Redex: Practical Semantics Engineering

Version 6.1.1

Robert Bruce Findler,
Casey Klein,
and Burke Fetscher

November 4, 2014

PLT Redex consists of a domain-specific language for specifying reduction semantics, plus a suite of tools for working with the semantics.

This manual consists of two parts: a tutorial introduction and a reference for Redex. Also see `http://redex.racket-lang.org/` and the `examples` subdirectory in the `redex` collection.

Contents

1 Amb: A Redex Tutorial

This tutorial is designed for those familiar with the call-by-value λ-calculus (and evaluation contexts), but not Redex. The tutorial works though a model of the λ-calculus extended with a variation on McCarthy's `amb` operator for ambiguous choice (McCarthy 1963; Zabih et al. 1987).

If you are not familiar with Racket, first try *Quick: An Introduction to Racket with Pictures* or *More: Systems Programming with Racket*.

The model includes a standard evaluation reduction relation and a type system. Along the way, the tutorial demonstrates Redex's support for unit testing, random testing, typesetting, metafunctions, reduction relations, and judgment forms. It also includes a number of exercises to use as jumping off points to explore Redex in more depth, and each of the functions and syntactic forms used in the examples are linked to more information.

1.1 Defining a Language

To get started, open DrRacket, and put the following two lines at the top of the file (if the first line is not there already, use the Language|Choose Language... menu item to make sure that DrRacket is set to use the language declaration in the source).

```
#lang racket
(require redex)
```

Those lines tell DrRacket that we're writing a program in the `racket` language and we're going to be using the `redex` DSL.

Next, enter the following definition.

```
(define-language L
  (e (e e)
     (λ (x t) e)
     x
     (amb e ...)
     number
     (+ e ...)
     (if0 e e e)
     (fix e))
  (t (→ t t) num)
  (x variable-not-otherwise-mentioned))
```

The `define-language` form gives a name to a grammar. In this case, `L` is the Racket-level name referring to the grammar containing the non-terminal `e`, with six productions

(application, abstraction, variables, `amb` expressions, numbers, and addition expressions), the non-terminal `t` with two productions, and the non-terminal `x` that uses the pattern keyword `variable-not-otherwise-mentioned`. This special pattern matches all symbols except those used as literals in the grammar (in this case: λ, `amb`, `+`, and →).

Once we have defined the grammar, we can ask Redex if specific terms match the grammar. This expression checks to see if the `e` non-terminal (from `L`) matches the object-language expression `(λ (x) x)`.

To do this, first click the Run button in DrRacket's toolbar and then enter the following expression after the prompt For the remainder of this tutorial, expressions prefixed with a > are intended to be run in the interactions window (lower pane), and expressions without the > prefix belong in the definitions window (the upper pane).

```
> (redex-match
   L
   e
   (term (λ (x) x)))
#f
```

In general, a `redex-match` expression first names a language, then a pattern, and then its third position is an arbitrary Racket expression. In this case, we use `term` to construct an Redex object-level expression. The `term` operator is much like Lisp's `quasiquote` (typically written ‘).

This term does not match `e` (since `e` insists the function parameters come with types), so Redex responds with `#f`, false.

When an expression does match, as with this one:

```
> (redex-match
   L
   e
   (term ((λ (x num) (amb x 1))
          (+ 1 2))))
(list
 (match
  (list
   (bind
    'e
    '((λ (x num) (amb x 1)) (+ 1 2))))))
```

Redex responds with bindings for all of the pattern variables. In this case, there is just one, `e`, and it matches the entire expression.

We can also use matching to extract sub-pieces. For example, we can pull out the function and argument position of an application expression like this:

```
> (redex-match
   L
   (e_1 e_2)
   (term ((λ (x num) (amb x 1))
          (+ 1 2))))
(list
 (match
  (list
   (bind 'e_1 '(λ (x num) (amb x 1)))
   (bind 'e_2 '(+ 1 2)))))
```

As you probably noticed, `redex-match` returns a list of matches, not just a single match. The previous matches each only matched a single way, so the corresponding lists only have a single element. But a pattern may be ambiguous, e.g., the following pattern which matches any non-empty sequence of expressions, but binds different elements of the sequence in different ways:

```
> (redex-match
   L
   (e_1 ... e_2 e_3 ...)
   (term ((+ 1 2)
          (+ 3 4)
          (+ 5 6))))
(list
 (match
  (list
   (bind 'e_1 '())
   (bind 'e_2 '(+ 1 2))
   (bind 'e_3 '((+ 3 4) (+ 5 6)))))
 (match
  (list
   (bind 'e_1 '((+ 1 2)))
   (bind 'e_2 '(+ 3 4))
   (bind 'e_3 '((+ 5 6)))))
 (match
  (list
   (bind 'e_1 '((+ 1 2) (+ 3 4)))
   (bind 'e_2 '(+ 5 6))
   (bind 'e_3 '()))))
```

Exercise 1

Use `redex-match` to extract the body of the λ expression from this object-language program:

```
((λ (x num) (+ x 1))
 17)
```

Exercise 2

Use `redex-match` to extract the range portion of the type `(→ num (→ num num))`.

Exercise 3

Redex's pattern language supports ambiguity through non-terminals, the `in-hole` pattern, and ellipsis placement (as in the example just above). Use the latter source of ambiguity to design a pattern that matches one way for each adjacent pair of expressions in a sequence. That is, if you match the sequence `(1 2 3 4)`, then you'd expect one match for 1 & 2, one match for 2 & 3, and one match for 3 & 4. In general, this pattern should produce n matches when there are n+1 expressions in the sequence.

To test your solution use `redex-match` like this:

```
(redex-match
 L
  ; your solution goes here
 (term (1 2 3 4)))
```

where you expect a result like this

```
(list
 (match (list (bind 'e_1 1) (bind 'e_2 2)))
 (match (list (bind 'e_1 2) (bind 'e_2 3)))
 (match (list (bind 'e_1 3) (bind 'e_2 4))))
```

but possibly with more pattern variables in the resulting match.

Exercise 4

The ellipsis pattern can also be "named" via subscripts that, when duplicated, force the lengths of the corresponding sequences to match. For example, the pattern

```
((λ (x ...) e) v ...)
```

matches application expressions where the function may have a different arity than the number of arguments it receives, but the pattern:

```
((λ (x ..._1) e) v ..._1)
```

ensures that the number of xs is the same as the number of vs.

Use this facility to write a pattern that matches odd length lists of expressions, returning one match for each pair of expressions that are equidistant from the ends of the sequence. For example, if matching the sequence `(1 2 3 4 5)`, there would be two matches, one for the pair `1` & `5` and another for the pair `2` & `4`. Your match should include the the bindings `e_left` and `e_right` that extract these pairs (one element of the pair bound to `e_left` and the other to `e_right`). Test your pattern with `redex-match`.

1.2 Typing

To support a type system for our language, we need to define type environments, which we do by extending the language `L` with a new non-terminal `Γ`, that we use to represent environments.

```
(define-extended-language L+Γ L
  [Γ · (x : t Γ)])
```

The `define-extended-language` form accepts the name of the new language, the name of the extended language and then a series of non-terminals just like `define-language`.

In the extended language, we can give all of the typing rules for our language. Ignoring the `#:mode` specification for a moment, the beginning of this use of `define-judgment-form` has a contract declaration indicating that the judgments all have the shape `(types Γ e t)`.

```
(define-judgment-form
  L+Γ
  #:mode (types I I O)
  #:contract (types Γ e t)

  [(types Γ e_1 (→ t_2 t_3))
   (types Γ e_2 t_2)
   -------------------------
   (types Γ (e_1 e_2) t_3)]

  [(types (x : t_1 Γ) e t_2)
   ----------------------------------
   (types Γ (λ (x t_1) e) (→ t_1 t_2))]

  [(types Γ e (→ (→ t_1 t_2) (→ t_1 t_2)))
   ---------------------------------------
   (types Γ (fix e) (→ t_1 t_2))]
```

```
[----------------------
 (types (x : t Γ) x t)]

[(types Γ x_1 t_1)
 (side-condition (different x_1 x_2))
 -----------------------------------
 (types (x_2 : t_2 Γ) x_1 t_1)]

[(types Γ e num) ...
 ----------------------
 (types Γ (+ e ...) num)]

[--------------------
 (types Γ number num)]

[(types Γ e_1 num)
 (types Γ e_2 t)
 (types Γ e_3 t)
 ----------------------------
 (types Γ (if0 e_1 e_2 e_3) t)]

[(types Γ e num) ...
 --------------------------
 (types Γ (amb e ...) num)])
```

The first clause gives the typing rule for application expressions, saying that if `e_1` has the type `(→ t_2 t_3)` and `e_2` has the type `t_2`, then the application expression has the type `t_3`.

Similarly, the other clauses give the typing rules for all of the other forms in the language.

Most of the rules use use `types`, or give base types to atomic expressions, but the fifth rule is worth a special look. It says that if a variable type checks in some environment, then it also type checks in an extended environment, provided that the environment extension does not use the variable in question.

The `different` function is a metafunction, defined as you might expect:

```
(define-metafunction L+Γ
  [(different x_1 x_1) #f]
  [(different x_1 x_2) #t])
```

The `#:mode` specification tells Redex how to compute derivations. In this case, the mode specification indicates that `Γ` and `e` are to be thought of as inputs, and the type position is to

be thought of as an output.

Redex then checks that spec, making sure that, given a particular Γ and e, it can compute a t or, perhaps, multiple ts (if the patterns are ambiguous, or if multiple rules apply to a given pair of Γ and e).

1.3 Testing Typing

The judgment-holds form checks to see if a potential judgment is derivable. For example,

```
> (judgment-holds
   (types · 
          ((λ (x num) (amb x 1))
           (+ 1 2))
          t)
   t)
'(num)
```

computes all of the types that the expression

```
((λ (x num) (amb x 1))
 (+ 1 2))
```

has, returning a list of them (in this case, just one).

In general, the judgment-holds form's first argument is an instance of some judgment-form that should have concrete terms for the I positions in the mode spec, and patterns in the positions labeled O. Then, the second position in judgment-holds is an expression that can use the pattern variables inside those O positions. The result of judgment-holds will be a list of terms, one for each way that the pattern variables in the O positions can be filled when evaluating judgment-holds's second position.

For example, if we wanted to extract only the range position of the type of some function, we could write this:

```
> (judgment-holds
   (types · 
          (λ (f (→ num (→ num num))) (f (amb 1 2)))
          (→ t_1 t_2))
   t_2)
'((→ num num))
```

The result of this expression is a singleton list containing the function type that maps numbers to numbers. The reason you see two open parentheses is that Redex exploits Racket's

s-expressions to reflect Redex terms as Racket values. Here's another way to write the same value

```
> (list (term (→ num num)))
'((→ num num))
```

Racket's printer does not know that it should use `term` for the inner lists and `list` (or `quote`) for the outer list, so it just uses the `quote` notation for all of them.

We can combine `judgment-holds` with Redex's unit test support to build a small test suite:

```
> (test-equal
   (judgment-holds
    (types · (λ (x num) x) t)
    t)
   (list (term (→ num num))))

> (test-equal
   (judgment-holds
    (types · (amb 1 2 3) t)
    t)
   (list (term num)))

> (test-equal
   (judgment-holds
    (types · (+ 1 2) t)
    t)
   (list (term (→ num num))))
FAILED :27.0
  actual: '(num)
expected: '((→ num num))
```

Redex is silent when tests pass and gives the source location for the failures, as above. The `test-equal` form accepts two expressions, evaluates them, and checks to see if they are `equal?` (structural equality).

To see a summary of the tests run so far, call `test-results`.

```
> (test-results)
1 test failed (out of 3 total).
```

Exercise 5

Remove the `different` side-condition and demonstrate how one expression now has multiple types, using `judgment-holds`. That is, find a use of `judgment-holds` that returns a list of length two, with two different types in it.

Exercise 6

The typing rule for `amb` is overly restrictive. In general, it would be better to have a rule like this one:

```
[(types Γ e t) ...
 ----------------------
 (types Γ (amb e ...) t)]
```

but Redex does not support this rule because the mode specification is not satisfied in the case that `amb` has no subexpressions. That is, any type should be okay in this case, but Redex cannot "guess" which type is the one needed for a particular derivation, so it rejects the entire `define-judgment-form` definition. (The error message is different, but this is the ultimate cause of the problem.)

Fix this by annotating `amb` expressions with their types, making suitable changes to the language as well as the `define-judgment-form` for `types`. Add new test cases to make sure you've done this properly.

1.4 Defining a Reduction Relation

To reduce terms, Redex provides `reduction-relation`, a form that defines unary relations by cases. To define a reduction relation for our `amb` language, we first need to define the evaluation contexts and values, so we extend the language a second time.

```
(define-extended-language Ev L+Γ
  (p (e ...))
  (P (e ... E e ...))
  (E (v E)
     (E e)
     (+ v ... E e ...)
     (if0 E e e)
     (fix E)
     hole)
  (v (λ (x t) e)
     (fix v)
     number))
```

To give a suitable notion of evaluation for `amb`, we define `p`, a non-terminal for programs. Each program consists of a sequence of expressions and we will use them to represent the

possible ways in which an amb expression could have been evaluated. Initially, we will simply wrap an expression in a pair of parentheses to generate a program that consists of that single expression.

The non-terminal P gives the corresponding evaluation contexts for ps and says that evaluation can occur in any of them, without restriction. The grammar for E dictates that reduction may occur inside application expressions and addition expressions, always from left to right.

To prepare for the reduction relation, we first define a metafunction for summation.

```
(define-metafunction Ev
  Σ : number ... -> number
  [(Σ number ...)
   ,(apply + (term (number ...)))])
```

This lifts the Racket function + to Redex, giving it the name Σ. The unquote (comma) in the definition of the metafunction escapes into Racket, using apply and + to sum up the sequence of numbers that were passed to Σ. As we've noted before, the term operator is like Racket's quasiquote operator, but it is also sensitive to Redex pattern variables. In this case, (term (number ...)) produces a list of numbers, extracting the arguments from the call to Σ.

To define a reduction relation, we also have to define substitution. Generally speaking, substitution functions are tricky to get right and, since they generally are not shown in papers, we have defined a workhorse substitution function in Racket that runs in near linear time. The source code is included with Redex. If you'd like to have a look, evaluate the expression below in the REPL to find the precise path on your system:

```
(collection-file-path "tut-subst.rkt" "redex")
```

(Test cases are in "test/tut-subst-test.rkt", relative to "tut-subst.rkt".)

That file contains the definition of the function subst/proc, which expects four arguments: a predicate for determining if an expression is a variable, a list of variables to replace, a list of terms to replace them with, and a term to do the replacement inside (the function has a hard-wired notion of the shape of all binding forms, but is agnostic to the other expression forms in the language).

To use this substitution function, we also need to lift it into Redex, just like we did for Σ.

```
(require redex/tut-subst)
(define-metafunction Ev
  subst : x v e -> e
  [(subst x v e)
   ,(subst/proc x? (list (term x)) (list (term v)) (term e))])
(define x? (redex-match Ev x))
```

In this case, we use `term` to extract the values of the Redex variables `x`, `v`, and `e` and then pass them to `subst/proc`.

The definition of `x?` uses a specialized, more efficient form of `redex-match`; supplying `redex-match` with only two arguments permits Redex to do some processing of the pattern, and it results in a predicate that matches the pattern in the given language (which we can supply directly to `subst/proc`).

Using that substitution function, we can now give the reduction relation.

```
(define red
  (reduction-relation
   Ev
   #:domain p
   (--> (in-hole P (if0 0 e_1 e_2))
        (in-hole P e_1)
        "if0t")
   (--> (in-hole P (if0 v e_1 e_2))
        (in-hole P e_2)
        (side-condition (not (equal? 0 (term v))))
        "if0f")
   (--> (in-hole P ((fix (λ (x t) e)) v))
        (in-hole P (((λ (x t) e) (fix (λ (x t) e))) v))
        "fix")
   (--> (in-hole P ((λ (x t) e) v))
        (in-hole P (subst x v e))
        "βv")
   (--> (in-hole P (+ number ...))
        (in-hole P (Σ number ...))
        "+")
   (--> (e_1 ... (in-hole E (amb e_2 ...)) e_3 ...)
        (e_1 ... (in-hole E e_2) ... e_3 ...)
        "amb")))
```

The `reduction-relation` form accepts the name of a language, the domain of the relation (`p` in this case), and then a series of rewriting rules, each of the form `(--> pattern pattern)`.

The first rule replaces `if0` expressions when the test position is `0` by the second subexpression (the true branch). It uses the `in-hole` pattern, the Redex notation for context decomposition. In this case, it decomposes a program into some `P` with an appropriate `if0` expression inside, and then the right-hand side of the rule places `e_1` into the same context.

The rule for the false branch should apply when the test position is any value except `0`. To establish this, we use a `side-condition`. In general, a side-condition is a Racket expression that is evaluated for expressions where the pattern matches; if it returns true, then the rule fires. In this case, we use `term` to extract the value of `v` and then compare it with `0`.

To explore the behavior of a reduction relation, Redex provides `traces` and `stepper`. They both accept a reduction relation and a term, and then show you how that term reduces in a GUI. The GUI that `traces` uses is better suited to a quick overview of the reduction graph and `stepper` is better for more detailed explorations of reduction graphs that have larger expressions in them.

Exercise 7

Evaluate

```
(traces red
        (term ((+ (amb 1 2)
                  (amb 10 20)))))
```

It does not show all of the terms by default, but one click the Reduce button shows them all.

If you have Graphviz installed, Redex can use it to lay out the graph; click Fix Layout and Redex will call out to `dot` to lay out the graph.

Exercise 8

Design a function that accepts a number `n` and evaluates (ambiguously) to any of the numbers between `n` and `0`. Call it with `10` and look at the results in both `traces` and `stepper`.

Hint: to subtract 1 from `n`, use `(+ n -1)`

1.5 Testing Reduction Relations

Redex provides `test-->>` for using testing the transitive closure of a reduction relation. If you supply it a reduction relation and two terms, it will reduce the first term and make sure that it yields the second.

```
> (test-->>
   red
   (term ((if0 1 2 3)))
   (term (3)))

> (test-->>
   red
   (term ((+ (amb 1 2)
             (amb 10 20))))
   (term (11 21 12 22)))

> (test-results)
Both tests passed.
```

The `test-->` form is like `test-->>`, except that it only reduces the term a single step.

```
> (test-->
   red
   (term ((+ (amb 1 2) 3)))
   (term ((+ 1 3) (+ 2 3))))

> (test-results)
One test passed.
```

If a term produces multiple results, then each of the results must be listed.

```
> (test-->
   red
   (term ((+ 1 2) (+ 3 4)))
   (term (3 (+ 3 4)))
   (term ((+ 1 2) 7)))

> (test-results)
One test passed.
```

Technically, when using `test-->>`, it finds all irreducible terms that are reachable from the given term, and expects them all to be listed, with one special case: when it detects a cycle in the reduction graph, then it signals an error. (Watch out: when the reduction graph is infinite and there are no cycles, then `test-->>` consumes all available memory.)

```
> (test-->>
   red
   (term (((fix (λ (x (→ num num)) x)) 1))))
FAILED :46.0
found a cycle in the reduction graph

> (test-results)
1 test failed (out of 1 total).
```

To suppress this behavior, pass `#:cycles-ok` to `test-->>`.

```
> (test-->>
   red #:cycles-ok
   (term (((fix (λ (x (→ num num)) x)) 1))))
```

```
> (test-results)
One test passed.
```

This test case has no expected results but still passes, since there are no irreducible terms reachable from the given term.

Exercise 9

Extend λ to support multiple arguments. Use the notation `(λ (x t) ... e)` for multi-arity λ expressions because the `subst/proc` function works properly with λ expressions of that shape. Use this definition of `subst`.

```
(define-metafunction Ev
  subst : (x v) ... e -> e
  [(subst (x v) ... e)
   ,(subst/proc x?
                (term (x ...))
                (term (v ...))
                (term e))])
```

Also, adjust the typing rules (and do not forget that an ellipsis can be named, as discussed in exercise 4).

1.6 Random Testing

Random testing is a cheap and easy way to find counter-examples to false claims. Unsurprisingly, it is hard to pin down exactly which false claims that random testing can provide counter-examples to. Hanford (1970) put it best (calling his random test case generator a syntax machine): "[a]lthough as a writer of test cases, the syntax machine is certainly unintelligent, it is also uninhibited. It can test a [language] processor with many combinations that would not be thought of by a human test case writer."

To get a sense of how random testing works, we define this Racket predicate

```
(define (progress-holds? e)
  (if (types? e)
      (or (v? e)
          (reduces? e))
      #t))
```

that captures the statement of the progress result.

The three helper functions `types?`, `v?`, and `reduces?` can be defined by using our earlier definitions of typing, the grammar, and the reduction relation, plus calls into Redex:

```
(define (types? e)
  (not (null? (judgment-holds (types · ,e t)
                              t))))

(define v? (redex-match Ev v))

(define (reduces? e)
  (not (null? (apply-reduction-relation
               red
               (term (,e))))))
```

The only new construct here is `apply-reduction-relation`, which accepts a reduction and a term, and returns a list of expressions that it reduces to in a single step. Thus, `reduces?` returns `#t` when the given term is reducible and `#f` otherwise.

Putting all of that together with `redex-check` will cause Redex to randomly generate 1,000 `e`s and attempt to falsify them:

```
> (redex-check Ev e (progress-holds? (term e)))
redex-check: no counterexamples in 1000 attempts
```

The `redex-check` form accepts the name of a language (`Ev` in this case), a pattern (`e` in this case), and a Racket expression that returns a boolean. It randomly generates expressions matching the pattern and then invokes the expression in an attempt to elicit `#f` from the Racket expression.

