

# Artifact: A low-level look at A-normal Form

Version 8.10

William J. Bowman <wjb@williamjbowman.com>

September 20, 2024

```
(require "main.rkt")
```

This artifact guide is available as a PDF and in HTML. The HTML version is recommended, as it features a more complete table of contents, navigation bar, hyperlinking to additional online documentation, and improved typesetting of example code.

# **1 Hardware Dependencies**

Standard consumer laptop hardware should be sufficient. An Intel or Arm 64-bit CPU is required to run standard builds of Racket, and at at least 4GBs of RAM to run some tests (2GB might suffice). There is no required number of cores nor is a GPU required, nor a required operating system.

## 2 Getting Started

This artifact requires Racket, and has been tested in Racket 8.7 and above. We recommend running this artifact locally, but it can also be run using the provided Docker files or QEMU image. An installer can be downloaded for most major systems at <https://download.racket-lang.org>, including both Intel and Arm hardware. It relies on the Redex package, which is included in the standard Racket install, but can be installed manually using `raco pkg install redex`.

To run the Docker images, you can build the image using *e.g.* `docker build -tag oopsla2024-159-artifact-amd64 -f Dockerfile.amd64 -platform linux/amd64 .`, then run `docker run -it oopsla2024-159-artifact-amd64`. The artifact is stored in `/artifact`, which should be the working directory for the Docker image.

To run the QEMU image, you can run *e.g.* `d2vm run qemu oopsla2024-159-artifact-amd64.qcow2` or `qemu-system-x86_64 -m 4G oopsla2024-159-artifact-amd64.qcow2`. The root password is `root`. The artifact is stored in `/artifact`.

The main entry point for this artifact, which provides all languages, examples, reduction relations, and compilers defined in the paper, is `main.rkt`. This is largely implemented in [redex](#), but this documentation tries to provide sufficient usage instructions that knowledge of Redex is not required.

You can use the artifact by running expressions in the REPL after importing `main.rkt`, or running any other module interactively with `(require "main.rkt")`, or running `racket -i -t main.rkt` on the command line, or opening `main.rkt` in DrRacket. If running on a headless machine (including running a Docker container), you need to launch using `xvfb-run`, such as `xvfb-run racket -it main.rkt`, as Redex provides some optional GUI components. Starting `main.rkt` may take a minute as importing this module will compile and run the test suite.

There are minor differences between the paper and the implementation:

- The syntax for function application uses the keyword `apply` rather than juxtaposition. This means some rules in the paper are split into two rules in the artifact: one for operators with juxtaposition syntax, one for `apply`.
- `let` requires a second pair of parenthesis, as in `(let ([x 5]) x)`.
- `if0` is instead written `ifz`.
- Fresh names may be generated or renamed differently.

After installing Racket, run `raco test ./` or `xvfb-run raco test ./` in the same directory as `main.rkt` to run the entire test suite; this should take several minutes. This should end with 55 tests passed. If it does, your installation should be working correctly.

To test interactively, run `racket -i` or `xvfb-run racket -i` in the same directory as `main.rkt`, then run the following expressions in the REPL:

Examples:

```
> (require "main.rkt")
> reg1-term
'(letregion
  r2
  (@
    r0
    (*
      (letregion r1 (@ r2 (* (@ r1 1) (@ r1 2))))
      (letregion r3 (@ r2 (* (@ r3 3) (@ r3 4)))))))
> (cesk-max-stack (apply-reduction-relation* #:all? #t cesk-
> (init-cesk (anf-compile reg1-term))))
3
> (cesk-max-stack (apply-reduction-relation* #:all? #t cesk-
> (init-cesk (abnormal-compile reg1-term))))
0
```

If you get the same return values as in the examples above, your installation should be working correctly.

### 3 Introduction

The theorems of the paper are testable via the artifact, and supported by tests. Step-by-step instructions are provided below to create and run additional tests. The theorems are not mechanically verified; Redex is not a theorem prover. However, conjuncts in theorems featuring existential quantification are proven via tests.

- Theorem 3.1: A-normalization optimizes the stack: tested in `3-machine-semantics.rkt` lines 189–218. See also [test-theorem-3.1?](#).
- Theorem 3.2: A-normalization is unsafe-for-scope: tested in `3-machine-semantics.rkt` line 650–668; this proves the inequality is strict in some case. See also [test-theorem-3.2?](#).
- Theorem 3.3: B-normalization is safe-for-scope: tested in `3-machine-semantics.rkt` line 664. See also [test-theorem-3.3?](#).
- Theorem 4.1: AB-normalization optimizes the stack: tested in `4-ab-normal-form.rkt` lines 240–246; this proves the inequality is strict in some case. See also [test-theorem-4.1?](#).
- Theorem 4.2: AB-normalization is safe-for-scope: tested in `4-ab-normal-form.rkt` lines 620–634; this proves the inequality is strict in some case. See also [test-theorem-4.2?](#).

In addition, all examples, machines, languages, and compilers are defined, so new examples can be run and implementations can be extended. These are documented in §5 “Reusability Guide”

## 4 Step-by-Step Instructions

```
(test-theorem-3.1? C) → #t
C : (redex-match? monadicL C)
```

Theorem 3.1 claims that A-normalizing a  $\lambda$  term in B-normal form results in at least no more stack usage, and strictly less in some cases. To test this, for `monadicL C`, the following expression should return `#t` (true):

```
(>=
 (ck-max-stack (apply-reduction-relation* #:all? #t bnf-ck-
 > (init-ck C)))
 (ck-max-stack (apply-reduction-relation* #:all? #t anf-ck-
 > (init-ck (a-normalize C)))))
```

For convenience, the function `test-theorem-3.1?` can be used. The function will validate its input and output. The function also tests the witness for strict equality case of the theorem.

```
(>
 (ck-max-stack (apply-reduction-relation* #:cache-all? #t bnf-ck-
 > (init-ck (b-normalize running-example)))))
 (ck-max-stack (apply-reduction-relation* #:cache-all? #t anf-ck-
 > (init-ck (a-normalize running-example)))))
```

Examples:

```
> (test-theorem-3.1? intro-example-monadic)
#t
> (test-theorem-3.1? (b-normalize fact-example))
#t
```

```
(test-theorem-3.2? C) → #t
C : (redex-match? bregionL C)
```

Theorem 3.2 claims that A-normalizing a  $\lambda r$  term in B-normal form results in at least as much maximum memory and as many maximum regions in use, and more in some cases. To test this, for any `bregionL` term `C`, the following expressions should return `#t` (true):

```
(>=
 (csk-max-memory (apply-reduction-relation* #:all? #t region-csk-
 > (init-csk (car (apply-reduction-relation* #:cache-all? #t ra-
 > * C)))))
 (csk-max-memory (apply-reduction-relation* #:all? #t region-csk-
 > (init-csk C)))))
```

```
(>=
  (csk-max-regions (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk (car (apply-reduction-relation* #:cache-all? #t ra-
      >* C))))))
  (csk-max-regions (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk C))))
```

