acmqueue

Creating Languages in Racket

Sometimes you just have to make a better mousetrap.

Matthew Flatt, University of Utah

Choosing the right tool for a simple job is easy: a screwdriver is usually the best option when you need to change the battery in a toy, and grep is the obvious choice to check for a word in a text document. For more complex tasks, the choice of tool is rarely so straightforward—all the more so for a programming task, where programmers have an unparalleled ability to construct their own tools. Programmers frequently solve programming problems by creating new tool programs, such as scripts that generate source code from tables of data.

Since programmers often build task-specific tools, one way to make them more productive is to give them better tool-making tools. When tools take the form of program generators, this idea leads to libraries for creating languages that are directly extensible. Programmers may even be encouraged to think about a problem in terms of a language that would better support the task. This approach is sometimes called *language-oriented programming*.[3]

Racket is both a programming language and a framework for building programming languages. A Racket program can contain definitions that extend the syntax of the language for use later in the same program, and language extensions can be packaged as modules for use in multiple programs. Racket supports a smooth path from relatively simple language extensions to completely new languages, since a programming tool, like any other piece of software, is likely to start simple and grow as demands on the language increase.

As an example task, consider the implementation of a text-adventure game (also known as interactive fiction), where a player types commands to move around in a virtual world and interact with objects:

```
You're standing in a meadow.
There is a house to the north.
> north
You are standing in front of a house.
There is a door here.
>  open door
The door is locked.
>
```

To make the game interesting, a programmer must populate the virtual world with places and things that have rich behavior. Most any programming language could implement this virtual world, but choosing the right language construct (i.e., the right tool) to represent each game element is a crucial step in the development process.

The right constructs allow commands, places, and things to be created easily—avoiding error-

prone boilerplate code to set up the world's state and connections—while also allowing the use of a programming language's full power to implement behaviors.

In a general-purpose programming language, no built-in language construct is likely to be a perfect fit. For example, places and things could be objects, while commands could be implemented as methods. The game's players, however, don't call methods but instead type commands that have to be parsed and dynamically mapped to responses for places and things. Similarly, saving and loading a game requires inspecting and restoring the state of places and things, which is partly a matter of object serialization but also of setting variables to unmarshaled values (or else using an indirection through a dictionary for each reference from one object to another).

Some programming languages include constructs—such as overloading or laziness—that a clever programmer can exploit to encode a domain-specific language. The design of Racket addresses the problem more directly; it gives programmers tools to explicitly extend the programming language with new syntax. Some tasks require only a small extension to the core language, while others benefit from the creation of an entirely new language. Racket supports both ends of the spectrum, and it does so in a way that allows a smooth progression from one end to the other. As a programmer's needs or ambitions grow for a particular task, the programmer can take advantage of ever more of Racket's unified framework for language extension and construction.

The text-adventure example presented here illustrates the progression from a simple embedding in Racket to a separate domain-specific language (including IDE support for syntax coloring), explaining relevant Racket details along the way; no prior knowledge of Racket is necessary. Readers who prefer a more complete introduction to the language should consult The Racket Guide.[1]

The example is a "toy" in multiple senses of the world, but it is also a scale model of industry practice. Most every video-game developer uses a custom language, including the Racket-based language that is used to implement content for the *Uncharted* video-game series.[2] Evidently, when billions of entertainment dollars are on the line, the choice of programming language matters—even to the point of creating new, special-purpose languages.

THE WORLD IN PLAIN RACKET

Our text adventure game contains a fixed set of *places*, such as a meadow, house, or desert, and a fixed set of *things*, such as a door, key, or flower. The player navigates the world and interacts with things using commands that are parsed as either one or two words: a single verb (i.e., an intransitive verb, since it does not have a target object), such as *help* or *look*; or a verb followed by the name of a thing (i.e., a transitive verb followed by a noun), such as *open door* or *get key*. Navigation words such as *north* or *in* are treated as verbs. A user can save the game using the *save* and *load* verbs, which work everywhere and prompt the user for a file name.