We can also ask `redex-check` how good of a job it is doing. Specifically, this expression re-runs the same random test, but this time sets up some instrumenting infrastructure to determine how many of the reduction rules fire during the testing. In this case, we create a coverage value that indicates that we're interested in how many of the rules in `red` fired, and then we install it using the `relation-coverage` parameter. In the dynamic extent of the `parameterize`, then, the relation will record how it gets tested. Once that returns we can use `covered-cases` to see exactly how many times each case fired.

```
> (let ([c (make-coverage red)])
    (parameterize ([relation-coverage (list c)])
      (redex-check Ev e (progress-holds? (term e))))
    (covered-cases c))
redex-check: no counterexamples in 1000 attempts
'(("+" . 18)
  ("amb" . 32)
  ("fix" . 0)
```

```
  ("if0f" . 0)
  ("if0t" . 9)
  ("βv" . 2))
```

Not many of them! To improve coverage, we can tell `redex-check` to try generating expressions using the patterns on the left-hand side of the rules to generate programs, and then check to see if progress for each of the expressions in the program:

```
> (check-reduction-relation
   red
   (λ (p) (andmap progress-holds? p)))
check-reduction-relation: no counterexamples in 1000 attempts
(with each clause)
```

The `check-reduction-relation` is a shorthand for using `redex-check` to generate elements of the domain of the given reduction relation (`red` in this case), and then pass them to the given function, attempting to elicit `#f`.

In this case, since the domain of `red` is `p`, the random generator produces sequences of `e` expressions, which are reflected into Redex as lists, and so we simply try to see if progress holds for each element of the list, using `andmap`.

Still no test failures, but installing the same coverage testing boilerplate around the call to `check-reduction-relation` tells us that we got much better coverage of the reduction system.

```
> (let ([c (make-coverage red)])
    (parameterize ([relation-coverage (list c)])
      (check-reduction-relation
       red
       (λ (p) (andmap progress-holds? p)))
      (covered-cases c)))
check-reduction-relation: no counterexamples in 1000 attempts
(with each clause)
'(("+" . 1970)
  ("amb" . 2096)
  ("fix" . 1)
  ("if0f" . 221)
  ("if0t" . 112)
  ("βv" . 42))
```

Exercise 10

Remove one of the productions from `E` (except `hole`) and find an expression in the revised system that causes `progress?` to return `#f`.

See if `redex-check` can also falsify progress for the same system.

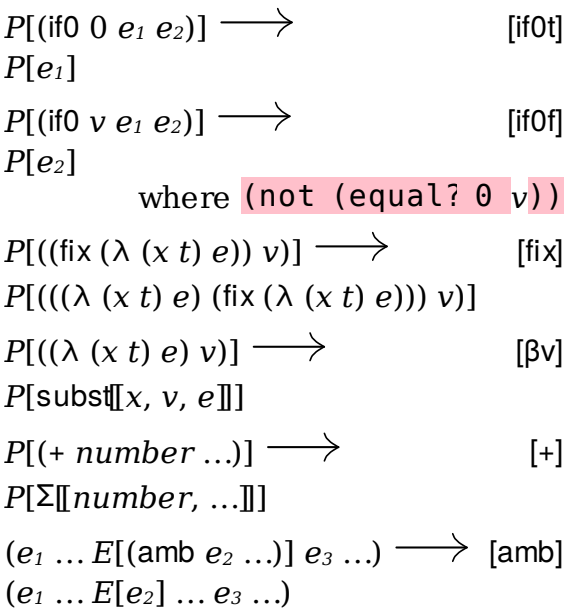
Exercise 11

Formulate and randomly check type preservation. Usually, this lemma says that if an expression has a type and it takes a step, then it produces an expression with the same type. In this case, however, formulate a predicate that accepts an expression and checks that, if it has a type and takes a step, then all of the resulting expressions in the new program have the same type.

1.7 Typesetting the Reduction Relation

Redex's typesetting facilities accept languages, metafunctions, reduction relations, and judgment-forms and produce typeset output that can be included directly into a figure in a paper.

```
> (render-reduction-relation red)
```



The result of `render-reduction-relation` is rendered directly in DrRacket's interactions window, and also can be saved as a `".ps"` file by passing the name of the file as the second argument to `render-reduction-relation`.

Redex's typesetting also interoperates with the `pict` library. If we pull it in with a `require`:

```
(require pict)
```

then we can use the pict primitives to combine typeset fragments into a larger whole.

```
> (scale (vl-append
          20
          (language->pict Ev)
          (reduction-relation->pict red))
         3/2)
```

$p ::= (e \ldots)$
$P ::= (e \ldots E\ e \ldots)$
$E ::= (v\ E)$
$\quad\quad | \ (E\ e)$
$\quad\quad | \ (+\ v \ldots E\ e \ldots)$
$\quad\quad | \ (\text{if0}\ E\ e\ e)$
$\quad\quad | \ (\text{fix}\ E)$
$\quad\quad | \ []$
$v ::= (\lambda\ (x\ t)\ e)$
$\quad\quad | \ (\text{fix}\ v)$
$\quad\quad | \ number$

$P[(\text{if0}\ 0\ e_1\ e_2)] \longrightarrow$ [if0t]
$P[e_1]$

$P[(\text{if0}\ v\ e_1\ e_2)] \longrightarrow$ [if0f]
$P[e_2]$
where `(not (equal? 0 v))`

$P[((\text{fix}\ (\lambda\ (x\ t)\ e))\ v)] \longrightarrow$ [fix]
$P[(((\lambda\ (x\ t)\ e)\ (\text{fix}\ (\lambda\ (x\ t)\ e)))\ v)]$

$P[((\lambda\ (x\ t)\ e)\ v)] \longrightarrow$ [βv]
$P[\text{subst}[\![x, v, e]\!]]$

$P[(+\ number \ldots)] \longrightarrow$ [+]
$P[\Sigma[\![number, \ldots]\!]]$

$(e_1 \ldots E[(\text{amb}\ e_2 \ldots)]\ e_3 \ldots) \longrightarrow$ [amb]
$(e_1 \ldots E[e_2] \ldots e_3 \ldots)$

Generally speaking, Redex has reasonable default ways to typeset its definitions, except when they escapes to Racket. In that case, it typesets the code in a fixed-width font and makes the background pink to call our attention to it. While it is possible to use `with-unquote-rewriter` to tell Redex how to typeset those regions, often it is easier to define a metafunction and call it. In this case, we can use `different` (defined earlier).

```
(define if0-false-rule
  (reduction-relation
   Ev #:domain p
   (--> (in-hole P (if0 v e_1 e_2))
        (in-hole P e_2)
        (side-condition (term (different v 0)))
        "if0f")))
```

Now when we typeset this reduction-relation there is no pink.

```
> (render-reduction-relation if0-false-rule)
```

$$P[(\text{if0}\ v\ e_1\ e_2)] \longrightarrow [\text{if0f}]$$
$$P[e_2]$$
$$\text{where different}[\![v, 0]\!]$$

Still, the typesetting is non-optimal, so we can use `with-compound-rewriter` to adjust the way calls to `different` typeset.

```
> (with-compound-rewriter
   'different
   (λ (lws)
     (list "" (list-ref lws 2) " ≠ " (list-ref lws 3) ""))
  (render-reduction-relation if0-false-rule))
```

$$P[(\text{if0}\ v\ e_1\ e_2)] \longrightarrow [\text{if0f}]$$
$$P[e_2]$$
$$\text{where } v \neq 0$$

The compound rewriter is given a list of `lw` structs that correspond to the untypeset sequence for a use of `different`, and then can replace them with a different set of strings and `lw`s. For more details on the structure of `lw` structs and to experiment with them, see `to-lw`.

Exercise 12

Redex uses the indentation and newlines in the program source code to determine where the line breaks in the printed output goes, instead of using a pretty-printer, so as to give Redex programmers fine-grained control over how their models typeset.

Exploit this facility so that this expression produces an expression with a minimum amount of whitespace within its bounding box. (The call to `frame` helps to clarify where the bounding box is.)

```
(frame
 (vl-append
  20
  (language->pict Ev)
  (reduction-relation->pict red)))
```

That is, adjust the whitespace in `Ev` so that it fills as much of the width established by rendering `red`.

Exercise 13

Typeset `types`. Use a compound rewriter so a use of `(type Γ e t)` is rendered as

```
Γ ⊢ e : t
```

2 The Redex Reference

To load Redex use:

```
(require redex)
```
package: redex-gui-lib

which provides all of the names documented in this library.

Alternatively, use the `redex/reduction-semantics` and `redex/pict` modules, which provide only non-GUI functionality (i.e., everything except `redex/gui`), making them suitable for programs which should not depend on `racket/gui/base`.

2.1 Patterns

```
(require redex/reduction-semantics)
```
package: redex-lib

This section covers Redex's *pattern* language, used in many of Redex's forms.

Note that pattern matching is caching (including caching the results of side-conditions). This means that once a pattern has matched a given term, Redex assumes that it will always match that term.

This is the grammar for the Redex pattern language. Non-terminal references are wrapped with angle brackets; otherwise identifiers in the grammar are terminals.

```
pattern = any
        | _
        | number
        | natural
        | integer
        | real
        | string
        | boolean
        | variable
        | (variable-except <id> ...)
        | (variable-prefix <id>)
        | variable-not-otherwise-mentioned
        | hole
        | symbol
        | (name <id> <pattern>)
        | (in-hole <pattern> <pattern>)
        | (hide-hole <pattern>)
        | (side-condition <pattern> guard)
        | (cross <id>)
        | (<pattern-sequence> ...)
```

```
                  | <racket-constant>

pattern-sequence = <pattern>
                  | ... ; literal ellipsis
                  | ..._id
```

- The `any` pattern matches any sexpression. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `_` pattern matches any sexpression, but does not bind _ as a name, nor can it be suffixed to bind a name.
- The `number` pattern matches any number. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `natural` pattern matches any exact non-negative integer. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `integer` pattern matches any exact integer. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `real` pattern matches any real number. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `string` pattern matches any string. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `boolean` pattern matches `#true` and `#false` (which are the same as `#t` and `#f`, respectively). This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `variable` pattern matches any symbol. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `variable-except` pattern matches any symbol except those listed in its argument. This is useful for ensuring that keywords in the language are not accidentally captured by variables.
- The `variable-prefix` pattern matches any symbol that begins with the given prefix.

- The `variable-not-otherwise-mentioned` pattern matches any symbol except those that are used as literals elsewhere in the language.

- The `hole` pattern matches anything when inside the first argument to an in-hole pattern. Otherwise, it matches only a hole.

- The `symbol` pattern stands for a literal symbol that must match exactly, unless it is the name of a non-terminal in a relevant language or contains an underscore.

  If it is a non-terminal, it matches any of the right-hand sides of that non-terminal. If the non-terminal appears twice in a single pattern, then the match is constrained to expressions that are the same, unless the pattern is part of a grammar, in which case there is no constraint.

  If the symbol is a non-terminal followed by an underscore, for example `e_1`, it is implicitly the same as a name pattern that matches only the non-terminal, `(name e_1 e)` for the example. Accordingly, repeated uses of the same name are constrained to match the same expression.

  If the symbol is a non-terminal followed by `_!_`, for example `e_!_1`, it is also treated as a pattern, but repeated uses of the same pattern are constrained to be different. For example, this pattern:

  ```
  (e_!_1 e_!_1 e_!_1)
  ```

  matches lists of three `e`s, but where all three of them are distinct.

  If the `_!_` is used under the ellipsis then the ellipsis is effectively ignored. That is, a pattern like this:

  ```
  (e_!_1 ... e_!_1)
  ```

  matches all sequences of `e`s that have at least one element and are all distinct.

  Unlike a `_` pattern, the `_!_` patterns do not bind names.

  If `_` names and `_!_` are mixed, they are treated as separate. That is, this pattern `(e_1 e_!_1)` matches just the same things as `(e e)`, but the second doesn't bind any variables.

  If the symbol otherwise has an underscore, it is an error.

- The pattern `(name symbol pattern)` matches *`pattern`* and binds using it to the name `symbol`.

- The `(in-hole pattern pattern)` pattern matches the first *`pattern`*. This match must include exactly one match against the second *`pattern`*. If there are zero matches or more than one match, an exception is raised.

  When matching the first argument of in-hole, the `hole` pattern matches any sexpression. Then, the sexpression that matched the hole pattern is used to match against the second pattern.

- The `(hide-hole pattern)` pattern matches what the embedded *pattern* matches but if the pattern matcher is looking for a decomposition, it ignores any holes found in that *pattern*.

- The `(side-condition pattern guard)` pattern matches what the embedded *pattern* matches, and then the guard expression is evaluated. If it returns `#f`, the pattern fails to match, and if it returns anything else, the pattern matches. Any occurrences of `name` in the pattern (including those implicitly there via `_` patterns) are bound using `term-let` in the guard.

- The `(cross symbol)` pattern is used for the compatible closure functions. If the language contains a non-terminal with the same name as `symbol`, the pattern `(cross symbol)` matches the context that corresponds to the compatible closure of that non-terminal.

- The `(pattern-sequence ...)` pattern matches a sexpression list, where each pattern-sequence element matches an element of the list. In addition, if a list pattern contains an ellipsis, the ellipsis is not treated as a literal, instead it matches any number of duplicates of the pattern that came before the ellipses (including 0). Furthermore, each `(name symbol pattern)` in the duplicated pattern binds a list of matches to `symbol`, instead of a single match. (A nested duplicated pattern creates a list of list matches, etc.) Ellipses may be placed anywhere inside the row of patterns, except in the first position or immediately after another ellipses.

  Multiple ellipses are allowed. For example, this pattern:

  ```
  ((name x a) ... (name y a) ...)
  ```

  matches this sexpression:

  ```
  (term (a a))
  ```

  three different ways. One where the first `a` in the pattern matches nothing, and the second matches both of the occurrences of `a`, one where each named pattern matches a single `a` and one where the first matches both and the second matches nothing.

  If the ellipses is named (i.e., has an underscore and a name following it, like a variable may), the pattern matcher records the length of the list and ensures that any other occurrences of the same named ellipses must have the same length.

  As an example, this pattern:

  ```
  ((name x a) ..._1 (name y a) ..._1)
  ```

  only matches this sexpression:

  ```
  (term (a a))
  ```

one way, with each named pattern matching a single a. Unlike the above, the two patterns with mismatched lengths is ruled out, due to the underscores following the ellipses.

Also, like underscore patterns above, if an underscore pattern begins with `..._!_`, then the lengths must be different.

Thus, with the pattern:

```
((name x a) ..._!_1 (name y a) ..._!_1)
```

and the expression

```
(term (a a))
```

two matches occur, one where `x` is bound to `'()` and `y` is bound to `'(a a)` and one where `x` is bound to `'(a a)` and `y` is bound to `'()`.

```
(redex-match lang pattern any)
(redex-match lang pattern)
```

If `redex-match` receives three arguments, it matches the pattern (in the language) against its third argument. If it matches, this returns a list of match structures describing the matches (see `match?` and `match-bindings`). If it fails, it returns `#f`.

If `redex-match` receives only two arguments, it builds a procedure for efficiently testing if expressions match the pattern, using the language *lang*. The procedures accepts a single expression and if the expression matches, it returns a list of match structures describing the matches. If the match fails, the procedure returns `#f`.

Examples:

```
> (define-language nums
    (AE number
        (+ AE AE)))

> (redex-match nums
               (+ AE_1 AE_2)
               (term (+ (+ 1 2) 3)))
(list (match (list (bind 'AE_1 '(+ 1 2)) (bind 'AE_2 3))))
> (redex-match nums
               (+ AE_1 (+ AE_2 AE_3))
               (term (+ (+ 1 2) 3)))
#f
> (redex-match nums
               (+ AE_1 AE_1)
               (term (+ (+ 1 2) 3)))
```

```
#f
```

```
(redex-match? lang pattern any)
(redex-match? lang pattern)
```

Like `redex-match`, except it returns only a boolean indicating if the match was successful.

Examples:

```
> (define-language nums
    (AE number
        (+ AE AE)))

> (redex-match? nums
                (+ AE_1 AE_2)
                (term (+ (+ 1 2) 3)))
#t
> (redex-match? nums
                (+ AE_1 AE_1)
                (term (+ (+ 1 2) 3)))
#f
```

```
(match? val) → boolean?
  val : any/c
```

Determines if a value is a `match` structure.

```
(match-bindings m) → (listof bind?)
  m : match?
```

This returns a list of `bind` structs that binds the pattern variables in this match.

```
(struct bind (name exp)
    #:extra-constructor-name make-bind)
  name : symbol?
  exp : any/c
```

Instances of this struct are returned by `redex-match`. Each `bind` associates a name with an s-expression from the language, or a list of such s-expressions, if the `(name ...)` clause is followed by an ellipsis. Nested ellipses produce nested lists.

```
(caching-enabled?) → boolean?
(caching-enabled? on?) → void?
  on? : boolean?
```

When this parameter is `#t` (the default), Redex caches the results of pattern matching and metafunction evaluation. There is a separate cache for each pattern and metafunction; when one fills (see `set-cache-size!`), Redex evicts all of the entries in that cache.

Caching should be disabled when matching a pattern that depends on values other than the in-scope pattern variables or evaluating a metafunction that reads or writes mutable external state.

```
(set-cache-size! size) → void?
  size : positive-integer?
```

Changes the size of the per-pattern and per-metafunction caches.

The default size is `350`.

```
(check-redudancy) → boolean?
(check-redudancy check?) → void?
  check? : boolean?
```

Ambiguous patterns can slow down Redex's pattern matching implementation significantly. To help debug such performance issues, set the `check-redundancy` parameter to `#t`. This causes Redex to, at runtime, report any redundant matches that it encounters.

2.2 Terms

Object language expressions in Redex are written using `term`. It is similar to Racket's `quote` (in many cases it is identical) in that it constructs lists as the visible representation of terms.

The grammar of *term*s is (note that an ellipsis stands for repetition unless otherwise indicated):

```
          term = identifier
               | (term-sequence ...)
               | ,racket-expression
               | (in-hole term term)
               | hole
               | #t
               | #f
               | string

 term-sequence = term
               | ,@racket-expression
               | ... ; literal ellipsis
```

- A term written *identifier* is equivalent to the corresponding symbol, unless the

identifier is bound by `term-let`, `define-term`, or a pattern variable or the identifier is `hole` (as below).

- A term written (*term-sequence* ...) constructs a list of the terms constructed by the sequence elements.
- A term written ,*racket-expression* evaluates the `racket-expression` and substitutes its value into the term at that point.
- A term written ,@*racket-expression* evaluates the `racket-expression`, which must produce a list. It then splices the contents of the list into the expression at that point in the sequence.
- A term written (`in-hole` *term term*) is the dual to the pattern `in-hole` – it accepts a context and an expression and uses `plug` to combine them.
- A term written `hole` produces a hole.
- A term written as a literal boolean or a string produces the boolean or the string.

```
(term term)
(term term #:lang lang-id)
```

Used for construction of a term.

It behaves similarly to `quasiquote`, except for a few special forms that are recognized (listed below) and that names bound by `term-let` are implicitly substituted with the values that those names were bound to, expanding ellipses as in-place sublists (in the same manner as syntax-case patterns).

The optional `#:lang` keyword must supply an identifier bound by `define-language`, and adds a check that any symbol containing an underscore in *term* could have been bound by a pattern in the language referenced by *lang-id*. In practice, this means that the underscore must be preceded by a non-terminal in that language or a built-in *pattern* such as `number`. This form of `term` is used internally by default, so this check is applied to terms that are constructed by Redex forms such as `reduction-relation` and `define-metafunction`.

For example,

```
(term-let ([body '(+ x 1)]
           [(expr ...) '(+ - (values * /))]
           [((id ...) ...) '((a) (b) (c d))])
  (term (let-values ([(id ...) expr] ...) body)))
```

evaluates to

```
'(let-values ([(a) +]
              [(b) -]
              [(c d) (values * /)])
   (+ x 1))
```

It is an error for a term variable to appear in an expression with an ellipsis-depth different from the depth with which it was bound by `term-let`. It is also an error for two `term-let`-bound identifiers bound to lists of different lengths to appear together inside an ellipsis.

`hole`

Recognized specially within `term`. A `hole` form is an error elsewhere.

`in-hole`

Recognized specially within `reduction-relation`. An `in-hole` form is an error elsewhere.

```
(term-let ([tl-pat expr] ...) body)

    tl-pat = identifier
           | (tl-pat-ele ...)

tl-pat-ele = tl-pat
           | tl-pat ... ; a literal ellipsis
```

Matches each given id pattern to the value yielded by evaluating the corresponding expression and binds each variable in the id pattern to the appropriate value (described below). These bindings are then accessible to the *term* syntactic form.

Note that each ellipsis should be the literal symbol consisting of three dots (and the ... elsewhere indicates repetition as usual). If *tl-pat* is an identifier, it matches any value and binds it to the identifier, for use inside `term`. If it is a list, it matches only if the value being matched is a list value and only if every subpattern recursively matches the corresponding list element. There may be a single ellipsis in any list pattern; if one is present, the pattern before the ellipses may match multiple adjacent elements in the list value (possibly none).

This form is a lower-level form in Redex, and not really designed to be used directly. For `let`-like forms that use Redex's full pattern matching facilities, see `redex-let`, `redex-let*`, `term-match`, `term-match/single`.

```
(redex-let language ([pattern expression] ...) body ...+)
```

Like `term-let` but the left-hand sides are Redex patterns, interpreted according to the specified language. It is a syntax error for two left-hand sides to bind the same pattern variable.

This form raises an exception recognized by exn:fail:redex? if any right-hand side does not match its left-hand side in exactly one way.

In some contexts, it may be more efficient to use term-match/single (lifted out of the context).

```
(redex-let* language ([pattern expression] ...) body ...+)
```

The let* analog of redex-let.

```
(define-term identifier term)
```

Defines *identifier* for use in term templates.

```
(term-match language [pattern expression] ...)
```

This produces a procedure that accepts term (or quoted) expressions and checks them against each pattern. The function returns a list of the values of the expression where the pattern matches. If one of the patterns matches multiple times, the expression is evaluated multiple times, once with the bindings in the pattern for each match.