For convenience, the function `test-theorem-3.2?` can be used. The function will validate its input and output. The function also tests the witnesses for the strict equality cases of the theorem:

```
(>
  (csk-max-memory (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk (car (apply-reduction-relation* #:cache-all? #t ra-
      >* reg1-term))))))
  (csk-max-memory (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk reg1-term))))

(>
  (csk-max-regions (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk (car (apply-reduction-relation* #:cache-all? #t ra-
      >* reg1-term))))))
  (csk-max-regions (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk reg1-term))))
```

Example:

```
> (test-theorem-3.2? bnf-reg1-term)
#t
```

```
(test-theorem-3.3? C) → #t
  C : (redex-match? regionL e)
```

Theorem 3.3 claims that B-normalizing any a  $\lambda r$  term has no effect on maximum memory nor maximum regions in use. To test this, for any `regionL e`, the following expressions should return `#t` (true):

```
(=
  (csk-max-memory (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk (car (apply-reduction-relation* #:cache-all? #t rb-
      >* e))))))
  (csk-max-memory (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk e))))
```

```
(=
  (csk-max-regions (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk (car (apply-reduction-relation* #:cache-all? #t rb-
      >* e))))))
  (csk-max-regions (apply-reduction-relation* #:all? #t region-csk-
    > (init-csk e))))
```

For convenience, the function `test-theorem-3.3?` can be used. The function will validate its input and output.

Example:

```
> (test-theorem-3.3? reg1-term)
#t
```

```
(test-theorem-4.1? C) → #t
  C : (redex-match? imonadicL t)
```

Theorem 4.1 claims that AB-normalizing an imperative monadic term results in at least no more stack usage, and strictly less in some cases. To test this, for any `imonadicL t`, the following expressions should return `#t` (true):

```
(>=
  (cek-max-stack (apply-reduction-relation* #:all? #t cek-> (init-
    cek t))))
  (cek-max-stack (apply-reduction-relation* #:all? #t cek-> (init-
    cek (ab-normalize t))))))
```

For convenience, the function `test-theorem-4.1?` can be used. The function will validate its input and output. The function also tests the witness for the strict equality case of the theorem.

```
(>
  (cek-max-stack (apply-reduction-relation* #:all? #t cek-> (init-
    cek monadic-large-eg-term))))
  (cek-max-stack (apply-reduction-relation* #:all? #t cek-> (init-
    cek (ab-normalize monadic-large-eg-term))))))
```

Example:

```
> (test-theorem-4.1? (monadic-code-gen (b-normalize nested-
  branches-example))))
#t
```



```
(test-theorem-4.2? C) → #t
C : (redex-match? rimonadicL t)
```

Theorem 4.2 claims that AB-normalizing has no effect on maximum memory nor maximum regions in use, and results in at least no more stack usage, and strictly less in some cases. To test this, for any `rimonadicL t`, the following expressions should return `#t` (true):

```
(>=
 (cesk-max-stack (apply-reduction-relation* #:all? #t cesk-
 > (init-cesk t)))
 (cesk-max-stack (apply-reduction-relation* #:all? #t cesk-
 > (init-cesk (abnf t)))))

(=
 (cesk-max-memory (apply-reduction-relation* #:all? #t cesk-
 > (init-cesk t)))
 (cesk-max-memory (apply-reduction-relation* #:all? #t cesk-
 > (init-cesk (abnf t)))))

(=
 (cesk-max-regions (apply-reduction-relation* #:all? #t cesk-
 > (init-cesk t)))
 (cesk-max-regions (apply-reduction-relation* #:all? #t cesk-
 > (init-cesk (abnf t)))))
```

For convenience, the function `test-theorem-4.2?` can be used. The function will validate its input and output. The function also tests the witness for the strict equality case of the theorem.

```
(>
 (cesk-max-stack (apply-reduction-relation* #:all? #t cesk-
 > (init-cesk lst:code-gen-mregion-eg1)))
 (cesk-max-stack (apply-reduction-relation* #:all? #t cesk-
 > (init-cesk (abnf lst:code-gen-mregion-eg1)))))
```

Example:

```
> (test-theorem-4.2? (abnf (monadic-code-gen bnf-reg1-term)))
#t
```

## 5 Reusability Guide

Complete interface and usage documentation is provided in later sections. The following list maps figures and listing from the paper to file and line numbers in the implementation, and briefly describes how models can be extended.

- Listing 1: `1-lc-syntax.rkt`, line 40 defines the listing as `intro-example`.
- Listing 2: `2-anf-and-monadic-form.rkt`, line 68 tests that Listing 1 A-normalizes to Listing 2.
- Listing 3: `3-machine-semantics.rkt`, line 716 defines the listing as `listing3`.
- Listing 4: `3-machine-semantics.rkt`, line 723 defines the listing as `listing4`; line 466 tests that Listing 3 A-normalizes to Listing 4.
- Listing 5: `1-lc-syntax.rkt`, line 40 defines the listing as `intro-example`; `2-anf-and-monadic-form.rkt`, line 65 tests that the example is not in ANF.
- Listing 6: `1-lc-syntax.rkt`, line 45 defines the listing as `intro-example-monadic`; `2-anf-and-monadic-form.rkt`, line 66 tests that the example is not in ANF.
- Listing 7: `1-lc-syntax.rkt`, line 50 defines the listing as `intro-example-anf`; `2-anf-and-monadic-form.rkt`, line 64 tests that the example is in ANF.
- Figure 1: `1-lc-syntax.rkt` lines 6–38 defines the syntax of  $\lambda$  terms `lambdaL` and `eval-lambdaL`. `2-anf-and-monadic.rkt`, line 11–43 defines the A-normalization reductions `a->` and `a->*`.
- Figure 2: `2-anf-and-monadic.rkt`, line 45–55 defines the language `ANFL`.
- Listing 8: `1-lc-syntax.rkt`, line 56 defines the listing as `running-example`.
- Listing 9: `2-anf-and-monadic-form.rkt` line 74 tests that Listing 8 A-normalizes to Listing 9.
- Listing 10: `1-lc-syntax.rkt`, line 60 defines the listing as `nested-branches-example`.
- Listing 11: `2-anf-and-monadic-form.rkt` line 77 tests that Listing 10 A-normalizes to Listing 11.
- Figure 3: `2-anf-and-monadic-form.rkt` lines 137–145 define the syntax of monadic form as the language `monadicL`.
- Figure 4: `2-anf-and-monadic-form.rkt` lines 104–133 define the B-normalization for  $\lambda$  terms as `b->` and `b->*`. The artifact splits the application rule into 2 rules, one for built-in operators, and one for functions.