To implement a text-adventure game in Racket, you would start by declaring structure types for each of the three game elements:

```
(struct verb
 (aliases             ;  list of symbols
  Desc                ;  string
  transitive?))       ;  Boolean

(struct thing
 (name                ;  symbol
  [state #:mutable]   ;  any value
  actions))           ;  list of verb-function pairs

(struct place
 (desc                ;  string
  [things #:mutable] ;  list of things
  actions))           ;  list of verb-function pairs
```

Racket is a dialect of Lisp and a descendant of Scheme, so its syntax uses parentheses and a liberal grammar of identifiers (e.g., `transitive?` is an identifier). A semicolon introduces a newline-terminated comment. Square brackets are interchangeable with parentheses but are used by convention in certain contexts, such as grouping a field name with modifiers. The `#:mutable` modifier declares a field as mutable, since fields are immutable by default.

The first `struct` form in the code binds verb to function, taking one argument for each field and creating a verb instance. For example, you can define a `south` verb with alias `s` as

```
(define south (verb (list 'south 's) "go south" #false))
```

Lisp and Racket programs tend to use strings for the text that is to be shown to an end user—for example, verb descriptions such as `"go south"`. A *symbol*, written with a leading single quote (e.g., `'south`), is more typically used for an internal name, such as a verb alias.

Given the definition of `south` and a thing, `flower`, you could define a `meadow` place where the `south` verb moves the player to a `desert` place:

```
(define meadow (place "You're in a meadow."
                      (list flower)
                      (list (cons south
                                  (lambda () desert)))))
```

The `list` function creates a list, while `cons` pairs two values. The `cons` function usually pairs an element with a list to form a new list, but here `cons` is used to pair a verb with a function that implements the verb's response. The lambda form creates an anonymous function, which in this case expects zero arguments.

When a verb's response function produces a place, such as `desert` in the example, the game execution engine will move the player to the returned place. The game engine's support for saving

and loading game state, meanwhile, requires a mapping between places and their names. (Places can be implemented as objects that can be serialized, but restoring a game requires both deserialization and updating Racket-level variables such as `meadow`.) The `record-element!` function implements mappings between names and places:

```
(define names (make-hash))    ; symbol to place/thing
(define elements (make-hash)) ; place/thing to symbol

(define (record-element! name val)
  (hash-set! names name val)
  (hash-set! elements val name))

(define (name->element name) (hash-ref names name))
(define (element->name obj) (hash-ref elements obj))
```

Consequently, the complete implementation of the meadow is:

```
(define meadow (place ....)) ; as above
(record-element! 'meadow meadow)
```

Things must be defined and registered in much the same way as places. Verbs must be collected into a list to be used by the game's command parser. Finally, the parsing and execution engine needs a set of verbs that work everywhere, each with its response function. All of those pieces form the interesting part of the game implementation, while the parsing and execution engine is a few dozen lines of static infrastructure.

The complete game implementation is available online:

http://queue.acm.org/downloads/2011/racket/0-longhand/txtadv+world.rkt
http://queue.acm.org/downloads/2011/racket/0-longhand/README.txt

Note that the code needed to construct the virtual world is particularly verbose.

SYNTACTIC ABSTRACTION

Although the data-representation choices of the previous section are typical for a Racket program, a Racket programmer is unlikely to write the repetitive code that directly defines and registers places, since it includes so many boilerplate `lists`, `conses`, and `lambdas`. Instead, a Racket programmer would write

```
(define-place meadow
  "You're in a meadow."
  [flower]
  ([south  desert]))
```

and would add a `define-place` form to Racket using a pattern-based macro. The simplest form of such a macro uses `define-syntax-rule`:

```
(define-syntax-rule (define-place id desc [thng] ([vrb expr]))
  (begin
    (define id (place desc
                      (list thng )
                      (list (cons vrb (lambda () expr )))))
    (record-element! 'id id )))
```

The form immediately after `define-syntax-rule` is a *pattern*, and the form after the pattern is a *template*. A use of a macro that matches its pattern is replaced by the macro's template, modulo substitutions of *pattern variables* for their matches. The *id*, *desc*, *thng*, *vrb*, and *expr* identifiers in this pattern are pattern variables.

