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

Version 8.10

Anon Y. Mous

April 2, 2024

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

This artifact requires Racket, and has been tested in Racket 8.7 and above. An installer can can be downloaded for most major systems at `https://download.racket-lang.org`.

It relies on the Redex package, which is included in the standard Racket install, but can be installed manually using `raco pkg install redex`.

The main entry point for this artifact, which provides all languages, examples,reduction relations, and compilers defined in the paper. 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 importing "main.rkt" in the REPL or any other module with `(require "main.rkt")`, or running `racket -i -t main.rkt`, or opening "main.rkt" in DrRacket. It 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.
- `let` requires a second pair of parenthesis, as in `(let ([x 5]) x)`.

Examples:

```
> (require "main.rkt")
> a->
#<reduction-relation>
> anf
#<procedure:anf>
> abnormal-compile
```

```
#<procedure:abnormal-compile>
> anf-compile
#<procedure:anf-compile>
> (anf '(apply (lambda (x) x) 5))
'(let ((x37423 (lambda (x) x)))
   (let ((x37424 5)) (let ((x37422 (apply x37423 x37424)))
x37422)))
> (anf '(apply (lambda (x) x) 5) (inc-var))
'(let ((x2 (lambda (x) x))) (let ((x3 5)) (let ((x1 (apply x2
x3))) x1)))
```

1 λ-calculus, intro, examples, etc.

`lambdaL : language?`

The syntax of the λ-calculus, as a Redex language (see `define-language`).

Examples:

```
> (redex-match?
   lambdaL
   e
   (term (apply (lambda (x) 5) 6)))
#t
> (redex-match?
   lambdaL
   ι
   '(apply (lambda (x) 5) 6))
#f
> (redex-match?
   lambdaL
   ι
   5)
#t
> (redex-match?
   lambdaL
   x
   'x1)
#t
```

`eval-lambdaL : language?`

The syntax of the λ-calculus 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 λ-calculus example from the intro.

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

The λ-calculus running example.

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

The λ-calculus 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 in the λ-calculus.

`lambda-ckL : language?`

The syntax of the λ 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 λ 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 λ 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
```

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-determinisic 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-determinisic 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 λ-calculus extendedwith 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-determinisic 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-determinisic 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
```

3 ANF and Monadic Machines

```
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 (apply f 1))) (+ 4 x2)) 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 lambda-ck-
> (init-ck running-example)))
2
```

```
regionL : language?
```

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

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:

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

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

An example term for λ-regions.

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

Produces an initial machine configuration for the λ-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 λ-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
```

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 single-step A-reductions for λ-regions (`aregionL`), as a Redex `reduction-relation`.

This is non-determinisic 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)
'()
```

`ra->* : reduction-relation?`

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

This is non-determinisic 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))))))))
> (car (apply-reduction-relation* ra->* #:cache-all? #t reg1-
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 λ-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
```

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`.

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

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

This is non-determinisic 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-determinisic 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)
'((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`

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
```

```
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 o38846))) (x (r0 o38846))) (() (r0 (() (o38846
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
```

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`.

```
(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))))
```

```
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 Σ 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
```

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 λ-calculus or λ-regions term. Optionally takes a source of fresh names.

Examples:

```
 > (anf intro-example)
 '(let ((x38884 f))
    (let ((x38885 5))
      (let ((x38883 (apply x38884 x38885)))
        (let ((x x38883))
          (let ((x38886 0)) (let ((x38887 6)) (+ x38886
 x38887)))))))
 > (anf reg1-term)
 '(letregion
   r2
   (letregion
    r1
    (let ((x38888 (@ r1 1)))
      (let ((x38889 (@ r1 2)))
        (let ((x38890 (@ r2 (* x38888 x38889))))
          (letregion
           r3
           (let ((x38891 (@ r3 3)))
             (let ((x38892 (@ r3 4)))
               (let ((x38893 (@ r2 (* x38891 x38892))))
                 (@ r0 (* x38890 x38893)))))))))))
 > (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 λ-calculus or λ-regions term.

Examples:

```
> (monadic intro-example)
'(let ((x38894
        (let ((x (let ((x38896 f)) (let ((x38897 5)) (apply x38896
x38897)))))
          0)))
   (let ((x38895 6)) (+ x38894 x38895)))
> (monadic reg1-term)
'(letregion
  r2
  (let ((x38898
         (letregion
          r1
          (let ((x38900 (@ r1 1)))
            (let ((x38901 (@ r1 2))) (@ r2 (* x38900 x38901)))))))
    (let ((x38899
           (letregion
            r3
            (let ((x38902 (@ r3 3)))
              (let ((x38903 (@ r3 4))) (@ r2 (* x38902
x38903)))))))
      (@ r0 (* x38898 x38899)))))
> (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))
```

Genernate 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! x38905 f)
   (set! x38906 5)
   (set! x38904 (call x38905 x38906))
   (set! x x38904)
   (set! x38907 0)
   (set! x38908 6)
   (+ x38907 x38908))
> (flatten-begin (anf-code-gen (anf reg1-term (inc-var))))
'(begin
   (ralloc r2)
   (set! x38909
     (begin
       (ralloc r1)
       (set! x38910
         (begin
           (set! x1 (alloc r1 1))
           (begin
             (set! x2 (alloc r1 2))
             (begin
               (set! x3 (alloc r2 (* x1 x2)))
               (begin
                 (ralloc r3)
                 (set! x38911
                   (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)
                 x38911)))))
       (rfree r1)
       x38910))
   (rfree r2)
   x38909)
```