- Listing 12: 1-lc-syntax.rkt, line 60 defines the listing as [nested-branches-example](#).
- Listing 13: 2-anf-and-monadic-form.rkt line 161 tests that Listing 12 B-normalizes to Listing 13.
- Listing 14: 2-anf-and-monadic-form.rkt line 172 tests that Listing 12 B- then A-normalizes to Listing 14.
- Figure 5: 3-machine-semantics.rkt lines 9–13 define the CK machine syntax as the language [lambda-ckL](#), and 15–50 define the CK machine [lambda-ck->](#).
- Figure 6: 3-machine-semantics.rkt lines 115–154 define the ANF CK machine [anf-ck->](#), and lines 67–110 define BNF CK machine [bnf-ck->](#).
- Figure 7: 3-machine-semantics.rkt lines 224–239 define the syntax  $\lambda r$  as [regionL](#).
- Figure 8: 3-machine-semantics.rkt lines 282–335 define the CSK machine for  $\lambda r$  as [regionCSKL](#).
- Inline definition: 3-machine-semantics.rkt lines 364–440 defines A-normalization for  $\lambda r$  as [ra->](#) and [ra->\\*](#).
- Listing 15: 3-machine-semantics.rkt line 448 defines the example as [anf-reg1-term](#).
- Listing 16: 3-machine-semantics.rkt line 602 defines the example as [bnf-reg1-term](#).
- Figure 9: 4-ab-normal-form.rkt lines 25–32 define the syntax of AB-normal form as [abnfL](#).
- Figure 10: 4-ab-normal-form.rkt lines 9–23 define the syntax of imperative monadic form as [imonadicL](#).
- Inline definition: 4-ab-normal-form.rkt lines 33–46 define the AB reductions as [ab->](#) and [ab->\\*](#).
- Figure 11: 4-ab-normal-form.rkt lines 107–125 defines the CEK machine syntax as [imonadic-cekL](#).
- Figure 12: 4-ab-normal-form.rkt lines 127–199 defines the CEK machine for imperative monadic form as [cek->](#).
- Figure 13: 4-ab-normal-form.rkt lines 248–281 defines imperative monadic with regions syntax as [rimonadicL](#).
- Figure 14: 4-ab-normal-form.rkt lines 333–438 defines the CESK machine for the imperative monadic w/ regions syntax as [cesk->](#).
- Listing 17: 4-ab-normal-form.rkt line 659 generates the example for the listing.

- Listing 18: 4-ab-normal-form.rkt line 660 generates the example for the listing.
- Figure 16: 5-compiler.rkt lines 26–106 implement the ANF compiler pass with join points as [anf](#).
- Figure 17: 5-compiler.rkt lines 112–179 implement the monadic compiler pass as [monadic](#).
- Figure 18a: 5-compiler.rkt lines 207–250 implement the code generator from ANF as [anf-code-gen](#).
- Figure 18b: 5-compiler.rkt lines 252–281 implement the code generator from monadic form as [monadic-code-gen](#).
- Figure 19: 5-compiler.rkt lines 286–344 implement the AB-normalizer from imperative code as [abnf](#).

All syntax and languages can be extended by modifying or adding new nonterminals, following the documentation for `define-language` and `define-extended-language`. All reduction relations can be extended by modifying or adding new rules, following the documentation for `reduction-relation` and `extend-reduction-relation`. New examples and terms can be defined using `define-term` or `term`; see [lambdaL](#) for examples using `redex-match?` to check that the term is syntactically valid.

## 5.1 $\lambda$ , intro, examples, etc.

[lambdaL](#) : language?

The syntax of  $\lambda$ , as a Redex language (see `define-language`).

Examples:

```
> (redex-match?
  lambdaL
  e
  (term (apply (lambda (x) 5) 6)))
#t
> (redex-match?
  lambdaL
  l
  '(apply (lambda (x) 5) 6))
#f
> (redex-match?
  lambdaL
  l
  5)
```

```

#t
> (redex-match?
  lambdaL
  x
  'x1)
#t

```

`eval-lambdaL` : language?

The syntax of  $\lambda$  extended with evaluation contexts and values, as a Redex language (see `define-language`).

Examples:

```

> (redex-match?
  eval-lambdaL
  (in-hole E v)
  '(apply (lambda (x) 5) 6))
#f
> (redex-match
  eval-lambdaL
  (in-hole E v)
  '(apply (lambda (x) 5) 6))
#f
> (redex-match?
  eval-lambdaL
  E
  '(apply hole 6))
#f

```

`intro-example` : (redex-match? `lambdaL` `e`)

The  $\lambda$  term example from the intro.

`intro-example-monadic` : (redex-match? `lambdaL` `e`)

The  $\lambda$  term example term in monadic form from the intro.

`intro-example-anf` : (redex-match? `lambdaL` `e`)

The  $\lambda$  term example term in ANF from the intro.

`running-example` : (redex-match? `lambdaL` `e`)

The  $\lambda$  term running example.

```
| nested-branches-example : (redex-match? lambdaL e)
```

The  $\lambda$  term example with nested case-of-case pattern.

Example:

```
> nested-branches-example
'(let ((x (ifz (ifz (ifz 0 0 1) 0 1) 0 1))) LARGE)
```

```
| fact-example : (redex-match? lambdaL e)
```

A definition of the factorial function as a  $\lambda$  term.

```
| lambda-ckL : language?
```

The syntax of the  $\lambda$  CK machine as a Redex language (see `define-language`).

Examples:

```
> (redex-match? lambda-ckL K 'mt)
#t
> (redex-match? lambda-ckL K (term ((+ 4 hole) :: mt)))
#t
```

```
| (init-ck term) → (redex-match? lambda-ckL (e K))
  term : (redex-match? lambdaL e)
```

Produces an initial machine configuration for the  $\lambda$  CK machine using `term` for the initial code.

In Redex, `term` behaves almost the same as `quote`, but there are exceptions. For example, you must use `term` to write a term with a `hole`.

Example:

```
> (init-ck intro-example)
'((+ (let ((x (apply f 5))) 0) 6) mt)
```

```
| lambda-ck-> : reduction-relation?
```

The  $\lambda$  CK abstract machine, as a Redex `reduction-relation`. Try using with `traces` to navigate the trace graphically.

Examples:

```
> (apply-reduction-relation* lambda-ck-> (init-ck intro-example))
'((f ((apply hole 5) :: ((let ((x hole)) 0) :: ((+ hole 6) ::
mt))))))
> (car (car (apply-reduction-relation* lambda-ck-> (init-ck fact-
example))))
120
```

## 5.2 A-normal form and Monadic form

`a-> : reduction-relation?`

The single-step A-reductions for the `lambdaL`, as a Redex reduction-relation. Try using with `traces` to navigate the trace graphically.

This is non-deterministic due to fresh name generation. The empty set of answers is returned when the program cannot single-step at the top-level.

Examples:

```
> (apply-reduction-relation a-> intro-example)
'((let ((x (apply f 5))) (+ 0 6)))
> (car (apply-reduction-relation a-> intro-example))
'(let ((x (apply f 5))) (+ 0 6))
```

`a->* : reduction-relation?`

The compatible closure of the `a->`, as a Redex reduction-relation. Try using with `traces` to navigate the trace graphically.