When evaluating a term-match expression, the patterns are compiled in an effort to speed up matching. Using the procedural result multiple times to avoid compiling the patterns multiple times.

```
(term-match/single language [pattern expression] ...)
```

This produces a procedure that accepts term (or quoted) expressions and checks them against each pattern. The function returns the expression behind the first successful match. If that pattern produces multiple matches, an error is signaled. If no patterns match, an error is signaled.

Raises an exception recognized by exn:fail:redex? if no clauses match or if one of the clauses matches multiple ways.

When evaluating a term-match/single expression, the patterns are compiled in an effort to speed up matching. Using the procedural result multiple times to avoid compiling the patterns multiple times.

```
(plug context expression) → any
  context : any/c
  expression : any/c
```

The first argument to this function is an sexpression to plug into. The second argument is the sexpression to replace in the first argument. It returns the replaced term. This is also used when a term sub-expression contains in-hole.

```
(variable-not-in t var) → symbol?
  t : any/c
  var : symbol?
```

This helper function accepts an sexpression and a variable. It returns a variable not in the sexpression with a prefix the same as the second argument.

```
(variables-not-in t vars) → (listof symbol?)
  t : any/c
  vars : (listof symbol?)
```

This function, like variable-not-in, makes variables that do no occur in its first argument, but it returns a list of such variables, one for each variable in its second argument.

Does not expect the input symbols to be distinct, but does produce variables that are always distinct.

```
(exn:fail:redex? v) → boolean?
  v : any/c
```

Returns `#t` if its argument is a Redex exception record, and `#f` otherwise.

2.3 Languages

```
(define-language lang-name
  non-terminal-def ...)

non-terminal-def = (non-terminal-name ...+ ::= pattern ...+)
                 | (non-terminal-name pattern ...+)
                 | ((non-terminal-name ...+) pattern ...+)
```

This form defines the grammar of a language. It allows the definition of recursive patterns, much like a BNF, but for regular-tree grammars. It goes beyond their expressive power, however, because repeated `name` patterns and side-conditions can restrict matches in a context-sensitive way.

A *non-terminal-def* comprises one or more non-terminal names (considered aliases) followed by one or more productions.

For example, the following defines *lc-lang* as the grammar of the λ-calculus:

```
(define-language lc-lang
  (e (e e ...)
```

```
   x
   v)
(c (v ... c e ...)
   hole)
(v (λ (x ...) e))
(x variable-not-otherwise-mentioned))
```

with non-terminals `e` for the expression language, `x` for variables, `c` for the evaluation contexts and `v` for values.

`::=`

A non-terminal's names and productions may be separated by the keyword `::=`. Use of the `::=` keyword outside a language definition is a syntax error.

```
(define-extended-language extended-lang base-lang
  non-terminal-def ...)

non-terminal-def = (non-terminal-name ...+ ::= pattern ...+)
                 | (non-terminal-name pattern ...+)
                 | ((non-terminal-name ...+) pattern ...+)
```

This form extends a language with some new, replaced, or extended non-terminals. For example, this language:

```
(define-extended-language lc-num-lang
  lc-lang
  (v ....      ; extend the previous `v' non-terminal
     number
     +)
  (x (variable-except λ +)))
```

extends *lc-lang* with two new alternatives (`+` and `number`) for the `v` non-terminal, carries forward the `e` and `c` non-terminals, and replaces the `x` non-terminal with a new one (which happens to be equivalent to the one that would have been inherited).

The four-period ellipses indicates that the new language's non-terminal has all of the alternatives from the original language's non-terminal, as well as any new ones. If a non-terminal occurs in both the base language and the extension, the extension's non-terminal replaces the originals. If a non-terminal only occurs in either the base language, then it is carried forward into the extension. And, of course, extend-language lets you add new non-terminals to the language.

If a language is has a group of multiple non-terminals defined together, extending any one of those non-terminals extends all of them.

```
(define-union-language L base/prefix-lang ...)

base/prefix-lang = lang-id
                 | (prefix lang-id)
```

Constructs a language that is the union of all of the languages listed in the *base/prefix-lang*.

If the two languages have non-terminals in common, then `define-union-language` will combine all of the productions of the common non-terminals. For example, this definition of *L*:

```
(define-language L1
  (e ::=
     (+ e e)
     number))
(define-language L2
  (e ::=
     (if e e e)
     true
     false))
(define-union-language L L1 L2)
```

is equivalent to this one:

```
(define-language L
  (e ::=
     (+ e e)
     number
     (if e e e)
     true
     false))
```

If a language has a prefix, then all of the non-terminals from that language have the corresponding prefix in the union language. The prefix helps avoid unintended collisions between the constituent language's non-terminals.

For example, with two these two languages:

```
(define-language UT
  (e (e e)
     (λ (x) e)
```

```
      x))

(define-language WT
  (e (e e)
     (λ (x t) e)
     x)
  (t (→ t t)
     num))
```

then this declaration:

```
(define-union-language B (ut. UT) (wt. WT))
```

will create a language named `B` containing the non-terminals `ut.e`, `wt.e`, and `wt.t` consisting of the productions listed in the original languages.

```
(language-nts lang) → (listof symbol?)
  lang : compiled-lang?
```

Returns the list of non-terminals (as symbols) that are defined by this language.

```
(compiled-lang? l) → boolean?
  l : any/c
```

Returns `#t` if its argument was produced by `language`, `#f` otherwise.

2.4 Reduction Relations

```
(reduction-relation language domain base-arrow
                    reduction-case ...
                    shortcuts)
```

```
        domain =
               | #:domain pattern

    base-arrow =
               | #:arrow base-arrow-name

reduction-case = (arrow-name pattern term red-extras ...)

    red-extras = rule-name
               | (fresh fresh-clause ...)
               | (side-condition racket-expression)
               | (where pattern term)
               | (judgment-holds (judgment-form-id pat/term ...))
               | (side-condition/hidden racket-expression)
               | (where/hidden pattern term)

     shortcuts =
               | with shortcut ...

      shortcut = [(old-arrow-name pattern term)
                  (new-arrow-name identifier identifier)]

     rule-name = identifier
               | string
               | (computed-name racket-expression)

  fresh-clause = var
               | ((var1 ...) (var2 ...))

      pat/term = pattern
               | term
```

Defines a reduction relation case-wise, one case for each of the *reduction-case* clauses.

The optional *domain* clause provides a contract for the relation, in the form of a pattern that defines the relation's domain and codomain.

The *arrow-name* in each *reduction-case* clause is either *base-arrow-name* (default `-->`) or an arrow name defined by *shortcuts* (described below). In either case, the pattern refers to *language* and binds variables in the corresponding term. Following the pattern and term can be the name of the reduction rule and declarations of fresh variables and side-conditions.

For example, the expression

```
(reduction-relation
```

```
lc-lang
(--> (in-hole c_1 ((λ (variable_i ...) e_body) v_i ...))
     (in-hole c_1 ,(foldl lc-subst
                          (term e_body)
                          (term (v_i ...))
                          (term (variable_i ...))))
     beta-v))
```

defines a reduction relation for the *lc-lang* grammar.

A rule's name (used in typesetting, the stepper, traces, and apply-reduction-relation/tag-with-names) can be given as a literal (an identifier or a string) or as an expression that computes a name using the values of the bound pattern variables (much like the rule's right-hand side). Some operations require literal names, so a rule definition may provide both a literal name and a computed name. In particular, only rules that include a literal name may be replaced using `extend-reduction-relation`, used as breakpoints in the stepper, and selected using render-reduction-relation-rules. The output of apply-reduction-relation/tag-with-names, traces, and the stepper prefers the computed name, if it exists. Typesetting a rule with a computed name shows the expression that computes the name only when the rule has no literal name or when it would not typeset in pink due to `with-unquote-rewriters` in the context; otherwise, the literal name (or nothing) is shown.

Fresh variable clauses generate variables that do not occur in the term being reduced. If the *fresh-clause* is a variable, that variable is used both as a binding in the term and as the prefix for the freshly generated variable. (The variable does not have to be a non-terminal in the language of the reduction relation.)

The second form of *fresh-clause*s generates a sequence of variables. In that case, the ellipses are literal ellipses; that is, you must actually write ellipses in your rule. The variable *var1* is like the variable in first case of a *fresh-clause*; namely it is used to determine the prefix of the generated variables and it is bound in the right-hand side of the reduction rule, but unlike the single-variable fresh clause, it is bound to a sequence of variables. The variable *var2* is used to determine the number of variables generated and *var2* must be bound by the left-hand side of the rule.

The expressions within side-condition clauses and side-condition/hidden clauses are collected with `and` and used as guards on the case being matched. The argument to each side-condition should be a Racket expression, and the pattern variables in the *pattern* are bound in that expression. A side-condition/hidden clause is the same as a side-condition clause, except that the condition is not rendered when typesetting via redex/pict.

Each *where clause* acts as a side condition requiring a successful pattern match, and it can bind pattern variables in the side-conditions (and where clauses) that follow and in the metafunction result. The bindings are the same as bindings in a `term-let` expression. A

`where/hidden` clause is the same as a `where` clause, but the clause is not rendered when typesetting via `redex/pict`.

Each `judgment-holds` clause acts like a `where` clause, where the left-hand side pattern incorporates each of the patterns used in the judgment form's output positions.

Each *`shortcut`* clause defines arrow names in terms of *`base-arrow-name`* and earlier *`shortcut`* definitions. The left- and right-hand sides of a *`shortcut`* definition are identifiers, not patterns and terms. These identifiers need not correspond to non-terminals in *`language`*.

For example, this expression

```
(reduction-relation
 lc-num-lang
 (==> ((λ (variable_i ...) e_body) v_i ...)
      ,(foldl lc-subst
              (term e_body)
              (term (v_i ...))
              (term (variable_i ...))))
 (==> (+ number_1 ...)
      ,(apply + (term (number_1 ...)))))

 with
 [(--> (in-hole c_1 a) (in-hole c_1 b))
  (==> a b)])
```

defines reductions for the λ-calculus with numbers, where the `==>` shortcut is defined by reducing in the context `c`.

A `fresh` clause in *`reduction-case`* defined by shortcut refers to the entire term, not just the portion matched by the left-hand side of shortcut's use.

```
(extend-reduction-relation reduction-relation language more ...)
```

This form extends the reduction relation in its first argument with the rules specified in *`more`*. They should have the same shape as the rules (including the `with` clause) in an ordinary *`reduction-relation`*.

If the original reduction-relation has a rule with the same name as one of the rules specified in the extension, the old rule is removed.

In addition to adding the rules specified to the existing relation, this form also reinterprets the rules in the original reduction, using the new language.

```
(union-reduction-relations r ...) → reduction-relation?
  r : reduction-relation?
```

Combines all of the argument reduction relations into a single reduction relation that steps when any of the arguments would have stepped.

```
(reduction-relation->rule-names r) → (listof symbol?)
  r : reduction-relation?
```

Returns the names of the reduction relation's named clauses.

```
(compatible-closure reduction-relation lang non-terminal)
```

This accepts a reduction, a language, the name of a non-terminal in the language and returns the compatible closure of the reduction for the specified non-terminal.

```
(context-closure reduction-relation lang pattern)
```

This accepts a reduction, a language, a pattern representing a context (i.e., that can be used as the first argument to `in-hole`; often just a non-terminal) in the language and returns the closure of the reduction in that context.

```
(reduction-relation? v) → boolean?
  v : any/c
```

Returns `#t` if its argument is a reduction-relation, and `#f` otherwise.

```
(apply-reduction-relation r t) → (listof any/c)
  r : reduction-relation?
  t : any/c
```

This accepts reduction relation, a term, and returns a list of terms that the term reduces to.

```
(apply-reduction-relation/tag-with-names r
                                         t)
 → (listof (list/c (union false/c string?) any/c))
  r : reduction-relation?
  t : any/c
```

Like `apply-reduction-relation`, but the result indicates the names of the reductions that were used.

```
(apply-reduction-relation*  r
                            t
                           [#:cache-all? cache-all?
                            #:stop-when stop-when])
 → (listof any/c)
  r : reduction-relation?
  t : any/c
  cache-all? : boolean? = (current-cache-all?)
  stop-when : (-> any/c any) = (λ (x) #f)
```

Accepts a reduction relation and a term. Starting from `t`, it follows every reduction path and returns all of the terms that do not reduce further. If there are infinite reduction sequences that do not repeat, this function will not terminate (it does terminate if the only infinite reduction paths are cyclic).

If the *cache-all?* argument is `#t`, then `apply-reduction-relation*` keeps a cache of all visited terms when traversing the graph and does not revisit any of them. This cache can, in some cases, use a lot of memory, so it is off by default and the cycle checking happens by keeping track only of the current path it is traversing through the reduction graph.

The *stop-when* argument controls the stopping criterion. Specifically, it is called with each term that `apply-reduction-relation*` encounters. If it ever returns a true value (anything except `#f`), then `apply-reduction-relation*` considers the term to be irreducible (and so returns it and does not try to reduce it further).

```
(current-cache-all?) → boolean?
(current-cache-all? cache-all?) → void?
  cache-all? : boolean?
```

Controls the behavior of `apply-reduction-relation*` and `test-->>`'s cycle checking. See `apply-reduction-relation*` for more details.

Examples:

```
> (define-language empty-lang)

> (define R
    (reduction-relation
     empty-lang
     (--> 0 1)
     (--> 0 2)
     (--> 2 3)
     (--> 3 3)))

> (apply-reduction-relation R 0)
'(2 1)
> (apply-reduction-relation* R 0)
'(1)
```

```
-->
```

Recognized specially within `reduction-relation`. A `-->` form is an error elsewhere.

```
fresh
```

Recognized specially within `reduction-relation`. A `fresh` form is an error elsewhere.

```
with
```

Recognized specially within `reduction-relation`. A `with` form is an error elsewhere.

2.5 Other Relations

```
(define-metafunction language
  metafunction-contract
  [(name pattern ...) term metafunction-extras ...]
  ...)

metafunction-contract =
                      | id : pattern-sequence ... -> range
                        maybe-pre-condition
                        maybe-post-condition

  maybe-pre-condition = #:pre term
                      |

 maybe-post-condition = #:post term
                      |

               range = pattern
                     | pattern or range
                     | pattern ∨ range
                     | pattern ∪ range

  metafunction-extras = (side-condition racket-expression)
                      | (side-condition/hidden racket-expression)
                      | (where pat term)
                      | (where/hidden pat term)
                      | (judgment-holds
                         (judgment-form-id pat/term ...))
                      | (clause-name name)
```

The `define-metafunction` form builds a function on sexpressions according to the pattern and right-hand-side expressions. The first argument indicates the language used to resolve non-terminals in the pattern expressions. Each of the rhs-expressions is implicitly wrapped in *term*.

The contract, if present, is matched against every input to the metafunction and, if the match fails, an exception is raised. If present, the term inside the *maybe-pre-condition* is evaluated after a successful match to the input pattern in the contract (with any variables from

the input contract bound). If it returns #f, then the input contract is considered to not have matched and an error is also raised. When a metafunction returns, the expression in the *maybe-post-condition* is evaluated (if present), with any variables from the input or output contract bound.

The side-condition, hidden-side-condition, where, and where/hidden clauses behave as in the reduction-relation form.

The resulting metafunction raises an exception recognized by exn:fail:redex? if no clauses match or if one of the clauses matches multiple ways (and that leads to different results for the different matches).

The side-condition extra is evaluated after a successful match to the corresponding argument pattern. If it returns #f, the clause is considered not to have matched, and the next one is tried. The side-condition/hidden extra behaves the same, but is not typeset.

The where and where/hidden extra are like side-condition and side-condition/hidden, except the match guards the clause.

The judgment-holds clause is like side-condition and where, except the given judgment must hold for the clause to be taken.

The *clause-name* is used only when typesetting. See metafunction-cases.

Note that metafunctions are assumed to always return the same results for the same inputs, and their results are cached, unless caching-enabled? is set to #f. Accordingly, if a metafunction is called with the same inputs twice, then its body is only evaluated a single time.

As an example, these metafunctions finds the free variables in an expression in the *lc-lang* above:

```
(define-metafunction lc-lang
  free-vars : e -> (x ...)
  [(free-vars (e_1 e_2 ...))
   (∪ (free-vars e_1) (free-vars e_2) ...)]
  [(free-vars x) (x)]
  [(free-vars (λ (x ...) e))
   (- (free-vars e) (x ...))])
```

The first argument to define-metafunction is the grammar (defined above). Following that are three cases, one for each variation of expressions (e in *lc-lang*). The free variables of an application are the free variables of each of the subterms; the free variables of a variable is just the variable itself, and the free variables of a λ expression are the free variables of the body, minus the bound parameters.

Here are the helper metafunctions used above.

```
(define-metafunction lc-lang
  ∪ : (x ...) ... -> (x ...)
  [(∪ (x_1 ...) (x_2 ...) (x_3 ...) ...)
   (∪ (x_1 ... x_2 ...) (x_3 ...) ...)]
  [(∪ (x_1 ...))
   (x_1 ...)]
  [(∪) ()])

(define-metafunction lc-lang
  - : (x ...) (x ...) -> (x ...)
  [(- (x ...) ()) (x ...)]
  [(- (x_1 ... x_2 x_3 ...) (x_2 x_4 ...))
   (- (x_1 ... x_3 ...) (x_2 x_4 ...))
   (side-condition (not (memq (term x_2) (term (x_3 ...)))))]
  [(- (x_1 ...) (x_2 x_3 ...))
   (- (x_1 ...) (x_3 ...))])
```

Note the side-condition in the second case of `-`. It ensures that there is a unique match for that case. Without it, `(term (- (x x) x))` would lead to an ambiguous match.

Changed in version 1.4 of package `redex-lib`: Added `#:post` conditions.

```
(define-metafunction/extension f language
  metafunction-contract
  [(g pattern ...) term metafunction-extras ...]
  ...)
```

Defines a metafunction *g* as an extension of an existing metafunction *f*. The metafunction *g* behaves as if *f*'s clauses were appended to its definition (with occurrences of *f* changed to *g* in the inherited clauses).

For example, `define-metafunction/extension` may be used to extend the free-vars function above to the forms introduced by the language *lc-num-lang*.

```
(define-metafunction/extension free-vars lc-num-lang
  free-vars-num : e -> (x ...)
  [(free-vars-num number)
   ()]
  [(free-vars-num (+ e_1 e_2))
   (∪ (free-vars-num e_1)
      (free-vars-num e_2))])
```

```
(in-domain? (metafunction-name term ...))
```

Returns `#t` if the inputs specified to *metafunction-name* are legitimate inputs according to *metafunction-name*'s contract, and `#f` otherwise.

```
(define-judgment-form language
  mode-spec
  contract-spec
  invariant-spec
  rule rule ...)
```

```
     mode-spec = #:mode (form-id pos-use ...)

 contract-spec =
               | #:contract (form-id pattern-sequence ...)

invariant-spec = #:inv term
               |

       pos-use = I
               | O

          rule = [premise
                  ...
                  dashes rule-name
                  conclusion]
               | [conclusion
                  premise
                  ...
                  rule-name]

    conclusion = (form-id pat/term ...)

       premise = (judgment-form-id pat/term ...) maybe-ellipsis
               | (where pattern term)
               | (where/hidden pattern term)
               | (side-condition term)
               | (side-condition/hidden term)

     rule-name =
               | string
               | non-ellipsis-non-hypens-var

      pat/term = pattern
               | term

maybe-ellipsis =
               | ...

        dashes = ---
               | ----
               | -----
               | etc.
```

Defines *form-id* as a relation on terms via a set of inference rules. Each rule must be such that its premises can be evaluated left-to-right without "guessing" values for any of their

pattern variables. Redex checks this property using the mandatory *mode-spec* declaration, which partitions positions into inputs `I` and outputs `O`. Output positions in conclusions and input positions in premises must be *term*s; input positions in conclusions and output positions in premises must be *pattern*s. When the optional *contract-spec* declaration is present, Redex dynamically checks that the terms flowing through these positions match the provided patterns, raising an exception recognized by `exn:fail:redex` if not. The term in the optional *invariant-spec* is evaluated after the output positions have been computed and the contract has matched successfully, with variables from the contract bound; a result of `#f` is considered to be a contract violation and an exception is raised.

For example, the following defines addition on natural numbers:

```
> (define-language nats
    (n ::= z (s n)))

> (define-judgment-form nats
    #:mode (sum I I O)
    #:contract (sum n n n)
    [-----------  "zero"
     (sum z n n)]

    [(sum n_1 n_2 n_3)
     ------------------------- "add1"
     (sum (s n_1) n_2 (s n_3))])
```

The `judgment-holds` form checks whether a relation holds for any assignment of pattern variables in output positions.

Examples:

```
> (judgment-holds (sum (s (s z)) (s z) (s (s (s z)))))
#t
> (judgment-holds (sum (s (s z)) (s z) (s (s (s n)))))
#t
> (judgment-holds (sum (s (s z)) (s z) (s (s (s (s n))))))
#f
```

Alternatively, this form constructs a list of terms based on the satisfying pattern variable assignments.

Examples:

```
> (judgment-holds (sum (s (s z)) (s z) (s (s (s n)))) n)
'(z)
> (judgment-holds (sum (s (s z)) (s z) (s (s (s (s n))))) n)
```

```
'()
> (judgment-holds (sum (s (s z)) (s z) (s (s (s n)))) (s n))
'((s z))
```

Declaring different modes for the same inference rules enables different forms of computation. For example, the following mode allows `judgment-holds` to compute all pairs with a given sum.

```
> (define-judgment-form nats
    #:mode (sumr O O I)
    #:contract (sumr n n n)
    [------------
     (sumr z n n)]

    [(sumr n_1 n_2 n_3)
     -------------------------
     (sumr (s n_1) n_2 (s n_3))])

> (judgment-holds (sumr n_1 n_2 (s (s z))) (n_1 n_2))
'(((s (s z)) z) ((s z) (s z)) (z (s (s z))))
```

A rule's `where` and `where/hidden` premises behave as in `reduction-relation` and `define-metafunction`.

