

## 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

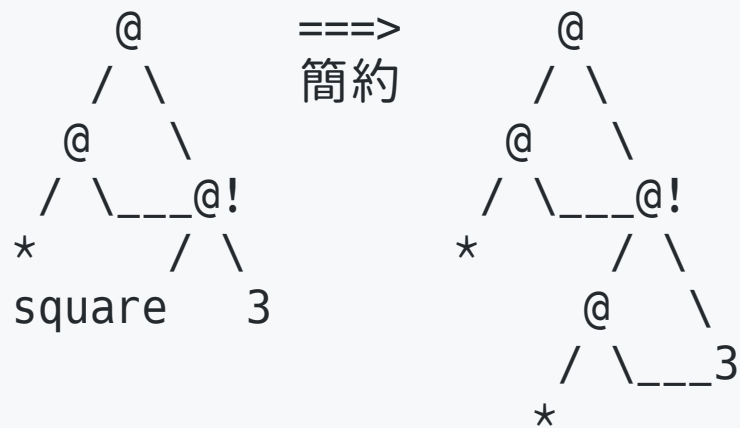
==>  
簡約

```
      @
     /\
square @
      /\
square 3
```

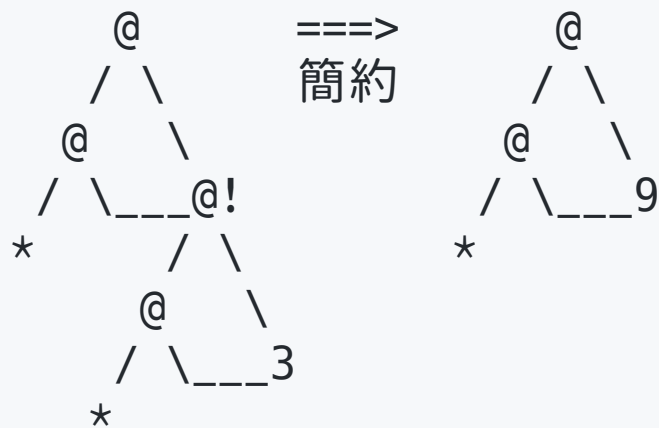
最外のredexは `square` の適用式。関数本体を具体化したものでredexを置き換える。仮引数の各出現を引数へのポインタで置き換える。



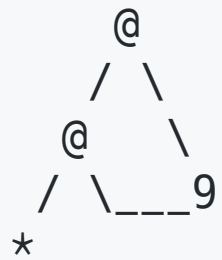
ここでredexは内側の `square` の適用式だけなので、これを簡約する。



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。



====>  
簡約

81

## 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

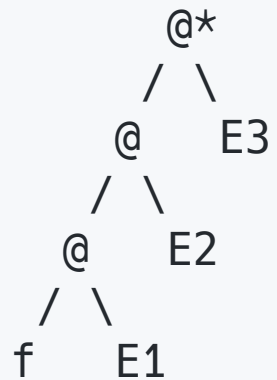
- スーパーコンビネータ適用の場合
  - この適用は必ずredexなので簡約 ( $\beta$ 簡約)
- 組込みのプリミティブ適用の場合
  - 2つの引数が共に正規形ならこの適用はredexなので簡約 ( $\delta$ 簡約)
  - そうでないなら、引数を正規形にして（この適用がredexになって）から簡約 ( $\delta$ 簡約)

