1. Principles

2. Reducing ambiguity in definite references

Let's consider a piece of code written by a student during their first programming lab and the error message they received.

```
(check-expect ([IMG] [IMG])(overlay/xy pizza 0 50 salad))
function call: expected a function after the open
parenthesis, but found an image
```

This error message has one definite reference, “the open parenthesis”. The word “the” is a definite article. In natural languages, the purpose of definite articles is to indicate that the reference is made to one particular item. The definite article implies that the reference is intended to have no ambiguity.

This is not true in our sentence. There are three open parentheses in the code. Which one is being referred to, we cannot tell. Indeed, helpers during lab sessions will often intervene at this point and help students resolve the definitive reference.

If we expand our analysis beyond the grammar of the sentence, we can find two more definite references.

First, the words “function call” are an elision for “this function call.” This is a demonstrative reference, which, in turn, is a kind of definite reference. This reference is just as ambiguous as the previous one. Just as there were three open parentheses, there are three function calls.

Then, we can consider the “an image” phrase. The article “an” indicates an indefinite reference. Indefinite references are used to indicate that the specific item is not known, that any one item of the right kind would do, or simply to indicate that the specific item does not matter in the moment. This is the case for our sample sentence. There is definitely a specific image causing the error, but the wording of the error message intends to stress that having a different image in this place would not help. It is having an image at all, any image, which is making the function call invalid. We can confirm that this is the case by replacing the “an” article by the demonstrative article “this” in the sentence.

```
function call: expected a function after the open
parenthesis, but found this image
```

The resulting error message is just as meaningful, though the connotation has changed.

Another way to find the definite reference underneath is to imaging pursuing a conversation about the error, perhaps with a lab helper. We could ask “which image was found?,” and receive the answer “this error,” while they point their finger at the screen.

DrRacket does not have any finger. To hold the same conversation with it, DrRacket would need to be extended with a mechanism for pointing at pieces code. With it, DrRacket could say, “this is the one I mean.” The color-coded error messages are one such mechanism. Each definite reference will be highlighted in the error message with a different color, and each referent piece of code will be highlight with the corresponding color.

In our sample sentence, the phrase “a function” is the other kind of indefinite reference. Here, the reference does indeed means any one function will do, and there is no further clarification possible. Neither the replacement trick nor the conversation work trick work. So the phrase “a function” does not receive a highlight.

```
(check-expect ([IMG] [IMG])(overlay/xy pizza 0 50 salad))
```

function call: expected a function after the open parenthesis, but found an image

3. Algorithm

An error message start with a string

```
(define-type ErrorMessage String)
```

The highlighting mechanism takes as an input a list of pairings between, on one side, substrings of the error message, and on the other side, a number of sexps, srclocs, or links to the documentation.

```
(struct: pairing ([subrange : (PairOf Number Number)]
                  [code : (ListOf (U Sexp Srcloc DocLink))]))
```

The sub ranges cannot overlap. This is not a burdensome restriction since it arises from nature of what we are annotating. Each sub range is delimiting the definite reference phrases in the sentence, and the grammar of English does not allow those to overlap. The most common case is when a sub range will be associated with exactly one sexp from the code. Three other cases are:

- If the definite reference mentions a number of items of the same kind, as in “3 arguments,” then three sexps will be associated with this reference.
- If the error message is a well-formness error, then it is not yet well-define to describe locations in terms of sexps. In this case, the destination of the highlight will be specified in terms of source location.
- If the error message mentions a function whose definition is imported from professional libraries (rather than being define in the student’s code,) the link will point to the documentation for this function, since linking to the implementation code would break the abstraction that *SL aims to maintain. It is possible that the context-switch to the

documentation might be overly jarring. The UI design will have to be worked design carefully to avoid this.

