DrRacket's Error Message Composition Guidelines for *SL

These guidelines distill our current thoughts on writing good error messages for novices, as informed by our research. Please apply these to all code you write, including libraries and Teachpacks. Inform Kathi, Shriram, and Guillaume of any cases that appear awkward under these guidelines.

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.

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 failed 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, call 'local-syntax-expand' on all sub-expressions to the left to trigger their errors.

— State the number of parts instead of saying "found too many parts." Write the code necessary to make plurals agree.

Vocabulary

Permitted Words

Use **only** the following vocabulary words to describe code:

```
function, variable, argument, function body, expression, part, clause, top level,
structure name, type name, field name, binding.
```

— Use '**binding**' for the square-braced pair in a `let` and other similar binding forms.

— Use '**argument**' for *actual* arguments and '**variable**' for *formal* arguments and in the body of the 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**'.

Prohibited Words

These guidelines use fewer terms than the current implementation to define code, 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 the use in the program
defined name	"function" or "variable"
sequence	"at least one (in parentheses)"
function header	"after define", "after the name", "after the first argument", …
keyword	mention the construct directly by name, such as "expected a variable but found a cond"
built-in	Nothing – avoid this term
macro	Nothing – avoid this term

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.

Punctuation

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