## 2.1.3 背骨を巻き戻して次のredexを見つける

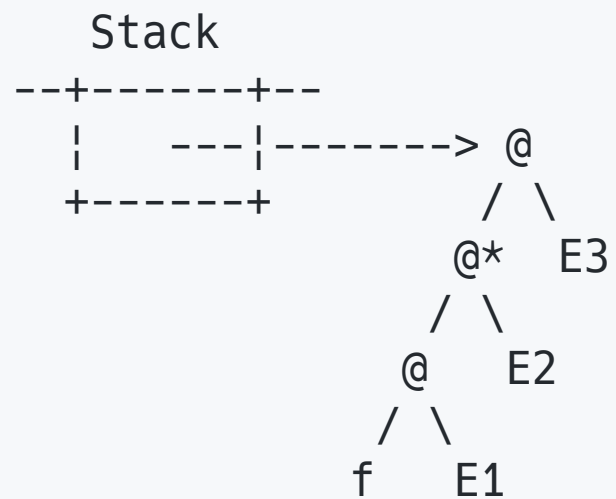
ルートから適用ノードの左側を辿る

Stack

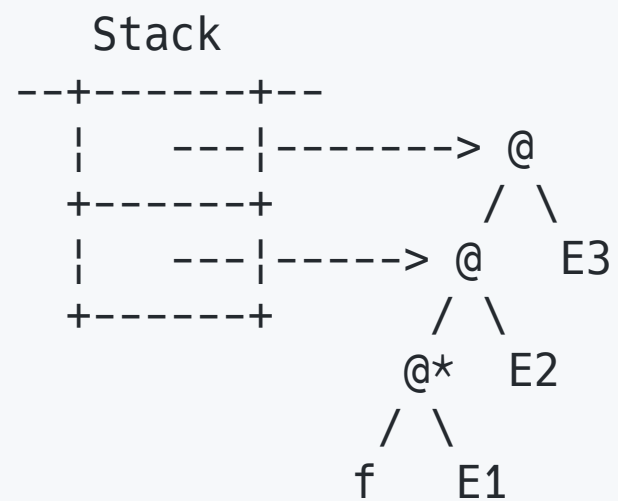
---+-----+---



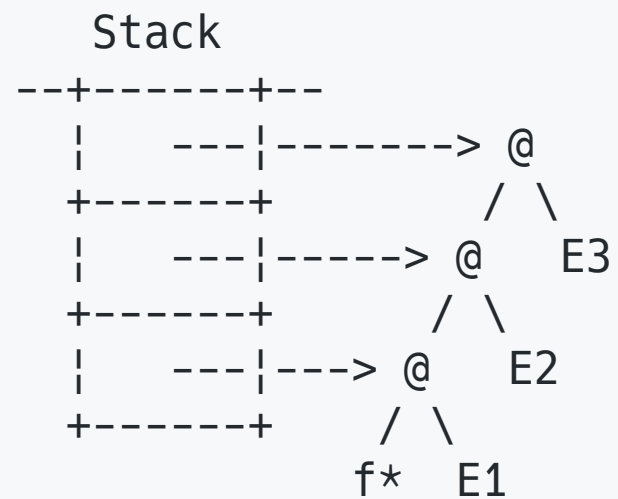
ルートの適用ノードをスタックに積んで、左へ降りる



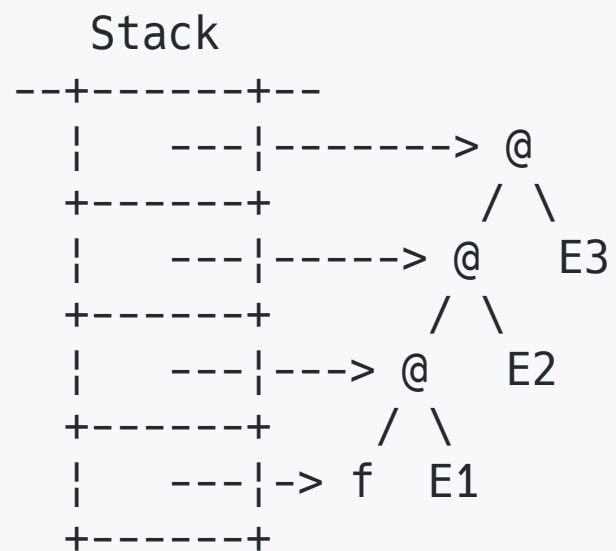
適用ノードをスタックに積んで、左に降りる



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

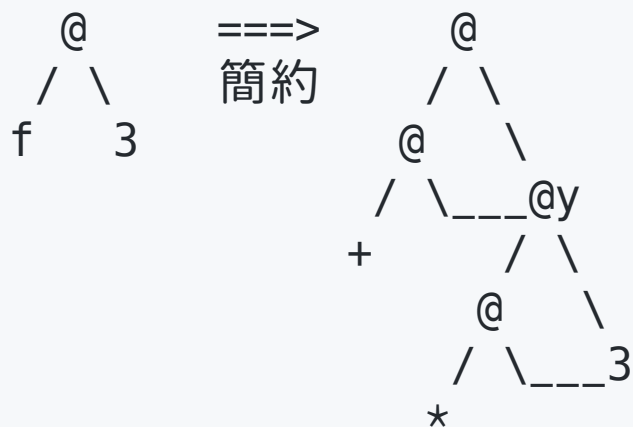
```
Stack
---+-----+---
|      ---|-----> @
+-----+      /\
|      ---|-----> @!  E3
+-----+      /\
|      ---|-----> @   E2
+-----+      /\
|      ---|-----> f   E1
+-----+
```

$f$  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。さもないければ、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

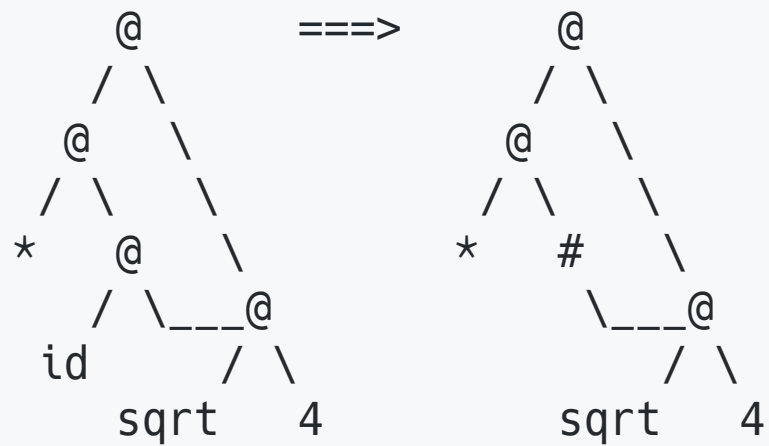
- スーパーコンビネータの簡約可能項は、仮パラメータの出現位置を対応する実引数項へのポインタに置き換えたスーパーコンビネータ本体で置き換える
- 実引数項をコピーせず、ポインタを使い、共有していることに注意



## 2.1.5 更新

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

f の簡約後



# 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

```
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。



## 2.3 Mark 1: 最小雛形具体化グラフ簡約器

マシン状態：  $(stack, dump, heap, globals)$  の4つ組

- *stack*: ヒープ上のノードを特定するアドレスのスタック