The colored message is the string bundled with the highlight specification:

```
(struct: colored-message ([msg : String]
                          [pairings : (ListOf pairing)]))
```

As an input, the algorithm also depends on a list of colors, taken from a color-blind-safe palette. There are 5 color-blind-safe colors that can be reliably distinguished across people with all types of color blindness. Error messages with more than 5 definite references should happen never, or have been very rarely. Additional colors will be added to the list to cover the eventuality of longer error messages, acknowledging that such messages would be more difficult to understand by some color-blind students. For completeness, in case the extended list is also depleted, the list of colors will be cyclic.

The following algorithm renders a colored message.

1 For each pairing, assign a color from the list of colors, in order.

2 For each pairing, for each sexp location to be highlighted, determine whether any one of descendant of that sexp is highlighted. Any sexp with a highlighted descendant will be highlighted only on the first and last character, with a fade effect.

3 If a pairing has more than one code location to be highlighted:

3.1 Divide the length of the subsequence of the error message evenly for each code location.

3.2 If the divisions are too short (shorter than two characters,) pad right side of the subsequence with spaces until there is enough space.

3.3 Select color for each division by:

3.3.1 Darkening the base color for this pairing by an amount proportional to the number of divisions. This becomes the base color of the first division.

3.3.2 Fading the color the same amount. This becomes the color of the last division.

3.3.3 Interpolating between these two colors for the other divisions.

4 Each substring and division in the error message gets highlighted with its color. They also are hyperlinked to the code location.

4.1 Clicking on the hyperlink scrolls the definition window so the code is centered, and moves the selection to the left edge of the code region.

4.2 If the code location is in a different file, DrRacket switches to its tab, opening the file if necessary.

4.3 If the destination of the pair is in the documentation, clicking pops the browser, or pops a tooltip window, or some other interface element, depending on a careful balance between the value of the information provided by the documentation and the risk of distraction.

4. Reducing unnecessary coloring clutter

Returning to our earlier example

```
(check-expect ([IMG] [IMG])(overlay/xy pizza 0 50 salad))
function call: expected a function after the open
parenthesis, but found an image
```

Here, the highlighted sexp `([IMG] [IMG])` contains another sexp which is also highlighted, namely the first `[IMG]`. In order to avoid the visual clutter that would happen when presenting a superposition of highlights, only the parentheses of the outer sexp are highlighted.

The phrase "the open parenthesis" refers to the same part as the previous phrase. Again, in order to avoid a superposition, this highlight is omitted entirely. An alternative would be to color "the open parenthesis" in pink as well, but this option has additional complexity that makes it undesirable, which we discuss next.

5. Coloring pattern

For fulfilling the purpose of using color as arrow from the reference to the referee, the particular color used doesn't matter. However, if there is our pattern to the colors (if the colors are chosen at random) students are likely to seek to attempt to uncover a pattern underneath governing the selection of the covers, before convincing themselves that there is none. Either is undesirable. So we need a pattern.

On the other hand, if the pattern is non-trivial, or complicated even, it might necessitate significant effort on the part of students to uncover it, perhaps even lecture time and TA intervention. So we should avoid more complicated patterns unless there is a compelling pedagogical justification for investing contact time in that direction.

At the moment, we have experimental results supporting using the colors as arrows, but we do not have support for any other function. This suggests that the coloring scheme should be the simplest pattern that satisfies the students' curiosity and thus prevents them from wondering further.

This is why I am suggesting that the colors appear always in the same order, rather than having them color-coding. As a rule, the order in the coloring is driven by the order of the error message text, since the text is shorter than the code.

6. Order of the highlights

The new coloring will require students to see and understand the pairwise correspondence between two highlights of the same color displayed on the screen. Although this task is not very demanding compared, say, to writing correct contracts, it is a non-zero amount which deserves some consideration. Subject to balancing this concern against others, we should seek to minimize the difficulties of developing the mental representation of the correspondence.

