2. 雛形具体化

雛形具体化を用いるグラフ簡約器

2.1 雛形具体化

Implementation of Functional Programming Languageの11章、12章

- 関数プログラムは**式を評価**によって実行される
- 式は**グラフ**で表現される
- 評価は一連の簡約を行うことで実施される
- 簡約はグラフ中の簡約可能式(redex)を簡約しその結果で置き換える
- 評価は対象とする式が**正規形**(normal form)になれば終了
- 簡約系列な複数ありうるが停止したときには同じ正規形になる
- 正規形に到達する簡約系列があれば最外簡約戦略で必ず停止する

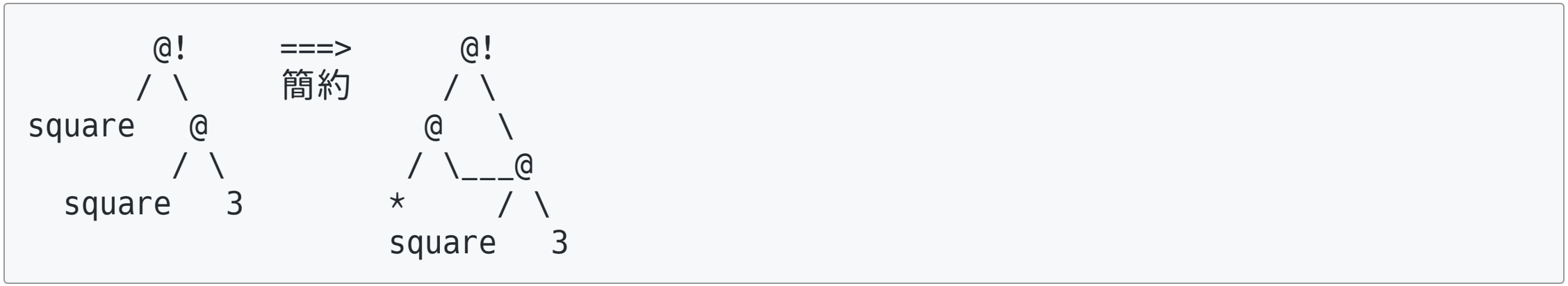
簡約例

```
main = square (square 3)
square x = x * x
```

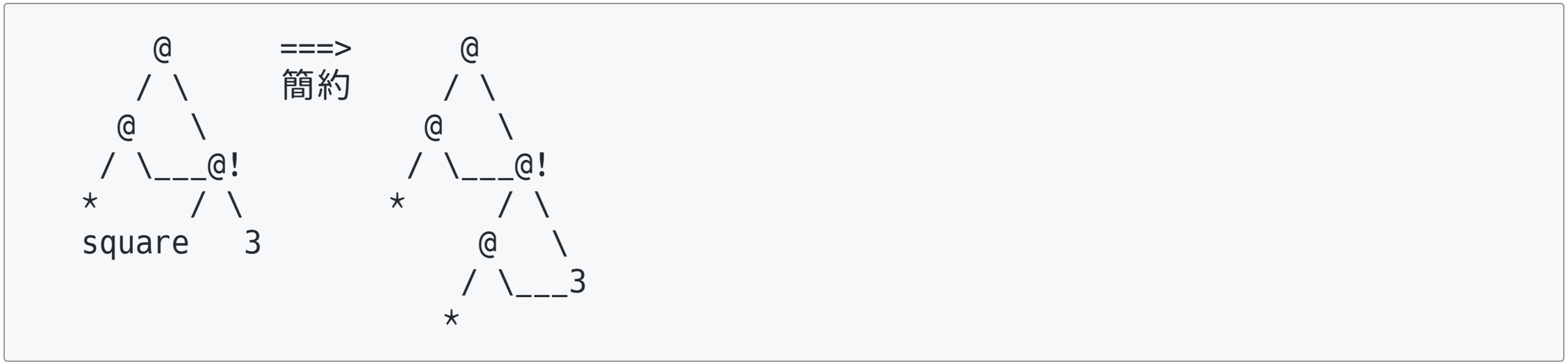
スーパーコンビネータ `main` には引数はなく、それ自身redexなので、ボディ部と置き換える。

```
main          ===>      @
              簡約     / \
                 square   @
                         / \
                   square   3
```