  - $a_1 : s$  という記法は、 $a_1$  がスタックトップ、 $s$  がのこりのスタックであることを示す
- *dump*: 正格なプリミティブ演算の引数評価に先立ち、スパインのスタックを記録
- *heap*: タグ付きノードを集めたもの
  - $h[a : node]$  という記法は、ヒープ  $h$  において、 $a$  はノード  $node$  のアドレスであることを示す
- *globals*: スーパーコンビネータおよびプリミティブを表すノードへのアドレス

## ノードの表現

```
data Node
  = NAp Addr Addr
  | NSupercomb Name [Name] CoreExpr
  | NNum Int
```

- `NAp  $a_1$   $a_2$`  はアドレス $a_1$ にあるノードのアドレス $a_2$ にあるノードへの適用を表す
- `NSupercomb  $args$   $body$`  は引数 $args$ と本体 $body$ をもつスーパーコンビネータを表す
- `NNum  $n$`  は整数 $n$ を表す

状態遷移規則

(2.1)

$$\Longrightarrow \begin{array}{cc} a : s & d \\ a_1 : a : s & d \end{array} \quad \begin{array}{cc} h[a : \mathbf{NAp} \ a_1 \ a_2] & f \\ h & f \end{array}$$

## 状態遷移規則

(2.2)

$$\begin{array}{c} a_0 : a_1 : \dots : a_n : s \quad d \quad h[a_0 : \text{NSupercomb } [x_1, \dots, x_n] \text{ body}] \quad f \\ \implies \quad \quad \quad a_r : s \quad d \quad h' \quad f \end{array}$$

ここで、 $(h', a_r) = \text{instantiate body } h \text{ } f[x_1 \mapsto a_1, \dots, x_n \mapsto a_n]$

関数 *instantiate* の引数は、

1. 具体化する式
2. ヒープ
3. 名前からヒープ上のアドレスへのグローバルマッピング  $f$  をスタックにある引数名からヒープアドレスへのマッピングで拡張したもの

返り値は、

- 新しいヒープと、新しく構成されたインスタンスの（ルートノードの）アドレス

## 2.3.2 実装の構造