Examples:

```
> (define-judgment-form nats
    #:mode (le I I)
    #:contract (le n n)
    [--------
     (le z n)]

    [(le n_1 n_2)
     --------------------
     (le (s n_1) (s n_2))])

> (define-metafunction nats
    pred : n -> n or #f
    [(pred z) #f]
    [(pred (s n)) n])

> (define-judgment-form nats
    #:mode (gt I I)
    #:contract (gt n n)
    [(where n_3 (pred n_1))
```

```
     (le n_2 n_3)
     ----------------------
     (gt n_1 n_2)])

> (judgment-holds (gt (s (s z)) (s z)))
#t
> (judgment-holds (gt (s z) (s z)))
#f
```

A rule's `side-condition` and `side-condition/hidden` premises are similar to those in `reduction-relation` and `define-metafunction`, except that they do not implicitly unquote their right-hand sides. In other words, a premise of the form `(side-condition term)` is equivalent to the premise `(where #t term)`, except it does not typeset with the "#t = ", as that would.

Judgments with exclusively `I` mode positions may also be used in *`term`*s in a manner similar to metafunctions, and evaluate to a boolean.

Examples:

```
> (term (le (s z) (s (s z))))
#t
> (term (le (s z) z))
#f
```

A literal ellipsis may follow a judgment premise when a template in one of the judgment's input positions contains a pattern variable bound at ellipsis-depth one.

Examples:

```
> (define-judgment-form nats
    #:mode (even I)
    #:contract (even n)

    [-------- "evenz"
     (even z)]

    [(even n)
     ---------------- "even2"
     (even (s (s n)))])

> (define-judgment-form nats
    #:mode (all-even I)
    #:contract (all-even (n ...))
    [(even n) ...
     ------------------
     (all-even (n ...))])
```

```
> (judgment-holds (all-even (z (s (s z)) z)))
#t
> (judgment-holds (all-even (z (s (s z)) (s z))))
#f
```

Redex evaluates premises depth-first, even when it doing so leads to non-termination. For example, consider the following definitions:

```
> (define-language vertices
    (v a b c))

> (define-judgment-form vertices
    #:mode (edge I O)
    #:contract (edge v v)
    [(edge a b)]
    [(edge b c)])

> (define-judgment-form vertices
    #:mode (path I I)
    #:contract (path v v)
    [----------
     (path v v)]

    [(path v_2 v_1)
     --------------
     (path v_1 v_2)]

    [(edge v_1 v_2)
     (path v_2 v_3)
     --------------
     (path v_1 v_3)])
```

Due to the second `path` rule, the follow query fails to terminate:

```
> (judgment-holds (path a c))
```

The `"examples"` directory demonstrates three use cases:

- `"typing-rules.rkt"` — defines a type system in a way that supports mechanized typesetting. When a typing judgment form can be given a mode, it can also be encoded as a metafunction using `where` clauses as premises, but Redex cannot typeset that encoding as inference rules.

- "sos.rkt" — defines an SOS-style semantics in a way that supports mechanized typesetting.
- "multi-val.rkt" — defines a judgment form that serves as a multi-valued meta-function.

```
(define-extended-judgment-form language judgment-form-id
  option ...
  rule ...)
```

Defines a new judgment form that extends *judgment-form-id* with additional rules. The *option*s and *rule*s are as in define-judgment-form.

The mode specification in this judgment form and the original must be the same.

```
(judgment-holds judgment)
(judgment-holds judgment term)

judgment = (judgment-form-id pat/term ...)
```

In its first form, checks whether *judgment* holds for any assignment of the pattern variables in judgment-id's output positions. In its second form, produces a list of terms by instantiating the supplied term template with each satisfying assignment of pattern variables.

```
> (judgment-holds (sum (s (s z)) (s z) n))
#t
> (judgment-holds (sum (s (s z)) (s z) n) n)
'((s (s (s z))))
```

See define-judgment-form for more examples.

```
(build-derivations judgment)
```

Constructs all of the derivation trees for *judgment*.

Example:

```
> (build-derivations (even (s (s z))))
(list
 (derivation
  '(even (s (s z)))
  "even2"
  (list (derivation '(even z) "evenz" '()))))
(struct derivation (term name subs)
    #:extra-constructor-name make-derivation)
  term : any/c
  name : (or/c string? #f)
  subs : (listof derivation?)
```

Represents a derivation from a judgment form.

The `term` field holds an s-expression based rendering of the conclusion of the derivation, the `name` field holds the name of the clause with `term` as the conclusion, and `subs` contains the sub-derivations.

See also `build-derivations`.

```
I
```

Recognized specially within `define-judgment-form`, the `I` keyword is an error elsewhere.

```
O
```

Recognized specially within `define-judgment-form`, the `O` keyword is an error elsewhere.

```
(define-relation language
  relation-contract
  [(name pattern ...)
   term ...
   metafunction-extras ...] ...)

relation-contract =
                  | form-id ⊂ pattern x ... x pattern
                  | form-id ⊆ pattern × ... × pattern
```

Similar to `define-judgment-form` but suitable only when every position is an input. There is no associated form corresponding to `judgment-holds`; querying the result uses the same syntax as metafunction application.

The contract specification for a relation restricts the patterns that can be used as input to a relation. For each argument to the relation, there should be a single pattern, using `x` or `×` to separate the argument contracts.

Examples:

```
> (define-language types
    ((τ σ) int
           num
           (τ → τ)))

> (define-relation types
    subtype ⊆ τ × τ
    [(subtype int num)]
    [(subtype (τ_1 → τ_2) (σ_1 → σ_2))
```

```
     (subtype σ_1 τ_1)
     (subtype τ_2 σ_2)]
    [(subtype τ τ)])

> (term (subtype int num))
#t
> (term (subtype (int → int) (num → num)))
#f
> (term (subtype (num → int) (num → num)))
#t
```

```
(current-traced-metafunctions) → (or/c 'all (listof symbol?))
(current-traced-metafunctions traced-metafunctions) → void?
  traced-metafunctions : (or/c 'all (listof symbol?))
```

Controls which metafunctions are currently being traced. If it is `'all`, all of them are. Otherwise, the elements of the list name the metafunctions to trace.

The tracing looks just like the tracing done by the `racket/trace` library, except that the first column printed by each traced call indicate if this call to the metafunction is cached. Specifically, a `c` is printed in the first column if the result is just returned from the cache and a space is printed if the metafunction call is actually performed.

Defaults to `'()`.

2.6 Testing

```
(test-equal e1 e2 option)

option = #:equiv pred-expr
       |

  pred-expr : (-> any/c any/c any/c)
```

Tests to see if *e1* is equal to *e2*, using *pred-expr* as the comparison. It defaults to `(default-equiv)`.

Examples:

```
> (define-language L
    (bt ::=
        empty
        (node any bt bt))
    (lt ::=
        empty
        (node any lt empty)))
```

```
> (define-metafunction L
    linearize/a : bt lt -> lt
    [(linearize/a empty lt) lt]
    [(linearize/a (node any_val bt_left bt_right) lt)
     (node any_val (linearize/a bt_left (linearize/a bt_right lt)) empty)])

> (define-metafunction L
    linearize : bt -> lt
    [(linearize bt) (linearize/a bt empty)])

> (test-equal (term (linearize empty))
              (term empty))

> (test-equal (term (linearize (node 1
                                     (node 2 empty empty)
                                     (node 3 empty empty))))
              (term (node 1 (node 2 (node 3 empty empty) empty) empty)))

> (test-results)
Both tests passed.
```

```
(test-->> rel-expr option ... e1-expr e2-expr ...)

option = #:cycles-ok
       | #:equiv pred-expr
       | #:pred pred-expr

  rel-expr : reduction-relation?
  pred-expr : (--> any/c any)
  e1-expr : any/c
  e2-expr : any/c
```

Tests to see if the term *e1-expr*, reduces to the terms *e2-expr* under *rel-expr*, using *pred-expr* to determine equivalence.

If `#:pred` is specified, it is applied to each reachable term until one of the terms fails to satisfy the predicate (i.e., the predicate returns `#f`). If that happens, then the test fails and a message is printed with the term that failed to satisfy the predicate.

This test uses `apply-reduction-relation*`, so it does not terminate when the resulting reduction graph is infinite, although it does terminate if there are cycles in the (finite) graph.

If `#:cycles-ok` is not supplied then any cycles detected are treated as a test failure. If a *pred-expr* is supplied, then it is used to compare the expected and actual results. If it isn't

supplied, then (default-equiv) is used.

```
(test--> rel-expr option ... e1-expr e2-expr ...)

option = #:equiv pred-expr

  rel-expr : reduction-relation?
  pred-expr : (--> any/c any/c any/c)
  e1-expr : any/c
  e2-expr : any/c
```

Tests to see if the term *e1-expr*, reduces to the terms *e2-expr* in a single *rel-expr* step, using *pred-expr* to determine equivalence (or (default-equiv) if *pred-expr* isn't specified).

Examples:

```
> (define-language L
    (i integer))

> (define R
    (reduction-relation
     L
     (--> i i)
     (--> i ,(add1 (term i)))))

> (define (mod2=? i j)
    (= (modulo i 2) (modulo j 2)))

> (test--> R #:equiv mod2=? 7 1)
FAILED :54.0
expected: 1
  actual: 8
  actual: 7

> (test--> R #:equiv mod2=? 7 1 0)

> (test-results)
1 test failed (out of 2 total).
```

```
(test-->>∃ option ... rel-expr start-expr goal-expr)

option = #:steps steps-expr
```

```
rel-expr : reduction-relation?
start-expr : any/c
goal-expr : (or/c (-> any/c any/c)
                  (not/c procedure?))
steps-expr : (or/c natural-number/c +inf.0)
```

Tests to see if the term *start-expr* reduces according to the reduction relation *rel-expr* to a term specified by *goal-expr* in *steps-expr* or fewer steps (default 1,000). The specification *goal-expr* may be either a predicate on terms or a term itself.

```
test-->>E
```

An alias for `test-->>∃`.

Examples:

```
> (define-language L
    (n natural))

> (define succ-mod8
    (reduction-relation
     L
     (--> n ,(modulo (add1 (term n)) 8))))

> (test-->>∃ succ-mod8 6 2)

> (test-->>∃ succ-mod8 6 even?)

> (test-->>∃ succ-mod8 6 8)
FAILED :61.0
term 8 not reachable from 6

> (test-->>∃ #:steps 6 succ-mod8 6 5)
FAILED :62.0
term 5 not reachable from 6 (within 6 steps)

> (test-results)
2 tests failed (out of 4 total).
```

```
(test-predicate p? e)
```

Tests to see if the value of `e` matches the predicate `p?`.

```
(test-results) → void?
```

Prints out how many tests passed and failed, and resets the counters so that next time this function is called, it prints the test results for the next round of tests.

```
(default-equiv) → (-> any/c any/c any/c)
(default-equiv equiv) → void?
  equiv : (-> any/c any/c any/c)
```

The value of this parameter is used as the default value of the equivalence predicates for `test-equal`, `test-->`, and `test-->>`.

It defaults to `equal?`.

```
(make-coverage subject)

subject = metafunction
        | relation-expr
```

Constructs a structure (recognized by `coverage?`) to contain per-case test coverage of the supplied metafunction or reduction relation. Use with `relation-coverage` and `covered-cases`.

```
(coverage? v) → boolean?
  v : any/c
```

Returns `#t` for a value produced by `make-coverage` and `#f` for any other.

```
(relation-coverage) → (listof coverage?)
(relation-coverage tracked) → void?
  tracked : (listof coverage?)
```

Redex populates the coverage records in *tracked* (default `null`), counting the times that tests exercise each case of the associated metafunction and relations.

```
(covered-cases c) → (listof (cons/c string? natural-number/c))
  c : coverage?
```

Extracts the coverage information recorded in `c`, producing an association list mapping names (or source locations, in the case of metafunctions or unnamed reduction-relation cases) to application counts.

Examples:

```
> (define-language empty-lang)

> (define-metafunction empty-lang
    [(plus number_1 number_2)
     ,(+ (term number_1) (term number_2))])

> (define equals
    (reduction-relation
     empty-lang
     (--> (+) 0 "zero")
     (--> (+ number) number)
     (--> (+ number_1 number_2 number ...)
          (+ (plus number_1 number_2)
             number ...)
          "add")))

> (let ([equals-coverage (make-coverage equals)]
        [plus-coverage (make-coverage plus)])
    (parameterize ([relation-coverage (list equals-coverage
                                            plus-coverage)])
      (apply-reduction-relation* equals (term (+ 1 2 3)))
      (values (covered-cases equals-coverage)
              (covered-cases plus-coverage))))
'(("#f:66:0" . 1) ("add" . 2) ("zero" . 0))
'(("#f:65:0" . 2))
```

```
(generate-term from-pattern)
(generate-term from-judgment-form)
(generate-term from-metafunction)
(generate-term from-reduction-relation)
```

```
          from-pattern = language pattern size-expr kw-args ...
                       | language pattern
                       | language pattern #:i-th index-expr
                       | language pattern #:i-th

     from-judgment-form = language #:satisfying
                          (judgment-form-id pattern ...)
                       | language #:satisfying
                          (judgment-form-id pattern ...)
                          size-expr

      from-metafunction = language #:satisfying
                          (metafunction-id pattern ...) = pattern
                       | language #:satisfying
                          (metafunction-id pattern ...) = pattern
                          size-expr
                       | #:source metafunction size-expr kw-args
                       | #:source metafunction

from-reduction-relation = #:source reduction-relation-expr
                          size-expr kw-args ...
                       | #:source reduction-relation-expr

               kw-args = #:attempt-num attempts-expr
                       | #:retries retries-expr

  size-expr : natural-number/c
  attempt-num-expr : natural-number/c
  retries-expr : natural-number/c
```

Generates terms in a number of different ways:

- *from-pattern*: In the first case, randomly makes an expression matching the given pattern whose size is bounded by *size-expr*; the second returns a function that accepts a size bound and returns a random term. Calling this function (even with the same size bound) may be more efficient than using the first case.

  Examples:

```
> (define-language L
    (e ::=
       (e e)
       (λ (x) e)
       x)
    (x ::= a b c))
```

```
> (for/list ([i (in-range 10)])
    (generate-term L e 3))
'(a
  (c ((λ (a) a) (λ (a) a)))
  (c ((λ (a) b) (a c)))
  (λ (b) ((λ (b) c) c))
  (λ (b) a)
  c
  (b (λ (c) c))
  (a a)
  (λ (b) a)
  ((λ (c) b) c))
```

The `#:i-th` option uses an enumeration of the non-terminals in a language. If *index-expr* is supplied, `generate-term` returns the corresponding term and if it isn't, `generate-term` returns a function from indices to terms.

Example:

```
> (for/list ([i (in-range 9)])
    (generate-term L e #:i-th i))
'(a (a a) (λ (a) a) b (a (a a)) (λ (a) (a a)) c ((a a) a) (λ
(b) a))
```

Base type enumerations such as `boolean`, `natural` and `integer` are what you might expect:

Examples:

```
> (for/list ([i (in-range 10)])
    (generate-term L boolean #:i-th i))
'(#t #f #t #f #t #f #t #f #t #f)
> (for/list ([i (in-range 10)])
    (generate-term L natural #:i-th i))
'(0 1 2 3 4 5 6 7 8 9)
> (for/list ([i (in-range 10)])
    (generate-term L integer #:i-th i))
'(0 1 -1 2 -2 3 -3 4 -4 5)
```

The `real` base type enumeration consists of all integers and flonums, and the `number` pattern consists of complex numbers with real and imaginary parts taken from the `real` enumeration.

Examples:

```
> (for/list ([i (in-range 20)])
    (generate-term L real #:i-th i))
```

```
'(0
  +inf.0
  1
  -inf.0
  -1
  +nan.0
  2
  0.0
  -2
  4.9406564584125e-324
  3
  -4.9406564584125e-324
  -3
  9.8813129168249e-324
  4
  -9.8813129168249e-324
  -4
  1.4821969375237e-323
  5
  -1.4821969375237e-323)
> (for/list ([i (in-range 20)])
    (generate-term L number #:i-th i))
'(0
  0+inf.0i
  +inf.0
  0+1i
  1
  +inf.0+inf.0i
  -inf.0
  +inf.0+1.0i
  -1
  0-inf.0i
  +nan.0
  +inf.0-inf.0i
  2
  1.0+inf.0i
  0.0
  1+1i
  -2
  1.0-inf.0i
  4.9406564584125e-324
  0-1i)
```

The `string` enumeration produces all single character strings before going on to strings with multiple characters. For each character it starts the lowercase Latin characters, then uppercase Latin, and then every remaining Unicode character. The `vari-`

`able` enumeration is the same, except it produces symbols instead of strings.

Examples:

```
> (generate-term L string #:i-th 0)
""
> (generate-term L string #:i-th 1)
"a"
> (generate-term L string #:i-th 26)
"aaaa"
> (generate-term L string #:i-th 27)
"baaa"
> (generate-term L string #:i-th 52)
"cc"
> (generate-term L string #:i-th 53)
"dc"
> (generate-term L string #:i-th 956)
"Eeba"
> (generate-term L variable #:i-th 1)
'a
> (generate-term L variable #:i-th 27)
'baaa
```

The `variable-prefix`, `variable-except`, and `variable-not-otherwise-mentioned` are defined similarly, as you expect.

Examples:

```
> (define-language L
    (used ::= a b c)
    (except ::= (variable-except a))
    (unused ::= variable-not-otherwise-mentioned))

> (for/list ([i (in-range 10)])
    (generate-term L (variable-prefix a:) #:i-th i))
'(a: a:a a:aa a:b a:ba a:aaa a:baa a:c a:ca a:caa)
> (for/list ([i (in-range 10)])
    (generate-term L except #:i-th i))
'(|| aa b ba aaa baa c ca caa ab)
> (for/list ([i (in-range 10)])
    (generate-term L unused #:i-th i))
'(|| aa ba aaa baa ca caa ab bb cb)
```

Finally, the `any` pattern enumerates sexpressions of the above base types.

Example:

```
> (for/list ([i (in-range 20)])
    (generate-term L any #:i-th i))
```

```
'(()
  (())
  0
  (() ())
  ""
  (((())))
  #t
  ((()) ())
  ||
  (() . 0)
  0+inf.0i
  ((()) . 0)
  "a"
  (0)
  #f
  (0 ())
  a
  (0 . 0)
  +inf.0
  (() () ()))
```

In addition, all other pattern types are supported except for mismatch repeat `..._!_` patterns and `side-condition` patterns.

The enumerators do not repeat terms unless the given pattern is ambiguous. Roughly speaking, the enumerator generates all possible ways that a pattern might be parsed and since ambiguous patterns have multiple ways they might be parsed, those multiple parsings turn into repeated elements in the enumeration.

Example:

```
> (for/list ([i (in-range 9)])
    (generate-term L (boolean_1 ... boolean_2 ...) #:i-th i))
'(() (#t) (#t) (#t #t) (#t #t) (#t #t #t) (#t #t) (#t #t #t)
(#t #t #t #t))
```

Other sources of ambiguity are `in-hole` and overlapping non-terminals.

Examples:

```
> (define-language L
    (e ::= (e e) (λ (x) e) x)
    (E ::= hole (e E) (E e))
    (x ::= a b c))

> (for/list ([i (in-range 9)])
    (generate-term L (in-hole E e) #:i-th i))
'(a
  (a a)
```

```
  (a a)
  (a (a a))
  (λ (a) a)
  (a (λ (a) a))
  (a a)
  ((a a) a)
  ((λ (a) a) a))
> (define-language L
    (overlap ::= natural integer))

> (for/list ([i (in-range 10)])
    (generate-term L overlap #:i-th i))
'(0 0 1 1 2 -1 3 2 4 -2)
```

For similar reasons, enumerations for mismatch patterns (using `_!_`) do not work properly when given ambiguous patterns; they may repeat elements of the enumeration.

Examples:

```
> (define-language Bad
    (ambig ::= (x ... x ...)))

> (generate-term Bad (ambig_!_1 ambig_!_1) #:i-th 4)
'((x) (x))
```

In this case, the elements of the resulting list are the same, even though they should not be, according to the pattern. Internally, the enumerator has discovered two different ways to generate `ambig` (one where the `x` comes from the first ellipses and one from the second) but those two different ways produce the same term and so the enumerator incorrectly produces `(x x)`.

- *`from-judgment-form`*: Randomly picks a term that satisfies the given use of the judgment form.

  Examples:

```
> (define-language L
    (nat ::= Z (S nat)))

> (define-judgment-form L
    #:mode (sum I I O)
    [---------------
     (sum Z nat nat)]
    [(sum nat_1 nat_2 nat_3)
     -------------------------------
     (sum (S nat_1) nat_2 (S nat_3))])
```

```
> (for/list ([i (in-range 10)])
    (generate-term L #:satisfying
                   (sum nat_1 nat_2 nat_3)
                   3))
'((sum (S (S Z)) Z (S (S Z)))
  (sum (S (S (S (S Z)))) Z (S (S (S (S Z)))))
  (sum Z (S Z) (S Z))
  (sum (S (S (S (S Z)))) Z (S (S (S (S Z)))))
  (sum (S (S (S (S Z)))) Z (S (S (S (S Z)))))
  (sum (S (S Z)) (S (S Z)) (S (S (S (S Z)))))
  (sum (S (S (S Z))) Z (S (S (S Z))))
  (sum (S (S Z)) (S Z) (S (S (S Z))))
  (sum (S (S (S Z))) Z (S (S (S Z))))
  (sum (S (S Z)) Z (S (S Z))))
```

- *from-metafunction*: The first form randomly picks a term that satisfies the given invocation of the metafunction, using techniques similar to how the *from-judgment-form* case works. The second form uses a more naive approach; it simply generates terms that match the patterns of the cases of the metafunction; it does not consider the results of the metafunctions, nor does it consider patterns from earlier cases when generating terms based on a particular case. The third case is like the second, except it returns a function that accepts the size and keywords arguments that may be more efficient if multiple random terms are generated.