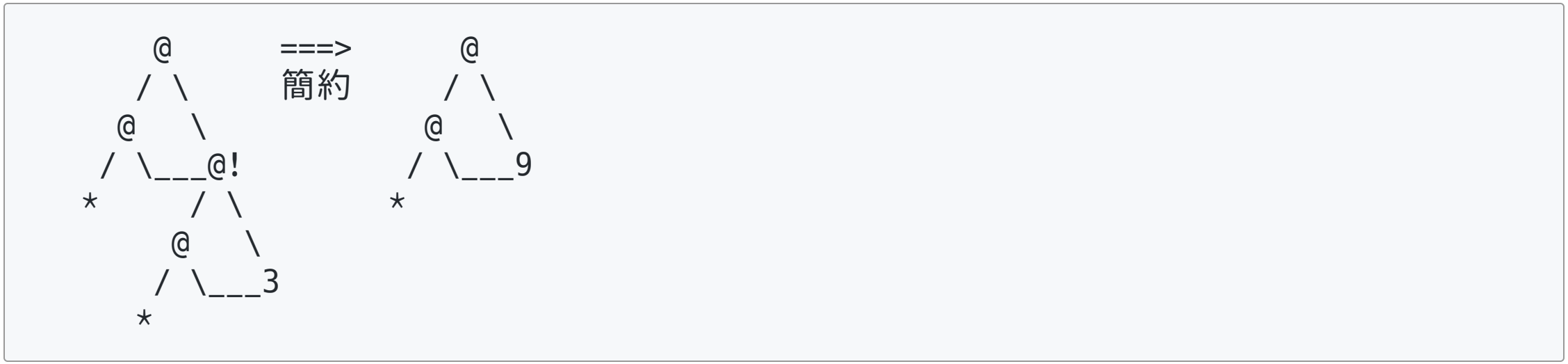
最外のredexは`square`の適用式。関数本体を具体化したものでredexを置き換える。仮引数の各出現を引数へのポインタで置き換える。



ここでredexは内側の `square` の適用式だけなので、これを簡約する。



ここで内側のかけ算が、唯一のredexとなるのでこれを簡約する。



最後の簡約は簡単。

2.1.2 簡約の3ステップ

以下を正規形が得られるまで繰り返す

1. 次に簡約するredexを見つける
2. そのredexを簡約する
3. redexのルートを結果で更新する

最外の関数適用が

- スーパーコンビネータ適用の場合
  - この適用は必ずredexなので簡約（β簡約）
- 組込みのプリミティブ適用の場合
  - 2つの引数が共に正規形ならこの適用はredexなので簡約（δ簡約）
  - そうでないなら、引数を正規形にして（この適用がredexになって）から簡約（δ簡約）