This is non-deterministic due to fresh name generation and (confluent) choices of evaluation contexts.

Examples:

```
> (apply-reduction-relation a->* intro-example)
'((let ((x (apply f 5))) (+ 0 6)))
> (apply-reduction-relation* a->* #:cache-all? #t nested-branches-
example)
'((ifz
  0
  (ifz
    0
    (ifz 0 (let ((x 0)) LARGE) (let ((x 1)) LARGE))
    (ifz 1 (let ((x 0)) LARGE) (let ((x 1)) LARGE)))
  (ifz
    1
    (ifz 0 (let ((x 0)) LARGE) (let ((x 1)) LARGE))
    (ifz 1 (let ((x 0)) LARGE) (let ((x 1)) LARGE)))))
```

`(a-normalize term) → (redex-match? ANFL m)`  
`term : (redex-match? lambdaL e)`

A-normalizes *term* by running *a->\** to a normal form.

Example:

```
> (a-normalize intro-example)
'(let ((x (apply f 5))) (+ 0 6))
```

**ANFL** : language?

The syntax of A-normal form, as a Redex language (see `define-language`).

Examples:

```
> (redex-match? ANFL M '(let ([x (apply f 1)]) (+ x 2)))
#t
> (redex-match? ANFL M '(+ (apply f 1) 2))
#f
```

**monadic-eval-lambdaL** : language?

The syntax of the  $\lambda$  extended with non-strict monadic evaluation contexts and values, as a Redex language (see `define-language`).

Examples:

```
> (redex-match? monadic-eval-lambdaL En (term (let ([x hole]) 5)))
#f
> (redex-match? eval-lambdaL E (term (let ([x hole]) 5)))
#t
> (redex-match? monadic-eval-lambdaL En (term (ifz hole 5 6)))
#t
> (redex-match? eval-lambdaL E (term (ifz hole 5 6)))
#t
```

**b->** : reduction-relation?

The single-step B-reductions for the **lambdaL**, as a Redex `reduction-relation`. Try using with `traces` to navigate the trace graphically.

This is non-deterministic due to fresh name generation and (confluent) choices of evaluation contexts. The empty set of answers is returned when the program cannot single-step at the top-level.

Examples:



```

> (apply-reduction-relation b-> intro-example)
'((let ((x (apply f 5))) (+ 0 6)))
> (apply-reduction-relation b-> nested-branches-example)
'()

```

**b->\*** : reduction-relation?

The compatible closure of the **b->**, as a Redex reduction-relation. Try using with **traces** to navigate the trace graphically.

This is non-deterministic due to fresh name generation and (confluent) choices of evaluation contexts.

Examples:

```

> (car (apply-reduction-relation b->* nested-branches-example))
'(let ((x (let ((x1 (ifz 0 0 1))) (ifz (ifz x1 0 1) 0 1)))) LARGE)
> (car (apply-reduction-relation* b->* nested-branches-example))
'(let ((x (let ((x1 (ifz 0 0 1))) (let ((x2 (ifz x1 0 1))) (ifz x2
0 1)))))
    LARGE)

```

**(b-normalize term)** → (redex-match? monadicL C)  
**term** : (redex-match? lambdaL e)

B-normalizes *term* by running **b->\*** to a normal form.

Example:

```

> (b-normalize nested-branches-example)
'(let ((x (let ((x1 (ifz 0 0 1))) (let ((x2 (ifz x1 0 1))) (ifz x2
0 1)))))
    LARGE)

```

**monadicL** : language?

The syntax of B-normal form, *i.e.*, monadic form, as a Redex language (see **define-language**).

Examples:

```

> (redex-match? monadicL C '(let ([x (apply f 1)]) (+ x 2)))
#t
> (redex-match? monadicL C '(+ (apply f 1) 2))
#f
> (redex-match? monadicL C '(let ((y (let ([x (apply f 1)]) (+ x 2)))) y))
#t

```

### 5.3 ANF and Monadic Machines

`bnf-ck-> : reduction-relation?`

The BNF (monadic form) CK abstract machine, as a Redex reduction-relation. Try using with `traces` to navigate the trace graphically.

Examples:

```
> (apply-reduction-relation* lambda-ck-> (init-ck running-
example))
'((f ((apply hole 1) :: ((+ 4 hole) :: mt))))
> (apply-reduction-relation* bnf-ck-> (init-ck (b-
normalize running-example)))
'(((apply f 1) ((let ((x2 hole)) (+ 4 x2)) :: mt)))
```

`anf-ck-> : reduction-relation?`

The ANF CK abstract machine, as a Redex reduction-relation. Try using with `traces` to navigate the trace graphically.

Examples:

```
> (apply-reduction-relation* lambda-ck-> (init-ck running-
example))
'((f ((apply hole 1) :: ((+ 4 hole) :: mt))))
> (apply-reduction-relation* anf-ck-> (init-ck (a-
normalize running-example)))
'(((let ((x2«999» (apply f 1))) (+ 4 x2«999»)) mt))
```

`(ck-max-stack states) → natural-number?`  
`states : (listof (redex-match? lambda-ckL (e K)))`

Returns the maximum stack length found in a CK machine trace (either `lambda-ck->` or `anf-ck->`).

Examples:

```
> (ck-max-stack (apply-reduction-relation* #:all? #t anf-ck-
> (init-ck (a-normalize running-example))))
0
> (ck-max-stack (apply-reduction-relation* #:all? #t bnf-ck-
> (init-ck (b-normalize running-example))))
1
> (ck-max-stack (apply-reduction-relation* #:all? #t lambda-ck-
> (init-ck running-example)))
2
```

```
| regionL : language?
```

The  $\lambda$ -region syntax, as a Redex language (see `define-language`).

Examples:

```
> (redex-match? regionL e '(@ r0 5))
#t
> (redex-match? regionL e '(letregion r1 (@ r0 5)))
#t
```

```
| regionCSKL : language?
```

The  $\lambda$ -region CSK abstract machine syntax, as a Redex language (see `define-language`).

Examples:

```
> (redex-match? regionCSKL K (term ((free r1) :: mt)))
#t
> (redex-match? regionCSKL S '())
#t
> (redex-match? regionCSKL S '(() (r0 ())))
#t
> (redex-match? regionCSKL S '(() (r0 (() (o1 5)))))
#t
> (redex-match? regionCSKL a '(r0 o1))
#t
```

```
| region-csk-> : reduction-relation?
```

The  $\lambda$ -region CSK abstract machine, as a Redex `reduction-relation`. Try using `with-traces` to navigate the trace graphically.

Example:

```
> (csk-read-val (apply-reduction-relation* region-csk-> (init-
csk reg1-term)))
24
```

```
| reg1-term : (redex-match? regionL e)
```

An example term for  $\lambda$ -regions.