  Examples:

```
> (define-language L
   (n number))

> (define-metafunction L
    [(F one-clause n) ()]
    [(F another-clause n) ()])

> (for/list ([i (in-range 10)])
    (generate-term #:source F 5))
'((another-clause 0)
  (one-clause 1)
  (one-clause 3)
  (another-clause 0)
  (another-clause 2)
  (one-clause 3)
  (one-clause 0)
  (one-clause 0)
  (another-clause 0)
  (another-clause 1))
```

- *from-reduction-relation*: In the first case, `generate-term` randomly picks a rule from the reduction relation and tries to pick a term that satisfies its domain pattern,

returning that. The second case returns a function that accepts the size and keyword arguments that may be more efficient if multiple random terms are generated.

Examples:

```
> (define-language L
   (n number))

> (for/list ([i (in-range 10)])
    (generate-term
     #:source
     (reduction-relation
      L
      (--> (one-clause n) ())
      (--> (another-clause n) ()))
     5))
'((one-clause 1)
  (one-clause 0)
  (one-clause 0)
  (one-clause 1)
  (one-clause 0)
  (another-clause 1)
  (another-clause 0)
  (one-clause 4)
  (one-clause 1)
  (another-clause 0))
```

The argument *size-expr* bounds the height of the generated term (measured as the height of its parse tree).

The optional keyword argument attempt-num-expr (default 1) provides coarse grained control over the random decisions made during generation; increasing attempt-num-expr tends to increase the complexity of the result. For example, the absolute values of numbers chosen for integer patterns increase with attempt-num-expr.

The random generation process does not actively consider the constraints imposed by side-condition or _!_ patterns; instead, it uses a "guess and check" strategy in which it freely generates candidate terms then tests whether they happen to satisfy the constraints, repeating as necessary. The optional keyword argument *retries-expr* (default 100) bounds the number of times that generate-term retries the generation of any pattern. If generate-term is unable to produce a satisfying term after *retries-expr* attempts, it raises an exception recognized by exn:fail:redex:generation-failure?.

```
(redex-check template property-expr kw-arg ...)
```

```
template = language pattern
         | language pattern #:enum
         | language #:satisfying
           (judgment-form-id pattern ...)
         | language #:satisfying
           (metafunction-id pattern ...) = pattern

 kw-arg = #:attempts attempts-expr
        | #:source metafunction
        | #:source relation-expr
        | #:retries retries-expr
        | #:print? print?-expr
        | #:attempt-size attempt-size-expr
        | #:prepare prepare-expr
        | #:keep-going? keep-going?-expr

 property-expr : any/c
 attempts-expr : natural-number/c
 relation-expr : reduction-relation?
 retries-expr : natural-number/c
 print?-expr : any/c
 attempt-size-expr : (-> natural-number/c natural-number/c)
 prepare-expr : (-> any/c any/c)
```

Searches for a counterexample to *property-expr*, interpreted as a predicate universally quantified over the pattern variables bound by the pattern(s) in *template*. redex-check constructs and tests a candidate counterexample by choosing a random term *t* based on *template* and then evaluating *property-expr* using the match-bindings produced by matching *t* against pattern. The form of *template* controls how *t* is generated:

- *language pattern*: In this case, redex-check uses an ad hoc strategy for generating *pattern*. For the first 10 seconds, it uses in-order enumeration to pick terms. After that, it alternates back and forth between in-order enumeration and the ad hoc random generator. After the 10 minute mark, it switches over to using just the ad hoc random generator.
- *language pattern* #:ad-hoc: In this case, redex-check uses an ad hoc random generator to generate terms that match *pattern*.
- *language pattern* #:in-order: In this case, redex-check uses an enumeration of *pattern*, checking each *t* one at a time
- *language pattern* #:uniform-at-random p-value: that to index into an enumeration of *pattern*. If the enumeration is finite, redex-check picks a natural

number uniformly at random; if it isn't, `redex-check` uses a geometric distribution with `p-value` as its probability of zero to pick the number of bits in the index and then picks a number uniformly at random with that number of bits.

- *`language pattern #:enum bound`*: This is similar to `#:uniform-at-random`, except that Redex always picks a random natural number less than `bound` to index into the enumeration
- *`language #:satisfying (judgment-form-id pattern ...)`*: Generates terms that match `pattern` and satisfy the judgment form.
- *`language #:satisfying (metafunction-id pattern ...) = pattern`*: Generates terms matching the two `pattern`s, such that if the first is the argument to the metafunction, the second will be the result.

`redex-check` generates at most *`attempts-expr`* (default `(default-check-attempts)`) random terms in its search. The size and complexity of these terms tend to increase with each failed attempt. The `#:attempt-size` keyword determines the rate at which terms grow by supplying a function that bounds term size based on the number of failed attempts (see `generate-term`'s `size-expr` argument). By default, the bound grows according to the `default-attempt-size` function.

When *`print?-expr`* produces any non-`#f` value (the default), `redex-check` prints the test outcome on `current-output-port`. When *`print?-expr`* produces `#f`, `redex-check` prints nothing, instead

- returning a `counterexample` structure when the test reveals a counterexample,
- returning `#t` when all tests pass, or
- raising a `exn:fail:redex:test` when checking the property raises an exception.

The optional `#:prepare` keyword supplies a function that transforms each generated example before `redex-check` checks *`property-expr`*. This keyword may be useful when *`property-expr`* takes the form of a conditional, and a term chosen freely from the grammar is unlikely to satisfy the conditional's hypothesis. In some such cases, the `prepare` keyword can be used to increase the probability that an example satisfies the hypothesis.

The `#:retries` keyword behaves identically as in `generate-term`, controlling the number of times the generation of any pattern will be reattempted. It can't be used together with `#:satisfying`.

If *`keep-going?-expr`* produces any non-`#f` value, `redex-check` will stop only when it hits the limit on the number of attempts showing all of the errors it finds. This argument is allowed only when *`print?-expr`* is not `#f`.

When passed a metafunction or reduction relation via the optional `#:source` argument, `redex-check` distributes its attempts across the left-hand sides of that metafunction/relation

by using those patterns, rather than `pattern`, as the basis of its generation. It is an error if any left-hand side generates a term that does not match `pattern`. `#:source` cannot be used with `#:satisfying`.

Examples:

```
> (define-language empty-lang)

> (random-seed 0)

> (redex-check
   empty-lang
   ((number_1 ...)
    (number_2 ...))
   (equal? (reverse (append (term (number_1 ...))
                            (term (number_2 ...))))
           (append (reverse (term (number_1 ...)))
                   (reverse (term (number_2 ...))))))
redex-check: counterexample found after 11 attempts:
((0) (0+inf.0i))

> (redex-check
   empty-lang
   ((number_1 ...)
    (number_2 ...))
   (equal? (reverse (append (term (number_1 ...))
                            (term (number_2 ...))))
           (append (reverse (term (number_2 ...)))
                   (reverse (term (number_1 ...)))))
   #:attempts 200)
redex-check: no counterexamples in 200 attempts

> (let ([R (reduction-relation
            empty-lang
            (--> (Σ) 0)
            (--> (Σ number) number)
            (--> (Σ number_1 number_2 number_3 ...)
                 (Σ ,(+ (term number_1) (term number_2))
                    number_3 ...)))])
    (redex-check
     empty-lang
     (Σ number ...)
     (printf "~s\n" (term (number ...)))
     #:attempts 3
     #:source R))
()
```

```
(0)
(2 0)
redex-check: no counterexamples in 1 attempt (with each clause)

> (redex-check
   empty-lang
   number
   (begin
     (printf "checking ~s\n" (term number))
     (positive? (term number)))
   #:prepare (λ (n)
               (printf "preparing ~s; " n)
               (add1 (abs n)))
   #:attempts 3)
preparing 0; checking 1
preparing 0+inf.0i; redex-check: fixing 0+inf.0i raises an excep-
tion
abs: contract violation
  expected: real?
  given: 0+inf.0i
> (define-language L
    (nat ::= Z (S nat)))

> (define-judgment-form L
    #:mode (sum I I O)
    [---------------
     (sum Z nat nat)]
    [(sum nat_1 nat_2 nat_3)
     ------------------------------
     (sum (S nat_1) nat_2 (S nat_3))])

> (redex-check L
               #:satisfying
               (sum nat_1 nat_2 nat_3)
               (equal? (judgment-holds
                        (sum nat_1 nat_2 nat_4) nat_4)
                       (term (nat_3)))
               #:attempts 100)
redex-check: no counterexamples in 100 attempts

> (redex-check L
               #:satisfying
               (sum nat_1 nat_2 nat_3)
               (equal? (term nat_1) (term nat_2)))
redex-check: counterexample found after 1 attempt:
(sum (S (S Z)) Z (S (S Z)))
```

```
(depth-dependent-order?) → boolean?
(depth-dependent-order? depth-dependent) → void?
  depth-dependent : boolean?
 = #t
```

Toggles whether or not redex will dynamically adjust the chance that more recursive clauses of judgment forms or metafunctions are chosen earlier when attempting to generate terms with forms that use `#:satisfying`. It is `#t` by default, which causes redex to favor more recursive clauses at lower depths and less recursive clauses at depths closer to the limit, in an attempt to generate larger terms. When it is `#f`, all clause orderings have equal probability above the bound.

```
(redex-generator language-id satisfying size-expr)

satisfying = (judgment-form-id pattern ...)
           | (metafunction-id pattern ...) = pattern

  size-expr : natural-number/c
```

WARNING: `redex-generator` is a new, experimental form, and its API may change.

Returns a thunk that, each time it is called, either generates a random s-expression based on *satisfying* or fails to (and returns `#f`). The terms returned by a particular thunk are guaranteed to be distinct.

Examples:

```
> (define-language L
    (nat ::= Z (S nat)))

> (define-judgment-form L
    #:mode (sum I I O)
    [---------------
     (sum Z nat nat)]
    [(sum nat_1 nat_2 nat_3)
     -------------------------------
     (sum (S nat_1) nat_2 (S nat_3))])

> (define gen-sum (redex-generator L (sum nat_1 nat_2 nat_3) 3))

> (for/list ([_ (in-range 5)])
    (gen-sum))
'((sum (S (S (S Z))) Z (S (S (S Z))))
  (sum (S (S (S (S Z)))) (S (S Z)) (S (S (S (S (S (S Z)))))))
```

```
(sum (S (S (S (S (S Z))))) Z (S (S (S (S (S Z))))))
(sum (S (S (S (S (S (S Z)))))) Z (S (S (S (S (S (S Z)))))))
(sum
 (S (S (S (S (S (S (S Z)))))))
 (S (S Z))
 (S (S (S (S (S (S (S (S (S Z))))))))))
```

```
(struct counterexample (term)
    #:extra-constructor-name make-counterexample
    #:transparent)
  term : any/c
```

Produced by `redex-check`, `check-reduction-relation`, and `check-metafunction` when testing falsifies a property.

```
(struct exn:fail:redex:test exn:fail:redex (source term)
    #:extra-constructor-name make-exn:fail:redex:test)
  source : exn:fail?
  term : any/c
```

Raised by `redex-check`, `check-reduction-relation`, and `check-metafunction` when testing a property raises an exception. The `exn:fail:redex:test-source` component contains the exception raised by the property, and the `exn:fail:redex:test-term` component contains the term that induced the exception.

```
(check-reduction-relation relation property kw-args ...)

kw-arg = #:attempts attempts-expr
       | #:retries retries-expr
       | #:print? print?-expr
       | #:attempt-size attempt-size-expr
       | #:prepare prepare-expr

  property : (-> any/c any/c)
  attempts-expr : natural-number/c
  retries-expr : natural-number/c
  print?-expr : any/c
  attempt-size-expr : (-> natural-number/c natural-number/c)
  prepare-expr : (-> any/c any/c)
```

Tests *relation* as follows: for each case of *relation*, `check-reduction-relation` generates `attempts` random terms that match that case's left-hand side and applies *property* to each random term.

This form provides a more convenient notation for

```
(redex-check L any (property (term any))
             #:attempts (* n attempts)
             #:source relation)
```

when *relation* is a relation on L with n rules.

```
(check-metafunction metafunction property kw-args ...)

kw-arg = #:attempts attempts-expr
       | #:retries retries-expr
       | #:print? print?-expr
       | #:attempt-size attempt-size-expr
       | #:prepare prepare-expr

  property : (-> (listof any/c) any/c)
  attempts-expr : natural-number/c
  retries-expr : natural-number/c
  print?-expr : any/c
  attempt-size-expr : (-> natural-number/c natural-number/c)
  prepare-expr : (-> (listof any/c) (listof any/c))
```

Like `check-reduction-relation` but for metafunctions. `check-metafunction` calls *property* with lists containing arguments to the metafunction. Similarly, *prepare-expr* produces and consumes argument lists.

Examples:

```
> (define-language empty-lang)

> (define-metafunction empty-lang
    Σ : number ... -> number
    [(Σ) 0]
    [(Σ number) number]
    [(Σ number_1 number_2) ,(+ (term number_1) (term number_2))]
    [(Σ number_1 number_2 ...) (Σ number_1 (Σ number_2 ...))])

> (check-metafunction Σ (λ (args)
                          (printf "trying ~s\n" args)
                          (equal? (apply + args)
                                  (term (Σ ,@args))))
                      #:attempts 2)
```

```
trying ()
trying ()
trying (0)
trying (0)
trying (2 1)
trying (0 1)
trying (0)
trying (1)
check-metafunction: no counterexamples in 2 attempts (with each
clause)
```

```
(default-attempt-size n) → natural-number/c
  n : natural-number/c
```

The default value of the `#:attempt-size` argument to `redex-check` and the other randomized testing forms, this procedure computes an upper bound on the size of the next test case from the number of previously attempted tests *n*. Currently, this procedure computes the base 5 logarithm, but that behavior may change in future versions.

```
(default-check-attempts) → natural-number/c
(default-check-attempts attempts) → void?
  attempts : natural-number/c
```

Determines the default value for `redex-check`'s optional `#:attempts` argument. By default, *attempts* is 1,000.

```
(redex-pseudo-random-generator) → pseudo-random-generator?
(redex-pseudo-random-generator generator) → void?
  generator : pseudo-random-generator?
```

`generate-term` and the randomized testing forms (e.g., `redex-check`) use the parameter *generator* to construct random terms. The parameter's initial value is `(current-pseudo-random-generator)`.

```
(exn:fail:redex:generation-failure? v) → boolean?
  v : any/c
```

Recognizes the exceptions raised by `generate-term`, `redex-check`, etc. when those forms are unable to produce a term matching some pattern.

Debugging PLT Redex Programs

It is easy to write grammars and reduction rules that are subtly wrong. Typically such mistakes result in examples that get stuck when viewed in a `traces` window.

The best way to debug such programs is to find an expression that looks like it should reduce, but doesn't, then try to find out which pattern is failing to match. To do so, use the `redex-match` form.

In particular, first check if the term in question matches the your system's main non-terminal (typically the expression or program non-terminal). If it does not match, simplify the term piece by piece to determine whether the problem is in the term or the grammar.

If the term does match your system's main non-terminal, determine by inspection which reduction rules should apply. For each such rule, repeat the above term-pattern debugging procedure, this time using the rule's left-hand side pattern instead of the system's main non-terminal. In addition to simplifying the term, also consider simplifying the pattern.

If the term matches the left-hand side, but the rule does not apply, then one of the rule's `side-condition` or `where` clauses is not satisfied. Using the bindings reported by `redex-match`, check each `side-condition` expression and each `where` pattern-match to discover which clause is preventing the rule's application.

2.7 GUI

`(require redex/gui)` package: `redex-gui-lib`

This section describes the GUI tools that Redex provides for exploring reduction sequences.

```
(traces reductions
        expr
       [#:multiple? multiple?
        #:reduce reduce
        #:pred pred
        #:pp pp
        #:colors colors
        #:racket-colors? racket-colors?
        #:scheme-colors? scheme-colors?
        #:filter term-filter
        #:x-spacing x-spacing
        #:y-spacing y-spacing
        #:layout layout
        #:edge-labels? edge-labels?
        #:edge-label-font edge-label-font
        #:graph-pasteboard-mixin graph-pasteboard-mixin])
 → void?
  reductions : reduction-relation?
  expr : (or/c any/c (listof any/c))
  multiple? : boolean? = #f
```

```
reduce : (-> reduction-relation? any/c
             (listof (list/c (union false/c string?) any/c)))
       = apply-reduction-relation/tag-with-names
pred : (or/c (-> sexp any)
             (-> sexp term-node? any))
       = (λ (x) #t)
pp : (or/c (any -> string)
           (any output-port number (is-a?/c text%) -> void))
   = default-pretty-printer
colors : (listof
          (cons/c string?
                  (and/c (listof (or/c string? (is-a?/c color%)))
                         (λ (x) (<= 0 (length x) 6)))))
       = '()
racket-colors? : boolean? = #t
scheme-colors? : boolean? = racket-colors?
term-filter : (-> any/c (or/c #f string?) any/c)
            = (λ (x y) #t)
x-spacing : real? = 15
y-spacing : real? = 15
layout : (-> (listof term-node?) void?) = void
edge-labels? : boolean? = #t
edge-label-font : (or/c #f (is-a?/c font%)) = #f
graph-pasteboard-mixin : (make-mixin-contract graph-pasteboard<%>)
                       = values
```

This function opens a new window and inserts each expression in *expr* (if *multiple?* is #t – if *multiple?* is #f, then *expr* is treated as a single expression). Then, it reduces the terms until at least reduction-steps-cutoff (see below) different terms are found, or no more reductions can occur. It inserts each new term into the gui. Clicking the reduce button reduces until reduction-steps-cutoff more terms are found.

The *reduce* function applies the reduction relation to the terms. By default, it is apply-reduction-relation/tag-with-names; it may be changed to only return a subset of the possible reductions, for example, but it must satisfy the same contract as apply-reduction-relation/tag-with-names.

The *pred* function indicates if a term has a particular property. If it returns #f, the term is displayed with a pink background. If it returns a string or a color% object, the term is displayed with a background of that color (using the-color-database to map the string to a color). If it returns any other value, the term is displayed normally. If the *pred* function accepts two arguments, a term-node corresponding to the term is passed to the predicate. This lets the predicate function explore the (names of the) reductions that led to this term, using term-node-children, term-node-parents, and term-node-labels.

The *pred* function may be called more than once per node. In particular, it is called each time an edge is added to a node. The latest value returned determines the color.

The *pp* function is used to specially print expressions. It must either accept one or four arguments. If it accepts one argument, it will be passed each term and is expected to return a string to display the term.

If the *pp* function takes four arguments, it should render its first argument into the port (its second argument) with width at most given by the number (its third argument). The final argument is the text where the port is connected – characters written to the port go to the end of the editor. Use `write-special` to send `snip%` objects or `2htdp/image` images (or other things that subscribe to `file/convertible` or `pict/convert`) directly to the editor.

The *colors* argument, if provided, specifies a list of reduction-name/color-list pairs. The traces gui will color arrows drawn because of the given reduction name with the given color instead of using the default color.

The `cdr` of each of the elements of *colors* is a list of colors, organized in pairs. The first two colors cover the colors of the line and the border around the arrow head, the first when the mouse is over a graph node that is connected to that arrow, and the second for when the mouse is not over that arrow. Similarly, the next colors are for the text drawn on the arrow and the last two are for the color that fills the arrow head. If fewer than six colors are specified, the specified colors are used and then defaults are filled in for the remaining colors.

The *racket-colors?* argument (along with *scheme-colors?*, retained for backward compatibility), controls the coloring of each window. When *racket-colors?* is `#t` (and *scheme-colors?* is `#t` too), `traces` colors the contents according to DrRacket's Racket-mode color scheme; otherwise, `traces` uses a black color scheme.

The *term-filter* function is called each time a new node is about to be inserted into the graph. If the filter returns false, the node is not inserted into the graph.

The *x-spacing* and *y-spacing* arguments control the amount of space put between the snips in the default layout.

The *layout* argument is called (with all of the terms) when new terms are inserted into the window. In general, it is called after new terms are inserted in response to the user clicking on the reduce button, and after the initial set of terms is inserted. See also `term-node-set-position!`.

If *edge-labels?* is `#t` (the default), then edge labels are drawn; otherwise not.

The *edge-label-font* argument is used as the font on the edge labels. If `#f` is supplied, the `dc<%>` object's default font is used.

The traces library uses an instance of the `mrlib/graph` library's `graph-pasteboard<%>` interface to layout the graphs. Sometimes, overriding one of its methods can help give finer-grained control over the layout, so the *graph-pasteboard-mixin* is applied to the class before it is instantiated. Also note that all of the snips inserted into the editor by this library

have a `get-term-node` method which returns the snip's `term-node`.