The simplest correspondence occurs when the pieces of highlighted code appear in the same order (when scanning top to bottom, left to right) as the highlights in the message. The question is then whether it is worth rewording the error message in order to make the orders correspond.

Readability needs to come first. On one hand, English provides grammatically correct ways to change the order of the phrases. On the other hand, it is often very clear which order reads best, and all other orders will be stilted and difficult to read. The error message

```
function call: expected a function after the open
parenthesis, but found an image
```

remains grammatically correct when rewritten as

```
function call: found an image where, after the open
parenthesis, a function was expected
```

but a lot of readability is lost.

However, in the case where it is possible to order the highlights in the message to match the order in the code without loss of readability, it should be done.

7. Shading

When the error message mentions the concrete numbers of items, such as “found 3 parts,” how should they be highlighted? They can be highlighted in a single block of one color, and multiple blocks of the same color, in multiple blocks of different colors, or of different shades.

The “single block” option misses an opportunity to reinforce in students a procedural definition of “part,” and perhaps even undermines it. Students are learning how to determine what consists of the beginning and the end of a part. By highlighting as one bloc, the ends and start are painted over, leaving the students to wonder where they might be. By maintaining the rule that “one highlighting block always highlights exactly one part,” we reinforce the definition.

The “multiple blocks of the same colors” option forces students into a more complicated mental model. Instead of having in mind a bipartite graph (the half of the nodes correspond to the highlights in the error message, the other half of the nodes correspond to the highlights in the code) with one-to-one edges, they now have to understand that highlights in the code can have an in-degree greater than one. This additional complexity risks delaying students' development of their understanding of the semantics of the highlight.

The “multiple blocks of different colors” option quickly gets overloaded when the number of parts is large. It makes the color error messages even more likely to fail due to the Christmas-light effect, where students are overwhelmed by the amount of information coded in color highlights. The Christmas light effect is the number one failure risks for the color highlight design, and we should be careful not to exacerbate it.

The “different shades” option addresses all the concern stated for the other options. In particular, it scales smoothly to very large number of parts if the shade of the first block in the last block are held fixed (independent of the number of blocks) and we interpolate the intermediate blocks.

8. Arg-count : Shuffle-01/Ryanmbrown—05-001

```
 (check-expect test expected ([IMG] [IMG])(overlay/xy pizza
0 50 salad))
```

```
check-expect: expects only 2 arguments, but found
4 arguments.
```

9. arg count : amdemart--tab-02--compile-033--(arg-count).beginning.rkt

```
;; update-ball-x: number -> image
(define (update-ball-x X-POS)
  (+ 3 X-POS))

(define (update-game a-game)
  (and (update-ball-x place-ball-x 100)
         (place-goalie-y 100)))
```

```
update-ball-x: expects only 1 argument, but found 2
```

10. arg count : aishak-tab-01-compile-04

```
(define (filter-funk a-filter)
  (filter a-filter (filter-brand a-filter) (filter-order a-
filter) (filter-type a-filter) (filter-frequency a-filter)
(filter-ripple a-filter) (filter-band a-filter)))
```

```
filter: expects only 2 argument, but found 7
```

11. paren matching : Rryanbrown-03-16

```
(define (label-near? label word basketball basketball2
basketball3)
  (cond [(>label-near1? basketball) (< (label-near2?
basketball2)) (>label-near3 basketball3)))
```

read: missing `]' to close preceding `[', found instead `)'

12. meaning of open : ryanbrown-05-02

```
(check-expect ([IMG] [IMG])(overlay/xy pizza 0 50 salad))
```

function call: expected a function after the open parenthesis, but found an image

13. meaning of open : dfrods42-01-20

```
(define (label-near1? label name word)
  (cond [(or (label word word word name)
             (name word word word label)) true]))
```

function call: expected a function after the open parenthesis, but found a variable

