Pyret Feedback Guidelines

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.

1. General guidelines

- Error messages are required to be *accurate* above all else. It is *not* necessary to suggest any particular course of action to correct the error, unless it is guaranteed to be the only way out. 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 messages if the text is too long, uses obscure words, or employs difficult grammar.
- Error messages do not in general address the user of the program as "you", nor do they refer to the program as "I" or "We". Current Pyret messages violate this policy, and the workarounds will be clear in the following text. There is however one important exception to this policy where the user is directly addressed as "you", and we will explore that too.

2. Message structure and form

An error message is split into at most three sections, viz.,

1. a start section,
2. a section repeating the program fragment containing error, and
3. the error message proper.

Not all error messages contain all three sections. Each non-empty section is followed by a newline (or a bit more vertical space?).

3. Types of errors

For convenience, we recognize two types of error:

1. *Syntactic* errors, which arise when the program is ill-formed
2. *Semantic* errors, which arise when the program is well-formed but fails because Pyret could not ascribe proper meaning to them

Syntactic errors get spotted by the Pyret parser. In contrast, a semantic error is encountered, if at all, when Pyret runs a successfully parsed program. Thus, from a purely implementational point of view, we can view syntactic errors as *parsing* (including lexing) errors, and semantic errors as *runtime* errors.

i **Info**

Shriram postulates a third, intermediate, type of error called *well-formedness* errors, where the program is syntactically valid but still not well formed. I think these are syntactic errors. A well-formed program is one that is grammatically correct (even if it fails to have meaning), i.e., syntactically correct. A well-formedness error (i.e., an error that violates well-formedness) is therefore a syntax error. But we can revisit this issue if further error categories are found to be useful.

4. Syntactic errors

These are issued when the progam is ill-formed, i.e., cannot be parsed (or worse, lexed). The error message starts off with

```
Pyret did not understand the program around
```

The offending program fragment is then repeated, highlighting the offending expression, or, if it is missing, the expression closest to the missing part.

Highlighting is done by changing the color background of the expression both in the error message and in the program, using the *same* color, and with blinking.

The choice of color has no other meaning, i.e., it is not *syntax* highlighting as is used in code editors (including Pyret's own IDE). It merely mark as related the matching expressions in the program and the error message. In general, the colors for different expressions are chosen to be as "far apart" as possible, to avoid conflation. (The "distance" metric is based on the colors' positions on a standard color wheel.) The algorithm for this color choice is described in Picking Colors for Pyret Error Messages.

Possible ways of correction are then suggested. At this point, the user is addressed as "you", in a departure from the overall policy. The "you" here is acceptable and even desirable because it is a call to action, and the direct address makes it less likely to be glossed over as boilerplate fuzz.

In case of total syntactic confusion, a catch-all message is issued:

```
You may need to add or remove text to fix your program.
Look at the highlighted text.
Is there a missing colon, comma, string marker or other text?
Is there something there that shouldn't be?
```

4.1. Instances of syntactic errors

Many syntactic errors are more easily spotted for what they are, and a more helpful, tailored recommendation can be given. Such errors include:

- mildly mistyped function definitions (i.e., with extra space)
- mildly mistyped function calls (i.e., with extra space)
- mildly mistyped operator calls (i.e., without surrounding space)
- mildly mistyped numbers (e.g., with 0 before decimal point)
- missing commas, colons, `end` (function definitions, `if`/`ask`/`check`/etc, function parameters)
- type annotations where they shouldn't be
- sudden end of program
- unterminated string (very common!)
- testing operators outside of check blocks

- empty blocks

An example recommendation (for the mistyped number) would be something like:

```
You probably meant to write a number at ...
Number literals in Pyret should have at least one digit before
the decimal point.
```

Admittedly, some of these syntactic errors – the ones complaining against too much or too little space – may seem unnecessarily fussbudgety. But it is Pyret policy to disallow too freewheeling a spacing style.

5. Semantic errors

These are issued when a program is well-formed (= syntactically correct) but fails because Pyret cannot ascribe a proper meaning to it. This can happen for four reasons (could there be more?):

1. A definition is missing
2. Expressions have the wrong type for the context they're in
3. Expected expressions are missing
4. Unexpected expressions are present

i **Info**

Exceptions could be considered a fifth type of semantic error: These happen when an expression takes on specific values for which the program fails. They are somewhat similar to type errors (#2 above), but are triggered only by specific values in a type. A canonical example is division by an expression that happens to evaluate to zero.

i **Info**

In some languages (e.g., JavaScript, Lua) #1 could also be classed as a type error (#2), since undefined variables are assumed to have a special "null" or "nil" value. In Pyret, undefinitions are immediately recognized as their own error.