```
run :: String -> String  
run = showResult . eval . compile . parse
```

1. `parse` はソースコード( `:: String` )を構文解析して `CoreProgram` を構成する関数（教科書とは意味を変更）

```
parse :: String -> CoreProgram
```

2. `compile` は `CoreProgram` を雛形具体化機械の初期状態に変換

```
compile :: CoreProgram -> TiState
```

3. `eval` はプログラム実行関数、初期状態から状態遷移を繰り返し、最終状態にまで遷移させ、結果は通過したすべての状態のリスト

```
eval :: TiState -> [TiState]
```

4. `showResult` は最終結果を整形して表示する

```
showResult :: [TiState] -> String
```

### 2.3.3 パーザ

Language モジュールをインポートする



## 2.3.4 コンパイラ

TiState

```
type TiState = (TiStack, TiDump, TiHeap, TiGlobals, TiStats)
```

# 1. TiStack スパインスタック、ヒープアドレス Addr のスタック

```
type TiStack = [Addr]
```

## 2. `TiDump` ダンプ、2.6節までは不要。ダミー

```
data TiDump = DummyTiDump  
initalTiDump :: TiDump  
initalTiDump = DummyTiDump
```

### 3. TiHeap は Node を格納するヒープ

```
type TiHeap = Heap Node
type Heap a = (Int, [Addr], [(Addr, a)])
```

Heap は使用アドレス数、未使用アドレス集合、アドレスと内容の2つ組のリスト、の3つを組にしたもの

4. `TiGlobal` はスーパーコンビネータ名とその定義が納められているヒープ上のアドレスの連想リスト

```
type TiGlobals = Assoc Name Addr
type Assoc a b = [(a, b)]
```

## 5. `TiStats` 実行時性能統計のためのデータ収集用、ひとまずステップカウンタ

```
type TiStats = Int

tiStatInitial :: TiStats
tiStatInitial = 0

tiStatIncSteps :: TiStats -> TiStats
tiStatIncSteps s = s + 1
tiStatGetSteps :: TiStats -> Int
tiStatGetSteps s = s

applyToStats :: (TiStats -> TiStats) -> TiState -> TiState
applyToStats f (stack, dump, heap, scDefs, stats)
  = (stack, dump, heap, scDefs, f stats)
```

compile

```
compile program
= (initialStack, initialTiDump, initialHeap, globals, tiStatInitial)
  where
    scDefs = program ++ preludeDefs ++ extraPreludeDefs
    (initialHeap, globals) = buildInitialHeap scDefs
    initialStack = [addressOfMain]
    addressOfMain = aLookup globals "main" (error "main is not defined")

extraPreludeDefs :: CoreProgram
extraPreludeDefs = []
```

`buildInitialHeap` プログラムから `NSupercomb` ノードを含むヒープと、スーパーコンビネータ名とヒープ上のアドレスの対応を示す連想リストを構成する。

```
buildInitialHeap :: [CoreScDefn] -> (TiHeap, TiGlobals)
buildInitialHeap scDefs = mapAccumL allocateSc hInitial scDefs

allocateSc :: TiHeap -> CoreScDefn -> (TiHeap, (Name, Addr))
allocateSc heap scDefn = case scDefn of
  (name, args, body) -> (heap', (name, addr))
    where
      (heap', addr) = hAlloc heap (NSupercomb name args body)
```



## 2.3.5 評価器