Note that the `define-place` form cannot be a function. The `desert` expression after `south` is, in general, an expression whose evaluation must be delayed until the `south` command is entered. More significantly, the form should bind the variable `meadow` so that Racket expressions for commands can refer to the place directly. In addition, the variable's source name (as opposed to its value) is used to register the place in the table of elements.

The `define-place` macro so far matches exactly one thing in a place and exactly one verb and response expression. To generalize to any number of things, verbs, and expressions, you add ellipses to the pattern:

```
(define-syntax-rule (define-place id desc
                      [thng ...]
                      ([vrb expr ] ...))
  (begin
    (define id (place desc
                      (list thng ...)
                      (list (cons vrb (lambda () expr ))
                            ...)))
    (record-element! 'id id )))
```

Ellipses work in the obvious way, and with this generalized `define-place`, you can put both a cactus and a key initially in the desert and respond to direction verbs other than `north` by staying in the desert:

```
(define-place desert
  "You're in a desert."
  [cactus key]
  ([north meadow]
   [south desert]
   [east desert]
   [west desert]))
```

The macro for a thing is similarly straightforward:

```
(define-syntax-rule (define-thing id
                      [vrb expr ]  ...)
  (begin
    (define id
      (thing 'id #false (list (cons vrb (lambda () expr )) ...)))
    (record-thing! 'id id )))
```

Verbs are slightly trickier, because you want to make simple verbs especially compact to specify, and you need one kind of pattern for intransitive verbs and another for transitive verbs. The following example illustrates the target syntax:

```
(define-verbs all-verbs
  [quit]
  [north (= n) "go north"]
  [knock _]
  [get _ (= grab take) "take"])
```

This example defines four verbs: `quit` as an intransitive verb with no aliases; `north` as an intransitive verb with alias `n` and a preferred description `go north`; `knock` as a transitive verb (as indicated by the underscore) with no aliases; and `get` as a transitive verb with aliases `grab` and `take` and preferred description `take`. Finally, all of these verbs are collected into a list that is bound to `all-verbs` for use by the game's command parser.

Implementing the `define-verbs` form requires a more general kind of pattern matching to support different shapes of verb specifications and to match `=` and `_` as literals. An implementation of `define-verbs` can defer the work of handling an individual verb to a `define-one-verb` macro, which uses `define-syntax` and `syntax-rules`:

```
(define-syntax define-one-verb
  (syntax-rules (= _)
    [(one-verb id (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #false))]
    [(one-verb id _ (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #true))]
    [(one-verb id)
     (define id (verb (list 'id ) (symbol->string 'id ) #false))]
    [(one-verb id _)
     (define id (verb (list 'id ) (symbol->string 'id ) #true))]))
```

The `=` and `_` in parentheses after `syntax-rules` indicate that `=` and `_` are literals, rather than pattern variables, in the patterns that follow. Each pattern afterward has a corresponding template.

Thus, in

```
(define-verbs all-verbs
  ....
  [get _ (= grab take) "take"])
```

the `define-verbs` expansion turns the last clause into a `define-one-verb` use:

```
(define-one-verb get _ (= grab take) "take")
```

This matches the first pattern of `one-verb` and expands into:

```
(define get (verb (list 'get 'grab 'take) "take" #true))
```

Finally, a `define-everywhere` form is created for defining verb responses that work throughout the world, as needed for verbs such as `save` and `load`:

```
(define-syntax-rule (define-everywhere id ([vrb expr ] ...))
  (define id (list (cons vrb (lambda () expr )) ...)))

(define-everywhere everywhere-actions
  ([quit (begin (printf "Bye!\n") (exit))]
   [save (save-game)]
   [load (load-game)]
   ....))
```

The `define-place`, `define-thing`, and `define-verb` macros are examples of *syntactic abstraction*. They abstract over repeated patterns of syntax, so that a programmer can avoid boilerplate code and concentrate on the creation of interesting verbs, places, and things.

