi-language with an ML-like fragment and an affine fragment. The typing rules must split the affine environment for all composite terms, even in the ML-like fragment of the language. In the context of extending a host language with an affine DSL, this would require that the extension be able to modify the host-language typing rules to split the affine environment.

Extending `syntax-spec`'s multi-language approach to address these additional facets of conventional syntax and typechecking would complete the promise of

4 https://agda.readthedocs.io/en/v2.8.0/language/reflection.html#macros

language-oriented programming. Each DSL could realize domain-specific advantages via custom concrete syntax and type systems, in addition to optimizing compilation. And finally, building such a system for a mainstream host language would at last bring quality tools for language-oriented programming to a broad audience.

BIBLIOGRAPHY

Allen, Eric, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyong Ryu (2009). "Growing a syntax." In: *Proc. Workshop on Foundations of Object-Oriented Languages*. URL: https://www.cs.cmu.edu/~aldrich/FOOL09/allen.pdf.

Andersen, Leif, Michael Ballantyne, and Matthias Felleisen (Nov. 2020). "Adding Interactive Visual Syntax to Textual Code." In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: 10.1145/3428290.

Andersen, Leif, Stephen Chang, and Matthias Felleisen (Aug. 2017). "Super 8 languages for making movies (functional pearl)." In: *Proc. ACM Program. Lang.* 1.ICFP. DOI: 10.1145/3110274.

Antwerpen, Hendrik van, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth (2016). "A constraint language for static semantic analysis based on scope graphs." In: *Proc. Partial Evaluation and Program Manipulation*. PEPM '16, 49–60. DOI: 10.1145/2847538.2847543.

Antwerpen, Hendrik van, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser (Oct. 2018). "Scopes as types." In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276484.

Atkey, Robert, Sam Lindley, and Jeremy Yallop (2009). "Unembedding domain-specific languages." In: *Proc. Symposium on Haskell*. Haskell '09, pp. 37–48. DOI: 10.1145/1596638.1596644.

Bachrach, Jonathan and Keith Playford (1999). *D-Expressions: Lisp Power, Dylan Style*. URL: https://people.csail.mit.edu/jrb/Projects/dexprs.pdf.

Ball, W. W. Rouse (1914). *Mathematical Recreations and Essays (6th Edition)*. MacMillan & Co., Limited.

Ballantyne, Michael (2024). *faster-minikanren*. https://github.com/michaelballantyne/faster-minikanren.

Ballantyne, Michael, Mitch Gamburg, and Jason Hemann (Aug. 2024). "Compiled, Extensible, Multi-language DSLs (Functional Pearl)." In: *Proc. ACM Program. Lang.* 8.ICFP. DOI: https://doi.org/10.1145/3674627.

Ballantyne, Michael, Alexis King, and Matthias Felleisen (Nov. 2020). "Macros for domain-specific languages." In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: 10.1145/3428297.

Ballantyne, Michael, Rafaello Sanna, Jason Hemann, William E. Byrd, and Nada Amin (2025). "Multi-stage relational programming." In: *Proc. ACM Program. Lang.* Vol. 9. PLDI. DOI: 10.1145/3729314.

Barrett, Langston, David Thrane Christiansen, and Samuel Gélineau (2020). "Predictable Macros for Hindley-Milner." In: *The Workshop on Type-Driven Develop-*

ment. TyDe '20. URL: https://davidchristiansen.dk/pubs/tyde2020-predictable-macros-abstract.pdf.

Barrett, Langston, David Thrane Christiansen, and Samuel Gélineau (2020). "Predictable Macros for Hindley-Milner." In: *Proc. Workshop on Type-Driven Development*. URL: https://davidchristiansen.dk/pubs/tyde2020-predictable-macros-abstract.pdf.

Barringer, Howard, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard (2012). "Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors." In: *Proc. Formal Methods*, pp. 68–84. DOI: 10.1007/978-3-642-32759-9_9.

Barzilay, Eli, Ryan Culpepper, and Matthew Flatt (2011). "Keeping it clean with syntax parameters." In: *Proc. Workshop on Scheme and Functional Programming*. URL: https://www.schemeworkshop.org/2011/papers/Barzilay2011.pdf.

Blinn, Andrew, Xiang Li, June Hyung Kim, and Cyrus Omar (Oct. 2024). "Statically Contextualizing Large Language Models with Typed Holes." In: *Proc. ACM Program. Lang.* 8.OOPSLA2. DOI: 10.1145/3689728.

Bocirnea, Sean and William J. Bowman (Oct. 2025). "Fast and Extensible Hybrid Embeddings with Micros." In: *Proc. Workshop on Scheme and Functional Programming*. Scheme '25. DOI: 10.1145/3759537.3762696.

Bünder, Hendrik and Herbert Kuchen (2020). "Towards Multi-editor Support for Domain-Specific Languages Utilizing the Language Server Protocol." In: *Model-Driven Engineering and Software Development*. MODELSWARD 2019, pp. 225–245. DOI: 10.1007/978-3-030-37873-8_10.

Burmako, Eugene (2013). "Scala macros: let our powers combine!: On how rich syntax and static types work with metaprogramming." In: *Proc. Workshop on Scala*. Scala '13. DOI: 10.1145/2489837.2489840.

Byrd, William E. (2009). "Relational programming in minikanren: techniques, applications, and implementations." PhD thesis. ISBN: 9781109504682.

Byrd, William E., Michael Ballantyne, Gregory Rosenblatt, and Matthew Might (Aug. 2017). "A Unified Approach to Solving Seven Programming Problems (Functional Pearl)." In: *Proc. ACM Program. Lang.* 1.ICFP. DOI: 10.1145/3110252.

Byrd, William E., Eric Holk, and Daniel P. Friedman (2012). "miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl)." In: *Proc. Workshop on Scheme and Functional Programming*. Scheme '12, pp. 8–29. DOI: 10.1145/2661103.2661105.

Chang, Stephen, Michael Ballantyne, Milo Turner, and William J. Bowman (Dec. 2019). "Dependent type systems as macros." In: *Proc. ACM Program. Lang.* 4.POPL. DOI: 10.1145/3371071.

Chang, Stephen, Alex Knauth, and Ben Greenman (2017). "Type systems as macros." In: *Proc. Principles of Programming Languages*. POPL 2017, 694––705. ISBN: 9781450346603. DOI: 10.1145/3009837.3009886.

Christiansen, David and Edwin Brady (2016). "Elaborator reflection: Extending Idris in Idris." In: *Proc. International Conference on Functional Programming*. ICFP '16, 284—297. DOI: 10.1145/2951913.2951932.

Claessen, Koen and Peter Ljunglöf (2001). "Typed Logical Variables in Haskell." In: *Electronic Notes in Theoretical Computer Science* 41.1, p. 37. DOI: 10.1016/S1571-0661(05)80544-4.

Clément, Dominique, Janet Incerpi, and Gilles Kahn (1989). "CENTAUR: towards a "software tool box" for programming environments." In: *International Workshop on Software Engineering Environments*, pp. 287–304. DOI: 10.1007/3-540-53452-0_51.

Culpepper, Ryan (Aug. 2012). "Fortifying macros." In: *Journal of Functional Programming* 22.4-5, pp. 439–476. DOI: 10.1017/s0956796812000275.

Culpepper, Ryan, Matthias Felleisen, Matthew Flatt, and Shriram Krishnamurthi (2019). "From macros to DSLs: The evolution of Racket." In: *Proc. Summit on Advances in Programming Languages*. SNAPL 2019, 5:1–5:19. DOI: 10.4230/LIPIcs.SNAPL.2019.5.

Delmonaco, Michael (2022). *Ocular Patdown: A Racket Optics Library*. URL: https://github.com/quasarbright/ocular-patdown.

Dimoulas, Christos, Sam Tobin-Hochstadt, and Matthias Felleisen (2012). "Complete Monitors for Behavioral Contracts." In: *Proc. European Symposium on Programming*, pp. 214–233. DOI: https://doi.org/10.1007/978-3-642-28869-2_11.

Disney, Tim, Nathan Faubion, David Herman, and Cormac Flanagan (2014). "Sweeten your JavaScript: Hygienic macros for ES5." In: *Proc. Symposium on Dynamic Languages*. DLS '14, pp. 35–44. DOI: 10.1145/2661088.2661097.

Dybvig, R. Kent (2004). *The guaranteed optimization clause of the macro-writer's bill of rights*. Presented at Dan Friedman's 60th birthday conference. URL: https://www.youtube.com/watch?v=LIEX3tUliHw.

Elliott, Conal, Sigbjørn Finne, and Oege de Moor (2003). "Compiling embedded languages." In: *Journal of Functional Programming* 13.3, 455–481. DOI: 10.1017/S0956796802004574.

Erdweg, Sebastian, Lennart C. L. Kats, Tillmann Rendel, Christian Kästner, Klaus Ostermann, and Eelco Visser (2011a). "Growing a language environment with editor libraries." In: *Proc. Generative Programming and Component Engineering*. GPCE '11, pp. 167–176. DOI: 10.1145/2047862.2047891.

Erdweg, Sebastian, Tillmann Rendel, Christian Kästner, and Klaus Ostermann (2011b). "SugarJ: library-based syntactic language extensibility." In: *Proc. Object-Oriented Programming Systems, Languages & Applications*. OOPSLA '11, pp. 391–406. DOI: 10.1145/2048066.2048099.

Felleisen, Matthias (1985). *Transliterating Prolog into Scheme*. Tech. rep. 182. Indiana University. URL: https://legacy.cs.indiana.edu/ftp/techreports/TR182.pdf.

Felleisen, Matthias (1990). “On the expressive power of programming languages.” In: *Proc. European Symposium on Programming*. ESOP ’90, pp. 134–151. DOI: 10.1007/3-540-52592-0_60.

– (1991). “On the expressive power of programming languages.” In: *Science of Computer Programming* 17.1-3, pp. 35–75. DOI: 10.1016/0167-6423(91)90036-W.

Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt (Feb. 2018). “A programmable programming language.” In: *Communications of the ACM* 61.3, pp. 62–71. DOI: 10.1145/3127323.

Feltey, Daniel, Spencer P. Florence, Tim Knutson, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen (2016). “Languages the Racket way.” In: *Language Workbench Challenge*. URL: https://users.cs.northwestern.edu/~robby/pubs/papers/lwc2016-ffkscfff.pdf.

Findler, Robert Bruce, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen (2002). “DrScheme: A programming environment for Scheme.” In: *Journal of Functional Programming* 12.2, pp. 159–182. DOI: 10.1017/S0956796801004208.

Findler, Robert Bruce and Matthias Felleisen (2002). “Contracts for higher-order functions.” In: *Proc. International Conference on Functional Programming*. ICFP ’02, 48–59. DOI: 10.1145/581478.581484.

Fisher, David and Olin Shivers (Sept. 2008). “Building language towers with Ziggurat.” In: *Journal of Functional Programming* 18.5/6, pp. 707–780. DOI: 10.1017/S0956796808006928.

Flatt, Matthew (2002). “Composable and compilable macros: you want it when?” In: *Proc. International Conference on Functional Programming*. ICFP ’02, pp. 72–83. DOI: 10.1145/581478.581486.

– (2016b). *Binding as sets of scopes (extended version)*. URL: https://users.cs.utah.edu/plt/scope-sets/.

– (2016a). “Binding as sets of scopes.” In: *Proc. Principles of Programming Languages*. POPL ’16, pp. 705–717. DOI: 10.1145/2837614.2837620.

Flatt, Matthew, Ryan Culpepper, David Darais, and Robert Bruce Findler (Mar. 2012). “Macros that work together: compile-time bindings, partial expansion, and definition contexts.” In: *Journal of Functional Programming* 22.2, pp. 181–216. DOI: 10.1017/S0956796812000093.

Flatt, Matthew, Robert Bruce Findler, and Matthias Felleisen (2006). “Scheme with classes, mixins, and traits.” In: *Proc. Asian Conference on Programming Languages and Systems*. APLAS ’06, pp. 270–289. DOI: 10.1007/11924661_17.

Flatt, Matthew and PLT (2010). *Reference: Racket*. Tech. rep. PLT-TR-2010-1. https://racket-lang.org/tr1/. PLT Design Inc.

Flatt, Matthew et al. (Oct. 2023). "Rhombus: A New Spin on Macros without All the Parentheses." In: *Proc. ACM Program. Lang.* 7.OOPSLA2. DOI: 10.1145/3622818.

Ford, Bryan (2004). "Parsing expression grammars: a recognition-based syntactic foundation." In: *Proc. Principles of Programming Languages*. POPL '04, pp. 111–122. DOI: 10.1145/964001.964011.

Foster, J. Nathan, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt (May 2007). "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem." In: *ACM Trans. Program. Lang. Syst.* 29.3, 17—es. DOI: 10.1145/1232420.1232424.

Friedman, Daniel P., William E. Byrd, Oleg Kiselyov, and Jason Hemann (Mar. 2018). *The Reasoned Schemer, Second Edition*. The MIT Press. ISBN: 9780262535519.

Gibbons, Jeremy and Nicolas Wu (2014). "Folding domain-specific languages: Deep and shallow embeddings (functional Pearl)." In: *Proc. International Conference on Functional Programming*. ICFP '14, 339–347. DOI: 10.1145/2628136.2628138.

Gierczak, Olek, Lucy Menon, Christos Dimoulas, and Amal Ahmed (Apr. 2024). "Gradually Typed Languages Should Be Vigilant!" In: *Proc. ACM Program. Lang.* 8.OOPSLA1. DOI: 10.1145/3649842.

Griffin, Timothy (1988). "Notational definition—a formal account." In: *Proc. Symposium on Logic in Computer Science*. LICS '88, pp. 372–383. DOI: 10.1109/LICS.1988.5134.

Grimm, Robert (2004). *Practical Packrat Parsing*. Tech. rep. TR2004-854. New York University.

Harel, David (1987). "Statecharts: a visual formalism for complex systems." In: *Science of Computer Programming* 8.3, pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9.

Hasu, Tero and Matthew Flatt (May 2016). "Source-to-source compilation via submodules." In: *Proc. European Lisp Symposium*. ELS '16, pp. 57–64. URL: https://www.european-lisp-symposium.org/static/proceedings/2016.pdf.

Havelund, Klaus, Giles Reger, Daniel Thoma, and Eugen Zălinescu (2018). "Monitoring Events that Carry Data." In: *Lectures on Runtime Verification: Introductory and Advanced Topics*, pp. 61–102. DOI: 10.1007/978-3-319-75632-5_3.

Herman, David and Mitchell Wand (2008). "A theory of hygienic macros." In: *Programming Languages and Systems*. ESOP '15, pp. 48–62. DOI: 10.1007/978-3-540-78739-6_4.

Hickey, Rich (2008). "The Clojure programming language." In: *Proc. Symposium on Dynamic Languages*. DLS '08. DOI: 10.1145/1408681.1408682.

Hinze, Ralf (1998). "Prological Features in a Functional Setting Axioms and Implementation." In: *Fuji International Symposium on Functional and Logic*

Programming. FLOPS '98, pp. 98–122. URL: https://www.cs.ox.ac.uk/ralf.hinze/publications/FLOPS98.ps.gz.

Kasivajhula, Siddhartha (2021). *Qi: An Embeddable Flow-Oriented Language*. URL: https://docs.racket-lang.org/qi.

Kats, Lennart C. L. and Eelco Visser (2010). "The Spoofax language workbench: rules for declarative specification of languages and IDEs." In: *Proc. Object-Oriented Programming Systems, Languages & Applications*. OOPSLA '10, pp. 444–463. DOI: 10.1145/1869459.1869497.

Keep, Andrew W, Michael D Adams, Lindsey Kuper, William E Byrd, and Daniel P Friedman (2009). "A pattern matcher for miniKanren or how to get into trouble with CPS macros." In: *Proc. Workshop on Scheme and Functional Programming*. Scheme '09, pp. 37–45. URL: https://digitalcommons.calpoly.edu/csse_fac/83.