For a more serious example of `traces`, please see §1 "Amb: A Redex Tutorial", but for a silly one that demonstrates how the *pp* argument lets us use images, we can take the pairing functions discussed in Matthew Szudzik's *An Elegant Pairing Function* presentation:

```
(define/contract (unpair z)
  (-> exact-nonnegative-integer?
      (list/c exact-nonnegative-integer? exact-nonnegative-
integer?))
  (define i (integer-sqrt z))
  (define i2 (* i i))
  (cond
    [(< (- z i2) i)
     (list (- z i2) i)]
    [else
     (list i (- z i2 i))]))

(define/contract (pair x y)
  (-> exact-nonnegative-integer? exact-nonnegative-integer?
      exact-nonnegative-integer?)
  (if (= x (max x y))
      (+ (* x x) x y)
      (+ (* y y) x)))
```

and build a reduction relation out of them:

```
(define-language L (n ::= natural))
(define red
  (reduction-relation
   L
   (--> (n_1 n_2)
        ,(unpair (+ 1 (pair (term n_1)
                            (term n_2)))))))
(traces red (term (0 0)))
```

We can then turn those two numbers into two stars, where the number indicates the number of points in the star:

```
(require 2htdp/image)
(define/contract (two-stars point-count1 point-count2)
  (-> (>=/c 2) (>=/c 2) image?)
  (overlay
```

```
    (radial-star (+ 2 point-count1)
                 10 60
                 "solid"
                 (make-color 255 0 255 150))
    (radial-star (+ 2 point-count2)
                 10 60
                 "solid"
                 "cornflowerblue")))
```

and then use the *pp* function to show those in the traces window instead of just the numbers.

```
(traces red
        (term (0 0))
        #:pp
        (λ (term port w txt)
          (write-special
           (two-stars (+ 2 (list-ref term 0))
                      (+ 2 (list-ref term 1)))
           port)))
```

```
(traces/ps  reductions
            expr
            file
           [#:multiple? multiple?
            #:reduce reduce
            #:pred pred
            #:pp pp
            #:colors colors
            #:filter term-filter
            #:layout layout
            #:x-spacing x-spacing
            #:y-spacing y-spacing
            #:edge-labels? edge-labels?
            #:edge-label-font edge-label-font
            #:graph-pasteboard-mixin graph-pasteboard-mixin]
            #:post-process post-process)
 → void?
  reductions : reduction-relation?
  expr : (or/c any/c (listof any/c))
  file : (or/c path-string? path?)
  multiple? : boolean? = #f
            (-> reduction-relation? any/c
  reduce :      (listof (list/c (union false/c string?) any/c)))
         = apply-reduction-relation/tag-with-names
```

```
  pred : (or/c (-> sexp any)
               (-> sexp term-node? any))
       = (λ (x) #t)
  pp : (or/c (any -> string)
             (any output-port number (is-a?/c text%) -> void))
     = default-pretty-printer
  colors : (listof
            (cons/c string?
                    (and/c (listof (or/c string? (is-a?/c color%)))
                           (λ (x) (<= 0 (length x) 6)))))
         = '()
  term-filter : (-> any/c (or/c #f string?) any/c)
              = (λ (x y) #t)
  layout : (-> (listof term-node?) void?) = void
  x-spacing : number? = 15
  y-spacing : number? = 15
  edge-labels? : boolean? = #t
  edge-label-font : (or/c #f (is-a?/c font%)) = #f
  graph-pasteboard-mixin : (make-mixin-contract graph-pasteboard<%>)
                         = values
  post-process : (-> (is-a?/c graph-pasteboard<%>) any/c)
```

This function behaves just like the function `traces`, but instead of opening a window to show the reduction graph, it just saves the reduction graph to the specified *file*.

All of the arguments behave like the arguments to `traces`, with the exception of the *post-process* argument. It is called just before the PostScript is created with the graph pasteboard.

```
(stepper reductions t [pp]) → void?
  reductions : reduction-relation?
  t : any/c
  pp : (or/c (any -> string)
             (any output-port number (is-a?/c text%) -> void))
     = default-pretty-printer
```

This function opens a stepper window for exploring the behavior of the term *t* in the reduction system given by *reductions*.

The *pp* argument is the same as to the `traces` function but is here for backwards compatibility only and should not be changed for most uses, but instead adjusted with `pretty-print-parameters`. Specifically, the highlighting shown in the stepper window can be wrong if `default-pretty-printer` does not print sufficiently similarly to how `pretty-print` prints (when adjusted by `pretty-print-parameters`'s behavior, of course).

```
(stepper/seed reductions seed [pp]) → void?
  reductions : reduction-relation?
```

```
  seed : (cons/c any/c (listof any/c))
       (or/c (any -> string)
  pp :       (any output-port number (is-a?/c text%) -> void))
     = default-pretty-printer
```

Like stepper, this function opens a stepper window, but it seeds it with the reduction-sequence supplied in *seed*.

```
(show-derivations  derivations
                  [#:pp pp
                   #:racket-colors? racket-colors?
                   #:init-derivation init-derivation]) → any
  derivations : (cons/c derivation? (listof derivation?))
       (or/c (any -> string)
  pp :       (any output-port number (is-a?/c text%) -> void))
     = default-pretty-printer
  racket-colors? : boolean? = #f
  init-derivation : exact-nonnegative-integer? = 0
```

Opens a window to show *derivations*.

The *pp* and *racket-colors?* arguments are like those to traces.

The initial derivation shown in the window is chosen by *init-derivation*, used as an index into *derivations*.

```
(derivation/ps  derivation
                filename
                [#:pp pp
                 #:racket-colors? racket-colors?]
                 #:post-process post-process)     → void?
  derivation : derivation?
  filename : path-string?
       (or/c (any -> string)
  pp :       (any output-port number (is-a?/c text%) -> void))
     = default-pretty-printer
  racket-colors? : boolean? = #f
  post-process : (-> (is-a?/c pasteboard%) any)
```

Like show-derivations, except it prints a single derivation in PostScript to *filename*.

```
(term-node-children tn) → (listof term-node?)
  tn : term-node?
```

Returns a list of the children (i.e., terms that this term reduces to) of the given node.

Note that this function does not return all terms that this term reduces to – only those that are currently in the graph.

```
(term-node-parents tn) → (listof term-node?)
  tn : term-node?
```

Returns a list of the parents (i.e., terms that reduced to the current term) of the given node.

Note that this function does not return all terms that reduce to this one – only those that are currently in the graph.

```
(term-node-labels tn) → (listof (or/c false/c string?))
  tn : term-node?
```

Returns a list of the names of the reductions that led to the given node, in the same order as the result of term-node-parents. If the list contains `#f`, that means that the corresponding step does not have a label.

```
(term-node-set-color! tn color) → void?
  tn : term-node?
  color : (or/c string? (is-a?/c color%) false/c)
```

Changes the highlighting of the node; if its second argument is `#f`, the coloring is removed, otherwise the color is set to the specified `color%` object or the color named by the string. The `color-database<%>` is used to convert the string to a `color%` object.

```
(term-node-color tn) → (or/c string? (is-a?/c color%) false/c)
  tn : term-node?
```

Returns the current highlighting of the node. See also `term-node-set-color!`.

```
(term-node-set-red! tn red?) → void?
  tn : term-node?
  red? : boolean?
```

Changes the highlighting of the node; if its second argument is `#t`, the term is colored pink, if it is `#f`, the term is not colored specially.

```
(term-node-expr tn) → any
  tn : term-node?
```

Returns the expression in this node.

```
(term-node-set-position! tn x y) → void?
  tn : term-node?
  x : (and/c real? positive?)
  y : (and/c real? positive?)
```

Sets the position of *tn* in the graph to (*x*,*y*).

```
(term-node-x tn) → real?
  tn : term-node?
```

Returns the x coordinate of *tn* in the window.

```
(term-node-y tn) → real?
  tn : term-node?
```

Returns the y coordinate of *tn* in the window.

```
(term-node-width tn) → real?
  tn : term-node?
```

Returns the width of *tn* in the window.

```
(term-node-height tn) → real?
  tn : term-node?
```

Returns the height of *tn* in the window.

```
(term-node? v) → boolean?
  v : any/c
```

Recognizes term nodes.

```
(reduction-steps-cutoff) → number?
(reduction-steps-cutoff cutoff) → void?
  cutoff : number?
```

A parameter that controls how many steps the `traces` function takes before stopping.

```
(initial-font-size) → number?
(initial-font-size size) → void?
  size : number?
```

A parameter that controls the initial font size for the terms shown in the GUI window.

```
(initial-char-width) → (or/c number? (-> any/c number?))
(initial-char-width width) → void?
  width : (or/c number? (-> any/c number?))
```

A parameter that determines the initial width of the boxes where terms are displayed (measured in characters) for both the stepper and traces.

If its value is a number, then the number is used as the width for every term. If its value is a function, then the function is called with each term and the resulting number is used as the width.

```
(dark-pen-color) → (or/c string? (is-a?/c color<%>))
(dark-pen-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(dark-brush-color) → (or/c string? (is-a?/c color<%>))
(dark-brush-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(light-pen-color) → (or/c string? (is-a?/c color<%>))
(light-pen-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(light-brush-color) → (or/c string? (is-a?/c color<%>))
(light-brush-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(dark-text-color) → (or/c string? (is-a?/c color<%>))
(dark-text-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(light-text-color) → (or/c string? (is-a?/c color<%>))
(light-text-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
```

These six parameters control the color of the edges in the graph.

The dark colors are used when the mouse is over one of the nodes that is connected to this edge. The light colors are used when it isn't.

The pen colors control the color of the line. The brush colors control the color used to fill the arrowhead and the text colors control the color used to draw the label on the edge.

```
(pretty-print-parameters) → (-> (-> any/c) any/c)
(pretty-print-parameters f) → void?
  f : (-> (-> any/c) any/c)
```

A parameter that is used to set other `pretty-print` parameters.

Specifically, whenever default-pretty-printer prints something it calls *f* with a thunk that does the actual printing. Thus, *f* can adjust pretty-print's parameters to adjust how printing happens.

```
(default-pretty-printer v port width text) → void?
  v : any/c
  port : output-port?
  width : exact-nonnegative-integer?
  text : (is-a?/c text%)
```

This is the default value of pp used by traces and stepper and it uses pretty-print.

This function uses the value of pretty-print-parameters to adjust how it prints.

It sets the pretty-print-columns parameter to *width*, and it sets pretty-print-size-hook and pretty-print-print-hook to print holes and the symbol 'hole to match the way they are input in a term expression.

2.8 Typesetting

```
(require redex/pict)
```
package: redex-pict-lib

The redex/pict library provides functions designed to typeset grammars, reduction relations, and metafunctions.

Each grammar, reduction relation, and metafunction can be saved in a ".ps" file (as encapsulated PostScript), or can be turned into a pict for viewing in the REPL or using with Slideshow (see the pict library).

For producing papers with Scribble, just include the picts inline in the paper and pass the the --dvipdf flag generate the ".pdf" file. For producing papers with LaTeX, create ".ps" files from Redex and use latex and dvipdf to create ".pdf" files (using pdflatex with ".pdf" files will work but the results will not look as good onscreen).

2.8.1 Picts, PDF, & PostScript

This section documents two classes of operations, one for direct use of creating postscript figures for use in papers and for use in DrRacket to easily adjust the typesetting: render-term, render-language, render-reduction-relation, render-relation, render-judgment-form, render-metafunctions, and render-lw, and one for use in combination with other libraries that operate on picts term->pict, language->pict, reduction-relation->pict, relation->pict, judgment-form->pict, metafunction->pict, and lw->pict. The primary difference between these functions is that the former list sets dc-for-text-size and the latter does not.

```
(render-term lang term)
(render-term lang term file)
```

Renders the term *term*. If *file* is #f or not present, render-term produces a pict; if *file* is a path, it saves Encapsulated PostScript in the provided filename, unless the filename ends with ".pdf", in which case it saves PDF.

The *term* argument must be a literal; it is not an evaluated position. For example, this:

```
(define-language L)
(define x (term (+ 1 2)))
(render-term L x)
```

will render the term x, not the term (+ 1 2).

See render-language for more details on the construction of the pict.

```
(term->pict lang term)
```

Produces a pict like render-term, but without adjusting dc-for-text-size.

The first argument is expected to be a compiled-language? and the second argument is expected to be a term (without the *term* wrapper). The formatting in the *term* argument is used to determine how the resulting pict will look.

This function is primarily designed to be used with Slideshow or with other tools that combine picts together.

```
(render-term/pretty-write lang
                          term
                          filename
                          #:width width) → void?
  lang : compiled-lang?
  term : any/c
  filename : path-string?
  width : #f
```

Like render-term, except that the *term* argument is evaluated, and expected to return a term. Then, pretty-write is used to determine where the line breaks go, using the *width* argument as a maximum width (via pretty-print-columns).

```
(term->pict/pretty-write lang
                         term
                         filename
                         #:width width) → pict?
```

```
  lang : compiled-lang?
  term : any/c
  filename : (or/c path-string? #f)
  width : #f
```

Like term->pict, but with the same change that render-term/pretty-write has from render-term.

```
(render-language lang [file #:nts nts]) → (if file void? pict?)
  lang : compiled-lang?
  file : (or/c false/c path-string?) = #f
  nts : (or/c false/c (listof (or/c string? symbol?)))
      = (render-language-nts)
```

Renders a language. If *file* is #f, it produces a pict; if *file* is a path, it saves Encapsulated PostScript in the provided filename, unless the filename ends with ".pdf", in which case it saves PDF. See render-language-nts for information on the *nts* argument.

This function parameterizes dc-for-text-size to install a relevant dc: a bitmap-dc% or a post-script-dc%, depending on whether *file* is a path.

See language->pict if you are using Slideshow or are otherwise setting dc-for-text-size.

```
(language->pict lang [#:nts nts]) → pict?
  lang : compiled-lang?
  nts : (or/c false/c (listof (or/c string? symbol?)))
      = (render-language-nts)
```

Produce a pict like render-language, but without adjusting dc-for-text-size.

This function is primarily designed to be used with Slideshow or with other tools that combine picts together.

```
(render-reduction-relation rel
                           [file
                            #:style style])
 → (if file void? pict?)
  rel : reduction-relation?
  file : (or/c false/c path-string?) = #f
  style : reduction-rule-style/c = (rule-pict-style)
```

Renders a reduction relation. If *file* is #f, it produces a pict; if *file* is a path, it saves Encapsulated PostScript in the provided filename, unless the filename ends with ".pdf", in which case it saves PDF. See rule-pict-style for information on the *style* argument.

This function parameterizes dc-for-text-size to install a relevant dc: a bitmap-dc% or a post-script-dc%, depending on whether *file* is a path. See also reduction-relation->pict.

The following forms of arrows can be typeset:

```
--> -+> ==> -> => ..> >-> ~~> ~> :-> :-> c-> ->>
  >- -< >>- -<<
```

```
(reduction-relation->pict r [#:style style]) → pict?
  r : reduction-relation?
  style : reduction-rule-style/c = (rule-pict-style)
```

Produces a pict like render-reduction-relation, but without setting dc-for-text-size.

This function is primarily designed to be used with Slideshow or with other tools that combine picts together.

```
(render-metafunction metafunction-name maybe-contract)
(render-metafunction metafunction-name filename maybe-contract)
(render-metafunctions metafunction-name ...
                      maybe-filename maybe-contract)

 maybe-filename =
                | #:file filename

maybe-contract? =
                | #:contract? bool-expr
```

Like render-reduction-relation but for metafunctions.

Similarly, render-metafunctions accepts multiple metafunctions and renders them together, lining up all of the clauses together.

If the metafunctions have contracts, they are typeset as the first lines of the output unless the expression following #:contract? evaluates to #f (which is the default).

This function sets dc-for-text-size. See also metafunction->pict and metafunctions->pict.

Changed in version 1.3 of package redex-pict-lib: Added #:contract? keyword argument.

```
(metafunction->pict metafunction-name)
```

This produces a pict, but without setting dc-for-text-size. It is suitable for use in Slideshow or other libraries that combine picts.

```
(metafunctions->pict metafunction-name ...)
```

Like `metafunction->pict`, this produces a `pict`, but without setting `dc-for-text-size` and is suitable for use in Slideshow or other libraries that combine `pict`s. Like `render-metafunctions`, it accepts multiple metafunctions and renders them together.

```
(render-relation relation-name)
(render-relation relation-name filename)
```

Like `render-metafunction` but for relations.

This function sets `dc-for-text-size`. See also `relation->pict`.

```
(render-judgment-form judgment-form-name)
(render-judgment-form judgment-form-name filename)
```

Like `render-metafunction` but for judgment forms.

This function sets `dc-for-text-size`. See also `judgment-form->pict`.

```
(relation->pict relation-name)
```

This produces a pict, but without setting `dc-for-text-size`. It is suitable for use in Slideshow or other libraries that combine `pict`s.

```
(judgment-form->pict judgment-form-name)
```

This produces a pict, but without setting `dc-for-text-size`. It is suitable for use in Slideshow or other libraries that combine `pict`s.

2.8.2 Customization

```
(render-language-nts) → (or/c false/c (listof symbol?))
(render-language-nts nts) → void?
  nts : (or/c false/c (listof symbol?))
```

The value of this parameter controls which non-terminals `render-language` and `language->pict` render by default. If it is `#f` (the default), all non-terminals are rendered. If it is a list of symbols, only the listed symbols are rendered.

See also `language-nts`.

```
(non-terminal-gap-space) → real?
(non-terminal-gap-space gap-space) → void?
  gap-space : real?
```

Controls the amount of vertical space between non-terminals in a typeset language.

Defaults to `0`.

Added in version 1.1 of package `redex-pict-lib`.

```
(extend-language-show-union) → boolean?
(extend-language-show-union show?) → void?
  show? : boolean?
```

A parameter that controls the rendering of extended languages. If the parameter value is `#t`, then a language constructed with extend-language is shown as if the language had been constructed directly with `language`. If it is `#f`, then only the last extension to the language is shown (with four-period ellipses, just like in the concrete syntax).

Defaults to `#f`.

Note that the `#t` variant can look a little bit strange if `....` are used and the original version of the language has multi-line right-hand sides.

```
(extend-language-show-extended-order) → boolean?
(extend-language-show-extended-order ext-order?) → void?
  ext-order? : boolean?
```

A parameter that controls the rendering of extended languages when `extend-language-show-union` has a true value. If this parameter's value is `#t`, then productions are shown as ordered in the language extension instead of the order of the original, unextended language.

Defaults to `#f`.

Added in version 1.2 of package `redex-pict-lib`.

```
(render-reduction-relation-rules)
    (or/c false/c
          (listof (or/c symbol?
 →                      string?
                        exact-nonnegative-integer?)))
(render-reduction-relation-rules rules) → void?
          (or/c false/c
                (listof (or/c symbol?
  rules :                     string?
                              exact-nonnegative-integer?)))
```

This parameter controls which rules in a reduction relation will be rendered. The strings and symbols match the names of the rules and the integers match the position of the rule in the original definition.

```
(rule-pict-style) → reduction-rule-style/c
(rule-pict-style style) → void?
  style : reduction-rule-style/c
```

This parameter controls the style used by default for the reduction relation. It can be `'horizontal`, where the left and right-hand sides of the reduction rule are beside each other or `'vertical`, where the left and right-hand sides of the reduction rule are above each other. The `'compact-vertical` style moves the reduction arrow to the second line and uses less space between lines. The `'vertical-overlapping-side-conditions` variant, the side-conditions don't contribute to the width of the pict, but are just overlaid on the second line of each rule. The `'horizontal-left-align` style is like the `'horizontal` style, but the left-hand sides of the rules are aligned on the left, instead of on the right. The `''horizontal-side-conditions-same-line` is like `'horizontal`, except that side-conditions are on the same lines as the rule, instead of on their own line below.

```
reduction-rule-style/c : flat-contract?
```

A contract equivalent to

```
(or/c 'vertical
      'compact-vertical
      'vertical-overlapping-side-conditions
      'horizontal
      'horizontal-left-align
      'horizontal-side-conditions-same-line)
```

```
(arrow-space) → natural-number/c
(arrow-space space) → void?
  space : natural-number/c
```

This parameter controls the amount of extra horizontal space around the reduction relation arrow. Defaults to 0.

```
(label-space) → natural-number/c
(label-space space) → void?
  space : natural-number/c
```

This parameter controls the amount of extra space before the label on each rule, except in the `'vertical` and `'vertical-overlapping-side-conditions` modes, where it has no effect. Defaults to 0.

```
(metafunction-pict-style)
    (or/c 'left-right
          'up-down
          'left-right/vertical-side-conditions
 →        'up-down/vertical-side-conditions
          'left-right/compact-side-conditions
          'up-down/compact-side-conditions
          'left-right/beside-side-conditions)
(metafunction-pict-style style) → void?
          (or/c 'left-right
                'up-down
                'left-right/vertical-side-conditions
  style :       'up-down/vertical-side-conditions
                'left-right/compact-side-conditions
                'up-down/compact-side-conditions
                'left-right/beside-side-conditions)
```

This parameter controls the style used for typesetting metafunctions. The `'left-right` style means that the results of calling the metafunction are displayed to the right of the arguments and the `'up-down` style means that the results are displayed below the arguments.

The `'left-right/vertical-side-conditions` and `'up-down/vertical-side-conditions` variants format side conditions each on a separate line, instead of all on the same line.

The `'left-right/compact-side-conditions` and `'up-down/compact-side-conditions` variants move side conditions to separate lines to avoid making the rendered form wider than it would be otherwise.

The `'left-right/beside-side-conditions` variant is like `'left-right`, except it puts the side-conditions on the same line, instead of on a new line below the case.

```
(delimit-ellipsis-arguments?) → any/c
(delimit-ellipsis-arguments? delimit?) → void?
  delimit? : any/c
```

This parameter controls the typesetting of metafunction definitions and applications. When it is non-`#f` (the default), commas precede ellipses that represent argument sequences; when it is `#f` no commas appear in those positions.

```
(linebreaks) → (or/c #f (listof boolean?))
(linebreaks breaks) → void?
  breaks : (or/c #f (listof boolean?))
```

This parameter controls which cases in the metafunction are rendered on two lines and which are rendered on one.

If its value is a list, the length of the list must match the number of cases plus one if there is a contract. Each boolean indicates if that case has a linebreak or not.

This parameter's value influences the `'left/right` styles only.

```
(metafunction-cases)
    (or/c #f (and/c (listof (or/c exact-nonnegative-integer?
 →                              string?))
                    pair?))
(metafunction-cases cases) → void?
          (or/c #f (and/c (listof (or/c exact-nonnegative-integer?
  cases :                                string?))
                          pair?))
```

Controls which cases in a metafunction are rendered. If it is `#f` (the default), then all of the cases appear. If it is a list, then only the selected cases appear. The numbers indicate the cases counting from `0` and the strings indicate cases named with `clause-name`.