The revised game implementation, which has a compact and readable implementation of the virtual world, is available online:

http://queue.acm.org/downloads/2011/racket/1-monolith/txtadv+world.rkt
http://queue.acm.org/downloads/2011/racket/1-monolith/README.txt

SYNTACTIC EXTENSION

A Racket programmer who is interested in writing a single text-adventure game would likely stop extending the language at this point. If the text-adventure engine should be reusable for multiple worlds, however, a Racket programmer is likely to take a step beyond syntactic abstraction to *syntactic extension*.

The difference between abstraction and extension is partly in the eye of the beholder, but extension suggests that functions such as `place` and `record-element!` can be kept private, while `define-place` is exported for use in the world-defining module with implementation-independent

semantics. In the world-defining module, macros such as `define-place` have the same status as built-in forms such as `define` and `lambda`.

To make this shift, you can put the `define-verbs`, `define-place`, `define-thing`, and `define-everywhere` definitions in their own module, called world.rkt.

```
#lang racket
(require "txtadv.rkt")

(define-verbs ....)
(define-everywhere ....)
(define-thing ....) ...
(define-place ....) ...
```

This module imports txtadv.rkt, which exports `define-verbs`, etc., as well as functions used in verb responses such as `save-game` and `load-game`. Meanwhile, txtadv.rkt keeps private the structures and other functions that implement the world data types.

```
#lang racket
(provide define-verbs define-thing
         define-place define-everywhere

         save-game
         load-game
         ....)

(struct verb ....)
....
(define-syntax-rule (define-verbs ....) ....)
....
```

The `#lang racket` line that starts each module indicates that the module is implemented in the `racket` language. In world.rkt, `require` additionally imports both the syntactic extensions and functions that are exported by the txtadv.rkt module.

Since macro binding is part of the Racket language, as opposed to being implemented as a separate preprocessor, macro bindings can work with module imports and exports in the same way as variable bindings. In particular, the definition of the `define-verbs` macro can see the `verb` constructor function because of the rules of lexical scope, while code in the world.rkt module cannot access `verb` directly because of the same scoping rules. Since a use of `define-verbs` in world.rkt expands to a use of `verb`, considerable language machinery is required for Racket to maintain lexical scope in the presence of macro expansion, but the result is that syntactic extension is easy for programmers.

The modular game implementation is available online:
http://queue.acm.org/downloads/2011/racket/2-modules/txtadv.rkt
http://queue.acm.org/downloads/2011/racket/2-modules/world.rkt
http://queue.acm.org/downloads/2011/racket/2-modules/README.txt

MODULE LANGUAGES

Although the world.rkt module cannot directly access constructor functions such as `verb`, the module still has access to all of the Racket language and, via `require`, any other module's exports. More constraints on world.rkt may be appropriate to ensure that assumptions of txtadv.rkt are satisfied.

To exert further control, you can convert txtadv.rkt from a module that exports a language extension to one that exports a language. Then, instead of starting with `#lang racket`, world.rkt starts with

```
#lang s-exp "txtadv.rkt"
```

For now, `s-exp` indicates that the language of world.rkt uses S-expression notation (i.e., parentheses), while txtadv.rkt defines syntactic forms. Later, the S-expression and syntactic-form specifications are combined into a single name, analogous to `#lang racket`.

Along with changing world.rkt, you can change txtadv.rkt to export everything from racket:

```
#lang racket
(provide define-verbs ....
         (all-from-out racket))
....
```

Instead of `(all-from-out racket)`, you could use `(except-out (all-from-out racket) require)` to withhold the `require` form from world.rkt. Alternatively, instead of using `all-from-out` and then naming bindings to withhold, you could explicitly export only certain pieces from racket.

The exports of txtadv.rkt completely determine the bindings that are available in world.rkt—not only the functions, but also syntactic forms such as `require` or `lambda`. For example, txtadv.rkt could supply a `lambda` binding to world.rkt that implements a different kind of function than the usual `lambda`, such as functions with lazy evaluation.