Keep, Andrew W. and R. Kent Dybvig (2013). "A nanopass framework for commercial compiler development." In: *Proc. International Conference on Functional Programming*. ICFP '13, 343—350. DOI: 10.1145/2500365.2500618.

Kessenich, John, Dave Baldwin, and Randi Roost (2023). *The OpenGL Shading Language*. Ed. by Graeme Leese. Broadcom. URL: https://registry.khronos.org/OpenGL/specs/gl/GLSLangSpec.4.60.pdf.

Keuchel, Steven, Stephanie Weirich, and Tom Schrijvers (2016). "Needle & Knot: binder boilerplate tied up." In: *Programming Languages and Systems*. ESOP '16, pp. 419–445. DOI: 10.1007/978-3-662-49498-1_17.

King, Alexis (2017). *The Hackett Programming Language*. URL: https://lexi-lambda.github.io/hackett/.

Kiselyov, Oleg, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan (2008). "Pure, declarative, and constructive arithmetic relations (declarative pearl)." In: *Proc. International Symposium on Functional and Logic Programming*, pp. 64–80. DOI: 10.1007/978-3-540-78969-7_7.

Kohlbecker, E. E. and M. Wand (1987). "Macro-by-example: Deriving syntactic transformations from their specifications." In: *Proc. Principles of Programming Languages*. POPL '87, 77–84. DOI: 10.1145/41625.41632.

Konat, Gabriël, Lennart Kats, Guido Wachsmuth, and Eelco Visser (2013). "Declarative name binding and scope rules." In: *Software Language Engineering*. SLE '13, pp. 311–331. DOI: 10.1007/978-3-642-36089-3_18.

Kosarev, Dmitrii and Dmitry Boulytchev (2018). "Typed Embedding of a Relational Language in OCaml." In: *Proc. ML Family Workshop*. ML/OCAML 2016. DOI: 10.4204/EPTCS.285.1.

Kowalski, Robert (1979). *Logic for Problem Solving*. Ediciones Díaz de Santos. ISBN: 0444003657.

Krishnamurthi, Shriram (2001). "Linguistic Reuse." PhD thesis. Rice University.

Landin, Peter J (1964). "The mechanical evaluation of expressions." In: *The computer journal* 6.4, pp. 308–320.

Lichtenstein, Orna, Amir Pnueli, and Lenore Zuck (1985). "The glory of the past." In: *Proc. Logics of Programs*, pp. 196–218. DOI: 10.1007/3-540-15648-8_16.

Lozov, Peter and Dmitry Boulytchev (2021). "Efficient fair conjunction for structurally-recursive relations." In: *Proc. Partial Evaluation and Program Manipulation*. PEPM 2021. DOI: 10.1145/3441296.3441397.

Mainland, Geoffrey (2007). "Why it's nice to be quoted: Quasiquoting for Haskell." In: *Proc. Haskell Workshop*. Haskell '07, 73–82. DOI: 10.1145/1291201.1291211.

Marriott, Kim and Harald Søndergaard (1989). "On Prolog and the occur check problem." In: *ACM SIGPLAN Notices* 24.5, pp. 76–82. DOI: 10.1145/66068.66075.

Martens, Chris, Robert J. Simmons, and Michael Arntzenius (Jan. 2025). "Finite-Choice Logic Programming." In: *Proc. ACM Program. Lang.* 9.POPL. DOI: 10.1145/3704849.

Matthews, Jacob and Robert Bruce Findler (2007). "Operational Semantics for Multi-Language Programs." In: *Proc. Principles of Programming Languages*. POPL '07, 3—10. DOI: 10.1145/1190216.1190220.

Mernik, Marjan, Jan Heering, and Anthony M. Sloane (2005). "When and How to Develop Domain-Specific Languages." In: *ACM Computing Surveys* 37.4, 316–344. DOI: 10.1145/1118890.1118892.

Meunier, Philippe and Daniel Silva (2003). "From Python to PLT Scheme." In: *Proc. Workshop on Scheme and Functional Programming*, pp. 24–29.

Microsoft Corporation (2022). *Language Server Protocol Specification v3.17*. URL: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/.

Moura, Leonardo de and Sebastian Ullrich (2021). "The Lean 4 Theorem Prover and Programming Language." In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe, pp. 625–635.

Moy, Cameron and Matthias Felleisen (2023). "Trace contracts." In: *Journal of Functional Programming* 33.e14. DOI: 10.1017/S0956796823000096.

Moy, Cameron, Ryan Jung, and Matthias Felleisen (2025). "Contract Systems Need Domain-Specific Notations." In: *Proc. European Conference on Object-Oriented Programming*, 42:1–42:24. DOI: 10.4230/LIPIcs.ECOOP.2025.42.

Moy, Cameron and Daniel Patterson (Aug. 2025). *Teaching Software Specification (Experience Report)*. DOI: 10.1145/3747533.

NVIDIA (2007). *CUDA Toolkit*. URL: https://developer.nvidia.com/cuda-toolkit.

Najd, Shayan, Sam Lindley, Josef Svenningsson, and Philip Wadler (2016). "Everything old is new again: Quoted domain-specific languages." In: *Proc. Partial Evaluation and Program Manipulation*. PEPM '16, 25–36. DOI: 10.1145/2847538.2847541.

Neron, Pierre, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth (2015). "A theory of name resolution." In: *Programming Languages and Systems*. ESOP '15, pp. 205–231. DOI: 10.1007/978-3-662-46669-8_9.

Omar, Cyrus and Jonathan Aldrich (2018). "Reasonably programmable literal notation." In: *Proc. ACM Program. Lang.* 2.ICFP. DOI: 10.1145/3236801.

Omar, Cyrus, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh (2021). "Filling Typed Holes with Live GUIs." In: *Proc. Programming Language Design and Implementation*. PLDI '21, 511—525. DOI: 10.1145/3453483.3454059.

Pacak, André, Sebastian Erdweg, and Tamás Szabó (Nov. 2020). "A systematic approach to deriving incremental type checkers." In: *Proc. ACM Program. Lang.* 4.OOPSLA. DOI: 10.1145/3428195.

Patterson, Daniel Baker (Aug. 2022). "Interoperability through Realizability: Expressing High-Level Abstractions Using Low-Level Code." PhD thesis. Boston, Massachusetts: Northeastern University. DOI: 10.17760/D20467221.

Patterson, Daniel, Noble Mushtak, Andrew Wagner, and Amal Ahmed (2022). "Semantic soundness for language interoperability." In: *Proc. Programming Language Design and Implementation*. PLDI '22, 609–624. DOI: 10.1145/3519939.3523703.

Patterson, Daniel, Jamie Perconti, Christos Dimoulas, and Amal Ahmed (2017). "FunTAL: reasonably mixing a functional language with assembly." In: *Proc. Programming Language Design and Implementation*. PLDI '17, pp. 495–509. DOI: 10.1145/3062341.3062347.

Pech, Vaclav, Alex Shatalin, and Markus Voelter (2013). "JetBrains MPS as a tool for extending Java." In: *Proc. Principles and Practices of Programming on the Java Platform*. PPPJ '13, 165–168. DOI: 10.1145/2500828.2500846.

Pelsmaeker, Daniel A. A., Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser (Apr. 2022). "Language-Parametric Static Semantic Code Completion." In: *Proc. ACM Program. Lang.* 6.OOPSLA1. DOI: pelsmaeker2022languageparametric.

Perconti, James T. and Amal Ahmed (2014). "Verifying an Open Compiler Using Multi-language Semantics." In: *European Symposium on Programming Languages and Systems*. ESOP '14, pp. 128–148. DOI: 10.1007/978-3-642-54833-8_8.

Piterkin, Andrey and Luke Jianu (2025). *Logical Student Language V2*. URL: https://github.com/AndreyPiterkin/lsl-v2.

Politz, Joe Gibbs, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi (2013). "Python: The full monty." In: *Proc. Object-Oriented Programming Systems, Languages & Applications*. OOPSLA '13, pp. 217–232. DOI: 10.1145/2509136.2509536.

Pombrio, Justin, Shriram Krishnamurthi, and Mitchell Wand (Aug. 2017). "Inferring scope through syntactic sugar." In: *Proc. ACM Program. Lang.* 1.ICFP. DOI: 10.1145/3110288.

Prakash, Ari and Zack Eisbach (2025). *miniDusa*. URL: https://github.com/ariscript/minidusa.

Rafkind, Jon and Matthew Flatt (2012). "Honu: Syntactic extension for algebraic notation through enforestation." In: *Proc. Generative Programming and Component Engineering*. GPCE '12, pp. 122–131. DOI: 10.1145/2371401.2371420.

Ramos, Pedro Palma and António Menezes Leitão (2014). "Implementing Python for DrRacket." In: *Proc. Symposium on Languages, Applications and Technologies*, pp. 127–141. DOI: 10.4230/OASIcs.SLATE.2014.127.

Rompf, Tiark, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky (2012). "Scala-Virtualized: Linguistic reuse for deep embeddings." In: *Higher-Order and Symbolic Computation* 25.1, pp. 165–207. DOI: 10.1007/s10990-013-9096-9.

Rozplokhas, Dmitry and Dmitry Boulytchev (2021). "A Complexity Study for Interleaving Search." In: *Proc. miniKanren and Relational Programming Workshop*. miniKanren '21. URL: http://minikanren.org/workshop/2021/minikanren-2021-final7.pdf.

Sampson, Adrian, Kathryn S. McKinley, and Todd Mytkowicz (Oct. 2017). "Static stages for heterogeneous programming." In: *Proc. ACM Program. Lang.* 1.OOPSLA. DOI: 10.1145/3133895.

Sarkar, DiPanwita, Oscar Waddell, and R. Kent Dybvig (2005). "Educational Pearl: A Nanopass framework for compiler education." In: *Journal of Functional Programming* 15.5, 653—667. DOI: 10.1017/S0956796805005605.

Savaton, Guillaume (2021). *Tiny-HDL*. URL: https://github.com/aumouvantsillage/Tiny-HDL-Racket.

Scherr, Maximilian and Shigeru Chiba (2014). "Implicit Staging of EDSL Expressions: A Bridge between Shallow and Deep Embedding." In: *Proc. European Conference on Object-Oriented Programming*. ECOOP '14, pp. 385–410. DOI: 10.1007/978-3-662-44202-9_16.

Sewell, Peter, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša (2007). "Ott: effective tool support for the working semanticist." In: *Proc. International Conference on Functional Programming*. ICFP '07, pp. 1–12. DOI: 10.1145/1291151.1291155.

Shaikhha, Amir, Vojin Jovanovic, and Christoph Koch (Aug. 2018). "A Compiler-Compiler for DSL Embedding." In: arXiv: 1808.01344.

Søndergaard, Harald (1986). "An application of abstract interpretation of logic programs: Occur check reduction." In: *Proc. European Symposium on Programming*. ESOP '86, pp. 327–338. DOI: 10.1007/3-540-16442-1_25.

Souza Amorim, Luis Eduardo de and Eelco Visser (2020). "Multi-purpose Syntax Definition with SDF3." In: *Proc. Software Engineering and Formal Methods*, pp. 1–23. DOI: 10.1007/978-3-030-58768-0_1.

Soy, Suzanne (2017). *Type expanders for Typed Racket*. URL: https://github.com/jsmaniac/type-expander.

Stansifer, Paul and Mitchell Wand (2014). "Romeo: a system for more flexible binding-safe programming." In: *Proc. International Conference on Functional Programming*. ICFP '14, 53–65. DOI: 10.1145/2628136.2628162.

Steele, Guy Lewis (1978). *Rabbit: A compiler for Scheme*. Tech. rep. MIT Artificial Intelligence Laboratory. URL: http://hdl.handle.net/1721.1/6913.

Svenningsson, Josef and Emil Axelsson (2015). "Combining deep and shallow embedding of domain-specific languages." In: *Computer Languages, Systems and Structures* 44.Part B, pp. 143–165. DOI: 10.1016/j.cl.2015.07.003.

Taha, Walid (1999). "Multi-Stage Programming: Its Theory and Applications." PhD thesis.

Thiemann, Peter and Matthias Neubauer (2008). "Macros for context-free grammars." In: *Proc. Principles and Practice of Declarative Programming*. PPDP '08, pp. 120–130. DOI: 10.1145/1389449.1389465.

Tobin-Hochstadt, Sam (2011). "Extensible pattern matching in an extensible language." In: eprint: 1106.2578v1.

Tratt, Laurence (2005). *The Converge programming language*. Tech. rep. TR-05-01. Department of Computer Science, King's College London. URL: https://nms.kcl.ac.uk/informatics/techreports/papers/TR-05-01.pdf.

Ullrich, Sebastian Andreas (2023). "An Extensible Theorem Proving Frontend." PhD thesis. Karlsruher Institut für Technologie (KIT). DOI: 10.5445/IR/1000161074.

Van Roy, Peter (1994). "1983–1993: The wonder years of sequential Prolog implementation." In: *The Journal of Logic Programming* 19-20, pp. 385–441. DOI: 10.1016/0743-1066(94)90031-0.

Van Wyk, Eric, Derek Bodin, Jimin Gao, and Lijesh Krishnan (2008). "Silver: An extensible attribute grammar system." In: *Proc. Workshop on Language Descriptions, Tools, and Applications*. LDTA '07, pp. 103–116. DOI: 10.1016/j.entcs.2008.03.047.

Verbitskaia, Ekaterina, Daniil Berezun, and Dmitry Boulytchev (2020). "An Empirical Study of Partial Deduction for miniKanren." In: *Proc. miniKanren and Relational Programming Workshop*. miniKanren '20. URL: http://minikanren.org/workshop/2020/minikanren-2020-paper2.pdf.

Verbitskaia, Ekaterina, Igor Engel, and Daniil Berezun (2023). "Semi-Automated Direction-Driven Functional Conversion." In: *Proc. miniKanren and Relational Programming Workshop*. miniKanren '23. URL: http://minikanren.org/workshop/2023/minikanren23-final2.pdf.

Voelter, Markus and Sascha Lisson (2014). "Supporting Diverse Notations in MPS' Projectional Editor." In: *Proc. International Workshop on The Globalization of Modeling Languages*. GEMOC 2014, pp. 7–16. URL: http://ceur-ws.org/Vol-1236/paper-03.pdf.

Walia, Rajan, Chung-chieh Shan, and Sam Tobin-Hochstadt (2021). *Sham: A DSL for Fast DSLs*. arXiv: 2005.09028v2 [cs.PL].

Wand, Mitchell (1984). "A semantic prototyping system." In: *Proc. Symposium on Compiler Construction*, 213–221. DOI: 10.1145/502874.502895.

Ward, Martin P (1994). "Language-oriented programming." In: *Software Concepts and Tools* 15.4, pp. 147–161. DOI: 10.1007/978-1-4302-2390-0-12.

Weirich, Stephanie, Brent A. Yorgey, and Tim Sheard (2011). "Binders Unbound." In: *Proc. International Conference on Functional Programming*. ICFP '11, pp. 333–345. DOI: 10.1145/2034773.2034818.

Weisenburger, Pascal, Johannes Wirth, and Guido Salvaneschi (Sept. 2020). "A Survey of Multitier Programming." In: *ACM Comput. Surv.* 53.4. DOI: 10.1145/3397495.

Wiersdorf, Ashton, Stephen Chang, Matthias Felleisen, and Ben Greenman (2024). "Type tailoring." In: *Proc. European Conference on Object-Oriented Programming*. DOI: 10.4230/LIPIcs.ECOOP.2024.44.

de Bruijn, Nicolaas Govert (1991). "Telescopic Mappings in Typed Lambda-Calculus." In: *Information and Computation* 91.2, pp. 189–204. DOI: 10.1016/0890-5401(91)90066-B.

