Implementing HtDP Teachpacks, Libraries, and Customized Teaching Languages

Version 9.4.0.5

November 16, 2026

DrRacket has two different mechanisms for making available additional functions and functionality to students using the teaching languages.

- HtDP Teachpacks are added to a student's program by clicking on the "Language" menu and selecting "add Teachpack". Students can then install a new Teachpack by clicking "Add Teachpack to List" and choosing the Teachpack file from the filesystem.
- HtDP Libraries are brought into the student's program using a `require` statement.

Under the hood, HtDP Teachpacks and HtDP Libraries are implemented the same way, using normal Racket §6 "Modules".

When implementing such an extension for students, pay a special attention to two aspects:

1. **choice of construct**: The teaching languages limit the expressive power in comparison to plain Racket. One goal is to teach "design subject to constraints," and the other one is to help restrict the set of explanations for student errors. With regard to the first, we consider it imperative that new teachpacks and libraries avoid features intended for upper-level students or professionals.
2. **error messages**: The error messages from the teaching languages go to great length to never confront students messages that uses vocabulary or phrases outside of the scope of the chosen level. While teachpacks and libraries can be used at all levels, they should ideally restrict the vocabulary in error message to the lowest level language in which they are to be used.

This manual describes library support for authors of HtDP Teachpacks, libraries, and customized teaching languages. Use the HtDP error reporting functions to create error messages

that integrate smoothly with those of the teaching languages. Before composing new error messages, we recommend you read the error message composition guidelines that informed the design of the error messages of DrRacket's teaching languages.

1 Error Message Composition Guidelines

This section lists some guidelines for writing good error messages for novices, as informed by our research. Please follow these guidelines when you write code that is intended for beginners, including libraries and teachpacks. It ensures that error messages from your code fits messages from the student languages and from other teachpacks.

1.1 General Guidelines

- Frustrated students will peer at the error message for clues on how to proceed. Avoid offering hints, and avoid proposing any specific modification. Students will follow well-meaning-but-wrong advice uncritically, if only because they have no reason to doubt the authoritative voice of the tool.
- Be concise and clear. Students give up reading error messages if the text is too long, uses obscure words, or employs difficult grammar.

1.2 Message Structure and Form

- Start the message with the name of the construct whose constraint is being violated, followed by a colon.
- State the constraint that was violated (*"expected a..."*), then contrast with what was found. For example, *"this function expects two arguments, but found only one"*. If needed, explain how what was found fails to satisfy the constraint. Write somewhat anthropomorphically with an objective voice that is neither friendly nor antagonistic.
- If an expression contains multiple errors, report the leftmost error first. E.g., the error in `(define 1 2 3)` is *"expected the variable name, but found a number"*, not *"expected 2 parts after define, but found 3"*. Before raising an error about a sub-part of a macro, call `syntax-local-expand-expression` on sub-expressions to its left, so that such errors are shown first.
- State the number of parts instead of saying *"found too many parts"*. Write the code necessary to make plurals agree.

1.3 Words For Describing Code

Use only the following vocabulary words to describe code:

function	variable	argument	function body
expression	part	clause	top level

structure nametype namefield namebinding

- Use binding for the square-braced pair in a `let` and similar binding forms.
- Use 'argument' for actual arguments and 'variable' for formal arguments in the header and body of a definition.
- Use 'part' when speaking about an s-expression that is not an expression, either because it is malformed, because it occurs in a non-expression position, or because it is a valid piece of syntax for a macro invocation. A well-formed and well-placed call to a function, primitive, or macro is not a 'part', it is an 'expression'.

1.4 Words For Describing Runtime Behavior

Use the following vocabulary words to describe how code runs:

- When specifying a function's behavior, say *"the function takes ... and produces ..."*
- When describing a contract violation, say *"the function expects ... but received ..."*
- As much as possible, identify expressions and the value they evaluate to, e.g. *"the value of `(f x)` is 5"*. If it is necessary to mention evaluation order, such as when the context discusses mutable state, say that the expression *"evaluates to"* a value. Function calls are a special case of expression. Prefer *"the function call produces ..."* to *"the function call evaluates to ..."*, except when trying to draw attention to the evaluation of the arguments.
- `set!` and `set-structure-name-field-name!` 'mutate' variables and structure instances, respectively. Avoid using the verb 'sets' when discussing mutation, and reserve the verbs 'changes' and 'updates' for functional updates.