In Redex, `term` behaves almost the same as `quote`, but there are exceptions. For example, you must use `term` to write a term with a hole.

```
(init-csk term) → (redex-match? regionCSKL (e S K))
term : (redex-match? regionL e)
```

Produces an initial machine configuration for the  $\lambda$ -region CSK machine using *term* for the initial code, and an empty initial store and kontinuation.

Example:

```
> (init-csk reg1-term)
'((letregion
  r2
  (@
    r0
    (*
      (letregion r1 (@ r2 (* (@ r1 1) (@ r1 2))))
      (letregion r3 (@ r2 (* (@ r3 3) (@ r3 4))))))
  ((r0 ()))
  mt))
```

```
(csk-read-val states) → (redex-match? regionL v)
states : (listof (redex-match? regionCSKL (e S K)))
```

Produces the final value of the CSK machine from its final configuration. Expects that *states* is a singleton set of terminal machine configurations, *i.e.*, that the machine normalized to a single final state, whose code is a valid address and whose kontinuation is empty.

Example:

```
> (csk-read-val '(((r0 o1) ((r0 ((o1 5)))) mt)))
5
```

```
(csk-max-regions trace) → natural-number?
trace : (listof (redex-match? regionCSKL (e S K)))
```

Returns the maximum number of regions allocated in the *region-csk->* trace.

Examples:

```
> (csk-max-regions (apply-reduction-relation* region-csk-> (init-csk reg1-term)))
1
> (csk-max-regions (apply-reduction-relation* region-csk-> (init-csk anf-reg1-term)))
1
```

```
> (csk-max-regions (apply-reduction-relation* region-csk-> (init-
csk bnf-reg1-term)))
1
```

```
(csk-max-memory trace) → natural-number?
  trace : (listof (redex-match? regionCSKL (e S K)))
```

Returns the maximum number of live addresses in the `region-csk->` trace.

Examples:

```
> (csk-max-memory (apply-reduction-relation* region-csk-> (init-
csk reg1-term)))
1
> (csk-max-memory (apply-reduction-relation* region-csk-> (init-
csk anf-reg1-term)))
1
> (csk-max-memory (apply-reduction-relation* region-csk-> (init-
csk bnf-reg1-term)))
1
```

```
| aregionL : language?
```

The  $\lambda$ -regions syntax extended with evaluation contexts for A-reduction, as a Redex language (see `define-language`).

Examples:

```
> (redex-match? aregionL E (term (@ r hole)))
#t
> (redex-match? aregionL E (term (let ([x hole]) e)))
#t
```

```
| ra-> : reduction-relation?
```

The single-step A-reductions for  $\lambda$ -regions (`aregionL`), as a Redex `reduction-relation`.

This is non-deterministic due to fresh name generation and (confluent) choices of evaluation contexts. The empty set of answers is returned when the program cannot single-step at the top-level.

Example:

```
> (apply-reduction-relation ra-> reg1-term)
'()
```

In Redex, `term` behaves almost the same as `quote`, but there are exceptions. For example, you must use `term` to write a term with a `hole`.

`ra->* : reduction-relation?`

The compatible closure of `ra->`, as a Redex reduction-relation.

This is non-deterministic due to fresh name generation and (confluent) choices of evaluation contexts. This can be rather slow due to non-determinism, so you probably want to enable `#:cache-all?`.

Examples:

```
> (take (apply-reduction-relation ra->* reg1-term) 3)
'((letregion
  r2
  (@
    r0
    (*
      (letregion r1 (@ r2 (let ((x (@ r1 1))) (* x (@ r1 2))))
      (letregion r3 (@ r2 (* (@ r3 3) (@ r3 4))))))
  (letregion
    r2
    (@
      r0
      (*
        (letregion r1 (let ((x (@ r1 1))) (@ r2 (* x (@ r1 2))))
        (letregion r3 (@ r2 (* (@ r3 3) (@ r3 4))))))
    (letregion
      r2
      (@
        r0
        (*
          (letregion r1 (@ r2 (* (@ r1 1) (@ r1 2)))
          (letregion r3 (@ r2 (let ((x (@ r3 3))) (* x (@ r3 4))))))))
  term))
'((letregion
  r2
  (letregion
    r1
    (let ((x (@ r1 1)))
      (let ((x1 (@ r1 2)))
        (let ((x2 (@ r2 (* x x1)))
          (letregion
            r3
            (let ((x3 (@ r3 3)))
              (let ((x4 (@ r3 4)))
```

```
(let ((x5 (@ r2 (* x3 x4)))) (@ r0 (* x2
x5)))))))))
```

```
| anf-reg1-term : (redex-match? regionL e)
```

The A-normal form of `reg1-term`

```
| bregionL : language?
```

The  $\lambda$ -regions syntax extended with non-strict monadic evaluation contexts for B-reduction, as a Redex language (see `define-language`).

Examples:

```
> (redex-match? bregionL En (term (@ r hole)))
#t
> (redex-match? bregionL En (term (let ([x hole]) e)))
#f
```

```
| rb-> : reduction-relation?
```

The single-step B-reductions for the  $\lambda$ -regions (`aregionL`), as a Redex `reduction-relation`.

This is non-deterministic due to fresh name generation and (confluent) choices of evaluation contexts. The empty set of answers is returned when the program cannot single-step at the top-level.

Example:

```
> (apply-reduction-relation rb-> reg1-term)
'()
```

```
| rb->* : reduction-relation?
```

The compatible closure of `rb->`, as a Redex `reduction-relation`.

This is non-deterministic due to fresh name generation and (confluent) choices of evaluation contexts. This can be rather slow due to non-determinism, so you probably want to enable `#:cache-all?`.

Examples:

```
> (take (apply-reduction-relation rb->* reg1-term) 3)
```

In Redex, `term` behaves almost the same as `quote`, but there are exceptions. For example, you must use `term` to write a term with a `hole`.

```

'((letregion
  r2
  (@
    r0
    (*
      (letregion r1 (@ r2 (let ((x (@ r1 1))) (* x (@ r1 2)))))
      (letregion r3 (@ r2 (* (@ r3 3) (@ r3 4)))))))
  (letregion
    r2
    (@
      r0
      (*
        (letregion r1 (let ((x (@ r1 1))) (@ r2 (* x (@ r1 2)))))
        (letregion r3 (@ r2 (* (@ r3 3) (@ r3 4)))))))
    (letregion
      r2
      (@
        r0
        (*
          (letregion r1 (@ r2 (* (@ r1 1) (@ r1 2))))
          (letregion r3 (@ r2 (let ((x (@ r3 3))) (* x (@ r3 4))))))))))
> (car (apply-reduction-relation* rb->* #:cache-all? #t reg1-
term))
'(letregion
  r2
  (let ((x
    (letregion
      r1
      (let ((x1 (@ r1 1))) (let ((x2 (@ r1 2))) (@ r2 (* x1
x2)))))))
    (let ((x3
      (letregion
        r3
        (let ((x4 (@ r3 3))) (let ((x5 (@ r3 4))) (@ r2 (* x4
x5)))))))
      (@ r0 (* x x3))))))

bnf-reg1-term : (redex-match? regionL e)

```

The B-normal form of `reg1-term`

## 5.4 Imperative A-normalization, Machines, and AB-normalization

```
imonadicL : language?
```



The imperative monadic syntax, as a Redex language (see `define-language`).