```
eval state = state : restStates
  where
    restStates
      | tiFinal state = []
      | otherwise     = eval nextState
    nextState = doAdmin (step state)

doAdmin :: TiState -> TiState
doAdmin state = applyToStats tiStatIncSteps state
```

## 最終状態の判定

- 計算はスタックに単一の数またはデータオブジェクトが含まれる状態になったときにのみ停止する

```
tiFinal :: TiState -> Bool
tiFinal state = case state of
  ([soleAddr], _, heap, _, _) -> isDataNode (hLookup heap soleAddr)
  ([], _, _, _, _)           -> error "Empty stack!"
  _                           -> False

isDataNode :: Node -> Bool
isDataNode node = case node of
  NNum _ -> True
  _      -> False
```

step ある状態から1つ次の状態への遷移

```
step :: TiState -> TiState
step state = case state of
  (stack, dump, heap, globals, stats) -> dispatch (hLookup heap (head stack))
  where
    dispatch (NNum n) = numStep state n
    dispatch (NAp a1 a2) = apStep state a1 a2
    dispatch (NSupercomb sc args body) = scStep state sc args body
```

NNum ノードと NAp ノードはやさしい

```
numStep :: TiState -> Int -> TiState
numStep state n = error "Number applied as a function"

apStep :: TiState -> Addr -> Addr -> TiState
apStep state a1 a2 = case state of
  (stack, dump, heap, globals, stats) -> (a1:stack, dump, heap, globals, stats)
```

スーパーコンビネータの適用：

1. 本体を具体化、引数名をスタックにあるアドレスと結びつける（規則2.2）
2. 簡約可能項のルートを含む引数をスタックから除去、簡約結果をスタックにプッシュ

(Mark 1 では更新は行わない)

```
scStep :: TiState -> Name -> [Name] -> CoreExpr -> TiState
scStep state scName argNames body = case state of
  (stack, dump, heap, globals, stats)
    -> (stack', dump, heap', globals, stats)
  where
    stack' = resultAddr : drop (length argNames + 1) stack
    (heap', resultAddr) = instantiate body heap env
    env = argBindings ++ globals
    argBindings = zip argNames (getargs heap stack)
```

`getargs` スタックにある引数名に対応するヒープ上の実引数ノードのアドレスを取得する

```
getargs :: TiHeap -> TiStack -> [Addr]
getargs heap stack = case stack of
  sc:stack' -> map getarg stack'
  where
    getarg addr = arg
      where
        NAp fun arg = hLookup heap addr
[] -> error "Empty stack"
```

## instantiate

```
instantiate :: CoreExpr      -- Body of suprercombinator
            -> TiHeap        -- Heap before instatiation
            -> Assoc Name Addr -- Association of names to address
            -> (TiHeap, Addr) -- Heap after instatiation, and address of root of instance

instantiate expr heap env = case expr of
  ENum n          -> hAlloc heap  (NNum n)
  EAp e1 e2       -> hAlloc heap2 (NAp a1 a2)
    where
      (heap1, a1) = instantiate e1 heap  env
      (heap2, a2) = instantiate e2 heap1 env
  EVar v          -> (heap, aLookup env v (error ("Undefined name " ++ show v)))
  EConstr tag arity -> instantiateConstr tag arity heap env
  ELet isrec defs body -> instantiateLet isrec defs body heap env
  ECase e alts      -> error "Can't instantiate case exprs"
  ELam vs e         -> error "Can't instantiate lambda abstractions"
```

## 2.3.6 結果の整形

showResults

```
showResults :: [TiState] -> String
showResults states
  = iDisplay (iConcat [ iLayn (map showState states)
                        , showStats (last states)
                        ])

```

showState

```
showState :: TiState -> IseqRep
showState (stack, dump, heap, globals, stats)
  = iConcat [ showStack heap stack, iNewline ]

```



## showStack