1.5 Prohibited Words

These guidelines use few terms intentionally, emphasizing commonality among concepts rather than technical precision (which most students do not appreciate anyway).

Instead of	**Use**
procedure, primitive name, primitive operator, predicate, selector, constructor	*"function""*
s-expression	*"expression"*
identifier	*"argument"* or *"variable"*, depending on th
defined name	*"function"* or *"variable"*
sequence	*"at least one (in parentheses)"*
function header	*"after define"*, *"after the name"*, *"after the*
keyword	mention the construct directly by name, suc

built-in	Nothing — avoid this term
macro	Nothing — avoid this term

1.6 General Vocabulary Guidelines

- Avoid modifiers that are not necessary to disambiguate. Write 'variable' instead of 'local variable', 'defined variable', or 'input variable'. Write 'clause' instead of 'question-answer clause'. If they appear necessary for disambiguation, try to find some other way to achieve this (and drop the modifier).
- When introducing macros with sub-parts, reuse existing vocabulary words, such as 'clause' or 'binding' (if appropriate), or just 'part', instead of defining new terms.
- Use 'name' only when describing the syntax of a definition form. For example, the define form in BSL should say *"expected at least one variable after the function name"*. Outside of the definition form, simply use the word 'function' rather than distinguish between (1) a function, (2) the variable that binds the function, and (3) the name of that variable.

  [Rationale: Students learn this distinction when they learn about lambda. The first is the lambda implicit in the definition, the second is the variable introduced by the definition that can appear as the first argument to `set!`, the third is the particular sequence of letters. But BSL should avoid this complexity, and ASL's error messages should maintain consistency with BSL.]
- Avoid introducing technical vocabulary, even if well-known to a mathematician.

1.7 Punctuation

- Do not use any punctuation beyond those of the normal English language. Do not write <> around type names, and do not write quotes around keywords.

1.8 Supporting Research

These guidelines arose from a collections of research studies held at the Worcester Polytechnic Institute, Brown University, and Northeastern University. Further experiment details and results are described in:

- Mind Your Language : On Novices' Interactions with Error Messages

  This paper reports on a series of studies that explore beginning students' interactions with the vocabulary and source-expression highlighting in DrRacket. Our findings

demonstrate that the error message DrRacket's old error messages significantly failed to convey information accurately to students.

- Measuring the Effectiveness of Error Messages Designed for Novice Programmers

  This paper presents a fine-grained grading rubric for evaluating the performance of individual error messages. We applied the rubric to a course worth of student work, which allowed us to characterize some ways error messages fail.

2 Error Reporting Functions

```
(require htdp/error)
```
package: htdp-lib

To provide uniform error messages from teachpacks, this module provides several functions:

```
(check-arg name chk expected position given) → void?
  name : (or/c symbol? string?)
  chk : boolean?
  expected : any/c
  position : (or/c (and/c positive? integer?) string?)
  given : any/c
```

Checks an flat-valued argument to function *name*. Reports an error for function *name* telling students what kind of data is *expected* at the *position*-th argument and displaying what value was actually *given*, unless *chk* is #true.

```
(check-arity name [arg#] args) → void?
  name : (or/c symbol? string?)
  arg# : (or/c (and/c positive? integer?) string?) = ?
  args : list?
```

Checks the arity of a procedure-valued argument to function *name*. Reports an error for function *name* telling students that (length *args*) arguments were provided but *arg#* were expected, unless (= (length *args*) *arg#*) produces #true.

```
(check-proc name proc expected arg# arg-err) → void?
  name : (or/c symbol? string?)
  proc : any/c
  expected : natural?
  arg# : (or/c (and/c positive? integer?) string?)
  arg-err : string?
```