Šinkarovs, Artjoms and Jesper Cockx (2021). *Choosing is Losing: How to combine the benefits of shallow and deep embeddings through reflection*. arXiv: 2105.10819.

A

SYNTAX-SPEC DOCUMENTATION

This appendix contains the documentation for the `syntax-spec-v3` Racket package which represents the most recent `syntax-spec` release at the time of writing. It includes a series of tutorials (Appendix A.1) and a reference manual (Appendix A.2). The documentation is also available online at `https://docs.racket-lang.org/syntax-spec-v3/index.html`. A more recent version of the system may be available at `https://github.com/michaelballantyne/syntax-spec`.

Michael Ballantyne <michael.ballantyne@gmail.com> Michael Delmonaco <mdelmonacochs@gmail.com>

This package provides a metalanguage for creating hosted DSLs. *Hosted DSLs* extend the syntax of Racket with their own grammar and have their own static semantics and compilers.

The metalanguage allows programmers to declare a DSL's grammar, binding rules, and integration points with Racket. Under the hood it produces a macro expander for the DSL that parses, checks name bindings, expands DSL macros, and produces syntax in the DSL's core language for compilation.

You can implement conventional macros that do all these same things, but it can take a lot of manual effort and a deep knowledge of Racket's syntax API.

You might find the metalanguage useful when you both:

- want to perform static analysis or optimizing compilation
- you want your DSL to be macro-extensible

A.1 TUTORIAL

The tutorial is broken down into illustrative examples:

A.1.1 *Basic Tutorial: State Machine Language*

This guide demonstrates use of syntax-spec via the case study of constructing a DSL for structuring code as state machines.

We will:

- Define the syntax (§1.1.1 “Grammar”)
- Add binding rules (§1.1.2 “Binding”)
- Integrate Racket subexpressions (§1.1.3 “Integrating Racket Subexpressions”)
- Compile to Racket code (§1.1.4 “Compilation”)
- Allow macros to extend the language (§1.1.5 “Macros”)

Here’s what using the DSL to define a controller for a vending machine looks like:

```
(define vending-machine
  (machine
   #:initial idle
   (state idle
     (on-enter (displayln "pay a dollar"))
     (on (dollar)
       (goto paid))
     (on (select-item item)
       (displayln "you need to pay before selecting an
item")
       (goto idle)))
   (state paid
     (on-enter (displayln "select an item"))
     (on (select-item item)
       (displayln (format "dispensing ~a" item))
       (goto idle)))))
```

The vending machine has two states: idle and paid. It reacts to two kinds of external events: a dollar being inserted, and an item being selected for purchase.

The `machine` declaration acts as a class. Racket code interacts with the machine by constructing an instance and calling methods corresponding to machine transitions such as `dollar`. The `get-state` method returns a symbol representing the current state.

Within the machine, Racket code can be run when certain states are entered or certain transitions occur. Within transitions, these actions can reference arguments to the transition event, such as `item` in the `select-item` event.

A.1.1.1 *Grammar*

The essential parts of a DSL implementation in `syntax-spec` are a specification of the DSL's syntax and a compiler that transforms DSL syntax to Racket. The syntax is specified in terms of *nonterminals* with associated binding rules. We'll introduce binding rules later in the tutorial. *Host interface macros* tie together the specification and the DSL compiler producing a Racket macro that forms the entry point to the language implementation.

Our initial specification with `syntax-spec` supplies the grammar:

```
#lang racket

(require syntax-spec-v3 (for-syntax syntax/parse racket/list))

(syntax-spec
  (host-interface/expression
    (machine #:initial initial-state:id s:state-spec ...)

    (error 'machine "compiler not yet implemented"))

  (nonterminal state-spec
    (state name:id transitions:transition-spec ...)
    (state name:id
      ((~datum on-enter) body:action-spec ...+)
      transitions:transition-spec ...))

  (nonterminal transition-spec
    (on (event-name:id arg:id ...)
      action:action-spec
      ...
      ((~datum goto) next-state:id)))

  (nonterminal action-spec
```

```
((~datum displayln) x:id)))
```

The `syntax-spec` form is the entry-point into the metalanguage. It must be used at the top-level of a module. In the example above, the language definition contains two kinds of definitions, for host interface macros and nonterminals.

The `host-interface/expression` form is used to define host interface macros that extend the language of Racket expressions. Here, it defines the `machine` syntax for creating a state machine implemented as a Racket class.

The first part of the host interface definition specifies the syntax of the host interface macro, beginning with the name of the form: `machine`. The remainder of the `machine` form's syntax specification describes the literal elements of the syntax and its subexpression positions. Literal elements include keywords like `#:initial`. A colon-separated name like `s:state-spec` indicates a subexpression position, where the first portion is the *spec variable* used to name the position and the latter portion is a reference to a nonterminal or binding class indicating the type of syntax that may appear in the subexpression.

The remainder of the host interface declaration is compile-time Racket code. Once the DSL syntax is checked and macro-expanded according to the syntax specification, this compile-time code is responsible for compiling from the DSL to Racket. For now it's a stub.

A.1.1.2 *Binding*

Consider this program:

```
(machine
 #:initial red
 (state red
   (on (event x)
     (goto green))
   (on (event x)
     (goto red))))
```

Our first transition is to `green`, but there is no `green` state. This should result in an unbound variable error.

However, let's say our compiler translates `(goto green)` to `(set! state 'green)` and doesn't produce any identifiers for `green`. Would we get an unbound reference error for `green`? No! We'd just have strange behavior at runtime, or maybe a runtime error, depending on the compiler.

We could adjust our compiler to check for unbound state references, but syntax-spec can do it for us. syntax-spec allows us to declare the binding and scoping rules for our language, and bindings and references will be checked before your compiler is even invoked, so your compiler can assume the program is not only grammatically correct, but also well-bound.

There are also several other benefits that we get by providing binding rules. We can use symbol tables to associate information with identifiers, we can allow our languages to have hygienic macros, we can compute the free identifiers of an expression, and many other identifier-related operations. We'll get more into these details later, but the point is you get a lot for free by declaring binding rules. This is why you should be excited!

Simple binding

First, let's declare that the arguments to an action are in scope in the guard expression:

```
(syntax-spec
  (binding-class event-var)

  ...

  (nonterminal transition-spec
    (on (event-name:id arg:event-var ...)
      action:action-spec
      ...
      ((~datum goto) next-state:id))
    #:binding (scope (bind arg) ... action ...))


  (nonterminal action-spec
    ((~datum displayln) x:event-var)))
```

We added a binding class, `event-var`, for an event's argument names. We also added a `#:binding` declaration to transition actions to declare that the `arg`s are bound in the `action` expressions and this binding introduces a new scope.

These simple binding rules behave like `let`:

```
(syntax-spec
  (binding-class my-var)
  (nonterminal my-expr
    (my-let ([x:my-var e:my-expr] ...) body:my-expr)
    #:binding [e ... (scope (bind x) ... body)]
```

```
    x:my-var
    n:number))
```

We could've just written `(scope (bind x) ... body)`. syntax-spec will automatically treat `e` as a reference position outside of the new scope. That's why we don't have to mention `event-name` in the binding rules for transitions. Additionally, for `action-spec` expressions, there is an implicit `#:binding` rule generated that treats `x` as a reference position.

Notice that there are ellipses in the binding spec corresponding to the ellipses in the syntax spec. Like with syntax patterns and syntax templates, ellipses allow us to control the binding structure of syntax with sequences like `[x:my-var e:my-expr] ...`.

All spec references in a binding spec must have the same depth as their syntax spec counterparts. This is stricter than syntax templates, where it is possible for a template variable to occur with greater ellipsis depth than its associated pattern variable.

Separate scope and binding forms

Now let's add binding rules for state names. We can't just use `scope` and `bind` since the binding of the state name comes from the `state-spec` nonterminal, and those bindings need to be in scope throughout the entire `machine` form. To use `bind`, we need to be able to refer to the name being bound directly. For this kind of binding structure, we use `export` to export bindings from the `state-spec` nonterminal and `import` to import those bindings into a scope in the `machine` host interface:

```
(binding-class state-name)


(host-interface/expression
  (machine #:initial initial-state:state-name s:state-spec ...)
  #:binding (scope (import s) ... initial-state)
  (error 'machine "compiler not yet implemented"))

(nonterminal/exporting state-spec
  (state name:state-name
    ((~datum on-enter) body:action-spec ...+)
    transition:transition-spec ...)
  #:binding (export name)

  (state name:state-name transition:transition-spec ...)
  #:binding (export name))
```

```
(nonterminal transition-spec
  (on (event-name:id arg:event-var ...)
    action:action-spec
    ...
    ((~datum goto) next-state:state-name))
  #:binding (scope (bind arg) ... action ...))
```

We use an exporting nonterminal for `state-spec`, which allows us to use the `export` binding rule. This binds `name` in `transition` and the other `state-spec` forms in the body of the machine, like `define` in a `class` body or a `block` form.

Similar to `bind` for a variable, we use `import` to declare that an exporting nonterminal's bindings should be in scope for the `initial-state` in the `machine`.

Nested binding

There is another type of binding rule that doesn't fit into our state machine language, but you might need it when creating a different language. This is nested binding and behaves like `let*`, where you have a sequence of variables being defined and each one is in scope for the subsequent definitions (but not previous ones). Here is an example:

```
(syntax-spec
  (binding-class my-var)
  (nonterminal my-expr
    (my-let* (b:binding-pair ...) body:my-expr)
    #:binding (nest b ... body)
    n:number
    x:my-var)
  (nonterminal/nesting binding-pair (nested)
    [x:my-var e:my-expr]
    #:binding (scope (bind x) nested)))
```

We create a nesting nonterminal for a binding pair, which has `nested`, which is like an argument for the nonterminal's binding rules. This represents the scope tree of the rest of the binding rules. In this case, the scope tree gets built up sort of like `foldr` on a list.

The *scope tree* is a first-class representation of the binding structure of the program. It's not something that you explicitly work with, but it's useful to know about. Conceptually, syntax-spec uses your language's binding rules to construct this scope tree during expansion.

From the simple nonterminal `my-expr`, we put the `binding-pair`'s bindings in scope using `nest`, providing `body` as the intial value of `nested`, like the base case value of `foldr`.

Since we're folding over the sequence of `b`s, the ellipses are inside of the `nest`.

A.1.1.3 *Integrating Racket Subexpressions*

In our state machine language, action expressions are very limited. Let's remind ourselves what the grammar for an action expression looks like:

```
(nonterminal action-spec
   ((~datum displayln) x:event-var))
```

An action expression can only `displayln` the value of a variable. What if we want something fancier, like using `format` inside the `displayln`? Really, it'd be ideal to be able to allow arbitrary racket expressions for the action. We can actually do that!

```
(syntax-spec
  ...

  (nonterminal/exporting state-spec
    (state name:state-name
      ((~datum on-enter) body:racket-body ...+)
      transition:transition-spec ...)
    #:binding [(export name) (scope (import body) ...)]

    (state name:state-name transition:transition-spec ...)
    #:binding (export name))

  (nonterminal transition-spec
    (on (event-name:id arg:event-var ...)
      body:racket-body
      ...
      ((~datum goto) next-state-name:state-name))
    #:binding (scope (bind arg) ... (import body) ...))

  ...)
```

Instead of using `action-spec` and defining our own nonterminal for action expressions, we can just use `racket-body`, which allows arbitrary racket expressions

and definitions. And our `event-var` identifiers will be in scope in the racket expression! We can control how references to our DSL-bound variables behave in Racket expressions and whether they're allowed at all using reference compilers, which we'll discuss in the §1.1.4 "Compilation" section.

In addition to `racket-body`, syntax-spec provides `racket-expr` for allowing Racket expressions, `racket-var` for allowing references to Racket-defined variables in DSL expressions, and `racket-macro` for allowing the language to be extended by arbitrary Racket macros. We'll talk more about macros in the §1.1.5 "Macros" section.

A.1.1.4 *Compilation*

Now that we have our grammar and binding rules defined, we must write a compiler to translate a state machine program to Racket. We already have a host interface macro defined, which is the entry point to our DSL:

```
(syntax-spec
  ...
  (host-interface/expression
    (machine #:initial initial-state:state-name s:state-spec ...)
    #:binding (scope (import s) ... initial-state)
    (error 'machine "compiler not yet implemented"))
  ...)
```

However, our compiler, which performs the actual translation, is not defined. The compiler is a macro that translates our state machine language to Racket code. In our compiler, we'll translate the state machine to Racket classes using the state machine pattern.

For example, let's imagine how we'd translate the example state machine:

```
(define vending-machine
  (machine
   #:initial idle
   (state idle
     (on-enter (displayln "pay a dollar"))
     (on (dollar)
       (goto paid))
     (on (select-item item)
       (displayln "you need to pay before selecting an
item")
       (goto idle)))
```

```
    (state paid
      (on-enter (displayln "select an item"))
      (on (select-item item)
        (displayln (format "dispensing ~a" item))
        (goto idle)))))
```

We'll create a class for the state machine, which acts as a context class, and a class for each state:

```
(let ()
  (define machine%
    (class object%
      (define state #f)
      (define/public (set-state state%)
        (set! state (new state% [machine this])))
      (define/public (get-state)
        (send state get-state))

      (define/public (dollar)
        (send/apply state dollar))

      (define/public (select-item item)
        (send/apply state select-item item))

      (send this set-state idle)
      (super-new)))

  (define idle
    (class object%
      (init-field machine)
      (define/public (get-state)
        'idle)

      (displayln "pay a dollar")

      (define/public (dollar)
        (send machine set-state paid))

      (define/public (select-item item)
        (displayln "you need to pay before selecting an
item")
        (send machine set-state idle))
      (super-new)))
```

```
(define paid
  (class object%
    (init-field machine)
    (define/public (get-state)
      'idle)

    (displayln "select an item")

    (define/public (select-item item)
      (displayln (format "dispensing ~a" item))
      (send machine set-state idle))
    (super-new)))

(new machine%))
```

The `machine%` class stores the current state instance and delegates to it. Each state class has methods for each defined transition. Transition actions go in the transition's method and `on-enter` actions go in the class body. When a state is entered, the `machine%` class creates a fresh instance of it, which runs the class body, and sets the current state to that instance. Finally, we return an instance of the machine class.

Now let's start to write the compiler:

```
(syntax-spec
  (binding-class event-var
    #:reference-compiler mutable-reference-compiler)
  ...
  (host-interface/expression
    (machine #:initial initial-state:state-name s:state-spec ...)
    #:binding (scope (import s) ... initial-state)
    #'(compile-machine initial-state s ...))
  ...)

(define-syntax compile-machine
  (syntax-parser
    #:datum-literals (machine state on-enter)
    [(_ initial-state
        (state state-name
          (~optional (on-enter action ...)
                     #:defaults ([(action 1) '()]))
          e ...)
        ...)
```

```
      #'(let ()
          (define machine%
            (class object%
              (define state #f)
              (define/public (set-state state%)
                (set! state (new state% [machine this])))
              (define/public (get-state)
                (send state get-state))

              (compile-proxy-methods (e ... ...) state)

              (send this set-state initial-state)
              (super-new)))

          (define state-name
            (class object%
              (init-field machine)
              (define/public (get-state)
                'state-name)
              action ...
              (compile-event-method e machine) ...
              (super-new)))
          ...

          (new machine%))]))
```

We defined a macro, `compile-machine`, which expands to something similar to what we wrote by hand above. One thing we have to do with syntax-spec is declare a reference compiler in the `binding-class` declaration. This allows us to control whether and how DSL identifiers behave in Racket expressions like actions. In our case, we use `mutable-reference-compiler`, which allows event arguments to be referenced and mutated. We don't specify a reference compiler for state names, so they cannot be referenced in Racket expressions. Only `goto`.