More commonly, a module language can replace the `#%module-begin` form that implicitly wraps the body of a module. Specifically, txtadv.rkt can provide an alternate `#%module-body` that forces world.rkt to have a single `define-verbs` form, a single `define-everywhere` form, a sequence of `define-thing` declarations, and a sequence of `define-place` declarations; if world.rkt has any other form, it can be rejected as a syntax error. Such constraints can enforce restrictions to limit the power of the txtadv.rkt language, but they can also be used to provide domain-specific checking and error messages.

The game implemented with a txtadv.rkt language is available online:

http://queue.acm.org/downloads/2011/racket/3-module-lang/txtadv.rkt
http://queue.acm.org/downloads/2011/racket/3-module-lang/world.rkt
http://queue.acm.org/downloads/2011/racket/3-module-lang/README.txt

The `#%module-begin` replacement in the implementation requires `define-verbs` followed by `define-everywhere`, then allows any number of other declarations. The module must end with a place expression, which is used as the starting location for the game.

STATIC CHECKS

The `define-verb`, `define-place`, and `define-thing` forms bind names in the same way as any other Racket definition, and each reference to a verb, place, or thing is a Racket-level reference to the defined name. This approach makes it easy for `verb-response` expressions, which are implemented in Racket, to refer to other things and places in the virtual world. It also means, however, that misusing a reference as a thing can lead to a runtime error. For example, the incorrect reference to `desert` as a thing in

```
(define-place room
  "You're in the house."
  [trophy desert]
  ([out  house-front]))
```

triggers a failure only when the player enters `room`, and the game engine fails when trying to print the things within the place.

Many languages provide type checking or other static types to ensure the absence of certain runtime errors. Racket macros can implement languages with static checks, and macros can even implement language extensions that perform static checks within a base language that defers similar checks to runtime. Specifically, you can adjust `define-verb`, `define-place`, and `define-thing` to check certain references, such as requiring that the list of initial things in a place contain only names that are defined as things. Similarly, names used as verbs with responses can be checked to ensure that they are declared as verbs, suitably transitive or intransitive.

Implementing static checks typically requires macros that are more expressive than pattern-matching macros. In Racket, arbitrary compile-time code can perform the role of expander for a syntactic form, because the most general form of a macro definition is

```
(define-syntax id transformer-expr )
```

where `transformer-expr` is a compile-time expression that produces a function. The function must accept one argument, which is a representation of a use of the `id` syntactic form, and the function must produce a representation of the use's expansion. In the same way that `define-syntax-rule` is shorthand for `define-syntax` plus `syntax-rules` and a single pattern, `syntax-rules` is shorthand for a function of one argument that pulls apart expressions of a certain shape (matching a pattern) and constructs a new expression for the result (based on a template).

The compile-time language that is used for *`transformer-expr`* can be different from the surrounding runtime language, but `#lang racket` seeds the language of compile-time expressions with essentially the same language as for runtime expressions. New bindings can be introduced to the compile-time phase with `(require (for-syntax ....))` instead of just `require`, and local bindings can be added to the compile-time phase through definitions wrapped with `begin-for-syntax`.

For example, to check for verbs, things, and places statically, `begin-for-syntax` can define a new typed structure:

```
(begin-for-syntax
  (struct typed
    (id      ; an identifier
      type) ; a string
    #:property prop:procedure (lambda (self stx) (typed-id self))))
```

An *identifier* is written as a symbol, but with a # prefix, so that

```
(typed #'gen-desert "place")
```

associates the binding `gen-desert` to the type "place". The `#:property prop:procedure` clause in the declaration of `typed` makes a typed instance act as a function (for reasons explained later). The function takes one argument in addition to the implicit `self` argument, but it ignores the argument and returns the `typed` instance's `id`.

You can use `typed` by changing the `define-place` form to bind a place name *id* to a compile-time `typed` record. At the same time, `define-place` binds a generated name *gen-id* to the runtime place record:

```
(define-syntax-rule (define-place id ....)
  (begin
    (define gen-id (place ....)) ; as before
    (define-syntax id (typed #'gen-id "place"))
    (record-element! 'id id )))
```

Since a `typed` record acts as a function, a use of *id* expands to *gen-id* , so *id* still can be used as a direct reference to the `place`. At the same time, other macros can look at the *id* binding and determine that its expansion will have the type "place".