Checks [the properties of] a procedure-valued argument to function *name*. Reports an error for function *name* telling students that a procedure was expected at position *arg#* and that this procedure should be of arity *expected*, unless the *proc* is a function and has the *expected* arity. The string *arg-err* is used to describe the higher-order argument.

```
(check-result name pred? kind returned ...+) → void?
  name : (or/c symbol? string?)
  pred? : (-> any/c boolean?)
  kind : (or/c symbol? string?)
  returned : any/c
```

Checks the expected result of a procedure-valued argument. If the result satisfies *pred?*, it is returned. Otherwise, the function reports an error for function *name* telling students what

kind of value is expected and what the *returned* value is. NOTE: if there is more than one *returned* value, the function uses the second value. (MF: I forgot why.)

```
(check-list-list name chk pred? given) → void?
  name : (or/c symbol? string?)
  chk : (or/c string? false/c)
  pred? : any/c
  given : any/c
```

Checks a list-of-lists-valued argument to function *name*. Reports an error for function *name* if a list-of-lists contains a value of the wrong kind—signaled via a string-valued *chk*. The *given* value is the element that went wrong. Rarely used.

```
(check-color name arg# given) → void?
  name : (or/c symbol? string?)
  arg# : natural?
  given : any/c
```

Checks a color-valued argument to function *name*. Deprecated. Use `image-color?` instead.

```
(check-fun-res f pred? type) → void?
  f : procedure?
  pred? : (-> any/c boolean?)
  type : (or/c symbol? string?)
```

Creates a callback from *f* and uses `check-result` to make sure the result is a piece of data that satisfies *pred?*, described as *type*.

```
(natural? o) → boolean?
  o : any/c
```

Determines whether the given value is a natural number.

```
(find-non pred? l) → (or/c any/c false/c)
  pred? : (-> any/c boolean?)
  l : list?
```

Find an element of *l* for which (*pred? l*) produces `#true`; otherwise return `#false`.

```
(check-dependencies name chk fmt arg ...) → void?
  name : (or/c symbol? string?)
  chk : boolean?
  fmt : format-string?
  arg : any/c
```

Unless *chk* is #true, it raises an error called *name* whose message is composed from *fmt* and the *arg*s.

```
(tp-error name fmt arg ...) → void?
  name : (or/c symbol? string?)
  fmt : format-string?
  arg : any/c
```

Signals an exn:fail:contract from *fmt* and *arg* for a function called *name*.

```
(tp-exn? o) → boolean?
  o : any/c
```

Determine whether the given object is a teachpack exception MF: Guillaume seems to have deprecated these structures.

```
(number->ord n) → string?
  n : natural?
```

Convert a position number into a string, e.g., 1 into “first” and so on.

MF: These library and its uses needs to be cleaned up.

3 Testing

```
(require htdp/testing)
```
package: `htdp-lib`

The library re-exports the identifiers from `test-engine/racket-tests`.

In addition, it exports:

```
(generate-report) → void?
```

The same as `test`.

4 HtDP Languages as Libraries

4.1 *HtDP* Beginning Student

```
(require lang/htdp-beginner)
```
package: htdp-lib

The lang/htdp-beginner module provides the Beginning Student Language; see §1 “Beginning Student”.

4.2 *HtDP* Beginning Student with Abbreviations

```
(require lang/htdp-beginner-abbr)
```
package: htdp-lib

The lang/htdp-beginner-abbr module provides the Beginning Student with Abbreviations language; see §2 “Beginning Student with List Abbreviations”.

4.3 *HtDP* Intermediate Student

```
(require lang/htdp-intermediate)
```
package: htdp-lib

The lang/htdp-intermediate module provides the Intermediate Student language; see §3 “Intermediate Student”.

4.4 *HtDP* Intermediate Student with Lambda

```
(require lang/htdp-intermediate-lambda)
```
package: htdp-lib

The lang/htdp-intermediate-lambda module provides the Intermediate Student with Lambda language; see §4 “Intermediate Student with Lambda”.

4.5 *HtDP* Advanced Student

```
(require lang/htdp-advanced)
```
package: htdp-lib

The lang/htdp-advanced module provides the Advanced Student language; see §5 “Advanced Student”.