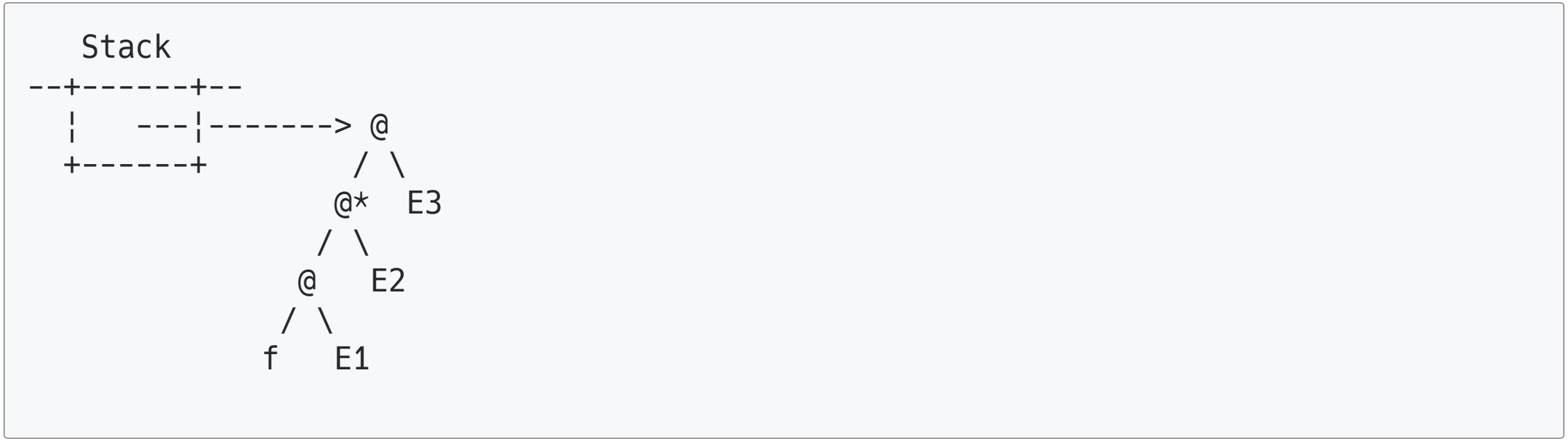
2.1.3 背骨を巻き戻して次のredexをみつける

ルートから適用ノードの左側を辿る

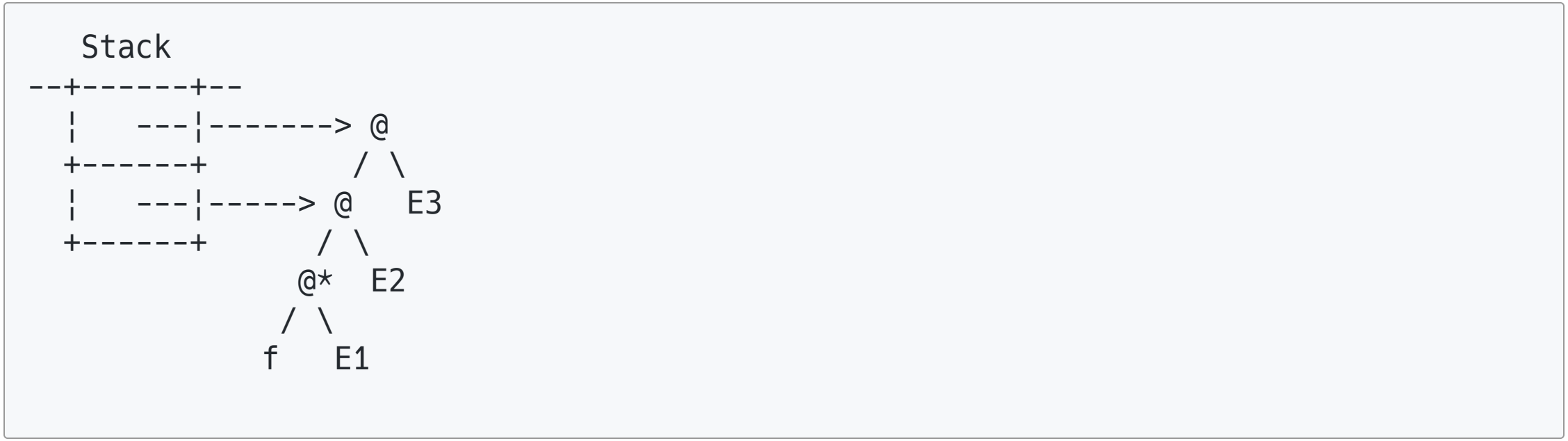
```
   Stack
--+------+--
                   @*  fizzbuzz:
    main:                  Main.hs
    source-dirs:           app/fizzbuzz/repl
    dependencies:
    - interaction-wrapper


                  / \
                 @   E3
                / \
               @   E2
              / \
             f   E1
```