We have helpers to define the proxy methods in the `machine%` class and transition methods in the state classes:

```
(define-syntax compile-proxy-methods
  (syntax-parser
    #:datum-literals (on goto)
    [(_ ((on (event-name . _) . _) ...) target)
     #:with (unique-event ...)
     (remove-duplicates (map syntax-e (attribute event-name)))
     #'(begin
```

```
          (define/public (unique-event . args)
            (send/apply target unique-event args))
          ...)]))

(define-syntax compile-event-method
  (syntax-parser
    #:datum-literals (on goto)
    [(_ (on (event-name arg ...)
          action ...
          (goto name))
        machine)
     #'(define/public (event-name arg ...)
         action ...
         (send machine set-state name))]))
```

For `compile-proxy-methods`, to generate one method definition for each possible transition, we gather up all the transitions in `compile-machine` with that `(e ... ...)`, remove the duplicate transition event names, and define a proxy method for each one that delegates to the state instance, which is passed in as `target`. #racket[compile-event-method] is pretty straightforward.

One thing to note is that Racket expressions like `action` in `compile-event-method` get wrapped in a `#%host-expression` form by syntax-spec. You can usually ignore this fact completely when writing a compiler, but if you try to inspect the contents of a Racket expression in a compiler, you'll have to account for it.

Now we have all the pieces to run programs using state machines:

```
> (require racket/class
           syntax-spec-v3/tests/dsls/state-machine-for-tutorial)
> (define vending-machine
    (machine
     #:initial idle
     (state idle
       (on-enter (displayln "pay a dollar"))
       (on (dollar)
         (goto paid))
       (on (select-item _)
         (displayln "you need to pay before selecting an
item")
         (goto idle)))
     (state paid
       (on-enter (displayln "select an item"))
       (on (select-item item)
```

```
          (displayln (format "dispensing ~a" item))
          (goto idle)))))
pay a dollar
> (send vending-machine get-state)
'idle
> (send vending-machine select-item "chips")
you need to pay before selecting an item
pay a dollar
> (send vending-machine get-state)
'idle
> (send vending-machine dollar)
select an item
> (send vending-machine get-state)
'paid
> (send vending-machine select-item "chips")
dispensing chips
pay a dollar
> (send vending-machine get-state)
'idle
```

Note: a bug in versions of Racket prior to the (future) v8.17 release creates a problem when a DSL defined with `syntax-spec` is used in the same module scope as it is defined. If you put the `syntax-spec` declaration, `compile-machine` macro, and example use above together in one module, expansion will raise an error for the `item` reference. As a workaround until the v8.17 release, put your uses of the DSL in another module or submodule. For example, you could put the vending machine example above in a main or test submodule.

Symbol Tables

Symbol tables and symbol sets allow us to associate information with identifiers, similar to §2.3 “Dictionaries with Identifier Keys” and §2.4 “Sets with Identifier Keys”, but for DSL identifiers.

In our language’s compiler, we can use symbol set to raise an error when a state is unreachable:

```
(syntax-spec
  ...
  (host-interface/expression
    (machine #:initial initial-state:state-name s:state-spec ...)
    #:binding (scope (import s) ... initial-state)
    (check-for-inaccessible-states #'initial-state (attribute s))
    #'(compile-machine initial-state s ...))
```

```
  ...)

(begin-for-syntax
  (define (check-for-inaccessible-states initial-state-id
                                         state-specs)
    (define accessible-states
      (get-accessible-states initial-state-id state-specs))
    (for/list ([state-spec state-specs]
               #:unless (symbol-set-member?
                         accessible-states
                         (state-spec-name state-spec)))
      (error 'machine "Inaccessible state: ~a"
             (syntax->datum (state-spec-name state-spec)))))

  (define (get-accessible-states initial-state-id state-specs)
    (define accessible-states (local-symbol-set))
    (define (find-state-spec state-name)
      (findf (lambda (state-spec)
               (compiled-identifier=?
                state-name
                (state-spec-name state-spec)))
             state-specs))
    (define (add-reachable-states! state-name)
      (unless (symbol-set-member? accessible-states state-name)
        (symbol-set-add! accessible-states state-name)
        (define state-spec (find-state-spec state-name))
        (for ([next-state-name
               (state-spec-next-state-names state-spec)])
          (add-reachable-states! next-state-name))))
    (add-reachable-states! initial-state-id)
    accessible-states)

  (define (state-spec-name state-spec)
    (syntax-parse state-spec
      [(state name . _) #'name]))

  (define (state-spec-next-state-names state-spec)
    (syntax-parse state-spec
      [(state name
         (~or ((~datum on-enter) . _)
              ((~datum on) ev
                  body
                  ...
```

```
                    (goto next-state-name)))
          ...)
        (attribute next-state-name)])))
```

We build up a symbol set of accessible states with a depth-first search over the possible transitions starting from the initial state, and if we find a state that isn't accessible, we error.

This static check runs before we generate the compiled code. Compilers may have many static analysis passes like this one, or even passes that emit an intermediate representation like ANF. There are some special considerations to be made when creating multi-pass compilers with intermediate representations in syntax-spec which are covered in §1.3 "Advanced Tutorial: A Compiler with Transformative Passes"

```
> (require racket/class
           syntax-spec-v3/tests/dsls/state-machine-for-tutorial)
> (define gas-tank
    (machine
      #:initial full

      (state empty
        (on (re-fuel)
          (goto full)))

      (state full)))
machine: Inaccessible state: empty
```

We forgot to add a transition to go from full to empty. And since we start on full, there is no way to get to empty.

A.1.1.5 *Macros*

syntax-spec allows us to make our DSLs macro-extensible. For example, let's allow users to create macros for defininig states:

```
(syntax-spec
  ...
  (extension-class state-macro)

  (nonterminal/exporting state-spec
    #:allow-extension state-macro
```

```
    ...))

(define-syntax-rule
  (define-state-syntax name trans)
  (define-dsl-syntax name state-macro trans))
```

By adding an extension class called `state-macro` and allowing `state-spec` to be extended by these state macros, transformers wrapped with `state-macro` can be used in `state-spec` positions. syntax-spec provides `define-dsl-syntax` for defining these wrapped transformers. These macros will be hygienic in our DSL. Since only certain nonterminals are extensible by certain extension classes, we can control what kinds of macros can be used where.

Now let's create a macro in our language!

```
> (require racket/class
           syntax-spec-v3/tests/dsls/state-machine-for-tutorial)
> (define-state-syntax simple-state
      (syntax-rules ()
        [(_ name [evt next] ...)
         (state name
                (on (evt) (goto next))
                ...)]))
> (define traffic-light
    (machine
     #:initial red
     (simple-state red [tick green])
     (simple-state green [tick yellow])
     (simple-state yellow [tick red])))
> (send traffic-light get-state)
'red
> (send traffic-light tick)
> (send traffic-light get-state)
'green
> (send traffic-light tick)
> (send traffic-light get-state)
'yellow
> (send traffic-light tick)
> (send traffic-light get-state)
'red
```

The full code for the state machine example is available at https://github.com/michaelballantyne/syntax-spec/blob/main/tests/dsls/state-machine-for-tutorial.rkt.

There is also an example of using the state machine language to create a CSV browser with a GUI at https://github.com/michaelballantyne/syntax-spec/blob/main/demos/minimal-state-machine/csv-browser.rkt

A.1.2 *Advanced Tutorial: Simply Typed Lambda Calculus*

This guide demonstrates advanced usage of syntax-spec via the case study of construscting a DSL for the simply typed lambda calculus.

Here is an example program in our language:

```
(let ([f (lambda ([x : Number]) x)])
  (f 1))
```

Let's start out with defining the grammar and binding rules for basic typed expressions:

```
(syntax-spec
  (binding-class typed-var)
  (extension-class typed-macro #:binding-space stlc)
  (nonterminal typed-expr
    #:allow-extension typed-macro
    #:binding-space stlc

    x:typed-var
    n:number

    (#%lambda ([x:typed-var (~datum :) t:type] ...)
      body:typed-expr)
    #:binding (scope (bind x) ... body)
    (#%app fun:typed-expr arg:typed-expr ...)

    (#%let ([x:typed-var e:typed-expr] ...) body:typed-expr)
    #:binding (scope (bind x) ... body)

    (~> (e (~datum :) t)
        #'(: e t))
```

```
    (: e:typed-expr t:type)

    (~> (fun arg ...)
        #'(#%app fun arg ...)))
  (nonterminal type
    Number
    ((~datum ->) arg-type:type ... return-type:type))
  (host-interface/expression
   (stlc/expr e:typed-expr)
   (infer-expr-type #'e)
   #'(compile-expr e))
  (host-interface/expression
   (stlc/infer e:typed-expr)
   (define t (infer-expr-type #'e))
   (define t-datum (type->datum t))
   #`'#,t-datum))
```

There are some features we've never seen here. Let's go through them one by one:

```
(binding-class typed-var)
(extension-class typed-macro #:binding-space stlc)
(nonterminal typed-expr
    ...

    #:binding-space stlc

    ...)
```

Syntax-spec supports binding spaces, which allow DSL forms to have the same names as regular Racket forms like `let` without shadowing them. Even DSL macros won't shadow Racket names. We will eventually write a macro for `let` so we don't have to write `#%let` when we use the DSL.

```
(nonterminal typed-expr
  ...

  (~> (e (~datum :) t)
      #'(: e t))
  (: e:typed-expr t:type)

  ...)
```

This is called a rewrite production. We have a DSL form, `:`, for type annotations like `(: 1 Number)`. We add a rewrite production to allow infix use like `(1 : Number)`

for better readability. The first part of a rewrite production is a `syntax/parse` pattern and the second part is the DSL form that the source syntax should transform into. The pattern cannot refer to binding classes, nonterminals, etc.

We have another rewrite production that converts function applications to `#%app` forms. It is important that this comes after the type annotation rewrite. Otherwise, infix usages would be treated as function applications.

In general, it is a good idea to tag most forms in your grammar like `#%app` to make your compiler less bug-prone. It also allows us to rely on datum literals for distinguishing between forms, which is useful when your form names are in a special binding space.

Now let's define `infer-expr-type`:

```
(begin-for-syntax
  (struct number-type [] #:prefab)
  (struct function-type [arg-types return-type] #:prefab)

  (define-persistent-symbol-table types)

  (define (infer-expr-type e)
    (syntax-parse e
      [n:number (number-type)]
      [x:id (get-identifier-type #'x)]
      [((~datum #%lambda) ([x:id _ t] ...) body)
       (define arg-types (map parse-type (attribute t)))
       (for ([x (attribute x)]
             [t arg-types])
         (extend-type-environment! x t))
       (define body-type (infer-expr-type #'body))
       (function-type arg-types body-type)]
      [((~datum #%app) f arg ...)
       (define f-type (infer-expr-type #'f))
       (match f-type
         [(function-type expected-arg-types return-type)
          (unless (= (length expected-arg-types)
                     (length (attribute arg)))
            (raise-syntax-error
             'infer-expr-type
             (format
              "arity error. expected ~a arguments, but got
~a"
              (length expected-arg-types)
```

```
              (length (attribute arg)))
             this-syntax))
           (for ([expected-type expected-arg-types]
                 [arg (attribute arg)])
             (check-expr-type arg expected-type))
           return-type]
          [_ (raise-syntax-error
              'infer-expr-type
              (format
               "type mismatch. expected a function type, but
got ~a"
               (type->datum f-type))
              #'f)])]
      [((~datum :) e t-stx)
       (define t (parse-type #'t-stx))
       (check-expr-type #'e t)
       t]
      [((~datum #%let) ([x e] ...) body)
       (for ([x (attribute x)]
             [e (attribute e)])
         (extend-type-environment! x (infer-expr-type e)))
       (infer-expr-type #'body)]))

  (define (get-identifier-type x)
    (symbol-table-ref
     types x
     (lambda () (raise-syntax-error #f "untyped identifier" x))))

  (define (extend-type-environment! x t)
    (void (symbol-table-ref
           types x
           (lambda () (symbol-table-set! types x t)))))

  (define (check-expr-type e expected-type)
    (define actual-type (infer-expr-type e))
    (unless (equal? expected-type actual-type)
      (raise-syntax-error
       'infer-expr-type
       (format "type mismatch. expected ~a, but got ~a"
               (type->datum expected-type)
               (type->datum actual-type))
       e)))
```

```
(define (parse-type t-stx)
  (syntax-parse t-stx
    [(~datum Number) (number-type)]
    [((~datum ->) arg-type ... return-type)
     (function-type (map parse-type (attribute arg-type))
                    (parse-type #'return-type))]))

(define (type->datum t)
  (match t
    [(number-type) 'Number]
    [(function-type arg-types return-type)
     (append (list '->)
             (map type->datum arg-types)
             (list (type->datum return-type)))]))
```

We use prefab structs for our compile-time representation of types and we have a persistent symbol table mapping identifiers to types. A persistent symbol table allows information like an identifier's type to be used between modules even if the providing module has been compiled. Eventually, we'll add definitions to our language, so when type-checking a module that requires a typed identifier, we'll need the identifier's type from the persistent symbol table.

We have to use prefab structs because persistent symbol tables can't persist non-prefab structs. The only values allowed in a symbol table are those which satisfy the `syntax-datum?` predicate.

`extend-type-environment!` uses a bit of a hack. By default, symbol tables error when setting an identifier's value after it has already been set. We will end up re-inferring an expression's type later on, so we use this hack to only set the type if it isn't already set.

The rest is a typical type checker, nothing syntax-spec-specific.

Now let's implement our compiler:

```
(define-syntax compile-expr
  (syntax-parser
    [(_ n:number) #'n]
    [(_ x:id) #'x]
    [(_ ((~datum #%lambda) ([x:id _ _] ...) body))
     #'(lambda (x ...) (compile-expr body))]
    [(_ ((~datum #%app) f arg ...))
     #'((compile-expr f) (compile-expr arg) ...)]
    [(_ ((~datum :) e _)) #'(compile-expr e)]
```

```
    [(_ ((~datum #%let) ([x e] ...) body))
     #'(let ([x (compile-expr e)] ...) (compile-expr body))]))
```

Nothing special here, it's a straightforward translation to Racket. We pretty much just throw away the types.

Finally, we can write macros for `let` and `lambda`:

```
(define-syntax define-stlc-syntax
  (syntax-parser
    [(_ name:id trans:expr)
     #'(define-dsl-syntax name typed-macro trans)]))

(define-stlc-syntax let
  (syntax-parser
    [(_ ([x e] ...) body) #'(#%let ([x e] ...) body)]))

(define-stlc-syntax lambda
  (syntax-parser
    [(_ ([x (~datum :) t] ...) body)
     #'(#%lambda ([x : t] ...) body)]))

(define-stlc-syntax let*
  (syntax-parser
    [(_ () body) #'(let () body)]
    [(_ ([x:id e] binding ...) body)
     #'(let ([x e]) (let* (binding ...) body))]))
```

Right now, these don't need to be macros. But when we add definitions, We will desugar multi-body `let` and `lambda` expressions to single-body ones.

Now we can run some programs:

```
> (require syntax-spec-v3/tests/dsls/simply-typed-lambda-calculus)
> (stlc/infer 1)
'Number
> (stlc/expr 1)
1
> (stlc/infer (lambda ([x : Number]) x))
'(-> Number Number)
> (stlc/expr (lambda ([x : Number]) x))
#<procedure:...lambda-calculus.rkt:221:7>
```

A.1.2.1 *Integrating Racket Expressions*

Let's add arbitrary Racket expressions to our language. These can evaluate to anything, so we can't infer their types. We can require the user to annotate the type, but we shouldn't just trust that the type is correct. Instead, we should add a contract check to ensure that the annotation is accurate.