Example:

```
> (redex-match? imonadicL t '(begin (set! x (begin (set! y 5) y)) x))
#t
```

`abnfL` : language?

The imperative ANF syntax, *i.e.*, AB-normal form, as a Redex language (see `define-language`).

Examples:

```
> (redex-match? imonadicL t '(begin (set! x (begin (set! y 5) y)) x))
#t
> (redex-match? abnfL t '(begin (set! x (begin (set! y 5) y)) x))
#f
> (redex-match? abnfL t '(begin (set! y 5) (set! x y) x))
#t
```

`ab->` : reduction-relation?

The single-step AB-reductions for the `imonadicL` *statements*, as a Redex reduction-relation. Try using with `traces` to navigate the trace graphically.

The empty set of answers is returned when the program cannot single-step at the top-level.

Example:

```
> (apply-reduction-relation ab-> '(set! x (begin (set! y 5) y)))
'((begin (set! y 5) (set! x y)))
```

`ab->*` : reduction-relation?

The compatible closure of `ab->`, as a Redex reduction-relation. This lifts `ab->` to work over tails, as well. Try using with `traces` to navigate the trace graphically.

Examples:

```
> (apply-reduction-relation ab->* '(begin (set! x (begin (set! y 5) y)) x))
'((begin (begin (set! y 5) (set! x y)) x))
> (apply-reduction-relation ab->* (monadic-code-gen (b-
normalize nested-branches-example)))
```

```
'((begin
  (begin
    (set! x1 (ifz 0 0 1))
    (set! x (begin (set! x2 (ifz x1 0 1)) (ifz x2 0 1))))
  LARGE)
(begin
  (set! x
    (begin
      (set! x1 (ifz 0 0 1))
      (begin (ifz x1 (set! x2 0) (set! x2 1)) (ifz x2 0 1))))
  LARGE)
(begin
  (set! x
    (begin
      (ifz 0 (set! x1 0) (set! x1 1))
      (begin (set! x2 (ifz x1 0 1)) (ifz x2 0 1))))
  LARGE))
```

```
(ab-normalize term) → (redex-match? abnfl t)
term : (redex-match? imonadicL t)
```

AB-normalizes *term* by running *ab->\** to a normal form.

Example:

```
> (ab-normalize (monadic-code-gen (b-normalize nested-branches-
example))))
'(begin
  (begin
    (ifz 0 (set! x1 0) (set! x1 1))
    (begin (ifz x1 (set! x2 0) (set! x2 1)) (ifz x2 (set! x 0)
(set! x 1))))
  LARGE)
```

```
imonadic-cekL : language?
```

The imperative CEK abstract machine syntax, as a Redex language (see *define-language*).

Examples:

```
> (redex-match? imonadic-cekL K (term ((begin (set! x hole) x) :: mt)))
#t
> (redex-match? imonadic-cekL Σ (term (() (x 5))))
#t
```

In Redex, `term` behaves almost the same as `quote`, but there are exceptions. For example, you must use `term` to write a term with a hole.

`cek-> : reduction-relation?`

The imperative CEK abstract machine, as a Redex reduction-relation. Try using with `traces` to navigate the trace graphically.

Examples:

```
> (apply-reduction-relation* cek-> (init-cek '(begin (set! x (begin (set! y 5) y)) x)))
'((x ((( (y 5)) (x 5)) mt))
> (car (car (apply-reduction-relation* cek-> (init-cek '(begin (set! x (begin (set! y 5) y)) x))))))
'x
```

```
(init-cek term) → (redex-match? imonadic-cekL (e Σ K))
term : (redex-match? imonadicL e)
```

Produces an initial machine configuration for the monadic imperative CEK machine using `term` for the initial code.

Example:

```
> (init-cek '(begin (set! x 5) x))
'((begin (set! x 5) x) () mt)
```

```
(cek-max-stack states) → natural-number?
states : (listof (redex-match? imonadic-cekL (e Σ K)))
```

Returns the maximum stack length found in a `cek->` trace.

Examples:

```
> (cek-max-stack
  (apply-reduction-relation* #:all? #t
    cek->
    (init-cek (monadic-code-gen (b-normalize nested-branches-example))))))
2
> (cek-max-stack
  (apply-reduction-relation* #:all? #t
    cek->
    (init-cek (ab-normalize (monadic-code-gen (b-normalize nested-branches-example))))))
0
```

```
| monadic-large-eg-term : (redex-match? imonadicL t)
```

An imperative monadic example with a large continuation after a branch.

```
| rimonadicL : language?
```

The imperative monadic with regions syntax, as a Redex language (see `define-language`).

Examples:

```
> (redex-match? rimonadicL t '(begin (set! x (begin (set! y 5) y)) x))
#f
> (redex-match? rimonadicL t '(begin (set! x (begin (set! y (alloc r0 5)) y)) x))
#t
```

```
| cesk-> : reduction-relation?
```

The imperative CESK abstract machine, as a Redex reduction-relation. Try using with `traces` to navigate the trace graphically.

Examples:

```
> (apply-reduction-relation* cesk-> (init-cesk '(begin (set! x (begin (set! y (alloc r0 5)) y)) x)))
'((x ((() (y (r0 o39432))) (x (r0 o39432))) ((() (r0 ((() (o39432 5)))) mt))
> (cesk-read-val (apply-reduction-relation* cesk-> (init-cesk '(begin (set! x (begin (set! y (alloc r0 5)) y)) x))))
5
```

```
| (init-cesk term) → (redex-match? rimonadicL (t Σ S K))
term : (redex-match? rimonadicL t)
```

Produces an initial machine configuration for the monadic imperative with regions CESK machine using `term` for the initial code. The machine starts with an empty environment, empty kontinuation, and an initial empty region `r0` where the final result must be allocated.

Example:

```
> (init-cesk '(begin (set! x (alloc r0 5)) x))
'((begin (set! x (alloc r0 5)) x) () ((() (r0 ()))) mt)
```

```
| (cesk-read-val states) → (redex-match? rimonadicL v)
states : (listof (redex-match? rimonadicL (t Σ S K)))
```

Produces the final value of the CESK machine from its final configuration. Expects that `states` is a singleton set of terminal machine configurations, *i.e.*, that the machine normalized to a single final state, whose code is a valid address and whose kontinuation is empty.

Example:

```
> (cesk-read-val (apply-reduction-relation* cesk-> (init-cesk '(begin (set! x (alloc r0 5)) x))))
5
```

```
(cesk-max-stack states) → natural-number?
states : (listof (redex-match? rimonadicL (t Σ S K)))
```

Returns the maximum stack length found in a `cesk->` trace.