Other macros inspect types by using a `check-type macro`. The implementation of `check-type` is in the complete code online, but its essential feature is that it uses a compile-time function `syntax-local-value` to obtain the compile-time value of an identifier; the `check-type` macro then uses `typed?` to check whether the compile-time value is a type declaration, in which case it uses `typed-type` to check whether the declared type is the expected one. As long as the type check passes, `check-type` expands to its first argument.

The `define-place` macro uses `check-typed` to check whether the list of things at the `place` contains only names that are defined as things. The `define-place` macro also uses `check-typed` to check whether verbs that have responses in the `place` are defined as intransitive verbs:

```
(define-syntax-rule (define-place id
                      desc
                      [thng  ...]
                      ([vrb  expr ]  ...))
```

```
(begin
  (define gen-id
    (place desc
           (list (check-type thng "thing")  ...)
           (list (cons (check-type vrb "intransitive verb")
                       (lambda () expr ))
                 ...)))
  (define-syntax id (typed #'gen-id "place"))
  (record-element! 'id id )))
```

The `define-one-verb` macro must change to similarly declare each verb as either type "`transitive verb`" or "`intransitive verb`". The `define-thing` macro changes to declare its binding as a "`thing`", and it checks that each handled verb is defined as a "`transitive verb`".

The code for the game with static checks is available online:

http://queue.acm.org/downloads/2011/racket/4-type/txtadv.rkt
http://queue.acm.org/downloads/2011/racket/4-type/world.rkt
http://queue.acm.org/downloads/2011/racket/4-type/README.txt

The implementation of `check-form` uses `syntax-case`, which provides the pattern-matching functionality of `syntax-rules`, but pairs each pattern with an expression rather than a fixed template.

NEW SYNTAX

A Racket programmer who defines a custom text-adventure language for other Racket programmers is especially likely to stop at this point. If the text-adventure language is to be used by others who are less familiar with Racket, however, a different notation may be appropriate. For example, others may prefer a notation such as the following from world.rkt:

```
#lang  reader  "txtadv-reader.rkt"

===VERBS===
north, n
  "go north"
get _, grab _, take _
  "take"
....
===EVERYWHERE===
save
  (save-game)
load
  (load-game)
```

```
....
===THINGS===
---cactus---
 get
  "Ouch!"
....
===PLACES===
---desert---
"You're in a desert."
  [cactus, key]
north   start
south   desert
....
```

In this notation, instead of forms such as `define-verbs` and `define-everywhere`, sections of the program are introduced by tags such as `===VERBS===` and `===EVERYWHERE===`. Names in the `===VERBS===` section implicitly define verbs, listing aliases afterward through a comma-separated sequence followed by an optional description of the verb. Similarly, each name in the `===EVERYWHERE===` section implicitly defines the response to a verb; the responses are still written as Racket expressions, but they could be in any alternate notation, if desired. Each thing and place is defined by its own subsection, such as `---cactus---`, with per-object verb responses in the same way as in `===EVERYWHERE===`.

Non-S-expression syntax is enabled in world.rkt by starting with `#lang reader "txtadv-reader.rkt"` instead of `#lang s-exp "txtadv.rkt"`. The `reader` language constructor, unlike the `s-exp` language constructor, defers parsing of the program's text to an arbitrary parsing function that is exported by the named module, which in this case is `txtadv-reader.rkt`. The parser from `txtadv-reader.rkt` is responsible for processing the rest of the text and converting it into S-expression notation, including the introduction of `txtadv.rkt` as the module language for the parsed `world.rkt` module.

More precisely, a reader function parses input into a syntax object, which is like an S-expression that is enriched with lexical-context and source-location information. It also acts as the representation of code for macro-transformer arguments and results. The syntax-object abstraction provides a clean separation of character-level parsing and tree-structured macro transformations. The source-location part of a syntax object automatically connects the result of macro expansion back to the original source; if a runtime error occurs in the code generated from `world.rkt`, then the error can point back to the relevant source.