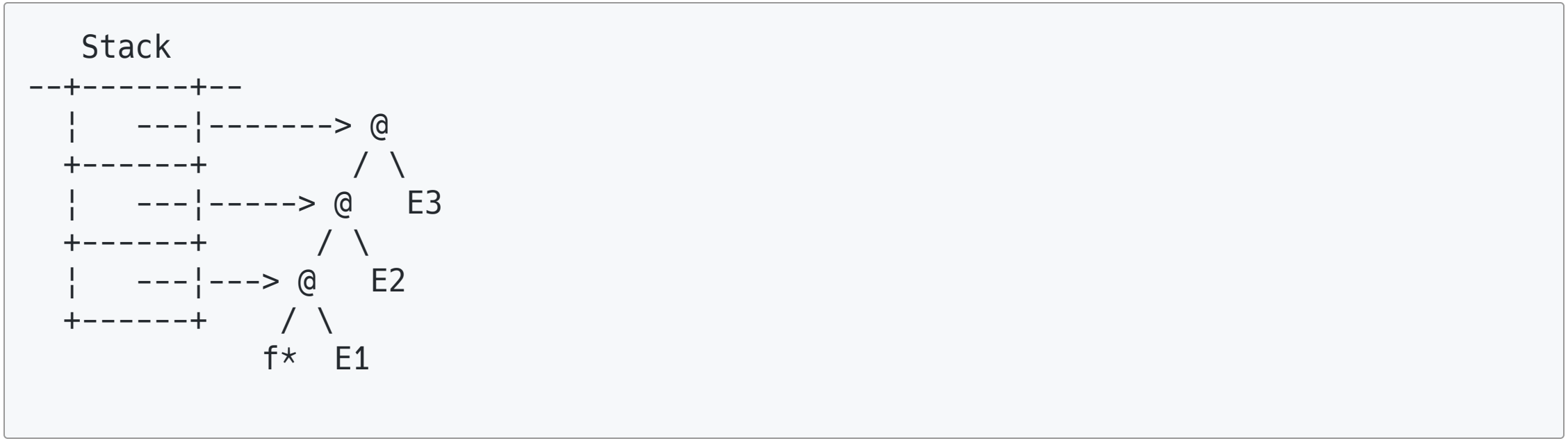
ルートの適用ノードをスタックに積んで、左へ降りる



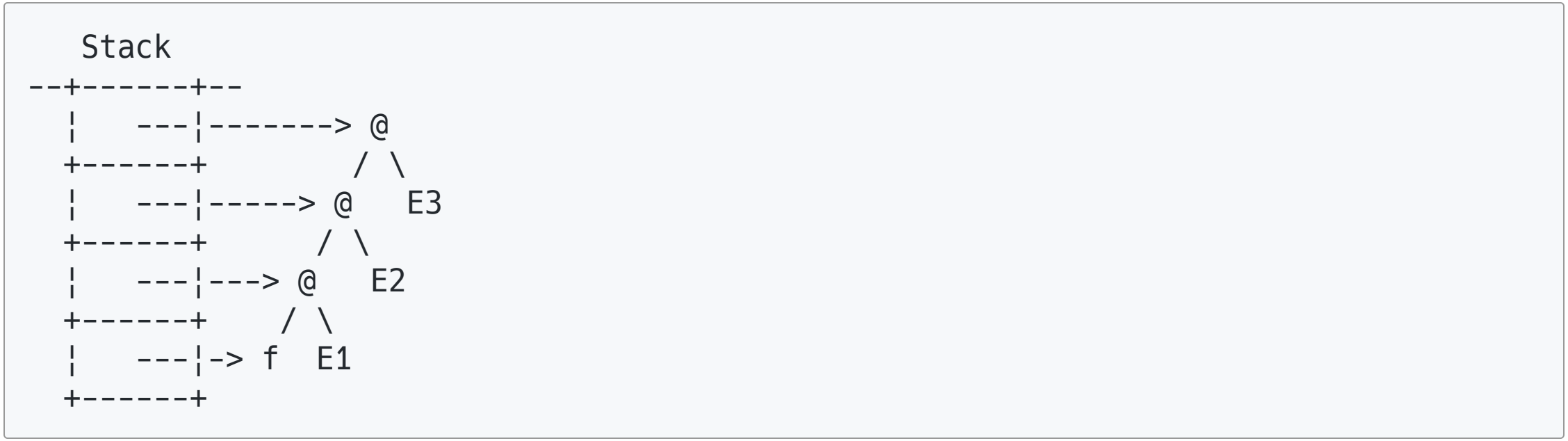
適用ノードをスタックに積んで、左に降りる



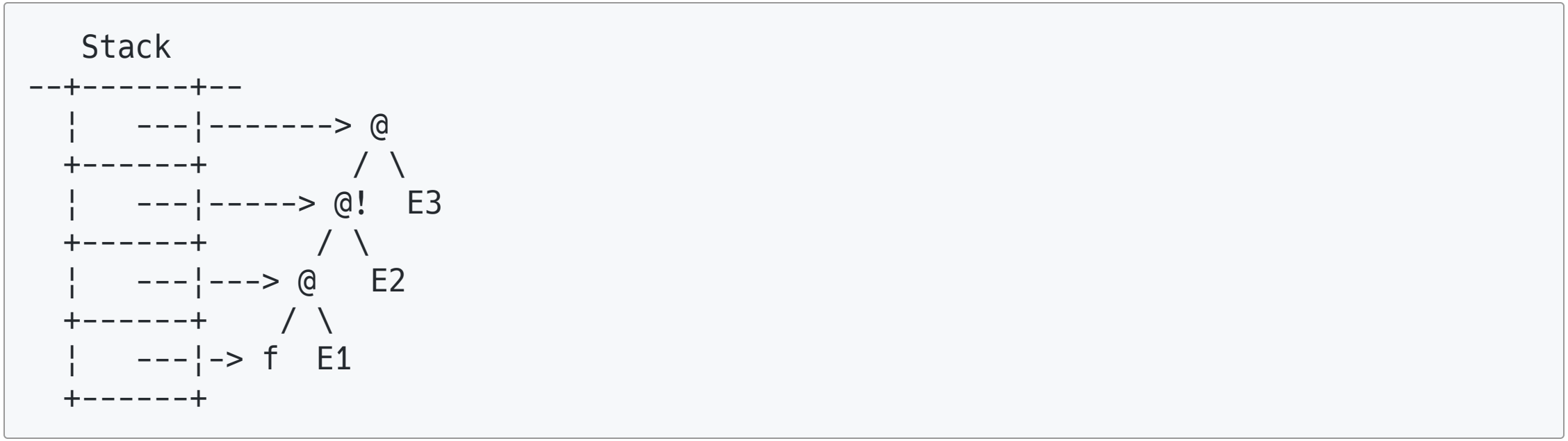
適用ノードをスタックに積んで、左に降りる



`f`がスーパーコンビネータの場合： `f` をスタックに積んで、 `f` のアリティ（ここでは2とする）を確認



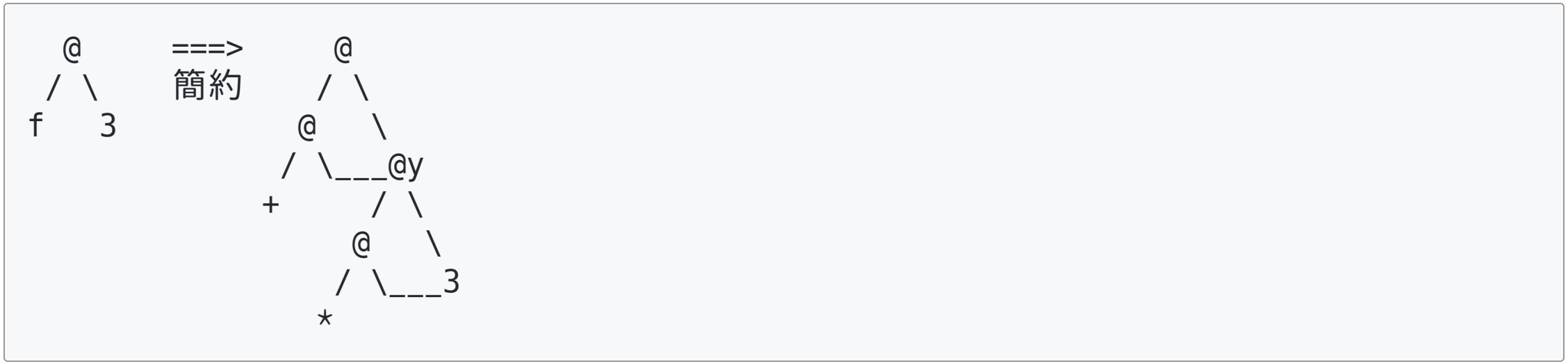
`f` がスーパーコンビネータの場合： アリティ（ここでは2とする）の分だけノードを上へもどったところが、最外の簡約可能項のルートノード



`f` がプリミティブ（算術2項演算子）の場合： 被演算子 `E1` および `E2` が共に正規形なら `!` のついたノードが最外の簡約可能項のルートノード。さもなければ、先に被演算子項を正規形にまで簡約する。

2.1.4 スーパーコンビネータの簡約可能項

- スーパーコンビネータの簡約可能項は、仮パラメータの出現位置を対応する実引数項へのポインタに置き換えたスーパーコンビネータ本体で置き換える
- 実引数項をコピーせず、ポインタを使い、共有していることに注意



2.1.5 更新

```
id x = x
f p = (id p) * p