As we can see, except for missing definitions, all semantic errors may be considered "failure of expectation" situations, and thus the error message has a standard format, viz.,

```
Expected ..., but found ...
```

E.g.,

```
Expected a number but found <expression>.

Expected an even number of arguments but found <expression>.

Expected 3 arguments but found <expression>, ...
```

Here `<expression>` is a repeat of the actual offending expression in the program, and it is highlighted to allow quicker recognition. (As we note in the Punctuation section, we *do not* use angular brackets to call attention to program fragments.)

Note that we do not mention who's doing the "expecting". Unlike for syntactic errors, semantic error messages neither address the user nor assume the first person for Pyret. This is because semantic errors can happen for deep reasons where it is risky to offer advice which is liable to be misleading.

Info

In the special case where something was expected but nothing was found, we could use

```
Expected but did not find ...
```

rather than

```
Expected ... but found nothing.
```

If the semantic error is a type error (#2), we have the option of specifying the erroneous type along with the erroneous expression. Currently we choose not to, but may revisit this (should we call attention to the wrong type, or be satisfied with suggesting the correct type). We always specify the expected type though.

If a program contains multiple errors, report them strictly left to right. Word the sentence so that the left-to-right order of the referrers matches the occurrence of the referents in the program. Permuting the order in the error message to something different from the lexical order in the program is confusing, even if may be better English.

Also, state the actual number of parts instead of saying "found too many parts". Numbers are always arabic numerals, never spelled out in English (even if they're "small"). Ensure number agreement (i.e., no "1 parts").

5.1. Permitted words

The nouns used in error messages to refer to the components of a program are intentionally parsimonious. The user cannot be expected to know the nuances of a fine-grained lexicon when they are already mired in the process of identifying the immediate predicament they're in with a failing program.

The permitted words are:

```
argument
array
binding
column
datatype
datatype name
datatype variant
definition
expression
field
function
function body
immutable
mutable
object
part
reference
row
table
type
```

```
type name
variable
```

Values (things that expressions evaluate to) are semantic entities. The words used for them are the (generic) "value" and, if we need to be specific, the type of that value, i.e., "number", "string", "boolean", "list", etc. The list of all such built-in types (whether simple or composite) can be gleaned from the Pyret manual. User-defined datatypes could swell this list.

5.1.1. Some important usage notes

- Use "variable" for *all* names! Do not use "identifier", even if the name is used for a binding that's immutable, or for naming things like check blocks, where mutability makes no sense. If the naming quality of the variable is important, use "name".
- Use "binding" for a pairing of a variable with its value. A value may be specified either at the variable's initial declaration, or via a subsequent assignment. Passive: A variable is bound to a value. Active: You bind a variable to a value. To avoid confusion, do not say "a value is bound to a variable", or "you bind a value to a variable". These inversions are common in informal parlance, but for the purposes of user feedback, it is best to avoid this, as the relation between variable and value is not symmetric: one is the namer, the other is the namee.
- The word "define" is used to mean that a variable is in play, even if it has not been bound to any value. Other systems often distinguish between "declare" and "define"; we don't. Use "unbound" to capture situations when a variable has been "declared" but has no value associated with it. Note that Pyret does not automatically associate a "null" value with variables that are defined but unbound.
- Use "argument" for actual arguments (in a function call) and "variable" for formal arguments and in the body of the definition. In general, we don't add a special name for something simply because it's used in a narrower context. Thus, variables when they are formal parameters are still referred to as variables.
- Use "part" when speaking about a contiguous part of an expression that is not an expression itself, either because it is malformed, or because it occurs in a non-expression position. A well-formed and well-placed call to a function or an operator is not a "part"; it is simply an expression. There is no need to call it a "sub"-expression: The fact that is inside another expression is obvious and doesn't need special vocabulary.