The game code with nonparentheses syntax is available online:

http://queue.acm.org/downloads/2011/racket/5-lang/txtadv-reader.rkt
http://queue.acm.org/downloads/2011/racket/5-lang/txtadv.rkt
http://queue.acm.org/downloads/2011/racket/5-lang/world.rkt
http://queue.acm.org/downloads/2011/racket/5-lang/README.txt

The parser in `txtadv-reader.rkt` is implemented in an especially primitive way with regular expressions. The Racket distribution includes better parsing tools such as Lex- and Yacc-style parser generators.

IDE SUPPORT

One of the benefits of S-expression notation is that a programming environment's functionality adapts easily to syntactic extension, since syntax coloring and parentheses matching can be independent of macro expansion. Some of those benefits are intact with the new syntax for describing a world, since the parser keeps source locations with identifiers and since the code ultimately expands to Racket-level binding forms. For example, the Check Syntax button in DrRacket can automatically draw arrows from the binding instance of `cactus` to each bound use of `cactus`.

DrRacket needs more help from the language implementer for IDE features, such as syntax coloring, that depend on the character-level syntax of the language. Filling in this piece of the sample text-adventure language takes two steps:

1. Install the language's reader as a `txtadv` library collection instead of relying on a relative path such as `txtadv-reader.rkt`. Moving to the namespace of library collections allows DrRacket and the program to agree on which language is being used (without requiring project-style configuration of the IDE).
2. Add a function to the `txtadv` reader module that identifies additional support for the language, such as a module that implements on-the-fly syntax coloring. Again, since DrRacket and the module use the same specification of the module's language, the syntax color can be precisely tailored to the module's language and content.

The code for the game with a DrRacket plug-in for syntax coloring is available online:

http://queue.acm.org/downloads/2011/racket/6-color/txtadv.rkt
http://queue.acm.org/downloads/2011/racket/6-color/world.rkt
http://queue.acm.org/downloads/2011/racket/6-color/README.txt
http://queue.acm.org/downloads/2011/racket/6-color/lang/color.rkt
http://queue.acm.org/downloads/2011/racket/6-color/lang/reader.rkt

This plug-in colors the program according to the game language's syntax instead of Racket's default rules, highlighting lexical syntax errors in red.

MORE LANGUAGES

The source code of the Racket distribution includes dozens of unique `#lang` lines. The most common is `#lang racket/base`, a stripped-down variant of `#lang racket`. Other common lines include `#lang scribble/manual` for documentation sources, `#lang racket/unit` for externally linkable components, `#lang scheme` for legacy modules, and `#lang setup/infotab` for library metadata. Most Racket languages use S-expression notation, but `scribble/manual` is a notable exception; even parentheses-loving Racketeers concede that an S-expression is a poor notation for documentation prose.

Different languages in the Racket distribution exist for different reasons, and they use Racket's language-creation facilities to different degrees. Racket developers do not create new languages lightly, but the benefits of a new language sometimes outweigh the cost of learning a language variant. These benefits are as readily available to Racket users as to the core Racket developers.

Racket's support for S-expression languages and language extensions is particularly rich, and the examples in this article only scratch the surface of that toolbox. Racket's toolbox for non-S-expression syntax is still evolving, especially with respect to composable parsers and language-triggered IDE plug-ins. Fortunately, Racket's `#lang` protocol moves most of the remaining work out of the core system and into libraries. This means that Racket users are as empowered as core Racket developers to develop improved syntax tools.

REFERENCES

1. Flatt, M., Findler, R. B., PLT. 2011. The Racket Guide.; http://docs.racket-lang.org/guide.
2. Liebgold, D. 2011. Functional mzScheme DSLs in game development. Presented at Commercial Users of Functional Programming.
3. Ward, M. 1994. Language-oriented programming. *Software – Concepts and Tools 15*(4): 147–161.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

MATTHEW FLATT is an associate professor in the school of computing at the University of Utah, where he works on extensible programming languages, runtime systems, and applications of functional programming. He is one of the developers of the Racket programming language and a coauthor of the introductory programming textbook *How to Design Programs* (MIT Press, 2001).