This parameter also controls how which clauses in judgment forms are rendered, but only in the case that `judgment-form-cases` is `#f` (and in that case, only the numbers are used).

```
(judgment-form-cases)
    (or/c #f
          (and/c (listof (or/c exact-nonnegative-integer?
 →                             string?))
                 pair?))
(judgment-form-cases cases) → void?
          (or/c #f
                (and/c (listof (or/c exact-nonnegative-integer?
  cases :                              string?))
                       pair?))
```

Controls which clauses in a judgment form are rendered. If it is `#f` (the default), then all of them are rendered. If it is a list, then only the selected clauses appear (numbers count from `0`, and strings correspond to the labels in a judgment form).

```
(label-style) → text-style/c
(label-style style) → void?
  style : text-style/c
(grammar-style) → text-style/c
(grammar-style style) → void?
  style : text-style/c
(paren-style) → text-style/c
(paren-style style) → void?
  style : text-style/c
(literal-style) → text-style/c
(literal-style style) → void?
  style : text-style/c
```

```
(metafunction-style) → text-style/c
(metafunction-style style) → void?
  style : text-style/c
(non-terminal-style) → text-style/c
(non-terminal-style style) → void?
  style : text-style/c
(non-terminal-subscript-style) → text-style/c
(non-terminal-subscript-style style) → void?
  style : text-style/c
(non-terminal-superscript-style) → text-style/c
(non-terminal-superscript-style style) → void?
  style : text-style/c
(default-style) → text-style/c
(default-style style) → void?
  style : text-style/c
```

These parameters determine the font used for various text in the picts. See `text` in the texpict collection for documentation explaining `text-style/c`. One of the more useful things it can be is one of the symbols `'roman`, `'swiss`, or `'modern`, which are a serif, sans-serif, and monospaced font, respectively. (It can also encode style information, too.)

The `label-style` is used for the reduction rule label names. The `literal-style` is used for names that aren't non-terminals that appear in patterns. The `metafunction-style` is used for the names of metafunctions. The `paren-style` is used for the parentheses (including "[", "]", "{", and "}", as well as "(" and ")"), but not for the square brackets used for in-hole decompositions, which use the `default-style`. The `grammar-style` is used for the "::=" and "|" in grammars.

The `non-terminal-style` parameter is used for the names of non-terminals. Two parameters style the text in the (optional) "underscore" component of a non-terminal reference. The first, `non-terminal-subscript-style`, applies to the segment between the underscore and the first caret (`^`) to follow it; the second, `non-terminal-superscript-style`, applies to the segment following that caret. For example, in the non-terminal reference `x_y^z`, `x` has style `non-terminal-style`, `y` has style `non-terminal-subscript-style`, and `z` has style `non-terminal-superscript-style`.

The `default-style` is used for parenthesis, the dot in dotted lists, spaces, the "where" and "fresh" in side-conditions, and other places where the other parameters aren't used.

```
(label-font-size) → (and/c (between/c 1 255) integer?)
(label-font-size size) → void?
  size : (and/c (between/c 1 255) integer?)
(metafunction-font-size) → (and/c (between/c 1 255)
                                  integer?)
(metafunction-font-size size) → void?
  size : (and/c (between/c 1 255)
                integer?)
```

```
(default-font-size) → (and/c (between/c 1 255) integer?)
(default-font-size size) → void?
  size : (and/c (between/c 1 255) integer?)
```

These parameters control the various font sizes. The default-font-size is used for all of the font sizes except labels and metafunctions.

```
(reduction-relation-rule-separation)
 → (parameter/c (and/c integer? positive? exact?))
(reduction-relation-rule-separation sep) → void?
  sep : (parameter/c (and/c integer? positive? exact?))
```

Controls the amount of space between clauses in a reduction relation. Defaults to 4.

```
(curly-quotes-for-strings) → boolean?
(curly-quotes-for-strings on?) → void?
  on? : boolean?
```

Controls if the open and close quotes for strings are turned into “ and ” or are left as merely ".

Defaults to `#t`.

```
(current-text) → (-> string? text-style/c number? pict?)
(current-text proc) → void?
  proc : (-> string? text-style/c number? pict?)
```

This parameter's function is called whenever Redex typesets some part of a grammar, reduction relation, or metafunction. It defaults to the `pict` library's `text` function.

```
(arrow->pict arrow) → pict?
  arrow : symbol?
```

Returns the pict corresponding to *arrow*.

```
(set-arrow-pict! arrow proc) → void?
  arrow : symbol?
  proc : (-> pict?)
```

This functions sets the pict for a given reduction-relation symbol. When typesetting a reduction relation that uses the symbol, the thunk will be invoked to get a pict to render it. The thunk may be invoked multiple times when rendering a single reduction relation.

```
(white-bracket-sizing)
 → (-> string? number? (values number? number? number? number?))
(white-bracket-sizing proc) → void?
  proc : (-> string? number? (values number? number? number? number?))
```

This parameter is used when typesetting metafunctions to determine how to create the ⟦⟧ characters. Rather than using those characters directly (since glyphs tend not to be available in PostScript fonts), they are created by combining two '[' characters or two ']' characters together.

The procedure accepts a string that is either "[" or "]", and returns four numbers. The first two numbers determine the offset (from the left and from the right respectively) for the second square bracket, and the second two two numbers determine the extra space added (to the left and to the right respectively).

The default value of the parameter is:

```
(λ (str size)
  (let ([inset-amt (floor/even (max 4 (* size 1/2)))])
    (cond
      [(equal? str "[")
       (values inset-amt
               0
               0
               (/ inset-amt 2))]
      [else
       (values 0
               inset-amt
               (/ inset-amt 2)
               0)])))
```

where `floor/even` returns the nearest even number below its argument. This means that for sizes 9, 10, and 11, `inset-amt` will be 4, and for 12, 13, 14, and 15, `inset-amt` will be 6.

```
(horizontal-bar-spacing)
 → (parameter/c exact-nonnegative-integer?)
(horizontal-bar-spacing space) → void?
  space : (parameter/c exact-nonnegative-integer?)
```

Controls the amount of space around the horizontal bar when rendering a relation (that was created by `define-relation`). Defaults to 4.

```
(relation-clauses-combine)
 → (parameter/c (-> (listof pict?) pict?))
(relation-clauses-combine combine) → void?
  combine : (parameter/c (-> (listof pict?) pict?))
```

The *combine* function is called with the list of picts that are obtained by rendering a relation; it should put them together into a single pict. It defaults to `(λ (l) (apply vc-append 20 l))`

```
(where-make-prefix-pict) → (parameter/c (-> pict?))
(where-make-prefix-pict make-prefix) → void?
  make-prefix : (parameter/c (-> pict?))
```

The *make-prefix* function is called with no arguments to generate a pict that prefixes `where` clauses. It defaults to a function that produces a pict for "where" surrounded by spaces using the default style.

```
(where-combine) → (parameter/c (-> pict? pict? pict?))
(where-combine combine) → void?
  combine : (parameter/c (-> pict? pict? pict?))
```

The *combine* function is called with picts for the left and right side of a where clause, and it should put them together into a single pict. It defaults to `(λ (l r) (hbl-append l =-pict r))`, where *=-pict* is an equal sign surrounded by spaces using the default style.

2.8.3 Removing the Pink Background

When reduction rules, a metafunction, or a grammar contains unquoted Racket code or side-conditions, they are rendered with a pink background as a guide to help find them and provide an alternative typesetting for them. In general, a good goal for a PLT Redex program that you intend to typeset is to only include such things when they correspond to standard mathematical operations, and the Racket code is an implementation of those operations.

To replace the pink code, use:

```
(with-unquote-rewriter proc expression)
```

Installs *proc* as the current unquote rewriter and evaluates *expression*. If that expression computes any picts, the unquote rewriter specified is used to remap them.

The *proc* must match the contract `(-> lw? lw?)`. Its result should be the rewritten version version of the input.

```
(with-atomic-rewriter name-symbol
                      string-or-thunk-returning-pict
                      expression)
```

This extends the current set of atomic-rewriters with one new one that rewrites the value of name-symbol to `string-or-pict-returning-thunk` (applied, in the case of a thunk), during the evaluation of expression.

name-symbol is expected to evaluate to a symbol. The value of *string-or-thunk-returning-pict* is used whenever the symbol appears in a pattern.

```
(with-compound-rewriter name-symbol
                        proc
                        expression)
```

This extends the current set of compound-rewriters with one new one that rewrites the value of name-symbol via proc, during the evaluation of expression.

name-symbol is expected to evaluate to a symbol. The value of proc is called with a (listof lw), and is expected to return a new (listof (or/c lw? string? pict?)), rewritten appropriately.

The list passed to the rewriter corresponds to the lw for the sequence that has name-symbol's value at its head.

The result list is constrained to have at most 2 adjacent non-lws. That list is then transformed by adding lw structs for each of the non-lws in the list (see the text just below the description of lw for a explanation of logical space):

- If there are two adjacent lws, then the logical space between them is filled with whitespace.
- If there is a pair of lws with just a single non-lw between them, a lw will be created (containing the non-lw) that uses all of the available logical space between the lws.
- If there are two adjacent non-lws between two lws, the first non-lw is rendered right after the first lw with a logical space of zero, and the second is rendered right before the last lw also with a logical space of zero, and the logical space between the two lws is absorbed by a new lw that renders using no actual space in the typeset version.

One useful way to take advantage of with-compound-rewriters is to return a list that begins and ends with "" (the empty string). In that situation, any extra logical space that would have been just outside the sequence is replaced with an lw that does not draw anything at all.

```
(with-compound-rewriters ([name-symbol proc] ...)
                         expression)
```

Shorthand for nested with-compound-rewriter expressions.

2.8.4 LWs

```
(struct lw (e
            line
            line-span
            column
            column-span
            unq?
            metafunction?)
    #:extra-constructor-name make-lw
    #:mutable)
      (or/c string?
            symbol?
  e :       pict?
            (listof (or/c (symbols 'spring) lw?)))
  line : exact-positive-integer?
  line-span : exact-positive-integer?
  column : exact-positive-integer?
  column-span : exact-positive-integer?
  unq? : boolean?
  metafunction? : boolean?
```

The lw data structure corresponds represents a pattern or a Racket expression that is to be typeset. The functions listed above construct lw structs, select fields out of them, and recognize them. The lw binding can be used with copy-struct.

The values of the unq? and metafunction? fields, respectively, indicate whether the lw represents an unquoted expression or a metafunction application. See to-lw for the meanings of the other fields.

```
(build-lw e line line-span column column-span) → lw?
      (or/c string?
            symbol?
  e :       pict?
            (listof (or/c (symbols 'spring) lw?)))
  line : exact-positive-integer?
  line-span : exact-positive-integer?
  column : exact-positive-integer?
  column-span : exact-positive-integer?
```

Like make-lw but specialized for constructing lws that do not represent unquoted expressions or metafunction applications.

```
(to-lw arg)
```

This form turns its argument into lw structs that contain all of the spacing information just as it would appear when being used to typeset.

Each sub-expression corresponds to its own lw, and the element indicates what kind of subexpression it is. If the element is a list, then the lw corresponds to a parenthesized sequence, and the list contains a lw for the open paren, one lw for each component of the sequence and then a lw for the close parenthesis. In the case of a dotted list, there will also be a lw in the third-to-last position for the dot.

For example, this expression:

```
(a)
```

becomes this lw (assuming the above expression appears as the first thing in the file):

```
(build-lw (list (build-lw "(" 0 0 0 1)
                        (build-lw 'a 0 0 1 1)
                        (build-lw ")" 0 0 2 1))
                  0 0 0 3)
```

If there is some whitespace in the sequence, like this one:

```
(a b)
```

then there is no lw that corresponds to that whitespace; instead there is a logical gap between the lws.

```
(build-lw (list (build-lw "(" 0 0 0 1)
                (build-lw 'a 0 0 1 1)
                (build-lw 'b 0 0 3 1)
                (build-lw ")" 0 0 4 1))
          0 0 0 5)
```

In general, identifiers are represented with symbols and parenthesis are represented with strings and `pict`s can be inserted to render arbitrary pictures.

The line, line-span, column, and column-span correspond to the logical spacing for the redex program, not the actual spacing that will be used when they are rendered. The logical spacing is only used when determining where to place typeset portions of the program. In the absence of any rewriters, these numbers correspond to the line and column numbers in the original program.

The line and column are absolute numbers from the beginning of the file containing the expression. The column number is not necessarily the column of the open parenthesis in a sequence – it is the leftmost column that is occupied by anything in the sequence. The line-span is the number of lines, and the column span is the number of columns on the last line (not the total width).

When there are multiple lines, lines are aligned based on the logical space (i.e., the line/column & line-span/column-span) fields of the lws. As an example, if this is the original pattern:

```
(all good boys
     deserve fudge)
```

then the leftmost edges of the words "good" and "deserve" will be lined up underneath each other, but the relative positions of "boys" and "fudge" will be determined by the natural size of the words as they rendered in the appropriate font.

When `'spring` appears in the list in the `e` field of a `lw` struct, then it absorbs all of the space around it. It is also used by `to-lw` when constructing the picts for unquoted strings. For example, this expression

```
,x
```

corresponds to these structs:

```
(build-lw (list (build-lw "" 1 0 9 0)
                'spring
                (build-lw x 1 0 10 1))
          1 0 9 2)
```

and the `'spring` causes there to be no space between the empty string and the `x` in the typeset output.

```
(to-lw/stx stx) → lw?
  stx : syntax?
```

This is the runtime variant on `to-lw`; it accepts a syntax object and returns the corresponding `lw` structs. It only uses the location information in the syntax object, so metafunctions will not be rendered properly.

```
(render-lw language/nts lw) → pict?
  language/nts : (or/c (listof symbol?) compiled-lang?)
  lw : lw?
```

Produces a pict that corresponds to the *lw* object argument, using *language/nts* to determine which of the identifiers in the *lw* argument are non-terminals.

This function sets `dc-for-text-size`. See also `lw->pict`.

```
(lw->pict language/ntw lw) → pict?
  language/ntw : (or/c (listof symbol?) compiled-lang?)
  lw : lw?
```

Produces a pict that corresponds to the `lw` object argument, using `language/nts` to determine which of the identifiers in the `lw` argument are non-terminals.

This does not set the `dc-for-text-size` parameter. See also `render-lw`.

```
(just-before stuff lw) → lw?
  stuff : (or/c pict? string? symbol?)
  lw : lw?
(just-after stuff lw) → lw?
  stuff : (or/c pict? string? symbol?)
  lw : lw?
```

These two helper functions build new lws whose contents are the first argument, and whose line and column are based on the second argument, making the new loc wrapper be either just before or just after that argument. The line-span and column-span of the new lw is always zero.

2.9 Macros and Typesetting

When you have a macro that abstracts over variations in Redex programs, then typesetting is unlikely to work without some help from your macros.

To see the issue, consider this macro abstraction over a Redex grammar:

```
> (define-syntax-rule
    (def-my-lang L prim ...)
    (define-language L
      (e ::=
         (λ (x) e)
         (e e)
         prim ...
         x)
      (x ::= variable-not-otherwise-mentioned)))

> (def-my-lang L + - *)

> (render-language L)
eject: lines going backwards (current-line 3 line 2 atom
#<pict> tokens (#(struct:string-token 0 1 "*" swiss)
#(struct:pict-token 1 0 #<pict>) #(struct:string-token 0 1
"-" swiss) #(struct:pict-token 1 0 #<pict>)
#(struct:string-token 0 1 "+" swiss) #(struct:pict-token 0 0
#<pict>) #(struct:spacer-token 0 0)))
```

Redex thinks that the grammar is going "backwards" because of the way macro expansion synthesizes source locations. In particular, in the result of the macro expansion, the third production for `e` appears to come later in the file than the fourth production and this confuses Redex, making it unable to typeset this language.

One simple, not-very-general work-around is to just avoid typesetting the parts that come from the macro arguments. For example if you move the primitives into their own non-terminal and then just avoid typesetting that, Redex can cope:

```
(define-syntax-rule
  (def-my-lang L prim ...)
  (define-language L
    (e ::=
       (λ (x) e)
       (e e)
       prims
       x)
    (prims ::= prim ...)
    (x ::= variable-not-otherwise-mentioned)))

> (def-my-lang L + - *)

> (render-language L #:nts '(e x))
```

$$\begin{array}{l} e ::= (\lambda\ (x)\ e) \\ \quad\quad \mid (e\ e) \\ \quad\quad \mid \mathit{prims} \\ \quad\quad \mid x \\ x ::= \mathit{variable\text{-}not\text{-}otherwise\text{-}mentioned} \end{array}$$

You can also, however, exploit Racket's macro system to rewrite the source locations in a way that tells Redex where the macro-introduced parts of the language are supposed to be, and then typesetting will work normally. For example, here is one way to do this with the original language:

```
(define-syntax (def-my-lang stx)
   (syntax-case stx ()
     [(_ L a ...)
      (let ()
        (define template
          #'(define-language L
              (e (λ (x) e)
                 (e e)
                 HERE
                 x)
              (x variable-not-otherwise-mentioned)))
```

```
      (car
       (let loop ([stx template])
         (syntax-case stx (HERE)
           [HERE
            (let loop ([as (syntax->list #'(a ...))]
                       [pos (syntax-position stx)]
                       [col (syntax-column stx)])
              (cond
                [(null? as) '()]
                [else
                 (define a (car as))
                 (define span
                   (string-length
                    (symbol->string (syntax-e a))))
                 (define srcloc
                   (vector (syntax-source stx)
                           (syntax-line stx)
                           col
                           pos
                           span))
                 (cons
                  (datum->syntax a
                                 (syntax-e a)
                                 srcloc
                                 a)
                  (loop (cdr as)
                        (+ pos span 1)
                        (+ col span 1)))]))]
           [(a ...)
            (list
             (datum->syntax
              stx
              (apply append (map loop (syntax->list #'(a ...))))
              stx
              stx))]
           [a
            (list stx)])))))
```

```
> (def-my-lang L + - *)
```

```
> (render-language L)
```

```
e ::= (λ (x) e)
    | (e e)
    | + | - | *
    | x
x ::= variable-not-otherwise-mentioned
```

And one final caveat: when Racket compiles source files to bytecode format, it discards source location information in syntax constants, which means that if a file containing a macro like the one above is compiled to bytecode, then it cannot properly adjust the source locations and the typeset language will not look right (the important constant whose source locations are lost is `template`, above).

The simplest way to avoid this problem is to just avoid creating bytecode for these files.

It is possible to write the constant in the source code, however, and then process it so the compiled version of the file contains different data structures that record the source locations of the expressions. Redex does this internally so that compiled files that use, e.g., `define-language`, still correctly typeset. But when you write macros that expand into `define-language`, you must also take this step yourself (or avoid `.zo` files).

3 Automated Testing Benchmark

```
(require redex/benchmark)
```
package: redex-benchmark

Redex's automated testing benchmark provides a collection of buggy models and falsifiable properties to test how efficiently methods of automatic test case generation are able to find counterexamples for the bugs.

Each entry in the benchmark contains a *check* function and multiple *generate* functions. The check function determines if a given example is a counterexample (i.e. if it uncovers the buggy behavior) and each of the generate functions generates candidate examples to be tried. There are multiple ways to generate terms for each model. They typically correspond to different uses of `generate-term`, but could be any way to generate examples. See `run-gen-and-check` for the precise contracts for generate and check functions.

Most of the entries in the benchmark are small differences to existing, bug-free models, where some small change to the model introduces the bug. These changes are described using `define-rewrite`.

To run a benchmark entry with a particular generator, see `run-gen-and-check/mods`.

3.1 The Benchmark Models

The programs in our benchmark come from two sources: synthetic examples based on our experience with Redex over the years and from models that we and others have developed and bugs that were encountered during the development process.

The benchmark has six different Redex models, each of which provides a grammar of terms for the model and a soundness property that is universally quantified over those terms. Most of the models are of programming languages and most of the soundness properties are type-soundness, but we also include red-black trees with the property that insertion preserves the red-black invariant, as well as one richer property for one of the programming language models (discussed in §3.1.3 "stlc-sub").

For each model, we have manually introduced bugs into a number of copies of the model, such that each copy is identical to the correct one, except for a single bug. The bugs always manifest as a term that falsifies the soundness property.

The table in figure 1 gives an overview of the benchmark suite, showing some numbers for each model and bug. Each model has its name and the number of lines of code for the bug-free model (the buggy versions are always within a few lines of the originals). The line number counts include the model and the specification of the property.

Each bug has a number and, with the exception of the rvm model, the numbers count from 1 up to the number of bugs. The rvm model bugs are all from Klein et al. (2013)'s work and

we follow their numbering scheme (see §3.1.8 "rvm" for more information about how we chose the bugs from that paper).

The **S/M/D/U** column shows a classification of each bug as:

- **S** (Shallow) Errors in the encoding of the system into Redex, due to typos or a misunderstanding of subtleties of Redex.
- **M** (Medium) Errors in the algorithm behind the system, such as using too simple of a data-structure that doesn't allow some important distinction, or misunderstanding that some rule should have a side-condition that limits its applicability.
- **D** (Deep) Errors in the developer's understanding of the system, such as when a type system really isn't sound and the author doesn't realize it.
- **U** (Unnatural) Errors that are unlikely to have come up in real Redex programs but are included for our own curiosity. There are only two bugs in this category.

The size column shows the size of the term representing the smallest counterexample we know for each bug, where we measure size as the number of pairs of parentheses and atoms in the s-expression representation of the term.

Each subsection of this section introduces one of the models in the benchmark, along with the errors we introduced into each model.