main = f (sqrt 4)

f の簡約後

     @       ===>       @
    / \                 / \
   @   \               @   \
  / \   \             / \   \
 *   @   \           *   #   \
    / \___@             \___@
  id     / \             / \
       sqrt   4        sqrt   4
```

`#` は間接参照ノード

2.1.6 CAF

```
fac20 = factorial 20
```

スーパーコンビネータ `fac20` は CAF でかつ redex のルートなので、 `fac20` の簡約結果で上書きする

2.2 状態遷移システム

かけ算機械

```haskell
type MultState = (Int, Int, Int, Int) -- ^ (n, m, d, t)

evalMult :: MultState -> [MultState]
evalMult state = if multFinal state
  then [state]
  else state : evalMult (stepMult state)

stepMult (n, m, d, t)
  | d >  0 = (n,   m, d-1, t+1)  -- ^ 規則 1
  | d == 0 = (n, m-1,   n, t  )  -- ^ 規則 2
```

練習問題 2.1

かけ算マシンを手で走らせよ。初期状態 (2,3,0,0) からはじめ、各ステップで発火する規則を特定し、最終状態が (2,0,0,6) であることを確かめよ。

練習問題 2.2

状態列の不変条件とは、すべての状態で真となる述語である。 n および m の初期値 N および M と現在の n、m、d、t の値のとの関係を見つけよ。これにより、このかけ算機械が、かけ算を実行するものであることを証明せよ。すなわち、以下を示せ。

1. 初期状態で不変条件が成り立つ
2. ある状態で、不変条件が成り立てば、次の状態でも不変状態が成り立つ
3. 不変条件と停止条件が成り立てば、$t = N \times M$ である
4. このかけ算機械は停止する

状態遷移システムは、以下の点で便利

- 低レベルの詳細にわずらわされない程度に抽象的
- 隠れた詳細に依存していないことが確認できる程度に具体的
- 状態遷移システムは直截に実行可能なHaskellのコードに変換できる

練習問題 2.3

状態が最終状態であるかを判断する述語 `multFinal :: multState -> Bool` を定義し、初期状態 `(2,3,0,0)` からかけ算機械を走らせると、最終状態が `(2,0,0,6)` になることを示せ。