We also need to add a contract check in the other direction, even if we don't allow arbitrary Racket expressions. Let's consider a program in our language:

```
(stlc/expr (lambda ([f : (-> Number Number)] [x : Number]) (f x)))
```

It evaluates to a function which takes in a function and a number and applies the function to a number. But `stlc/expr` gives us a raw procedure that we can pass anything into!

```
((stlc/expr (lambda ([f : (-> Number Number)] [x : Number]) (f x)))
 "not a function"
 1)
```

This produces a runtime type error from inside the typed code! This should be impossible. And if we allow DSL variables to be referenced in Racket expressions, we'll need to insert contract checks on references to make sure they're used properly. We can do this by creating a custom reference compiler.

Let's do it!

```
(syntax-spec
  (binding-class typed-var
    #:reference-compiler typed-var-reference-compiler)
  ...

  (nonterminal typed-expr
    ...

    (rkt e:racket-expr (~datum :) t:type)

    ...)

  ...

  (host-interface/expression
    (stlc/expr e:typed-expr)
    (define/syntax-parse t (infer-expr-type #'e))
```

```
    #'(compile-expr/top e t))

  ...)

(begin-for-syntax
  (define (infer-expr-type e)
    (syntax-parse e
      ...

      [((~datum rkt) e (~datum :) t)
       (parse-type #'t)]

      ...)))

(define-syntax compile-expr/top
  (syntax-parser
    [(_ e t-stx)
     (define t (syntax->datum #'t-stx))
     (define/syntax-parse e^
       #'(compile-expr e))
     #`(contract #,(type->contract-stx t)
                 e^
                 'stlc 'racket
                 #f #'e^)]))

(begin-for-syntax
  (define typed-var-reference-compiler
    (make-variable-like-reference-compiler
     (lambda (x)
       #`(contract #,(type->contract-stx (get-identifier-type x))
                   #,x
                   'stlc 'racket
                   '#,x #'#,x))))

  (define (type->contract-stx t)
    (match t
      [(number-type) #'number?]
      [(function-type arg-types return-type)
       (define/syntax-parse (arg-type-stx ...)
         (map type->contract-stx arg-types))
       (define/syntax-parse return-type-stx
         (type->contract-stx return-type))
       #'(-> arg-type-stx ... return-type-stx)])))
```

```
(define-syntax compile-expr
  (syntax-parser
    ...

    [(_ ((~datum rkt) e (~datum :) t))
     #`(contract #,(type->contract-stx (parse-type #'t))
                 e
                 'racket 'stlc
                 #f #'e)]))
```

We added a new form to our language, `rkt`, which contains a racket expression and a type annotation. The compilation of this experssion involves a contract check to make sure the value is of the expected type. We also added a contract check in the other direction when a typed value flows out of the host interface and created a custom reference compiler using `make-variable-like-reference-compiler` which inserts a contract check when a DSL variable is referenced in racket. These contract checks ensure typed values (particularly procedures) are used properly in untyped code.

This implementation is far from efficient. Instead of generating the syntax for a contract check everywhere, we should defer to a runtime function and have the type flow into the runtime since it's a prefab struct. We should also avoid inserting a contract check every time a DSL variable is referenced in Racket and just do it once per variable. But for this tutorial, we'll keep it simple.

Let's run some example programs now:

```
> (require syntax-spec-v3/tests/dsls/simply-typed-lambda-calculus)
> (stlc/expr
    (let ([add (rkt + : (-> Number Number Number))])
      (add 1 2)))
3
> (stlc/expr
    (rkt "not a number" : Number))
broke its own contract
  promised: number?
  produced: "not a number"
  in: number?
  contract from: racket
  blaming: racket
   (assuming the contract is correct)
  at: eval:3:0
> (stlc/expr
```

```
     (let ([add (rkt <= : (-> Number Number Number))])
       (add 1 2)))
broke its own contract
  promised: number?
  produced: #t
  in: the range of
      (-> number? number? number?)
  contract from: racket
  blaming: racket
   (assuming the contract is correct)
  at: eval:4:0
> ((stlc/expr (lambda ([f : (-> Number Number)] [x : Number]) (f x)))
   "not a function"
   1)
contract violation
  expected: a procedure
  given: "not a function"
  in: the 1st argument of
      (-> (-> number? number?) number? number?)
  contract from: stlc
  blaming: racket
   (assuming the contract is correct)
  at: <pkgs>/syntax-spec-v3/tests/dsls/simply-typed-lambda-c
alculus.rkt:288:39
> (stlc/expr
   (let ([app (lambda ([f : (-> Number Number)] [x : Number])
                (f x))])
     (rkt (app "not a function" 1) : Number)))
app: contract violation
  expected: a procedure
  given: "not a function"
  in: the 1st argument of
      (-> (-> number? number?) number? number?)
  contract from: stlc
  blaming: racket
   (assuming the contract is correct)
  at: eval:6:0
```

Our contract checks protect typed-untyped interactions.

A.1.2.2 *Adding Definitions*

Next, let's add definitions to our language:

```
(syntax-spec
  ...
  (nonterminal typed-expr
    ...

    (block d:typed-definition-or-expr ... e:typed-expr)
    #:binding (scope (import d) ... e)

    ...)

  ...

  (nonterminal/exporting typed-definition-or-expr
    #:allow-extension typed-macro
    #:binding-space stlc
    (#%define x:typed-var t:type e:typed-expr)
    #:binding (export x)
    (begin defn:typed-definition-or-expr ...+)
    #:binding [(re-export defn) ...]
    e:typed-expr)

  ...

  (host-interface/definitions
   (stlc body:typed-definition-or-expr ...+)
   #:binding [(re-export body) ...]
   (type-check-defn-or-expr/pass1 #'(begin body ...))
   (type-check-defn-or-expr/pass2 #'(begin body ...))
   #'(compile-defn-or-expr/top (begin body ...))))
```

We added a new nonterminal for forms that can be used in a definition context. Since definitions inside of a `begin` should spliced in to the surrounding definition context, we use the binding rule `re-export`, which we haven't seen yet. As the name suggests, it takes all exported names from an exporting nonterminal sub-expression and re-exports them. Here is an example of this splicing in regular Racket:

```
> (begin
    (begin
```

```
      (define a 1))
    (define b 2))
> (+ a b)
3
```

We also added the `block` form to our expression nonterminal so we can use definitions in expressions. To make the bindings from the definitions accessible within the `block` form, we use `scope` and `import`.

To support top-level definitions, we added a new host interface using `host-interface/definitions`, which we've never seen before. This defines a special type of host interface that can only be used in a definition context. This type of host interface can be used to define module-level variables that can be used with `provide` and `require`. Now that this is possible, it is important that we're using a persistent symbol table to store type information.

Now let's update the rest of our code:

```
(begin-for-syntax
  (define (infer-expr-type e)
    (syntax-parse e
      ...

      [((~datum block) d ... e)
       (type-check-defn-or-expr/pass1 #'(begin d ...))
       (type-check-defn-or-expr/pass2 #'(begin d ...))
       (infer-expr-type #'e)]

      ...))
  ...

  (define (type-check-defn-or-expr/pass1 e)
    (syntax-parse e
      [((~datum #%define) x:id t _)
       (extend-type-environment! #'x (parse-type #'t))]
      [((~datum begin) body ...)
       (for ([body (attribute body)])
         (type-check-defn-or-expr/pass1 body))]
      [_ (void)]))

  (define (type-check-defn-or-expr/pass2 e)
    (syntax-parse e
      [((~datum #%define) _ t e)
       (check-expr-type #'e (parse-type #'t))]
```

```
      [((~datum begin) body ...)
       (for ([body (attribute body)])
         (type-check-defn-or-expr/pass2 body))]
      [e (void (infer-expr-type #'e))])))

(define-syntax compile-expr/top
  (syntax-parser
    [(_ e t-stx (~optional should-skip-contract?))
     (define t (syntax->datum #'t-stx))
     (define/syntax-parse e^
       #'(compile-expr e))
     (if (attribute should-skip-contract?)
         #'e^
         #`(contract #,(type->contract-stx t)
                     e^
                     'stlc 'racket
                     #f #'e))]))

(define-syntax compile-expr
  (syntax-parser
    ...

    [(_ ((~datum block) d ... e))
     #'(let ()
         (compile-defn-or-expr d)
         ...
         (compile-expr e))]))

(define-syntax compile-defn-or-expr/top
  (syntax-parser
    [(_ ((~datum #%define) x:id _ body))
     #`(define x (compile-expr/top
                  body #,(get-identifier-type #'x) #t))]
    [(_ ((~datum begin) body ...+))
     #'(begin (compile-defn-or-expr/top body) ...)]
    [(_ e)
     #`(compile-expr/top e #,(infer-expr-type #'e) #t)]))

(define-syntax compile-defn-or-expr
  (syntax-parser
    [(_ ((~datum #%define) x:id _ body))
     #`(define x (compile-expr body))]
    [(_ ((~datum begin) body ...+))
```

```
     #'(begin (compile-defn-or-expr body) ...)]
    [(_ e)
     #'(compile-expr e)]))

(define-stlc-syntax let
  (syntax-parser
    [(_ ([x e] ...) body)
     #'(#%let ([x e] ...) body)]
    [(_ ([x e] ...) body ...+)
     #'(#%let ([x e] ...) (block body ...))]))

(define-stlc-syntax lambda
  (syntax-parser
    [(_ ([x (~datum :) t] ...) body)
     #'(#%lambda ([x : t] ...) body)]
    [(_ ([x (~datum :) t] ...) body ...+)
     #'(#%lambda ([x : t] ...) (block body ...))]))

(define-stlc-syntax let*
  (syntax-parser
    [(_ () body) #'(let () body)]
    [(_ ([x:id e] binding ...) body)
     #'(let ([x e]) (let* (binding ...) body))]))

(define-stlc-syntax define
  (syntax-parser
    [(_ x:id (~datum :) t e)
     #'(#%define x t e)]
    [(_ (f:id [arg:id (~datum :) arg-type] ...)
        (~datum ->) return-type body ...)
     #'(#%define f (-> arg-type ... return-type)
                 (lambda ([arg : arg-type] ...)
                   body
                   ...))]))
```

To type-check a group of definitions, we must take two passes. The first pass must record the type information of all defined identifiers, and the second pass checks the types of the bodies of definitions. Since mutual recursion is possible, we need the types of all identifiers before we can start checking the types of definition bodies which may reference variables before their definitions. This is a common pattern when working with mutually recursive definition contexts in general.

We added an optional flag to disable the contract check for `compile-expr/top` when compiling top-level definitions since it is unnecessary.

We also added support for multi-body `let`, `lambda`, and `let*`, and we added a macro around `#%define` for syntactic sugar.

Let's run it!

```
> (require syntax-spec-v3/tests/dsls/simply-typed-lambda-calculus)
> (stlc
    (begin
     (define two : Number
             2)
     (define three : Number
             3))
    (define add : (-> Number Number Number)
      (rkt + : (-> Number Number Number))))
> (stlc/expr (add two three))
5
```

A.1.3 *Advanced Tutorial: A Compiler with Transformative Passes*

Many DSLs need a compiler that transforms syntax in several passes. Some passes may just be static checks, and others may actually transform the program, often to a restricted subset of the surface language. When using syntax-spec, some special care needs to be taken with transformative passes. To demonstrate how such a DSL can be implemented, we will create a language with an A-normal form transformation and an unused variable pruning optimization.

A.1.3.1 *Expander*

Here is the syntax-spec of our language:

```
#lang racket
(require syntax-spec
         (for-syntax syntax/parse racket/syntax
                     racket/match racket/list))
(syntax-spec
  (binding-class var
                 #:reference-compiler immutable-reference-compiler)
  (nonterminal full-expr
    #:binding-space anf
```

```
  n:number
  x:var
  (let ([x:var e:full-expr]) body:full-expr)
  #:binding (scope (bind x) body)
  (+ a:full-expr b:full-expr)
  (* a:full-expr b:full-expr)
  (/ a:full-expr b:full-expr)
  (rkt e:racket-expr))
(nonterminal anf-expr
  #:binding-space anf
  ((~datum let) ([x:var e:rhs-expr]) body:anf-expr)
  #:binding (scope (bind x) body)
  e:rhs-expr)
(nonterminal rhs-expr
  #:binding-space anf
  ((~datum +) a:immediate-expr b:immediate-expr)
  ((~datum *) a:immediate-expr b:immediate-expr)
  ((~datum /) a:immediate-expr b:immediate-expr)
  ((~datum rkt) e:racket-expr)
  e:immediate-expr)
(nonterminal immediate-expr
  #:binding-space anf
  x:var
  n:number)

(host-interface/expression
 (eval-expr e:full-expr)
 #'(compile-expr e)))
```

Our language supports arithmetic, local variables, and Racket subexpressions.

We have the following nonterminals:

- `full-expr`: The surface syntax of a program
- `anf-expr`: An expression in A-normal form. Users will not be writing these expressions; the compiler will transform `full-expr`s the user writes into `anf-expr`s.
- `rhs-expr`: An expression which is allowed to be on the right-hand side of a binding pair in an expression when it is in A-normal form. Conceptually, these expressions take at most one "step" of reduction to evaluate. In other words, no nested expressions (except for `rkt` expressions).

- `immediate-expr`: Atomic expressions that can immediately be evaluated.

A-normal form makes the evaluation order of the program completely unambiguous and simplifies compilation to a language like assembly. Now, let's transform our surface syntax to it!

A.1.3.2 *A-normal Form Transformation*

The core idea of transforming to A-normal form is extracting nested subexpressions into temporary variables. For example:

```
(+ (+ 1 2) (+ 3 4))
~>
(let ([tmp1 (+ 1 2)])
  (let ([tmp2 (+ 3 4)])
    (+ tmp1 tmp2)))
```

To follow our grammar for an `anf-expr`, the arguments to functions like `+` must be immediate expressions, like variable references or numbers. Our source program did not obey this rule, so we had to create temporary variables for subexpressions and replace each subexpression with a reference to its temporary variable.

Now let's automate this process:

```
(begin-for-syntax
  ;; full-expr -> anf-expr
  ;; convert an expression to A-normal form
  (define (to-anf e)
    (define bindings-rev '())
    ;; Identifier rhs-expr -> Void
    ;; record a variable binding pair
    (define (lift-binding! x e)
      (set! bindings-rev (cons (list x e) bindings-rev)))
    (define e^ (to-rhs! e lift-binding!))
    (wrap-lets e^ (reverse bindings-rev)))

  ;; full-expr (Identifier rhs-expr -> Void) -> rhs-expr
  ;; convert an expr to an rhs-expr, potentially recording
bindings
  (define (to-rhs! e lift-binding!)
    (syntax-parse e
      [((~datum let) ([x e]) body)
       (define e^ (to-rhs! #'e lift-binding!))
```

```
       (lift-binding! #'x e^)
       (to-rhs! #'body lift-binding!)]
      [(op a b)
       (define/syntax-parse a^ (to-immediate! #'a lift-binding!))
       (define/syntax-parse b^ (to-immediate! #'b lift-binding!))
       #'(op a^ b^)]
      [(~or ((~datum rkt) _)
            x:id
            n:number)
       this-syntax]))

  ;; full-expr (Identifier rhs-expr -> Void) -> immediate-expr
  ;; convert a full-expr to an immediate-expr, potentially
recording bindings
  (define (to-immediate! e lift-binding!)
    (syntax-parse e
      [(~or x:id n:number) this-syntax]
      [_
       (define/syntax-parse tmp (generate-temporary 'tmp))
       (define e^ (to-rhs! this-syntax lift-binding!))
       (lift-binding! #'tmp e^)
       #'tmp]))

  ;; rhs-expr (Listof (List Identifier rhs-expr)) -> anf-expr
  ;; wrap the innermost expression with `let`s for the
bindings that were recorded
  (define (wrap-lets e bindings)
    (match bindings
      [(cons binding bindings)
       (with-syntax ([x (first binding)]
                     [rhs (second binding)]
                     [body (wrap-lets e bindings)])
         #'(let ([x rhs]) body))]
      ['() e]))
```

Our transformation goes through the expression, recording the temporary variable bindings to lift. The final `rhs-expr` returned by `to-rhs` will be the body of the innermost `let` at the end of the transformation. Converting to an `rhs-expr` or an `immediate-expr` has the side effect of recording a binding pair to be lifted, and the result of replacing complex subexpressions with temporary variable references is returned from each helper.