3.1.1 stlc

A simply-typed λ-calculus with base types of numbers and lists of numbers, including the constants `+`, which operates on numbers, and `cons`, `head`, `tail`, and `nil` (the empty list), all of which operate only on lists of numbers. The property checked is type soundness: the combination of preservation (if a term has a type and takes a step, then the resulting term has the same type) and progress (that well-typed non-values always take a reduction step).

We introduced nine different bugs into this system. The first confuses the range and domain types of the function in the application rule, and has the small counterexample: `(hd 0)`. We consider this to be a shallow bug, since it is essentially a typo and it is hard to imagine anyone with any knowledge of type systems making this conceptual mistake. Bug 2 neglects to specify that a fully applied `cons` is a value, thus the list `((cons 0) nil)` violates the progress property. We consider this be be a medium bug, as it is not a typo, but an oversight in the design of a system that is otherwise correct in its approach.

We consider the next three bugs to be shallow. Bug 3 reverses the range and the domain of function types in the type judgment for applications. This was one of the easiest bug for all of our approaches to find. Bug 4 assigns `cons` a result type of `int`. The fifth bug returns the head of a list when `tl` is applied. Bug 6 only applies the `hd` constant to a partially

Model	LoC	Bug#	S/M/D/U	Size	Description of Bug
stlc	211	1	S	3	app rule the range of the function is matched to the argument
		2	M	5	the ((cons v) v) value has been omitted
		3	S	8	the order of the types in the function position of application has been swapped
		4	S	9	the type of cons is incorrect
		5	S	7	the tail reduction returns the wrong value
		6	M	7	hd reduction acts on partially applied cons
		7	M	9	evaluation isn't allowed on the rhs of applications
		8	U	12	lookup always returns int
		9	S	15	variables aren't required to match in lookup
poly-stlc	277	1	S	6	app rule the range of the function is matched to the argument
		2	M	11	the (([cons @ τ] v) v) value has been omitted
		3	S	14	the order of the types in the function position of application has been swapped
		4	S	15	the type of cons is incorrect
		5	S	16	the tail reduction returns the wrong value
		6	M	16	hd reduction acts on partially applied cons
		7	M	9	evaluation isn't allowed on the rhs of applications
		8	U	15	lookup always returns int
		9	S	18	variables aren't required to match in lookup
stlc-sub	241	1	S	8	forgot the variable case
		2	S	13	wrong order of arguments to replace call
		3	S	10	swaps function and argument position in application
		4	D	22	variable not fresh enough
		5	SM	17	replace all variables
		6	S	8	forgot the variable case
		7	S	13	wrong order of arguments to replace call
		8	S	10	swaps function and argument position in application
		9	SM	17	replace all variables
let-poly	640	1	S	8	use a lambda-bound variable where a type variable should have been
		2	D	28	the classic polymorphic let + references bug
		3	M	3	mix up types in the function case
		4	S	8	misspelled the name of a metafunction in a side-condition, causing the occurs cl
		5	M	3	eliminate-G was written as if it always gets a Gx as input
		6	M	6	$\vee$ has an incorrect duplicated variable, leading to an uncovered case
		7	D	12	used let --> left-left-λ rewrite rule for let, but the right-hand side is less polymo
list-machine	256	1	S	22	confuses the lhs value for the rhs value in cons type rule
		2	M	22	var-set may skip a var with matching id (in reduction)
		3	S	29	cons doesn't actually update the store
rbtrees	187	1	M	13	ins does no rebalancing
		2	M	15	the first case is removed from balance
		3	S	51	doesn't increment black depth in non-empty case
delim-cont	287	1	M	46	guarded mark reduction doesn't wrap results with a list/c
		2	M	25	list/c contracts aren't applied properly in the cons case
		3	S	52	the function argument to call/comp has the wrong type
rvm	712	2	M	24	stack offset / pointer confusion
		3	D	33	application slots not initialized properly
		4	M	17	mishandling branches when then branch needs more stack than else branch; bug
		5	M	23	mishandling branches when then branch needs more stack than else branch; bug
		6	M	15	forgot to implement the case-lam branch in verifier
		14	M	27	certain updates to initialized slots could break optimizer assumptions
		15	S	21	neglected to restrict case-lam to accept only 'val' arguments

Figure 1: Benchmark Overview

constructed list (i.e., the term `(cons 0)` instead of `((cons 0) nil)`). Only the grammar based random generation exposed bugs 5 and 6 and none of our approaches exposed bug 4.

The seventh bug, also classified as medium, omits a production from the definition of evaluation contexts and thus doesn't reduce the right-hand-side of function applications.

Bug 8 always returns the type `int` when looking up a variable's type in the context. This bug (and the identical one in the next system) are the only bugs we classify as unnatural. We included it because it requires a program to have a variable with a type that is more complex that just `int` and to actually use that variable somehow.

Bug 9 is simple; the variable lookup function has an error where it doesn't actually compare its input to variable in the environment, so it effectively means that each variable has the type of the nearest enclosing lambda expression.

3.1.2 poly-stlc

This is a polymorphic version of §3.1.1 "stlc", with a single numeric base type, polymorphic lists, and polymorphic versions of the list constants. No changes were made to the model except those necessary to make the list operations polymorphic. There is no type inference in the model, so all polymorphic terms are required to be instantiated with the correct types in order for the function to type check. Of course, this makes it much more difficult to automatically generate well-typed terms, and thus counterexamples. As with **stlc**, the property checked is type soundness.

All of the bugs in this system are identical to those in **stlc**, aside from any changes that had to be made to translate them to this model.

This model is also a subset of the language specified in Pałka et al. (2011), who used a specialized and optimized QuickCheck generator for a similar type system to find bugs in GHC. We adapted this system (and its restriction in **stlc**) because it has already been used successfully with random testing, which makes it a reasonable target for an automated testing benchmark.

3.1.3 stlc-sub

The same language and type system as §3.1.1 "stlc", except that in this case all of the errors are in the substitution function.

Our own experience has been that it is easy to make subtle errors when writing substitution functions, so we added this set of tests specifically to target them with the benchmark. There are two soundness checks for this system. Bugs 1-5 are checked in the following way: given a candidate counterexample, if it type checks, then all βv-redexes in the term are reduced (but not any new ones that might appear) using the buggy substitution function to get a

second term. Then, these two terms are checked to see if they both still type check and have the same type and that the result of passing both to the evaluator is the same.

Bugs 4-9 are checked using type soundness for this system as specified in the discussion of the §3.1.1 "stlc" model. We included two predicates for this system because we believe the first to be a good test for a substitution function but not something that a typical Redex user would write, while the second is something one would see in most Redex models but is less effective at catching bugs in the substitution function.

The first substitution bug we introduced simply omits the case that replaces the correct variable with the term to be substituted. We considered this to be a shallow error, and indeed all approaches were able to uncover it, although the time it took to do so varied.

Bug 2 permutes the order of arguments when making a recursive call. This is also categorized as a shallow bug, although it is a common one, at least based on our experience writing substitutions in Redex.

Bug 3 swaps the function and argument positions of an application while recurring, again essentially a typo and a shallow error, although one of the more difficult to find in this model.

The fourth substitution bug neglects to make the renamed bound variable fresh enough when recurring past a lambda. Specifically, it ensures that the new variable is not one that appears in the body of the function, but it fails to make sure that the variable is different from the bound variable or the substituted variable. We categorized this error as deep because it corresponds to a misunderstanding of how to generate fresh variables, a central concern of the substitution function.

Bug 5 carries out the substitution for all variables in the term, not just the given variable. We categorized it as SM, since it is essentially a missing side condition, although a fairly egregious one.

Bugs 6-9 are duplicates of bugs 1-3 and bug 5, except that they are tested with type soundness instead. (It is impossible to detect bug 4 with this property.)

3.1.4 let-poly

A language with ML-style `let` polymorphism, included in the benchmark to explore the difficulty of finding the classic let+references unsoundness. With the exception of the classic bug, all of the bugs were errors made during the development of this model (and that were caught during development).

The first bug is simple; it corresponds to a typo, swapping an `x` for a `y` in a rule such that a type variable is used as a program variable.

Bug number 2 is the classic let+references bug. It changes the rule for `let`-bound variables in such a way that generalization is allowed even when the initial value expression is not a

value.

Bug number 3 is an error in the function application case where the wrong types are used for the function position (swapping two types in the rule).

Bugs 4, 5, and 6 were errors in the definition of the unification function that led to various bad behaviors.

Finally, bug 7 is a bug that was introduced early on, but was only caught late in the development process of the model. It used a rewriting rule for `let` expressions that simply reduced them to the corresponding `((`λ expressions. This has the correct semantics for evaluation, but the statement of type-soundness does not work with this rewriting rule because the let expression has more polymorphism that the corresponding application expression.

3.1.5 list-machine

An implementation of Appel et al. (2012)'s list-machine benchmark. This is a reduction semantics (as a pointer machine operating over an instruction pointer and a store) and a type system for a seven-instruction first-order assembly language that manipulates `cons` and `nil` values. The property checked is type soundness as specified in Appel et al. (2012), namely that well-typed programs always step or halt. Three mutations are included.

The first list-machine bug incorrectly uses the head position of a cons pair where it should use the tail position in the cons typing rule. This bug amounts to a typo and is classified as simple.

The second bug is a missing side-condition in the rule that updates the store that has the effect of updating the first position in the store instead of the proper position in the store for all of the store update operations. We classify this as a medium bug.

The final list-machine bug is a missing subscript in one rule that has the effect that the list cons operator does not store its result. We classify this as a simple bug.

3.1.6 rbtrees

A model that implements the red-black tree insertion function and checks that insertion preserves the red-black tree invariant (and that the red-black tree is a binary search tree).

The first bug simply removes the re-balancing operation from insert. We classified this bug as medium since it seems like the kind of mistake that a developer might make in staging the implementation. That is, the re-balancing operation is separate and so might be put off initially, but then forgotten.

The second bug misses one situation in the re-balancing operation, namely when a black

node has two red nodes under it, with the second red node to the right of the first. This is a medium bug.

The third bug is in the function that counts the black depth in the red-black tree predicate. It forgets to increment the count in one situation. This is a simple bug.

3.1.7 delim-cont

Takikawa et al. (2013)'s model of a contract and type system for delimited control. The language is Plotkin's PCF extended with operators for delimited continuations, continuation marks, and contracts for those operations. The property checked is type soundness. We added three bugs to this model.

The first was a bug we found by mining the model's git repository's history. This bug fails to put a list contract around the result of extracting the marks from a continuation, which has the effect of checking the contract that is supposed to be on the elements of a list against the list itself instead. We classify this as a medium bug.

The second bug was in the rule for handling list contracts. When checking a contract against a cons pair, the rule didn't specify that it should apply only when the contract is actually a list contract, meaning that the cons rule would be used even on non-list contacts, leading to strange contract checking. We consider this a medium bug because the bug manifests itself as a missing `list/c` in the rule.

The last bug in this model makes a mistake in the typing rule for the continuation operator. The mistake is to leave off one-level of arrows, something that is easy to do with so many nested arrow types, as continuations tend to have. We classify this as a simple error.

3.1.8 rvm

A existing model and test framework for the Racket virtual machine and bytecode verifier (Klein et al. 2013). The bugs were discovered during the development of the model and reported in section 7 of that paper. Unlike the rest of the models, we do not number the bugs for this model sequentially but instead use the numbers from Klein et al. (2013)'s work.

We included only some of the bugs, excluding bugs for two reasons:

- The paper tests two properties: an internal soundness property that relates the verifier to the virtual machine model, and an external property that relates the verifier model to the verifier implementation. We did not include any that require the latter properties because it requires building a complete, buggy version of the Racket runtime system to include in the benchmark.
- We included all of the internal properties except those numbered 1 and 7 for practical

reasons. The first is the only bug in the machine model, as opposed to just the verifier, which would have required us to include the entire VM model in the benchmark. The second would have required modifying the abstract representation of the stack in the verifier model in contorted way to mimic a more C-like implementation of a global, imperative stack. This bug was originally in the C implementation of the verifier (not the Redex model) and to replicate it in the Redex-based verifier model would require us to program in a low-level imperative way in the Redex model, something not easily done.

These bugs are described in detail in Klein et al. (2013)'s paper.

This model is unique in our benchmark suite because it includes a function that makes terms more likely to be useful test cases. In more detail, the machine model does not have variables, but instead is stack-based; bytecode expressions also contain internal pointers that must be valid. Generating a random (or in-order) term is relatively unlikely to produce one that satisfies these constraints. For example, of the first 10,000 terms produced by the in-order enumeration only 1625 satisfy the constraints. The ad hoc random generator generators produces about 900 good terms in 10,000 attempts and the uniform random generator produces about 600 in 10,000 attempts.

To make terms more likely to be good test cases, this model includes a function that looks for out-of-bounds stack offsets and bogus internal pointers and replaces them with random good values. This function is applied to each of the generated terms before using them to test the model.

3.2 Managing Benchmark Modules

This section describes utilities for making changes to existing modules to create new ones, intended to assist in adding bugs to models and keeping buggy models in sync with changes to the original model.

```
(define-rewrite id from ==> to
  [#:context (context-id ...)
   #:variables (variable-id ...)
   #:once-only
   #:exactly-once])
```

Defines a syntax transformer bound to `id`, the effect of which is to rewrite syntax matching the pattern `from` to the result expression `to`. The `from` argument should follow the grammar of a `syntax-case` pattern, and `to` acts as the corresponding result expression. The behavior of the match is the same as `syntax-case`, except that all identifiers in `from` are treated as literals with the exception of an identifier that has the same binding as a `variable-id` appearing in the `#:variables` keyword argument, which is treated as a pattern variable. (The reverse of the situation for `syntax-case`, where literals must be specified instead.)

The rewrite will only be applied in the context of a `module` form, but it will be applied wherever possible within the module body, subject to a few constraints.

The rest of the keyword arguments control where and how often the rewrite may be applied. The `#:once-only` option specifies that the rewrite can be applied no more than once, and the `#:exactly-once` option asserts that the rewrite must be applied once (and no more). In both cases a syntax error is raised if the condition is not met. The `#:context` option searches for syntax of the form `(some-id . rest)`, where the binding of `some-id` matches that of the first `context-id` in the `#:context` list, at which point it recurs on `rest` but drops the first id from the list. Once every `context-id` has been matched, the rewrite can be applied.

```
(define-rewrite/compose id rw-id ...)
```

Defines a syntax transformer bound to `id`, assuming that every `rw-id` also binds a syntax transformer, such that `id` has the effect of applying all of the `rw-id`s.

```
(include/rewrite path-spec mod-id rw-id ...)
```

If the syntax designated by `path-spec` is a module, the module syntax is inlined as a submodule with the identifier `mod-id`. Assumes each `rw-id` binds a syntax transformer, and applies them to the resulting module syntax. The syntax of `path-spec` must be same as for `include`.

For example, if the contents of the file mod-fx.rkt are:

"mod-fx.rkt"

```
#lang racket/base

(provide f)
(define x 'X!)
(define (f x) x)
```

Then:

```
> (define-rewrite xy-rw
    x ==> y
    #:context (f)
    #:once-only)

> (require "mod-fx.rkt")

> (f 3)
3
> (include/rewrite "mod-fx.rkt" submod-fx xy-rw)
```

```
> (require (prefix-in s: 'submod-fx))

> (s:f 3)
'X!
```

3.3 Running Benchmark Models

```
(run-gen-and-check  get-gen
                    check
                    seconds
                   [#:name name
                    #:type type]) → run-results?
  get-gen : (-> (-> any/c))
  check : (-> any/c boolean?)
  seconds : natural-number/c
  name : string? = "unknown"
  type : symbol? = 'unknown
```

Repeatedly generates random terms and checks if they are counterexamples to some property defined by *check*, where a term is considered a counterexample if *check* returns `#f` for that term.

The *get-gen* thunk is called to build a generator of random terms (which may close over some state). A new generator is created each time the property is found to be false.

Each generated term is passed to *check* to see if it is a counterexample. The interval in milliseconds between counterexamples is tracked, and the process is repeated either until the time specified by *seconds* has elapsed or the standard error in the average interval between counterexamples is less than 10% of the average.

The result is an instance of `run-results` containing the total number of terms generated, the total elapsed time, and the number of counterexamples found. More detailed information can be obtained using the benchmark logging facilities, for which *name* is refers to the name of the model, and *type* is a symbol indicating the generation type used.

```
(struct run-results (tries time cexps))
  tries : natural-number/c
  time : natural-number/c
  cexps : natural-number/c
```

Minimal results for one run of a generate and check pair.

```
(run-gen-and-check/mods  gen-mod-path
                         check-mod-path
                         seconds
                        [#:name name])   → run-results?
  gen-mod-path : module-path?
  check-mod-path : module-path?
  seconds : natural-number/c
  name : string? = "unknown"
```

Just like `run-gen-and-check`, except that *gen-mod-path* and *check-mod-path* are module paths to a *generator module* and a *check module*, which are assumed to have the following characteristics:

- A *generator module* provides the function `get-generator`, which meets the specification for the `get-gen` argument to `run-gen-and-check`, and `type`, which is a symbol designating the type of the generator.
- A *check module* provides the function `check`, which meets the specification for the `check` argument to `run-gen-and-check`.

3.4 Logging

```
(struct bmark-log-data (data))
  data : any/c
```

Contains data logged by the benchmark, as described below.

Detailed information gathered during a benchmark run is logged to the `current-logger`, at the `'info` level, with the message `"BENCHMARK-LOGGING"`. The `data` field of the log message contains a `bmark-log-data` struct, which wraps data of the form:

```
log-data = (list event timestamp data-list)
```

Where `event` is a symbol that designates the type of event, and `timestamp` is symbol that contains the `current-date` of the event in ISO-8601 format. The information in `data-list` depends on the event, but must be in the form of a list alternating between a keyword and a datum, where the keyword is a short description of the datum.

The following events are logged (the symbol designating the event is in parentheses, and the form of the data logged for each event is shown):

- Run starts (`'start`), logged when beginning a run with a new generate/check pair.
  ```
  data-list = (list '#:model model '#:type gen)
  ```

- Run completions (`'finished`), logged at the end of a run.

```
data-list = (list '#:model model '#:type gen '#:time-ms time
                  '#:attempts tries
                  '#:num-counterexamples countxmps
                  '#:rate-terms/s rate '#:attempts/cexp atts)
```

- Every counterexample found (`'counterexample`).

```
data-list = (list '#:model model '#:type gen
                  '#:counterexample term '#:iterations tries
                  '#:time time)
```

- New average intervals between counterexamples (`'new-average`), which are recalculated whenever a counterexample is found.

```
data-list = (list '#:model model '#:type gen
                  '#:average avg '#:stderr err)
```

- Major garbage collections (`'gc-major`).

```
data-list = (list '#:amount amount '#:time time)
```

- Heartbeats (`'hearbeat`) are logged every 10 seconds by the benchmark as a way to be sure that the benchmark has not crashed.

```
data-list = (list '#:model model '#:type gen)
```

- Timeouts (`'timeout`), which occur when generating or checking a single takes term longer than 5 minutes.

```
data-list = (list '#:during 'check '#:term term '#:model model
                  '#:type gen)
          | (list '#:during 'generation '#:model model '#:type gen)
```

```
(benchmark-logging-to filename thunk) → any/c
  filename : string?
  thunk : (-> any/c)
```

Intercepts events logged by the benchmark and writes the data specified by the `log-data` production above to *`filename`*.

```
(bmark-log-directory)
 → (or/c path-string? path-for-some-system? 'up 'same)
(bmark-log-directory directory) → void?
  directory : (or/c path-string? path-for-some-system? 'up 'same)
 = (current-directory)
```

Controls the directory where `filename` in `benchmark-logging-to` is located.

3.5 Plotting

Plotting and analysis tools consume data of the form produced by the benchmark logging facilities (see §3.4 “Logging”).

TODO!

3.6 Finding the Benchmark Models

The models included in the distribution of the benchmark are in the "redex/benchmark/models" subdirectory of the redex-benchmark package. Each such subdirectory contains an info file named according to the pattern "<name>-info.rkt", defining a module that provides the function:

```
(all-mods)
 → (listof (list/c string? module-path? module-path?))
```

Returns a list of generate and check pairs for a given model or set of models, such that for each pair the first element is the name of the model, the second is a module defining a generator, and the third is a module defining a check function.

The file "redex/benchmark/models/all-info.rkt" provides an all-mods function listing all of the generate and check pairs included in the benchmark.

A command line interface is provided by the file "redex/benchmark/run-benchmark.rkt", which takes an “info” file as described above as its primary argument and provides options for running the listed tests. It automatically writes results from each run to a separate log file, all of which are located in a temporary directory. (The directory path is printed to standard out at the beginning of the run).

Bibliography

Andrew W. Appel, Robert Dockins, and Xavier Leroy. A list-machine benchmark for mechanized metatheory. *Journal of Automated Reasoning* 49(3), pp. 453–491, 2012. `http://www.cs.princeton.edu/~appel/listmachine/`

Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal* 9(4), pp. 244–257, 1970. `http://dl.acm.org/citation.cfm?id=1663480`

Casey Klein, Robert Bruce Findler, and Matthew Flatt. The Racket virtual machine and randomized testing. *Higher-Order and Symbolic Computation*, 2013. `http://plt.eecs.northwestern.edu/racket-machine/`

John McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming And Formal Systems* by P. Braffort and D. Hirschberg (Ed.), 1963. `http://www-formal.stanford.edu/jmc/basis.html`

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proc. International Workshop on Automation of Software Test*, 2011. `http://dl.acm.org/citation.cfm?id=1982615`

Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. Constraining Delimited Control with Contracts. In *Proc. European Symposium on Programming*, pp. 229–248, 2013. `http://dl.acm.org/citation.cfm?id=2450287`

Ramin Zabih, David McAllester, and David Chapman. Non-deterministic Lisp with dependency-directed backtracking. In *Proc. Proceedings of the Sixth National Conference on Artificial Intelligence*, pp. 59–64, 1987.

Index