4.6 Pretty Big Text (Legacy Language)

```
(require lang/plt-pretty-big-text)
```
package: htdp-lib

The lang/plt-pretty-big-text module is similar to the *HtDP* Advanced Student language, but with more of Racket's libraries in legacy form. It provides the bindings of mzscheme, mzlib/etc, mzlib/file, mzlib/list, mzlib/class, mzlib/unit, mzlib/include, mzlib/defmacro, mzlib/pretty, mzlib/string, mzlib/thread, mzlib/math, mzlib/match, mzlib/shared, and lang/posn.

4.7 Pretty Big (Legacy Language)

```
(require lang/plt-pretty-big)
```
package: htdp-lib

The lang/plt-pretty-big module extends lang/plt-pretty-big-text with racket/gui/base and lang/imageeq. This language corresponds to the Pretty Big legacy language in DrRacket.

4.8 posns in *HtDP* Languages

```
(require lang/posn)
```
package: htdp-lib

```
(struct posn (x y)
    #:extra-constructor-name make-posn)
  x : any/c
  y : any/c
```

The posn structure type that is also provided by lang/htdp-beginner.

```
Posn
```

Signature for posns.

```
(PosnOf x-sig y-sig) → signature
  x-sig : signature
  y-sig : signature
```

Creates a parametric signature for posns from signatures for its fields.

4.9 Image Equality in *HtDP* Languages

```
(require lang/imageeq)
```
package: htdp-lib

```
(image=? i1 i2) → boolean?
  i1 : (is-a?/c image-snip%)
  i2 : (is-a?/c image-snip%)
```

The image-comparison operator that is also provided by `lang/htdp-beginner`.

4.10 Primitives in *HtDP* Beginner

```
(require lang/prim)
```
package: htdp-lib

The `lang/prim` module several syntactic forms for use by the implementors of teachpacks, when the teachpack is to be used with the Beginner Student Language. In Beginner Student, primitive names (for built-in procedures) are distinguished from other types of expressions, so that they can be syntactically restricted to application positions.

```
(define-primitive id proc-id)
```

Defines *id* to be a primitive operator whose implementation is *proc-id*, and that takes no procedures as arguments. Normally, *id* is exported from the teachpack and *proc-id* is not.

```
(provide-primitive id)
```

Like `define-primitive`, but the existing function *id* is exported as the primitive operator named *id*. An alternative to `define-primitive`.

```
(provide-primitives id ...)
```

Multiple-identifier version of `provide-primitive`.

```
(define-higher-order-primitive id proc-id (arg ...))
```

Defines *id* to be a primitive operator whose implementation is *proc-id*. Normally, *id* is exported from the teachpack and *proc-id* is not.

For each non-procedure argument, the corresponding *arg* should be an underscore. For each procedure argument, the corresponding *arg* should be the usual name of the procedure.

Examples:

```
(define-higher-order-primitive convert-gui convert-gui/proc (f2c))
```

```
(provide-higher-order-primitive id (arg ...))
```

Like `define-higher-order-primitive`, but the existing function *id* is exported as the primitive operator named *id*. An alternative to `define-higher-order-primitive`.

```
(first-order->higher-order expression)
```

If *expression* is the name of a first-order function (either a primitive or a function defined within Beginner Student), produces the function as a value; otherwise, the form is equivalent to *expression*.

This form is mainly useful for implementing syntactic forms that, like the application of a higher-order primitive, allow first-order bindings to be used in an expression position.

5 Color and Alpha Color Structs

```
(require htdp/color-structs)
```
package: htdp-lib

```
(struct color (red green blue)
    #:extra-constructor-name make-color)
  red : any/c
  green : any/c
  blue : any/c
```

This is the color struct that is also exported by htdp/image, but here it is exported via (provide (struct-out color)).

```
(struct alpha-color (alpha red green blue)
    #:extra-constructor-name make-alpha-color)
  alpha : any/c
  red : any/c
  green : any/c
  blue : any/c
```

This is the color struct that is also exported by htdp/image, but here it is exported via (provide (struct-out alpha-color)).