Notice that the code generation pass is implemented as macro, while the intermediate passes are implemented as compile-time functions. Using a Racket macro for the code generator is convenient because it provide hygiene for any temporary names we introduce. For the intermediate passes we must use compile-time functions rather than macros, for three reasons:

- The intermediate passes do not generate Racket syntax that can be further expanded by the Racket macro expander. Instead, they generate code in our DSL's intermediate representation.
- Compiler passes may need additional arguments and return values, which may not be syntax objects. This is possible with a compile-time function, but not with a macro. For example, our A-normal form transformation receives the `bind!` procedure as an argument.
- Compiler passes may use side effects, and rely on a particular order of evaluation. For our A-normal form pass, we want to create `let`-bindings for the innermost subexpressions first. We accomplish this via the way we order calls to the `bind!` procedure.

A.1.3.3 *Pruning unused variables*

Using syntax-spec's symbol tables and binding operations, we can add an optimizing pass that removes unused variables.

For example:

```
(let ([x (+ 2 2)])
  (let ([y (+ 3 3)])
  x))
~>
(let ([x (+ 2 2)])
  x)
```

Since `y` is not referenced, we can just remove its definition from the program. Note that this optimization only makes sense when the right-hand-side of a definition is free of side-effects. For example, pruning `y` in this example would change the behavior of the program:

```
(let ([x (+ 2 2)])
  (let ([y (rkt (begin (displayln "hello!") (+ 3 3)))])
  x))
~>
(let ([x (+ 2 2)])
```

```
  x)
```

Without pruning, this would print something, but with pruning, it would not. Our optimization shouldn't change the behavior of the program. This DSL is designed with the requirement that `rkt` forms only have pure computations inside, but this cannot easily be checked. As such, we will assume Racket subexpressions are free of side effects, and our optimization will only be sound for side-effect-free Racket subexpressions.

```
(begin-for-syntax
  ;; anf-expr -> anf-expr
  ;; reconstruct the expression, excluding definitions of
unused variables
  (define (prune-unused-variables e)
    (define used-vars (get-used-vars e))
    (remove-unused-vars e used-vars))

  ;; anf-expr -> ImmutableSymbolSet
  ;; compute the set of used variables
  (define (get-used-vars e)
    (syntax-parse e
      [((~datum let) ([x e]) body)
       (define body-vars (get-used-vars #'body))
       (if (symbol-set-member? body-vars #'x)
           (symbol-set-union body-vars (get-used-vars #'e))
           body-vars)]
      [(op a b)
       (symbol-set-union (get-used-vars #'a) (get-used-vars #'b))]
      [x:id
       (immutable-symbol-set #'x)]
      [(~or ((~datum rkt) _) n:number) (immutable-symbol-set)]))

  ;; anf-expr ImmutableSymbolSet -> anf-expr
  ;; reconstruct the expression, excluding definitions of
specified unused variables
  (define (remove-unused-vars e used-vars)
    (syntax-parse e
      [((~and let (~datum let)) ([x e]) body)
       (define/syntax-parse body^
         (remove-unused-vars #'body used-vars))
       (if (symbol-set-member? used-vars #'x)
           #'(let ([x e])
               body^)
```

```
          #'body^)]
       [_ this-syntax])))
```

First, we figure out which variables are referenced, using a bottom-up traversal. We only include consider variables in the right-hand-side of a `let` used if we have determined that the variable bound by the `let` is used in its body. For now, we ignore references in Racket subexpressions.

Then, with that knowledge, we reconstruct the program, only including bindings for used variables.

This optimization is slightly simplified by having already transformed the program to A-normal form. We can see this in `remove-unused-vars`: We don't need to recur on the right-hand-side of a let-binding because we know there are no variable bindings to be removed from that expression.

A.1.3.4 *Putting it all Together*

Due to the nature of expansion and binding structure, some special care needs to be taken in sequencing multiple transformative compiler passes. Since our A-normal form transformation adds new bindings, we need to re-expand the result so syntax-spec can compute and check binding information for use in later passes/compilation:

```
(begin-for-syntax
  (define local-expand-anf (nonterminal-expander anf-expr)))

(define-syntax compile-expr
  (syntax-parser
    [(_ e)
     (define e/anf
       (local-expand-anf (to-anf #'e) #:should-rename? #t))
     (define e/pruned
       (prune-unused-variables e/anf))
     (define/syntax-parse e/pruned^
       (local-expand-anf e/pruned #:should-rename? #t))
     #'(compile-anf e/pruned^)]))
```

We perform this re-expansion using `nonterminal-expander`. This function expects DSL syntax of a specified nonterminal (here, `anf-expr`) and expands macros in the DSL code, checks binding structure, etc. It's kind of like `local-expand` but for a particular nonterminal. This is what happens in a host interface that

produces the expanded, core syntax that your compiler works with. We use `#:should-rename? #t` to ensure that we re-compile and rename identifiers in this expansion.

The expansion after pruning is technically unnecessary for this example since we are only removing bindings in that pass, but it is good to always make sure your compiler is receiving freshly expanding syntax. This extra expansion also makes sure your optimization produces valid syntax. In general, even if your compiler just has a single transformative pass before compilation, you should expand the result of the pass.

An additional caveat is that identifiers need to undergo the same number of expansions for things to work properly. The easiest way to do this is to expand only the entire DSL expression at once, rather than expanding subexpressions individually.

Finally, we must implement compilation of A-normal form expressions to Racket, which is straightforward:

```
(define-syntax compile-anf
  (syntax-parser
    [(_ ((~datum let) ([x e]) body))
     #'(let ([x (compile-anf e)]) (compile-anf body))]
    [(_ (op a b)) #'(op a b)]
    [(_ ((~datum rkt) e))
     #'(let ([x e])
         (if (number? x)
             x
             (error 'rkt "expected a number, got ~a" x)))]
    [(_ e) #'e]))

> (eval-expr 1)
1
> (eval-expr (let ([x 1]) (let ([y 2]) x)))
1
> (eval-expr (let ([unused (rkt (displayln "can anyone hear
me?"))])
    42))
42
```

To summarize the key points:

- We used compile-time functions for compiler passes, rather than macros.
- We can have multiple passes in a compiler simply by sequencing compile-time functions that operate on expanded DSL expressions.

- Since we have transformative passes in our compiler, we must re-expand resulting syntax using `nonterminal-expander` after each transformation.

A.1.4 *Advanced Tutorial: An Interpreted Language*

This guide demonstrates how to use syntax-spec to create an interpreted language, as well as the benefits of this approach.

Typically, syntax-spec is used to create languages that compile to Racket. However, it's possible to use it to create an interpreted language as well. In such an implementation, syntax-spec will enforce the grammar, check binding, macro-expand the source program, etc. and pass off the expanded, core syntax to an interpreter that evaluates it to a value. As an example, let's create an interpreted implementation of the lambda calculus.

A.1.4.1 *Expander*

Here is the syntax-spec:

```
#lang racket
(require syntax-spec (for-syntax syntax/parse))
(syntax-spec
  (binding-class lc-var #:binding-space lc)
  (extension-class lc-macro #:binding-space lc)
  (nonterminal lc-expr
    #:binding-space lc
    #:allow-extension lc-macro
    n:number
    (+ e1:lc-expr e2:lc-expr)
    x:lc-var
    (lambda (x:lc-var) e:lc-expr)
    #:binding (scope (bind x) e)
    (~> (e1 e2)
        (syntax/loc this-syntax (#%app e1 e2)))
    (#%app e1:lc-expr e2:lc-expr))

  (host-interface/expression
    (lc-expand e:lc-expr)
    #'#'e))
```

We have numbers, binary addition, variables, lambdas, and applications. The interesting bit is the host interface `lc-expand`. The result is the expanded syntax itself! For example:

```
> (lc-expand ((lambda (x) x) 1))
#<syntax:eval:4:0 (#%app (lambda (x) x) 1)>
```

The host-interface could also just invoke the interpreter directly on the syntax:

```
(host-interface/expression
    (lc e:lc-expr)
    #'(lc-eval #'e empty-env))
```

This is what you'd normally do, but we haven't implemented `lc-eval` yet so we'll stick with `lc-expand`.

Our language is also macro-extensible, so the result of `lc-expand` expands macros away:

```
(require (for-syntax syntax/parse))

(define-dsl-syntax let lc-macro
  (syntax-parser
    [(_ ([x:id e:expr]) body:expr)
     #'((lambda (x) body) e)]))

> (lc-expand (let ([x 1]) (+ x x)))
#<syntax:eval:6:0 (#%app (lambda (x) (+ x x)) 1)>
```

A.1.4.2 *Interpreter*

We can use the output of `lc-expand` as the input of an interpreter. But first, let's define some helpers.

We will be building a strict, environment-based interpreter for this language, so we need to define what our environment will look like.

The environment needs to map variables to values, and we have the result of syntax-spec expansion, so we can use symbol tables! This means our environment will map identifiers to values and will respect hygiene. syntax-spec gives us this benefit of hygienic environments for free, which is important. If we used a hash from symbols to values, which would contain no binding/hygiene information, an example like this would break:

```
(define-dsl-syntax m lc-macro
  (syntax-parser
    [(_ e) #'(let ([tmp 2]) e)]))
(lc (let ([tmp 1]) (m tmp)))
```

The macro-introduced `tmp` would shadow the surface syntax `tmp` and we'd get `2` instead of `1`.

```
(require (for-template syntax-spec))

(define empty-env (immutable-symbol-table))

(define (env-lookup env x)
  (if (symbol-table-has-key? env x)
      (symbol-table-ref env x)
      x))

(define (env-extend env x v)
  (symbol-table-set env x v))
```

One more thing we'll need is the ability to raise errors. Luckily, since we're operating on syntax, we can report the source location of an error.

```
(require racket/syntax-srcloc)

(define (lc-error stx msg)
  (define loc (syntax-srcloc stx))
  (if loc
      (raise-user-error (format "~a: ~a" (srcloc->string loc) msg))
      (raise-user-error 'lc msg)))

> (lc-error #'x "something went wrong")
eval:16:0: something went wrong
```

Alright, now let's define our interpreter:

```
(require syntax/parse)

(define-syntax-rule (lc e)
  (lc-eval (lc-expand e) empty-env))

(define (lc-eval stx env)
  (syntax-parse stx
```

```
    #:datum-literals (+ lambda #%app)
    [n:number
     (syntax->datum #'n)]
    [(+ e1 e2)
     (define v1 (lc-eval #'e1 env))
     (unless (number? v1)
       (lc-error this-syntax "+ expects number"))
     (define v2 (lc-eval #'e2 env))
     (unless (number? v2)
       (lc-error this-syntax "+ expects number"))
     (+ v1 v2)]
    [x:id
     (env-lookup env #'x)]
    [(lambda (x:id) e:expr)
     (lambda (v) (lc-eval #'e (env-extend env #'x v)))]
    [(#%app e1 e2)
     (match (lc-eval #'e1 env)
       [(? procedure? f)
        (f (lc-eval #'e2 env))]
       [_
        (lc-error this-syntax "applied non-function")])]))

> (lc 1)
1
> (lc (+ 1 1))
2
> (lc (let ([x 1]) (+ x x)))
2
> (lc ((lambda (x) (+ x 1)) 3))
4
> (lc (1 2))
eval:24:0: applied non-function
> (define-dsl-syntax m lc-macro
    (syntax-parser
      [(_ e) #'(let ([tmp 2]) e)]))
> (lc (let ([tmp 1]) (m tmp)))
1
```

Pretty cool!

To recap, we are using syntax-spec as a frontend for our interpreter, which operates on expanded syntax and uses symbol tables as an environment.

Here are some of the benefits of writing an interpreter in this style:

- We operate on syntax, which means we easily get source locations in errors and we can use syntax/parse to perform case analysis on expressions
- Our interpreter can assume that the program is grammatically valid and well-bound since we are operating on the result of expansion from syntax-spec
- We can use symbol tables for environments, which are hygienic
- Our language is macro-extensible and our interpreter only has to operate on core forms

A.1.4.3 *Supporting Racket Subexpressions*

We can add limited support for Racket subexpressions to our language:

```
(syntax-spec
  (nonterminal lc-expr
    #:binding-space lc
    #:allow-extension lc-macro
    n:number
    (+ e1:lc-expr e2:lc-expr)
    x:lc-var
    (lambda (x:lc-var) e:lc-expr)
    #:binding (scope (bind x) e)
    (rkt e:expr)
    (~> (e1 e2)

        (syntax/loc this-syntax (#%app e1 e2)))
    (#%app e1:lc-expr e2:lc-expr)))
```

We added `(rkt e:expr)` to the productions. Usually for racket expressions, we use `racket-expr`, which wraps the expression with `#%host-expression`. This does some work behind the scenes to make sure we can refer to DSL bindings in the Racket expression. But for this syntax interpreter, that won't work, so we'll just use `expr` to avoid wrapping the expression in a `#%host-expression`. Evaluation is simple:

```
(define (lc-eval stx env)
  (syntax-parse stx
    #:datum-literals (+ lambda #%app rkt)
    [n:number
     (syntax->datum #'n)]
```

```
     [(+ e1 e2)
      (define v1 (lc-eval #'e1 env))
      (unless (number? v1)
        (lc-error this-syntax "+ expects number"))
      (define v2 (lc-eval #'e2 env))
      (unless (number? v2)
        (lc-error this-syntax "+ expects number"))
      (+ v1 v2)]
     [x:id
      (env-lookup env #'x)]
     [(lambda (x:id) e:expr)
      (lambda (v) (lc-eval #'e (env-extend env #'x v)))]
     [(#%app e1 e2)
      (match (lc-eval #'e1 env)
        [(? procedure? f)
         (f (lc-eval #'e2 env))]
        [_
         (lc-error this-syntax "applied non-function")])]
     [(rkt e)
      (eval #'e)]))

> (lc (rkt (* 4 2)))
8
```

We just add a case that calls `eval` on the Racket expression! However, there are some limitations with this method. In particular, we have access to top-level names like `*`, but not local variables defined outside of the Racket subexpression, because `eval` is evaluating against the global namespace and not capturing local variable definitions.

```
> (define top-level-x 2)
> (lc (rkt top-level-x))
2
> (let ([local-x 3])
    (lc (rkt local-x)))
local-x: undefined;
 cannot reference an identifier before its definition
  in module: top-level
```

Similarly, we cannot reference `lc-var`s:

```
> (lc (let ([lc-x 4]) (rkt lc-x)))
lc-x: undefined;
```

cannot reference an identifier before its definition
in module: top-level

A.2 REFERENCE

```
(require syntax-spec-v3)
```
package: syntax-spec-v3

A.2.1 *Specifying Languages*

This section describes the syntax of the `syntax-spec` metalanguage, used to describe the grammar, binding structure, and host interface of a DSL.

Language specifications are made via the subforms of `syntax-spec`, which must be used at module-level.

```
(syntax-spec spec-def ...)

spec-def = binding-class
         | extension-class
         | nonterminal
         | host-interface
```

The following subsections address each kind of declaration allowed within the `syntax-spec` form.

A.2.1.1 *Binding classes*

Binding classes distinguish types of binding. When a reference resolves to a binder, it is an error if the binding class declared for the reference position does not match the binding class of the binding position.

```
(binding-class id binding-class-option ...)

binding-class-option = #:description string-literal
                     | #:binding-space space-symbol
                     | #:reference-compiler reference-compiler-expr
```

The `#:description` option provides a user-friendly phrase describing the kind of binding. This description is used in error messages.