14. meaning of open : jaredandrews-00-18

```
(define (label-near? name affiliation word1 word2 word3)
  (string=? name word1 or word2 or word3)
  )
```

or: expected an open parenthesis before or, but found none

15. meaning of open : Jaredandrews-00-36

```
(define (label-near? name affiliation word1 word2 word3)
  (cond [(
  (string=? name word1)
  (or(string=? name word2)
     )


  )"true"]))
```

```
function call: expected a function after the open
parenthesis, but found a part
```

16. meaning of open : plataniasangel-00-07

```
(define (label-near label name word1 word2 word3)
          (and (or [(string=? name word1) ]
                    [(string=? name word2) ]
                    [(string=? name word3) ]
                    [else ])
               (or [(string=? label word1) ]
                    [(string=? label word2) ]
                    [(string=? label word3) ]
                    [else ])))
```

```
function call: expected a function after the open
parenthesis, but found a part
```

17. syntax cond : Ryanbrown-03-21

```
(define (label-near label word basketball basketball2
basketball3)
  (cond [(string=? label basketball) (string=? label
basketball2) (string=? label basketball3)]))
```

```
cond: expected a clause with a question and an answer, but
found a clause with 3 parts
```

18. syntax cond : Aishak-00-03

```
(define (recommend-sport temp activity-type)
  (cond [(and (> (celsius->fahrenheit temp) 0) (< (celsius->fahrenheit temp) 32))
  (cond [(symbol=? activity-type 'motorized) "snow mobiling"] [(symbol=? activity-type 'feet)
"skiing"] [else "ice-fishing"])][(and (> (celsius->fahrenheit temp) 50) (< (celsius-
>fahrenheit temp) 80)) "hiking"]
  [else (cond [(symbol=? activity-type 'motorized) "stationary bike"]I LOVE YOU [(symbol=?
activity-type 'feet) "ping-pong indoors"] [else "tv"])]))
```

```
cond: expected a clause with a question and an answer, but
found something else
```

19. syntax cond : Aishak-00-07

```
(define (label-near? name label word1 word2 word3)
 (cond [(string=? word1 name)
        (cond [(string=? word2 label) true]
              [(string=? word3 label) true])
       [(string=? word2 name)
        (cond [(string=? word1 label) true]
              [(string=? word2 label) true])
       [(string=? word3 name)
        (cond [(string=? word2 label) true]
              [(string=? word3 label) true])
        [else false]]]]))
```

cond: expected a clause with a question and an answer, but found a clause with 3 parts

20. syntax cond : Apnittel-00-06

```
(define (label-near? label name word1 word2 word3)
  (cond [(string=? label word1)]
        [(string=? label word2)]
        [(string=? label word3)]
        (and (cond [(string=? name word1)]
                   [(string=? name word2)]
                   [(string=? name word3)]))))
```

cond: expected a clause with a question and an answer, but found a clause with only one part

21. syntax cond : Bgaffey-00-09

```
(define (label-near? label name word-one word-two word-
three)
  (cond [(and (string=? "name" "word-one")
              (string=? "label" "word-two") "true")]
        [(and (string=? "name" "word-one")
              (string=? "label" "word-three") "true")]
        [(and (string=? "name" "word-two")
              (string=? "label" "word-one") "true")]
        [(and (string=? "name"  "word-two")
              (string=? "label" "word-three") "true")]
        [else "false"]))
```

```
cond: expected a clause with a question and an answer, but
found a clause with only one part
```

22. syntax cond : Jaredandrews-00-37

```
(define (label-near? name affiliation word1 word2 word3)
  (cond [(
  (string=? name word1)
  (or(string=? name word2)
     )
  "true"

  )]))
```

```
cond: expected a clause with a question and an answer, but
found a clause with only one part
```

23. syntax cond : Ryanbrown-03-13

```
(define (label-near? label word basketball basketball2
basketball3)
  (cond [(and (>label-near1? basketball) (< (label-near2?
basketball2)) (>label-near3 basketball3))]))
```

```
cond: expected a clause with a question and an answer, but
found a clause with only one part
```