Examples:

```
> (cesk-max-stack (apply-reduction-relation* #:all? #t cesk->
> (init-cesk '(begin (set! x (alloc r0 5)) x))))
0
> (define D_b
  (apply-reduction-relation* #:all? #t cesk->
    (init-cesk (monadic-code-gen bnf-reg1-term))))
> (define D_a
  (apply-reduction-relation* #:all? #t cesk->
    (init-cesk (anf-code-gen anf-reg1-term))))
> (define D_ab
  (apply-reduction-relation* #:all? #t cesk->
    (init-cesk (abnf (monadic-code-gen bnf-reg1-term)))))
> (cesk-max-stack D_b)
3
> (cesk-max-stack D_a)
3
> (cesk-max-stack D_ab)
0
```

```
(cesk-max-memory states) → natural-number?
states : (listof (redex-match? rimonadicL (t Σ S K)))
```

Returns the maximum number of live addresses in the `cesk->` trace.

Examples:

```
> (cesk-max-memory (apply-reduction-relation* #:all? #t cesk->
> (init-cesk '(begin (set! x (alloc r0 5)) x))))
```

A minor bug exists in the ANF code generator for regions which unnecessarily introduces a stack frame. Note that this bug does not affect the `cek-max-stack`.

```

1
> (cesk-max-memory D_b)
4
> (cesk-max-memory D_a)
7
> (cesk-max-memory D_ab)
4

```

```

(cesk-max-regions states) → natural-number?
states : (listof (redex-match? rimonadicL (t  $\Sigma$  S K)))

```

Returns the maximum number of regions allocated in the `cesk->` trace.

Examples:

```

> (cesk-max-regions (apply-reduction-relation* #:all? #t cesk-
> (init-cesk '(begin (set! x (alloc r0 5)) x))))
1
> (cesk-max-regions D_b)
3
> (cesk-max-regions D_a)
4
> (cesk-max-regions D_ab)
3

```

```

lst:code-gen-mregion-egl : (redex-match? rimonadicL t)

```

An imperative monadic with regions term with two independent regions.

## 5.5 Compiler

```

(anf term [fresh])
→ (or/c (redex-match? regionL e) (redex-match? ANFL M))
term : (or/c (redex-match? regionL e) (redex-match? lambdaL e))
fresh : (-> symbol? symbol?) = gensym

```

Produces the A-normal form of a  $\lambda$  or  $\lambda$ -regions term. Optionally takes a source of fresh names.

Examples:

```

> (anf intro-example)

```

```

'(let ((x39459 f))
  (let ((x39460 5))
    (let ((x39458 (apply x39459 x39460)))
      (let ((x x39458))
        (let ((x39461 0)) (let ((x39462 6)) (+ x39461
x39462))))))))
> (anf reg1-term)
'(letregion
  r2
  (letregion
    r1
    (let ((x39463 (@ r1 1)))
      (let ((x39464 (@ r1 2)))
        (let ((x39465 (@ r2 (* x39463 x39464))))
          (letregion
            r3
            (let ((x39466 (@ r3 3)))
              (let ((x39467 (@ r3 4)))
                (let ((x39468 (@ r2 (* x39466 x39467))))
                  (@ r0 (* x39465 x39468))))))))))
> (anf intro-example (inc-var))
'(let ((x2 f))
  (let ((x3 5))
    (let ((x1 (apply x2 x3)))
      (let ((x x1)) (let ((x4 0)) (let ((x5 6)) (+ x4 x5))))))

(monadic term [fresh])
→ (or/c (redex-match? regionL e) (redex-match? monadicL C))
term : (or/c (redex-match? regionL e) (redex-match? lambdaL e))
fresh : (-> symbol? symbol?) = gensym

```

Produces the monadic form of a  $\lambda$  or  $\lambda$ -regions term.

Examples:

```

> (monadic intro-example)
'(let ((x39469
  (let ((x (let ((x39471 f)) (let ((x39472 5)) (apply x39471
x39472))))))
    0)))
  (let ((x39470 6)) (+ x39469 x39470)))
> (monadic reg1-term)
'(letregion
  r2
  (let ((x39473

```

```

        (letregion
          r1
          (let ((x39475 (@ r1 1)))
            (let ((x39476 (@ r1 2))) (@ r2 (* x39475 x39476))))))
      (let ((x39474
        (letregion
          r3
          (let ((x39477 (@ r3 3)))
            (let ((x39478 (@ r3 4))) (@ r2 (* x39477
x39478))))))
          (@ r0 (* x39473 x39474))))))
> (monadic reg1-term (inc-var))
'(letregion
  r2
  (let ((x1
    (letregion
      r1
      (let ((x3 (@ r1 1))) (let ((x4 (@ r1 2))) (@ r2 (* x3
x4))))))
    (let ((x2
      (letregion
        r3
        (let ((x5 (@ r3 3))) (let ((x6 (@ r3 4))) (@ r2 (* x5
x6))))))
      (@ r0 (* x1 x2))))))

(anf-code-gen term)
→ (or/c (redex-match? rimonadicL t) (redex-match? abnfl t))
term : (or/c (redex-match? regionL e) (redex-match? ANFL M))

```

Generate the CESK machine code of the ANF term *term*. The output is AB-normal form if the input was A-normal.

Examples:

```

> (flatten-begin (anf-code-gen (anf intro-example)))
'(begin
  (set! x39480 f)
  (set! x39481 5)
  (set! x39479 (call x39480 x39481))
  (set! x x39479)
  (set! x39482 0)
  (set! x39483 6)
  (+ x39482 x39483))
> (flatten-begin (anf-code-gen (anf reg1-term (inc-var))))

```



```

'(begin
  (ralloc r2)
  (set! x39484
    (begin
      (ralloc r1)
      (set! x39485
        (begin
          (set! x1 (alloc r1 1))
          (begin
            (set! x2 (alloc r1 2))
            (begin
              (set! x3 (alloc r2 (* x1 x2)))
              (begin
                (ralloc r3)
                (set! x39486
                  (begin
                    (set! x4 (alloc r3 3))
                    (begin
                      (set! x5 (alloc r3 4))
                      (begin
                        (set! x6 (alloc r2 (* x4 x5)))
                        (alloc r0 (* x3 x6))))))
                (rfree r3)
                x39486))))))
            (rfree r1)
            x39485))
        (rfree r2)
        x39484)
    )
  )
)

(monadic-code-gen term)
→ (or/c (redex-match? rimonadicL t) (redex-match? imonadicL t))
term : (or/c (redex-match? regionL e) (redex-match? monadicL C))

```

A minor bug exists in the ANF code generator for regions which unnecessarily introduces a stack frame. Note that this bug does not affect the `cek-max-stack`.

Generate the CESK machine code of the monadic term *term*.