The `#:binding-space` option specifies a binding space to use for all bindings and references declared with this class. Operationally, the binding space declaration

causes the syntax-spec expander to add the binding space scope to bindings and references. The scope is added to the scope sets of all binding occurrences. When parsing a reference position declared with a binding class that has an associated binding space, the name that is looked up is augmented with the binding class scope in order to give it access to bindings defined in the space.

1

The `#:reference-compiler` option specifies a reference compiler for controlling how references to variables of this binding class are treated in Racket code.

A.2.1.2 *Extension classes*

Extension classes distinguish types of extensions to languages. A syntax transformer is tagged with an extension class using `define-dsl-syntax`. Nonterminals can be declared extensible by a certain extension class using `#:allow-extension`. These extensions are expanded away into core DSL forms before compilation.

```
(extension-class id extension-class-option)

extension-class-option = #:description string-literal
                       | #:binding-space space-symbol
```

The `#:description` option provides a user-friendly phrase describing the kind of extension. This description is used in error messages.

The `#:binding-space` option specifies a binding space to use for all extensions with this class.

A.2.1.3 *Nonterminals*

```
(nonterminal id nonterminal-option production ...)
```

Defines a nonterminal supporting `let`-like binding structure.

Example:

```
(syntax-spec
  (binding-class my-var)
```

1 See §2.2.1 “Compiling references to DSL bindings within Racket code” for more information about reference compilers

```
  (nonterminal my-expr
    n:number
    x:my-var
    (my-let ([x:my-var e:my-expr] ...) body:my-expr)
    #:binding (scope (bind x) ... body)))
```

```
(nonterminal/nesting id (nested-id) nonterminal-
option production ...)
```

Defines a *nesting nonterminal* supporting nested, `let*`-like binding structure. Nesting nonterminals may also be used to describe complex binding structures like for `match`.

Example:

```
(syntax-spec
  (binding-class my-var)
  (nonterminal my-expr
    n:number
    x:my-var
    (my-let* (b:binding-pair ...) body:my-expr)
    #:binding (nest b ... body))
  (nonterminal/nesting binding-pair (nested)
    [x:my-var e:my-expr]
    #:binding (scope (bind x) nested)))
```

```
(nonterminal/exporting id nonterminal-option production ...)
```

Defines an *exporting nonterminal* which can export bindings, like `define` and `begin`.

Example:

```
(syntax-spec
  (binding-class my-var)
  (nonterminal/exporting my-defn
    (my-define x:my-var e:my-expr)
    #:binding (export x)

    (my-begin d:my-defn ...)
```

```
    #:binding [(re-export d) ...])
  (nonterminal my-expr
    n:number))
```

Nonterminal options

```
nonterminal-option = #:description string-literal
                   | #:allow-extension extension-class-spec
                   | #:binding-space space-symbol

extension-class-spec = extension-class-id
                     | (extension-class-id ...)
```

The `#:description` option provides a user-friendly phrase describing the kind of nonterminal. This description is used in error messages.

The `#:allow-extension` option makes the nonterminal extensible by macros of the given extension class(es).

The `#:binding-space` option specifies a binding space to use for all bindings and references declared with this nonterminal.

Productions

```
          production = rewrite-production
                     | form-production
                     | syntax-production

  rewrite-production = (~> syntax-pattern
                           pattern-directive ...
                           body ...+)

     form-production = (form-id . syntax-spec) maybe-binding-spec
                     | form-id

   syntax-production = syntax-spec maybe-binding-spec

  maybe-binding-spec = #:binding binding-spec
                     |
```

A *rewrite production* allows certain terms to be re-written into other forms. For example, you might want to tag literals:

```
(syntax-spec
  (nonterminal peg
    (~> (~or s:string s:char s:number s:regexp)
        #:with #%peg-datum (datum->syntax #'s '#%peg-datum)
        #'(#%peg-datum s))

    ...))
```

Rewrite productions don't have binding specs since they declare an expansion of surface syntax into a another DSL form. The don't necessarily have to expand into a core form like one declared by a form production or a syntax production. A rewrite production can expand into a DSL macro usage or another rewrite production.

Form productions and syntax productions declare core forms in the nonterminal which may have binding specs. If a binding spec is not provided, one is implicitly created. In this case, or if any spec variable is excluded from a binding spec in general, it will be treated as a reference position and implicitly added to the binding spec.

A form production defines a form with the specified name. You may want to use a syntax production if you are re-interpreting racket syntax. For example, if you are implementing your own `block` macro using syntax-spec:

```
(syntax-spec
  (nonterminal/exporting block-form
    #:allow-extension racket-macro

    ((~literal define-values) (x:racket-var ...) e:racket-expr)
    #:binding [(export x) ...]

    ((~literal define-syntaxes) (x:racket-macro ...) e:expr)
    #:binding (export-syntaxes x ... e)

    e:racket-expr))
```

When a form production's form is used outside of the context of a syntax-spec DSL, like being used directly in Racket, a syntax error is thrown.

A.2.1.4 *Syntax specs*

```
syntax-spec = ()
            | keyword
            | ...
            | ...+
            | (~literal id maybe-space)
            | (~datum id)
            | (syntax-spec . syntax-spec)
            | spec-variable-id:binding-class-id
            | spec-variable-id:nonterminal-id
            | spec-variable-id:extension-class-id

maybe-space = #:space space-name
            |
```

Syntax specs declare the grammar of a DSL form.

- `()` specifies an empty list.
- `keyword` specifies a keyword like `#:key` .
- `...` specifies zero or more repetitions of the previous syntax spec.
- `...+` specifies one or more repetitions of the previous syntax spec.
- `(~literal id maybe-space)` specifies a literal identifier and optionally allows the specification of a binding space for the identifier.
- `(~datum id)` specifies a datum literal.
- `(syntax-spec . syntax-spec)` specifies a pair of syntax specs.
- `spec-variable-id:binding-class-id` specifies an identifier sub-expression belonging to the specified binding class. For example: `x:my-var` specifies an identifier of the `my-var` binding class.
- `spec-variable-id:nonterminal-id` specifies a sub-expression that conforms to the specified nonterminal's grammar. For example: `e:my-expr` specifies a sub-expression that is a `my-expr`, where `my-expr` is a nonterminal name.

- `spec-variable-id:extension-class-id` specifies a sub-expression that is treated as a phase-1 transformer expression. This is used when your language has macro definitions.

A.2.1.5 *Binding specs*

```
binding-spec = spec-variable-id
             | (bind spec-variable-id)
             | (bind-syntax spec-variable-id spec-variable-id)
             | (bind-syntaxes spec-variable-id ooo ...
                 spec-variable-id)
             | (scope spec-or-ooo ...)
             | [spec-or-ooo ...]
             | (nest spec-variable-id ooo ... binding-spec)
             | (import spec-variable-id)
             | (export spec-variable-id)
             | (export-syntax spec-variable-id spec-variable-id)
             | (export-syntaxes spec-variable-id ooo ...
                 spec-variable-id)
             | (re-export spec-variable-id)

        ooo = ...

spec-or-ooo = binding-spec
             | ooo
```

Binding specs declare the binding rules of a DSL's forms. They allow us to control the scope of bound variables and to check that programs are well-bound before compilation. A binding spec is associated with a production and refers to spec variables from the production.

Similar to syntax patterns and templates, syntax specs and binding specs have a notion of ellipsis depth. However, all spec references in binding specs must have the exact same ellipsis depth as their syntax spec counterparts. Ellipses in binding specs are used to declare the scoping structure of syntax that includes sequences.

- For a `spec-variable-id` binding spec, if the specified sub-expression is an identifier (the syntax spec would be something like `x:my-var`), then the identifier is treated as a reference. If the specified sub-expression is

something else (the syntax spec would be something like `x:my-expr`), then the resulting binding structure depends on the content of the sub-expression. If a sub-expression of a production is excluded from its binding spec it is implicitly added as this type of binding spec.

- `(bind x)` declares that the variable specified by `x` is bound in the current scope. `bind` must be used inside of a `scope` and `x` must be specified with a binding class in its syntax spec like `x:my-var`.

  Example:

  ```
  (syntax-spec
    (binding-class my-var)
    (nonterminal my-expr
      n:number
      x:my-var
      (my-let ([x:my-var e:my-expr] ...) body:my-expr)
      #:binding (scope (bind x) ... body)))
  ```

  Notice how there are ellipses after the `(bind x)` since `x` occurred inside of an ellipsized syntax spec.

- `(bind-syntax x e)` declares that the variable specified by `x` is bound to the transformer specified by `e`. `bind-syntax` must be used inside of a `scope`, `x` must be specified with a binding class in its syntax spec like `x:my-var`, and `e` must be specified with an extension class in its syntax spec like `e:my-macro`.

  Example:

  ```
  (syntax-spec
    (binding-class my-var)
    (extension-class my-macro)
    (nonterminal my-expr
      #:allow-extension my-macro
      n:number
      (my-let-syntax ([x:my-var trans:my-macro]) body:my-expr)
      #:binding (scope (bind-syntax x trans) body)))
  ```

- `bind-syntaxes` is similar to `bind-syntax`, except it binds multiple identifiers.

  Example:

  ```
  (syntax-spec
    (binding-class my-var)
  ```

```
(extension-class my-macro)
(nonterminal my-expr
  #:allow-extension my-macro
  n:number
  (my-let-syntaxes ([(x:my-var ...) trans:my-macro])
    body:my-expr)
  #:binding (scope (bind-syntaxes x ... trans) body)))
```

Here, `trans` should evaluate to multiple transformers using `values`.

Note that the ellipses for `x` occur inside of the `bind-syntaxes`.

- `scope` declares that bindings and sub-expressions in the sub-specs are in a particular scope. Local bindings binding specs like `bind` must occur directly in a `scope` binding spec.

- `[spec ...]` is called a group, and it groups binding specs together. For example, when we write

```
(my-let ([x:my-var e:my-expr]) body:my-expr)
#:binding (scope (bind x) e)
```

We could instead write

```
(my-let ([x:my-var e:my-expr]) body:my-expr)
#:binding [(scope (bind x) body) e]
```

Which adds that `e` is a sub-expression outside of the scope of the `let`. All un-referenced syntax spec variables get implicitly added to a group with the provided binding spec, so the former example is equivalent to the latter.

Ellipses can occur after a binding spec in a group.

- `nest` is used with nesting nonterminals. In particular, the first argument to nest must be a spec variable associated with a nesting nonterminal. The second argument is treated as the "base case" of the "fold".

Example:

```
(syntax-spec
  (binding-class my-var)
  (nonterminal my-expr
    n:number
    x:my-var
    (my-let* (b:binding-pair ...) body:my-expr)
```

```
    #:binding (nest b ... body))
  (nonterminal/nesting binding-pair (nested)
    [x:my-var e:my-expr]
    #:binding (scope (bind x) nested)))
```

The `nest` binding spec sort of folds over the binding pairs. In this example, it'll produce binding structure like

```
[e1 (scope (bind x1) [e2 (scope (bind x2)
           [... [en (scope (bind xn) body)]])])]
```

`nest` does not necessarily have to be used with a sequence.

Example:

```
(syntax-spec
  (binding-class pattern-var)
  (nonterminal clause
    [p:pat e:racket-expr]
    #:binding (nest p e))
  (nonterminal/nesting pat (nested)
    x:pattern-var
    #:binding (scope (bind x) nested)
    ((~literal cons) car-pat:pat cdr-pat:pat)
    #:binding (nest car-pat (nest cdr-pat nested))))
```

However, the first arguemnt of `nest` cannot have ellipsis depth exceeding one.

- `(import d)` imports the bindings exported from the sub-expression specified by `d`. `import` must be used inside of a `scope` and must refer to a syntax spec associated with an exporting nonterminal.

  Example:

```
(syntax-spec
  (binding-class my-var)
  (nonterminal/exporting my-defn
    (my-define x:my-var e:my-expr)
    #:binding (export x)

    (my-begin d:my-defn ...)
    #:binding [(re-export d) ...])
  (nonterminal my-expr
```

```
n:number
(my-local [d:my-defn ...] body:my-expr)
#:binding (scope (import d) ... body)))
```

The argument to `import` cannot have ellipsis depth exceeding one.

- `(export x)` exports the variable specified by `x`. `x` must refer to a syntax spec variable associated with a binding class and `export` can only be used in an exporting nonterminal. See `import` for an example of usage.
- `export-syntax` is like `bind-syntax`, except it exports the binding instead of binding the identifier locally to the current scope. Like `export`, it can only be used in an exporting nonterminal.
- `export-syntaxes` is like `bind-syntaxes`, except it exports the bindings instead of binding the identifiers locally to the current scope. Like `export`, it can only be used in an exporting nonterminal.
- `(re-export d)` `export`s all bindings that are exported by `d`. `d` must be associated with an exporting nonterminal and `re-export` can only be used in an exporting nonterminal. See `import` for an example of usage.

There are several other constraints on binding specs:

- Specs of different categories cannot occur within the same ellipsis. The categories of specs are:
  - `refs+subexps` include references, `nest`, and `scope`.
  - `binds` include `bind`, `bind-syntax`, and `bind-syntaxes`.
  - `imports` include `import`.
  - `exports` include `export`, `export-syntax`, `export-syntaxes`, and `re-export`.

  For example, the spec `(scope [(bind x) e] ...)` is illegal since it mixes `refs+subexps` and `binds` in an ellipsis.
- `binds` and `imports` can only occur within a `scope`
- `exports` cannot occur within a scope.
- Within a scope, there can be zero or more `binds`, followed by zero or more `imports`, followed by zero or more `refs+subexps`.
- The second argument to nest must be `refs+subexps`.

- Spec variables can be used at most once. For example, `(scope (bind x) e e)` is illegal.

A.2.1.6 *Host interface forms*

Host interface forms are the entry point to the DSL from the host language. They often invoke a compiler macro to translate the DSL forms into Racket expressions.

```
(host-interface/expression
     (id . syntax-spec)
     maybe-binding-spec
     pattern-directive ...
     body ...+)
```

Defines a host interface to be used in expression positions.

Can only be used inside of a `syntax-spec` block.

An example from the miniKanren DSL:

```
(syntax-spec
  ...

  (host-interface/expression
    (run n:expr q:term-variable g:goal)
    #:binding (scope (bind q) g)

    #`(let ([q (var 'q)])
        (map (reify q)
             (run-goal n (compile-goal g))))))
```

This defines `run`, which takes in a Racket expression representing a number, a term variable, and a goal, and invokes the compiler `compile-goal` to translate the DSL forms into Racket.

```
(host-interface/definition
  (id . syntax-spec)
  maybe-binding-spec
  #:lhs
  [pattern-directive ...
   body ...+]
  #:rhs
  [pattern-directive ...
   body ...+])
```

Defines a host interface to be used in a definition context.

`#:lhs` runs before the right-hand-sides of definitions in the current context expand and must produce the identifier being defined.

`#:rhs` runs after the left-hand-sides of definitions and must produce the Racket expression whose value will be bound to the identifier (don't emit the definition syntax, just the syntax for producing the value).

Can only be used inside of a `syntax-spec` block.

An example from the miniKanren DSL:

```
(syntax-spec
  ...

  (host-interface/definition
    (defrel (name:relation-name x:term-variable ...) g:goal)
    #:binding [(export name) (scope (bind x) ... g)]

    #:lhs
    [(symbol-table-set!
      relation-arity
      #'name
      (length (syntax->list #'(x ...))))
     #'name]

    #:rhs
    [#`(lambda (x ...)
         (lambda (s)
           (lambda ()
             (#%app (compile-goal g) s))))]))
```

This defines `defrel`, which defines a relation. In the `#:lhs`, We record arity information about the identifier before producing it. Since the left-hand-sides all run before the right-hand-sides, even if there is mutual recursion, all arity information will be available before any goals are compiled. Note that the `#:rhs` produces a lambda expression, not a `define`.

```
(host-interface/definitions
     (id . syntax-spec)
     maybe-binding-spec
     pattern-directive ...
     body ...+)
```

Defines a host interface to be used in a definition context.