24. syntax cond : Ryanbrown-03-21

```
(define (label-near label word basketball basketball2
basketball3)
  (cond [(string=? label basketball) (string=? label
basketball2) (string=? label basketball3)]))
```

```
cond: expected a clause with a question and an answer, but
found a clause with 3 parts
```

25. syntax define : Jdelprete-00-28

```
(define (label-near? political-label "name" word1 word2
word3) (string<? "name" political-label "word1" [string<?
"name" political-label "word2"] [string<? "name" political-
label "word3"]))
```

```
(define (political-label "name") (string=? "name"
"conservative" (string=? "name" "liberal")))
```

define: expected a variable, but found a string

26. syntax define : amdemart-00-15

```
(define (label-near? label name word1 word2 word3)
  (cond [(string=? name 'bob) "liberal"])
   [(string=? label 'liberal) "true"]))
```

define: expected only one expression for the function body, but found 1 extra part

or

```
(define (label-near? label name word1 word2 word3)
  (cond [(string=? name 'bob) "liberal"])
   [(string=? label 'liberal) "true"]))
```

define: found the function body, but also found 1 extra part.

27. syntax define : drods42-01-25

```
(define ((label-near1? label)
  (label-near1? name)
  (label-near1? word))
  (cond [(or (label word word word name)
             (name word word word label)) true]))
```

define: expected the name of the function, but found a part

28. syntax define : jaredandrews-00-15

```
(define (label-near? name affiliation word1 word2 word3)
  (cond[(string=? name word1) "true"])
  (cond[(string=? name word2) "true"])
  (cond[(string=? name word3) "true"])
  )
```

define: expected only one expression for the function body, but found 2 extra parts

29. unbound id : be_binion-00-16

```
(define (label-near? word1 word2 word3)
  [cond (str=? "liberal" "liberal" true)])
```

str=?: this variable is not defined

30. unbound id : dfrod42-01-30

```
(define (label-near1? label name word)
  (cond [(or (string=? label word word word name)
             (string=? name word word word label)) true]))

(label-near1 label name word)
```

label-near1: this function is not defined

31. define duplicate : bmurray-tab-07-compile-05

```
;; A dna-fingerprint is a
;; (make-dna-fingerprint string string dna-sequence)
(define-struct dna-fingerprint (first-name last-name dna-seq))

;; [lots of code]

(define-struct dna-fingerprint (first-name last-name dna-seq))
```

dna-fingerprint: this nam has a previous definition and cannot be re-defined

32. define duplicate : dukes777-tab-01-compile-02

```
(define-struct polar (name weight food1 food2 food3 location))

(define-struct black (name weight food1 food2 food3 location))

(define-struct grizzly (name weight fur-color food1 food2 food3
location))

(define-struct location (loc-name average-temp))
```

make-polar: this name already has a definition and cannot be re-defined

33. define order : Used before definition

```
(define x (foo 4))
(define (foo x) 3)
```

foo is used here before its definition

34. Runtime type : dfrods42-01-38

```
     (define (label-near1? label name word)
       (cond [(or (string=? label word word word name)
                  (string=? name word word word label)) true]))
     (label-near1? 2 3 1)
```

string=?: expects a string as 1st argument, given 2

35.

36. Runtime type : jobrien-01-13

```
(cond ...
        [(grizzly? a-bear)(make-grizzly (grizzly-name a-bear)
                                        (grizzly-location location)
                                        (- (grizzly-weight a-bear)
                                           (* .10 (grizzly-weight a-bear)))
                                        (grizzly-diet1 a-bear)
                                        (grizzly-diet2 a-bear)
                                        (grizzly-diet3 a-bear)
                                        (grizzly-color-fur a-bear))])
```

grizzly-location: expects a grizzly for the first argument; given (make-location "reserve" 70)

[[set! Examples]]