Examples:

```

> (flatten-begin (monadic-code-gen (monadic intro-example)))
'(begin
  (set! x39487
    (begin
      (set! x
        (begin (set! x39489 f) (begin (set! x39490 5) (call
x39489 x39490))))))

```

```

    0))
    (set! x39488 6)
    (+ x39487 x39488))
> (flatten-begin (monadic-code-gen (monadic reg1-term)))
'(begin
  (ralloc r2)
  (set! x39497
    (begin
      (set! x39491
        (begin
          (ralloc r1)
          (set! x39498
            (begin
              (set! x39493 (alloc r1 1))
              (begin
                (set! x39494 (alloc r1 2))
                (alloc r2 (* x39493 x39494))))))
            (rfree r1)
            x39498))
        (begin
          (set! x39492
            (begin
              (ralloc r3)
              (set! x39499
                (begin
                  (set! x39495 (alloc r3 3))
                  (begin
                    (set! x39496 (alloc r3 4))
                    (alloc r2 (* x39495 x39496))))))
                (rfree r3)
                x39499))
              (alloc r0 (* x39491 x39492))))))
      (rfree r2)
      x39497)

```

```

(abnf term)
→ (or/c (redex-match? rimonadicL t) (redex-match? abnfl t))
term : (or/c (redex-match? rimonadicL t) (redex-match? imonadicL t))

```

AB-normalize the CESK machine code of the monadic term *term*.

Examples:

```

> (abnf (monadic-code-gen (monadic intro-example)))
'(begin

```

```

(begin
  (begin
    (set! x39502 f)
    (begin (set! x39503 5) (set! x (call x39502 x39503))))
    (set! x39500 0))
  (begin (set! x39501 6) (+ x39500 x39501)))
> (flatten-begin (abnf (monadic-code-gen (monadic reg1-term)))))
'(begin
  (ralloc r2)
  (ralloc r1)
  (set! x39506 (alloc r1 1))
  (set! x39507 (alloc r1 2))
  (set! x39511 (alloc r2 (* x39506 x39507)))
  (rfree r1)
  (set! x39504 x39511)
  (ralloc r3)
  (set! x39508 (alloc r3 3))
  (set! x39509 (alloc r3 4))
  (set! x39512 (alloc r2 (* x39508 x39509)))
  (rfree r3)
  (set! x39505 x39512)
  (set! x39510 (alloc r0 (* x39504 x39505)))
  (rfree r2)
  x39510)
> (flatten-begin (anf-code-gen (anf reg1-term)))
'(begin
  (ralloc r2)
  (set! x39519
    (begin
      (ralloc r1)
      (set! x39520
        (begin
          (set! x39513 (alloc r1 1))
          (begin
            (set! x39514 (alloc r1 2))
            (begin
              (set! x39515 (alloc r2 (* x39513 x39514)))
              (begin
                (ralloc r3)
                (set! x39521
                  (begin
                    (set! x39516 (alloc r3 3))
                    (begin
                      (set! x39517 (alloc r3 4))
                      (begin
                        (set! x39518 (alloc r2 (* x39516

```

```

x39517)))
                                (alloc r0 (* x39515 x39518))))))
                                (rfree r3)
                                x39521))))))
                                (rfree r1)
                                x39520))
                                (rfree r2)
                                x39519)
> (flatten-begin (abnf (anf-code-gen (anf reg1-term))))
'(begin
  (ralloc r2)
  (ralloc r1)
  (set! x39522 (alloc r1 1))
  (set! x39523 (alloc r1 2))
  (set! x39524 (alloc r2 (* x39522 x39523)))
  (ralloc r3)
  (set! x39525 (alloc r3 3))
  (set! x39526 (alloc r3 4))
  (set! x39527 (alloc r2 (* x39525 x39526)))
  (set! x39530 (alloc r0 (* x39524 x39527)))
  (rfree r3)
  (set! x39529 x39530)
  (rfree r1)
  (set! x39528 x39529)
  (rfree r2)
  x39528)
> (flatten-begin (anf-code-gen (anf intro-example)))
'(begin
  (set! x39532 f)
  (set! x39533 5)
  (set! x39531 (call x39532 x39533))
  (set! x x39531)
  (set! x39534 0)
  (set! x39535 6)
  (+ x39534 x39535))
> (flatten-begin (abnf (anf-code-gen (anf intro-example))))
'(begin
  (set! x39537 f)
  (set! x39538 5)
  (set! x39536 (call x39537 x39538))
  (set! x x39536)
  (set! x39539 0)
  (set! x39540 6)
  (+ x39539 x39540))

| (anf-compile term [fresh])

```

```

→ (or/c (redex-match? rimonadic t) (redex-match? abnfl t))
term : (or/c (redex-match? regionL e) (redex-match? lambdaL e))
fresh : (-> symbol? symbol?) = gensym

```

Compile a term with the ANF compiler. Optionally takes a source of fresh names.

Example:

```

> (flatten-begin (anf-compile nested-branches-example (inc-var)))
'(begin
  (set! x7 0)
  (set! j6
    (lambda (x5)
      (begin
        (set! x8 x5)
        (begin
          (set! j4
            (lambda (x3)
              (begin
                (set! x9 x3)
                (begin
                  (set! j2 (lambda (x1) (begin (set! x x1)
LARGE)))
                  (ifz
                    x9
                    (begin (set! x10 0) (call j2 x10))
                    (begin (set! x11 1) (call j2 x11)))))))
                (ifz
                  x8
                  (begin (set! x12 0) (call j4 x12))
                  (begin (set! x13 1) (call j4 x13)))))))
                (ifz
                  x7
                  (begin (set! x14 0) (call j6 x14))
                  (begin (set! x15 1) (call j6 x15))))))

```

```

(abnormal-compile term [fresh])
→ (or/c (redex-match? rimonadic t) (redex-match? abnfl t))
term : (or/c (redex-match? regionL e) (redex-match? lambdaL e))
fresh : (-> symbol? symbol?) = gensym

```

Compile a term with the AB-normal compiler. Optionally takes a source of fresh names.

Example:

```
> (flatten-begin (abnormal-compile nested-branches-example (inc-
var)))
'(begin
  (set! x3 0)
  (ifz x3 (set! x2 0) (set! x2 1))
  (ifz x2 (set! x1 0) (set! x1 1))
  (ifz x1 (set! x 0) (set! x 1))
  LARGE)
```

```
(inc-var [init]) → (-> symbol? symbol?)
init : natural? = 1
```

A predictable name generator factory. Optionally takes an starting number to append to a generated name.

May not actually generate fresh names, depending on the context in which it is used.

Examples:

```
> (define gsym (inc-var 2))
> (gsym 'x)
'x2
> (gsym 'x)
'x3
```

```
(flatten-begin term) → any/c
term : any/c
```

Flatten all nested begins; normalizes the admin1 and admin2 transitions. Not used in any of the compilers; useful for pretty-printing.

Example:

```
> (flatten-begin '(begin (begin (set! x y) (begin (set! x 5))) x))
'(begin (set! x y) (set! x 5) x)
```