```
showStack :: TiHeap -> TiStack -> IseqRep
showStack heap stack
  = iConcat
    [ iStr "Stack ["
    , iIndent (iInterleave iNewline (map showStackItem stack))
    , iStr " ]"
    ]
  where
    showStackItem addr
      = iConcat [ showFWAddr addr, iStr ": "
                  , showStkNode heap (hLookup heap addr)
                  ]
```

## showStkNode

```
showStkNode :: TiHeap -> Node -> IseqRep
showStkNode heap (NAp funAddr argAddr)
  = iConcat [ iStr "NAp ", showFWAddr funAddr
             , iStr " ", showFWAddr argAddr, iStr " ("
             , showNode (hLookup heap argAddr), iStr ")"
             ]
showStkNode heap node = showNode node
```

## showNode

```
showNode :: Node -> IseqRep
showNode node = case node of
  NAp a1 a2 -> iConcat [ iStr "NAp ", showAddr a1
                        , iStr " ",   showAddr a2
                        ]
  NSupercomb name args body
    -> iStr ("NSupercomb " ++ name)
  NNum n    -> iStr "NNum " `iAppend` iNum n
```

showAddr

```
showAddr :: Addr -> IseqRep
showAddr addr = iStr (showaddr addr)

showFWAddr :: Addr -> IseqRep
showFWAddr addr = iStr (space (4 - length str) ++ str)
  where
    str = show addr
```

showStats

```
showStats :: TiState -> IseqRep
showStats (stack, dump, heap, globals, stats)
  = iConcat [ iNewline, iNewline, iStr "Total number of steps = "
             , iNum (tiStatGetSteps stats)
             ]
```

## 練習問題 2.4

ここまでの実装をテストせよ

```
testProg0, testProg1, testProg2 :: String
testProg0 = "main = S K K 3"
testProg1 = "main = S K K" -- wrong (not saturated)
testProg2 = "id = S K K ;\n\
            \main = twice twice twice id 3"

test :: String -> IO ()
test = putStrLn . showResults . eval . compile . parse
```

## 練習問題 2.5

`showState` を改変してヒープの内容をすべて表示するようにせよ。

```
showState :: TiState -> IseqRep
showState (stack, dump, heap, globals, stats)
  = iConcat [ showStack heap stack, iNewline
             , showHeap heap, iNewline
             ]

showHeap :: TiHeap -> IseqRep
showHeap heap = case heap of
  (_,_,contents) -> iConcat
    [ iStr "Heap ["
    , iIndent (iInterleave iNewline (map showHeapItem contents))
    , iStr " ]"
    ]
  where
    showHeapItem (addr, node)
      = iConcat [ showFWAddr addr, iStr ": "
                  , showNode node
                  ]
```

## 練習問題 2.6

`scStep` が引数が充満でない場合に適切なエラーを表示するようにせよ。

```
scStep :: TiState -> Name -> [Name] -> CoreExpr -> TiState
scStep state scName argNames body = case state of
  (stack, dump, heap, globals, stats)
    | length stack < length argNames + 1
      -> error "Too few arguments given"
    | otherwise
      -> (stack', dump, heap', globals, stats)
  where
    stack' = resultAddr : drop (length argNames + 1) stack
    (heap', resultAddr) = instantiate body heap env
    env = argBindings ++ globals
    argBindings = zip argNames (getargs heap stack)
```



## 練習問題 2.7

より多くの実行時情報を収集するようにせよ。

- プリミティブの簡約とスーパーコンビネータの簡約を分けて計数
- ヒープ操作（とくにアロケーション）の回数
- スタックの最大深さ

## 練習問題 2.8

`instantiate` に渡す環境 `env` は、

```
env = argBindings ++ globals
```

と定義されているが、これを

```
env = globals ++ argBindings
```

とするとどうなるか

## 練習問題 2.9

`eval` の定義を

```
eval state
| tiFinal state = [state]
| otherwise    = state : eval nextState
```

としたほうが、わかりやすそうに見えるが、この定義には欠点がある。それはどのようなものか。