Can be used to produce multiple definitions.

Can only be used inside of a `syntax-spec` block.

An example from the PEG DSL:

```
(syntax-spec
  (host-interface/definitions
   (define-pegs [name:nonterm p:peg] ...)
   #:binding [(export name) ...]
   (run-leftrec-check! (attribute name) (attribute p))
   #'(begin (define name (lambda (in) (compile-peg p in)))
            ...)))
```

Unlike `host-interface/definition`, the definitions are directly produced by the host interface.

A.2.1.7 *Defining macros for DSLs*

```
(define-dsl-syntax name extension-class-id transformer-expr)
```

Defines a macro of the specified extension class. The transformer expression can be any Racket expression that evaluates to a `(-> syntax? syntax?)` procedure, so it is possible to use `syntax-rules`, `syntax-case`, `syntax-parse`, etc.

Example:

```
(define-dsl-syntax conj goal-macro
```

```
(syntax-parser
  [(_ g) #'g]
  [(_ g1 g2 g* ...) #'(conj (conj2 g1 g2) g* ...)]))
```

This defines a macro `conj` that expands to a goal in miniKanren.

A.2.1.8 *Embedding Racket syntax*

`racket-expr`

A nonterminal that allows arbitrary host language expressions. Such *host expressions* are wrapped with `#%host-expression` during DSL expansion. This nonterminal does not support definitions.

`racket-body`

A nonterminal that allows arbitrary host language expressions and definitions. This is an exporting nonterminal, so it must be explicitly mentioned in a binding spec, usually with `import`.

Example:

```
(syntax-spec
  (nonterminal my-expr
    (my-let ([x:racket-var e:racket-expr]) body:racket-body ...+)
    #:binding (scope (bind x) (import body) ...)))
```

`racket-var`

A binding class for host language bindings.

`racket-macro`

A binding class for arbitrary host language transformers.

A.2.2 *Compiling Languages*

A.2.2.1 *Compiling references to DSL bindings within Racket code*

2

2 §1.1.4 “Compilation” in the §1.1 “Basic Tutorial: State Machine Language” introduces the use of reference compilers.

By default, Racket code cannot reference names bound with DSL binding classes. To allow such references, specify a *reference compiler* for each class of bindings that should be usable in Racket code. The reference compiler is a syntax transformer that will be applied to compile the syntax including each reference.

When a reference appears in the head of a form, such as `x` in `(x 1 2)`, the reference compiler receives the entire form to transform. If the reference appears in a `set!`, the reference compiler will be invoked if it is a `set!-transformer?`; otherwise the expansion of the `set!` form results in an error.

In all cases, the reference identifier in the syntax provided to the reference compiler is a compiled identifier.

Reference compilers can be specified in the `#:reference-compiler` option of a `binding-class` form. For local control over reference compiler behavior, syntax parameters are recommended.

```
(make-variable-like-reference-compiler reference-stx
                                       [setter-stx])
 → set!-transformer?
  reference-stx : (or/c syntax? (-> identifier? syntax?))
  setter-stx : (or/c syntax? (-> syntax? syntax?)) = #f
```

Like `make-variable-like-transformer`, but works properly as a reference compiler that receives compiled identifiers.

If `reference-stx` is syntax, references expand to that syntax. If `reference-stx` is a procedure, it is called with the reference identifier to produce the reference's expansion.

If `setter-stx` is syntax, it should be syntax that evaluates to a procedure. The procedure will be invoked with the new value for the variable.

If `setter-stx` is a procedure, it is called with the entire `set!` expression to produce its expansion.

If `setter-stx` is not provided, references within `set!` position raise a syntax error.

When a reference is used in the head position of a form such as `(x 1 2)`, the variable-like reference compiler ensures that the form is parsed as a function application (not a macro call) and uses the `reference-stx` to expand only the identifier in head position.

Here is an example for a `match` DSL where pattern-bound variables cannot be mutated:

```
(syntax-spec
 (binding-class pat-var
   #:reference-compiler (make-variable-like-reference-compiler
                         (lambda (id) id)))
 (host-interface/expression
  (match target:racket-expr c:clause ...)
  #'(let ([target-pv target])
      (match-clauses target-pv c ...))))
```

Alternately we could provide `immutable-reference-compiler` as the reference compiler, which behaves the same.

`immutable-reference-compiler` : `set!-transformer?`

A variable-like reference compiler that allows references but raises a syntax error when identifiers are used in `set!` expressions.

References expand to their compiled identifier.

`mutable-reference-compiler` : `set!-transformer?`

A variable-like reference compiler that allows references as well as mutations via `set!` expressions.

References expand to their compiled identifier.

```
(#%host-expression rkt-expr)
```

Racket subexpressions are wrapped with `#%host-expression` during DSL expansion, which delays the expansion of the Racket subexpression until after compilation, allowing context like syntax parameters to be established by the compiler, which can be used by reference compilers.

A.2.2.2 *Compiled identifiers vs surface syntax*

The syntax of a DSL program in its initial, un-expanded, un-compiled state is called the *surface syntax*. During the expansion of a host interface usage, before your compiler is invoked, syntax-spec renames and compiles surface identifiers. The resulting identifiers are called *compiled identifiers* and have unique names and have special scopes according to your DSL's binding rules.

Another concept that comes up when discussing identifiers and compilation is positive vs negative space. This has to do with macro-introduction scopes. To ensure macro hygiene, the Racket expander distinguishes between syntax that was introduced by a macro and syntax that originated from elsewhere. To do this, it adds an introduction scope to the macro invocation's syntax, expands the invocation by running the macro's transformer, and then flips the scope, removing it from syntax that has it and adding it to syntax that doesn't.

During the expansion of the invocation, when a macro's transformer is running, the macro transformer will see this introduction scope on the incoming syntax. This syntax is in *negative space*. After the scope is flipped off on the result, the syntax is in *positive space*. It's important to note that positive vs negative space depends on the *current* introduction scope, as there may be many introduction scopes floating around. It is also possible to manually flip this scope in a transformer to convert syntax between positive and negative space.

A.2.2.3 *Symbol collections*

Symbol collections allow compilers to track information related to dsl variables. Symbol collections expect to receive compiled identifiers in negative space.

Symbol tables

```
(symbol-table? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a symbol table, `#f` otherwise.

```
(in-symbol-table table)
 → (sequence/c identifier? (or/c syntax-datum? syntax?))
  table : symbol-table?
```

Like `in-free-id-table`.

```
(mutable-symbol-table? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a mutable symbol table, `#f` otherwise.

```
(define-persistent-symbol-table id)
```

Defines a (mutable) symbol table for global use. For example, if your DSL has a static type checker and you're requiring typed identifiers between modules, you can store each identifier's type in a persistent symbol table.

Can only be used at the top-level of a module.

```
(local-symbol-table) → mutable-symbol-table?
```

Creates a (mutable) symbol table for local use.

```
(syntax-datum? v) → boolean?
  v : any/c
```

Roughly, returns `#t` if `v` is something that could be the result of `syntax->datum`, `#f` otherwise.

This includes pairs, vectors, symbols, numbers, booleans, etc.

```
(symbol-table-set!  table
                    id
                    v
                   [#:allow-overwrite? allow-overwrite?]) → void?
  table : mutable-symbol-table?
  id : identifier?
  v : (or/c syntax? syntax-datum?)
  allow-overwrite? : any/c = #t
```

Like `free-id-table-set!`. Errors by default when setting the value of an identifier already present in the table. Pass `#:allow-overwrite? #t` to allow this. However, persistent symbol tables do not support this flag.

```
(symbol-table-ref table id failure) → any/c
  table : symbol-table?
  id : identifier?
  failure : any/c
```

Like `free-id-table-ref`

```
(symbol-table-has-key? table id) → boolean?
  table : symbol-table?
  id : identifier?
```

Returns `#t` if `table` has an entry for `id`, `#f` otherwise.

```
(immutable-symbol-table? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is an immutable symbol table, `#f` otherwise.

```
(immutable-symbol-table) → immutable-symbol-table?
```

Creates an immutable, local symbol table. There are no persistent immutable symbol tables.

```
(symbol-table-set  table
                   id
                   v
                  [#:allow-overwrite? allow-overwrite?])
 → immutable-symbol-table?
  table : immutable-symbol-table?
  id : identifier?
  v : (or/c syntax? syntax-datum?)
  allow-overwrite? : any/c = #t
```

like `free-id-table-set`. Errors by default when setting the value of an identifier already present in the table. Pass `#:allow-overwrite? #t` to allow this.

```
(symbol-table-remove table id) → immutable-symbol-table?
  table : immutable-symbol-table?
  id : identifier?
```

like `free-id-table-remove`

Symbol sets

```
(symbol-set? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a symbol set, `#f` otherwise.

```
(in-symbol-set table) → (sequence/c identifier?)
  table : symbol-set?
```

Like `in-free-id-set`.

```
(mutable-symbol-set? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is a mutable symbol set, `#f` otherwise.

```
(define-persistent-symbol-set id)
```

Defines a (mutable) symbol set for global use like `define-persistent-symbol-table`.

```
(local-symbol-set id ...) → mutable-symbol-set?
  id : identifier?
```

Creates a local (mutable) symbol set containing the given identifiers.

```
(symbol-set-add! s id) → void?
  s : mutable-symbol-set?
  id : identifier?
```

Like `free-id-set-add!`

```
(symbol-set-member? s id) → boolean?
  s : mutable-symbol-set?
  id : identifier?
```

Like `free-id-set-member?`

```
(immutable-symbol-set? v) → boolean?
  v : any/c
```

Returns `#t` if `v` is an immutable symbol set, `#f` otherwise.

```
(immutable-symbol-set id ...) → immutable-symbol-set?
  id : identifier?
```

Creates a (local) immutable symbol set containing the given identifiers. There are no persistent immutable symbol sets.

```
(symbol-set-add s id) → immutable-symbol-set?
  s : immutable-symbol-set?
  id : identifier?
```

Like `free-id-set-add`

```
(symbol-set-remove s id) → immutable-symbol-set?
  s : immutable-symbol-set?
  id : identifier?
```

Like `free-id-set-remove`

```
(symbol-set-union s ...) → immutable-symbol-set?
  s : immutable-symbol-set?
```

Like `free-id-set-union`

```
(symbol-set-intersect s0 s ...) → immutable-symbol-set?
  s0 : immutable-symbol-set?
  s : immutable-symbol-set?
```

Like `free-id-set-intersect`.

```
(symbol-set-subtract s0 s ...) → immutable-symbol-set?
  s0 : immutable-symbol-set?
  s : immutable-symbol-set?
```

Like `free-id-set-subtract`.

A.2.2.4 *Binding Operations*

```
(compiled-identifier=? a-id b-id) → boolean?
  a-id : identifier?
  b-id : identifier?
```

Returns `#t` if the two compiled DSL identifiers correspond to the same binding from the surface syntax. Returns `#f` otherwise. Similar to `free-identifier=?`.

This is the equality used by symbol tables.

```
(free-identifiers  stx
                   [#:allow-host? allow-host?])
 → (listof identifier?)
  stx : syntax?
  allow-host? : boolean? = #f
```

Get a DSL expression's free identifiers (deduplicated).

Analysis of host expressions is currently not supported. When given syntax that contains a host expression, the operation raises an error if `allow-host?` is `#f`, or ignores that portion is syntax if `allow-host?` is `#t`.

```
(binding-identifiers  stx
                      [#:allow-host? allow-host?])
 → (listof identifier?)
  stx : syntax?
  allow-host? : boolean? = #f
```

Get a DSL expression's binding identifiers.

Analysis of host expressions is currently not supported. When given syntax that contains a host expression, the operation raises an error if `allow-host?` is `#f`, or ignores that portion is syntax if `allow-host?` is `#t`.

```
(alpha-equivalent?  stx-a
                    stx-b
                   [#:allow-host? allow-host?]) → boolean?
  stx-a : syntax?
  stx-b : syntax?
  allow-host? : boolean? = #f
```

Returns `#t` if the two DSL expressions are alpha-equivalent, `#f` otherwise.

Analysis of host expressions is currently not supported. When given syntax that contains a host expression, the operation raises an error if `allow-host?` is `#f`, or ignores that portion is syntax if `allow-host?` is `#t`.

```
(subst stx target replacement) → syntax?
  stx : syntax?
  target : syntax?
  replacement : syntax?
```

Substitutes occurences of (expressions `alpha-equivalent?` to) `target` with `replacement` in `stx`.

All arguments must be the result of DSL expansion, not just plain racket expressions.

In the case that `target` is an identifier from a binding position, references will be replaced by `replacement`.

Host expressions are left unchanged.

NOTE: In order to avoid hygiene issues, it may be necessary to re-expand using `nonterminal-expander` after substitution.

```
(get-racket-referenced-identifiers [binding-class-id ...] expr)
```

Returns an immutable symbol set containing identifiers of the specified binding classes that were referenced in racket (host) expressions in `expr`. If `expr` is not a host expression, an exception is raised.

A.2.2.5 *Expansion*

```
(nonterminal-expander nonterminal-id)

  nonterminal-id : identifier?
```

Produces an expander procedure for the specified nonterminal. This procedure expands macros down to the DSL's core forms, binds identifiers in binding positions, and can be configured to compile and rename identifiers. It does not expand host expressions.

Expander procedure has contract `(->* (syntax?) (#:should-rename? boolean?) syntax?)`. The default behavior is not to re-compile and re-rename identifiers. To do this, pass in `#:should-rename? #t`.

Can only be used with simple non-terminals.

Example:

```
> (module arithmetic racket
    (require syntax-spec-v3)
    (syntax-spec
      (extension-class arithmetic-macro)
      (nonterminal arithmetic
        #:allow-extension arithmetic-macro
        ((~literal +) a:arithmetic b:arithmetic)
        ((~literal *) a:arithmetic b:arithmetic)
        n:number))
    (define-syntax sqr
```

```
      (arithmetic-macro
       (syntax-rules ()
         [(sqr n) (* n n)])))
    (begin-for-syntax
      (define local-expand-arithmetic
        (nonterminal-expander arithmetic))
      (displayln (local-expand-arithmetic #'(sqr 1)))))
#<syntax:eval:1:0 (* 1 1)>
```

A.2.3 *Release Notes*

This package is periodically released as a package on the package server with a versioned package and collection name, like `syntax-spec-dev`. The unversioned package name `syntax-spec` is used for the current unstable development version.

Breaking changes may occur between differently-named versions. This page documents the history of breaking changes. Other new features are not mentioned here.

The version used in the paper "Compiled, Extensible, Multi-language DSLs (Functional Pearl)" was `syntax-spec-v2`.

Version 3

Binding specifications now require ellipses matching the ellipsis depth of pattern variables in the syntax spec. See the PR description for more details.

With this change the new `nest` syntax accomplishes the behavior of the old `nest` syntax when the first form is followed by ellipses and the behavior of the old `nest-one` syntax when no ellipses are used.

Reference compilers are now specified as part of binding class declarations, rather than with `with-reference-compilers`. If you previously used `with-reference-compilers` to create reference compilers with contextual behavior, you can typically use syntax parameters to accomplish the same with the new design.

Version 2

Some forms were renamed:

Old name	New name
`recursive`	`import`
`nonterminal/two-pass`	`nonterminal/exporting`

Scopes in binding specifications are now indicated by the `scope` form rather than `{}` curly braces.

Reference compilers are now invoked at application forms like `(x y z)` where `x` is the DSL reference. Use `make-variable-like-reference-compiler` if you only want to transform references in reference or `set!` positions.