G-machine

この教科書での、コンパイラベースの最初の実装は G-machine による

3.1 G-machine の紹介

雛形具体化機械の基本的な操作 `instantiate`

- スーパーコンビネータの本体のインスタンスを構成する
- 具体化のたびに `instantiate` が本体の式を再帰的に走査

G-machine のアイデア

- スーパーコンビネータの本体を命令列に変換する（コンパイル時）
- 命令列を実行し、スーパーコンビネータの本体が具体化（実行時）

3.1.1 例

ソースコード

```
f g x = K (g x)
```

G-code

```
Push 1
Push 1
Mkap
Pushglobal K
Mkap
Slide 3
Unwind
```

(a) 初期状態

```
--+------+--
  ¦   ---¦-----> @
  +------+      / \
  ¦   ---¦---> @   x
  +------+    / \
  ¦   ---¦-> f   g
  +------+
```

(b) Push 1 ： スタックトップから1飛した先のアプリケーションノードの引数側のポインタを Push

```
--+------+--
  ¦   ---¦-----> @
  +------+      / \
  ¦   ---¦---> @   x
  +------+    / \  ↑
  ¦   ---¦-> f   g ¦
  +------+         ¦
  ¦   ---¦---------+
  +------+
```

(c) Push 1 ： スタックトップから1飛した先のアプリケーションノードの引数側のポインタを Push

```
--+------+--
  ¦   ---¦-----> @
  +------+      / \
  ¦   ---¦---> @   x
  +------+    / \  ↑
  ¦   ---¦-> f   g ¦
  +------+       ↑ ¦
  ¦   ---¦-------¦-+
  +------+       ¦
  ¦   ---¦-------+
  +------+
```

(d) Mkap ： スタックトップ 2 つのポインタをポップして、関数適用ノードをアロケートし、それへのポインタを Push

```
--+------+--
  ¦   ---¦-----> @
  +------+      / \
  ¦   ---¦---> @   x
  +------+    / \  ↑
  ¦   ---¦-> f   g ¦
  +------+       ↑ ¦
  ¦   ---¦-> @   ¦ ¦
  +------+  / \  ¦ ¦
           ¦   --¦-+
           +-----+
```

(e) Pushglobal K ： スーパーコンビネータ K をスタックにプッシュ

```
--+------+--
  ¦   ---¦---------> @
  +------+          / \
  ¦   ---¦-------> @   x
  +------+        / \  ↑
  ¦   ---¦-----> f   g ¦
  +------+           ↑ ¦
  ¦   ---¦-----> @   ¦ ¦
  +------+      / \  ¦ ¦
  ¦   ---¦-> K ¦   --¦-+
  +------+     +-----+
```

(f) Mkap

```
--+------+--
  ¦   ---¦-------> @
  +------+        / \
  ¦   ---¦-----> @   x
  +------+      / \  ↑
  ¦   ---¦---> f   g ¦
  +------+         ↑ ¦
  ¦   ---¦-> @     ¦ ¦
  +------+  / \    ¦ ¦
           K   @   ¦ ¦
              / \  ¦ ¦
             ¦   --¦-+
             +-----+
```

(g) Slide 3 ： スタックトップより先に積んだポインタ 3 つを破棄

```
--+------+--
  ¦   ---¦-> @
  +------+  / \
           K   @
              / \
             g   x
```

3.2 雛形構築のためにコード列

- `instantiate` は再帰的に定義された式の木構造にそって、式を再帰的に走査して具体化する
- 線形の命令列をコンパイルして式の具体化を行いたい

3.2.1 算術式の後置評価

```
data AExpr = Num Int
           ¦ Plus AExpr AExpr
           ¦ Mult AExpr AExpr
```

意味は `aInterpret :: AExpr -> Int` で与えられる

```
aInterpret :: AExpr -> Int
aInterpret (Num n)    = n
aInterpret (Plus e1 e2) = aInterpret e1 + aInterpret e2
aInterpret (Mult e1 e2) = aInterpret e1 * aInterpret e2
```

この算術式を後置算術演算子列にコンパイルし、スタックをつかって評価

たとえば $2 + 3 \times 4$ は `[INum 2, INum 3, INum 4, IMult, IPlus]` にコンパイルする

```
data AInstruction = INum Int
                  | IPlus
                  | IMult
```

(3.1)

$$\boxed{\begin{array}{lrr} & [\,] & [n] \\ \Longrightarrow & & n \end{array}}$$

(3.2)

$$\boxed{\begin{array}{lrr} & \texttt{INum}\ n : i & ns \\ \Longrightarrow & i & n : ns \end{array}}$$

(3.3)

$$\boxed{\begin{array}{rrr} & \texttt{IPlus} : i & n_0 : n_1 : ns \\ \Longrightarrow & i & (n_0 + n_1) : ns \end{array}}$$

(3.4)

$$\boxed{\begin{array}{rrr} & \texttt{IMult} : i & n_0 : n_1 : ns \\ \Longrightarrow & i & (n_0 \times n_1) : ns \end{array}}$$

```haskell
aEval :: ([AInstruction], [Int]) -> Int
aEval ([],       [n]) = n
aEval (INum n:is, s) = aEval (is, n : s)
aEval (IPlus :is, n0:n1:s) = aEval (is, (n0 + n1) : s)
aEval (IMult :is, n0:n1:s) = aEval (is, (n0 * n1) : s)
```

```haskell
aCompile :: AExpr -> [AInstruction]
aCompile (Num n) = [INum n]
aCompile (Plus e1 e2) = aCompile e1 ++ aCompile e2 ++ [IPlus]
aCompile (Mult e1 e2) = aCompile e1 ++ aCompile e2 ++ [IMult]
```

練習問題 3.1

任意の e :: AExpr について以下が成立つことを構造帰納法により証明せよ

```
aInterprete e = aEval (aCompile e, [])
```

練習問題 3.2

`let` 式が扱えるように `aInterpret` 、 `aCompile` 、 `aEval` を拡張せよ。このとき

```
aInterpret e = aEval (aCompile e, [])
```

であることを証明せよ

言語をさらに複雑な式、たとえば `letrec` 式を扱えるように拡張できるかできるか？拡張できるなら、実装が正しいことを証明できるか。