Info

Should we use "clause" to refer to the parts of: ask, cases, check, data, if? For some of these, clauses are separated by pipe signs. The documentation currently calls them "branches". Clauses/branches may be further divided into "tests" (or "conditions") and "actions". The only context where we may need to refer to them is in syntactic errors. (Semantic errors can easily refer to the expressions within them.) Syntactic error messages can likely get away with asking the user to check for missing colons, commas, and pipes.

5.2. Prohibited words

It is easy to fall into the trap of using words that seem reasonable and intuitive and seem to be widely recognized in programming-language manuals and implementations. *Don't!*

Examples of such attractive but nevertheless prohibited words are:

```
built-in
constructor
declaration
defined name
form
(actual, formal) parameter
function header
identifier
keyword
(left, right) (hand) side
predicate
primitive (name, operator)
procedure
selector
sequence
subexpression
```

5.3. 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).
- Similarly, no “formal parameter”, “actual parameter”, or indeed even “parameter”. No “local” or “global” or “input” “variable”: just “variable”. Use “argument” when it’s necessary to emphasize that something’s a function’s formal parameter (term to avoid), otherwise use the generic “variable”.
- Use “name” only when describing the syntax of a function definition. Outside of a definition, simply use the word “function”: In such contexts, no need to distinguish between (a) a function “value”, (b) the variable that is bound to the function value.
- Use “reference” when identifying something that is mutable, typically when it is in a context where it is being mutated. A reference can be a variable; field in an object; row/column/cell in table; or indexed location in an array.
- Avoid introducing technical vocabulary, even if well-known to a mathematician or computer scientist.

5.4. Punctuation, font, and color

Do not use any punctuation beyond those of the normal English language. Do not use attention-calling enclosers like <> around type names, and do not use quotes (smart or otherwise) around keywords.

Keywords may be set in a different font face (viz., monospace), or font style (e.g., bold) to stand out from their containing prose.

Program fragments, or English references to them (e.g., “this expression”) are color-coded to match their presence elsewhere.

Colors are used simply to match equivalent descriptions of the same thing within a region of interest. They have no intrinsic meaning, e.g., numbers don’t get a particular color across the board, etc. For more info on how colors are chosen, see Picking Colors for Pyret Error Messages.

Color-coding matching expressions and their referrers serves to immediately guide the user to what is intended without having to use specialized vocabulary to describe them.

For now, do use sentence-ending periods for every sentence in a message, even if it is the only or final sentence. (There is a modern convention that seeks to avoid this final period so as not to appear brusque, but it have the disadvantage of making the last sentence inconsistent with the rest)

5.5. Current Pyret practice

Pyret's current error messages are not consistent with these guidelines. For a list of the various error message templates used, please see the file `src/runtime-arr/error.arr` in the `pyret-lang` repository. It should be sufficient to modify this file to bring Pyret's error messages in line with our desired guidelines.

For instance, when an undefined variable (say x) is used, Pyret currently says:

```
The identifier x is unbound.
```

followed by the program fragment, and the explanation

```
It is used but not previously defined.
```

The explanation is perhaps redundant given the opening message. Furthermore, we're using the prohibited word "identifier" rather than the preferred "variable".

Pyret uses "declaration" when user tries to mutate an immutable variable (it's an oxymoron, but we're committed to using variable throughout for error messages). Say the variable is already bound or defined, or that it is not mutable. E.g.,

```
x = 3
x = 4
```

produces the error message

```
The declaration of x shadows a previous declaration of x.
```

First, change "declaration" to "definition". Second, we have the option of either keeping the word "shadow", or rewrite the message as

```
The variable x is already bound.
The variable x is already defined.
The variable x is not a mutable reference.
```

(The words definition/defined may be enough, so we could arguably avoid bound/binding entirely.)

Is shadowing something we want to mention in error messages? (In the case of the error, we're not actually shadowing. We failed to shadow, in fact, and therefore bailed.)

Type errors currently state the types of the arguments of the operator or function, at least one of them must be incorrectly typed. Pyret's current message lists their types. For binary operators, it uses phrases such as "left side" and "right side", states their types, followed by what the operator expects. This is long-winded, but perhaps so so as not violate the strictly left-to-right reference to the components of the program fragment. (For an infix operator, the operator follows its left-side.) However, we can't simply say

```
The expression x is not of type t.
```

Because of overloaded operators like +, where the failure is that the operands don't *match* type, not that they're not a particular type.

```
The expression x has type t1.
The expression y has type t2.
The operator o expects its arguments to be two t1s or two t2s.
```