}

```
(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`.

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

Examples:

```
> (flatten-begin (monadic-code-gen (monadic intro-example)))
'(begin
   (set! x38912
     (begin
       (set! x
         (begin (set! x38914 f) (begin (set! x38915 5) (call
x38914 x38915))))
       0))
   (set! x38913 6)
   (+ x38912 x38913))
> (flatten-begin (monadic-code-gen (monadic reg1-term)))
'(begin
   (ralloc r2)
   (set! x38922
     (begin
       (set! x38916
         (begin
           (ralloc r1)
           (set! x38923
             (begin
               (set! x38918 (alloc r1 1))
               (begin
                 (set! x38919 (alloc r1 2))
                 (alloc r2 (* x38918 x38919)))))
           (rfree r1)
           x38923))
       (begin
         (set! x38917
           (begin
             (ralloc r3)
             (set! x38924
               (begin
                 (set! x38920 (alloc r3 3))
                 (begin
                   (set! x38921 (alloc r3 4))
                   (alloc r2 (* x38920 x38921)))))
             (rfree r3)
             x38924))
         (alloc r0 (* x38916 x38917)))))
   (rfree r2)
   x38922)
```

```
(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! x38927 f)
       (begin (set! x38928 5) (set! x (call x38927 x38928))))
     (set! x38925 0))
   (begin (set! x38926 6) (+ x38925 x38926)))
> (flatten-begin (abnf (monadic-code-gen (monadic reg1-term))))
'(begin
   (ralloc r2)
   (ralloc r1)
   (set! x38931 (alloc r1 1))
   (set! x38932 (alloc r1 2))
   (set! x38936 (alloc r2 (* x38931 x38932)))
   (rfree r1)
   (set! x38929 x38936)
   (ralloc r3)
   (set! x38933 (alloc r3 3))
   (set! x38934 (alloc r3 4))
   (set! x38937 (alloc r2 (* x38933 x38934)))
   (rfree r3)
   (set! x38930 x38937)
   (set! x38935 (alloc r0 (* x38929 x38930)))
   (rfree r2)
   x38935)
> (flatten-begin (anf-code-gen (anf reg1-term)))
'(begin
   (ralloc r2)
   (set! x38944
     (begin
       (ralloc r1)
       (set! x38945
         (begin
           (set! x38938 (alloc r1 1))
           (begin
             (set! x38939 (alloc r1 2))
             (begin
               (set! x38940 (alloc r2 (* x38938 x38939)))
               (begin
                 (ralloc r3)
                 (set! x38946
                   (begin
                     (set! x38941 (alloc r3 3))
                     (begin
                       (set! x38942 (alloc r3 4))
                       (begin
                         (set! x38943 (alloc r2 (* x38941
x38942)))
                         (alloc r0 (* x38940 x38943))))))
                 (rfree r3)
                 x38946)))))
       (rfree r1)
       x38945))
   (rfree r2)
   x38944)
> (flatten-begin (abnf (anf-code-gen (anf reg1-term))))
'(begin
   (ralloc r2)
   (ralloc r1)
   (set! x38947 (alloc r1 1))
   (set! x38948 (alloc r1 2))
   (set! x38949 (alloc r2 (* x38947 x38948)))
   (ralloc r3)
   (set! x38950 (alloc r3 3))
   (set! x38951 (alloc r3 4))
   (set! x38952 (alloc r2 (* x38950 x38951)))
   (set! x38955 (alloc r0 (* x38949 x38952)))
   (rfree r3)
   (set! x38954 x38955)
   (rfree r1)
   (set! x38953 x38954)
   (rfree r2)
   x38953)
> (flatten-begin (anf-code-gen (anf intro-example)))
'(begin
   (set! x38957 f)
   (set! x38958 5)
   (set! x38956 (call x38957 x38958))
   (set! x x38956)
   (set! x38959 0)
   (set! x38960 6)
   (+ x38959 x38960))
> (flatten-begin (abnf (anf-code-gen (anf intro-example))))
'(begin
   (set! x38962 f)
```

```
    (set! x38963 5)
    (set! x38961 (call x38962 x38963))
    (set! x x38961)
    (set! x38964 0)
    (set! x38965 6)
    (+ x38964 x38965))
```

```
(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? lsymbol?)
```

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)
```