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

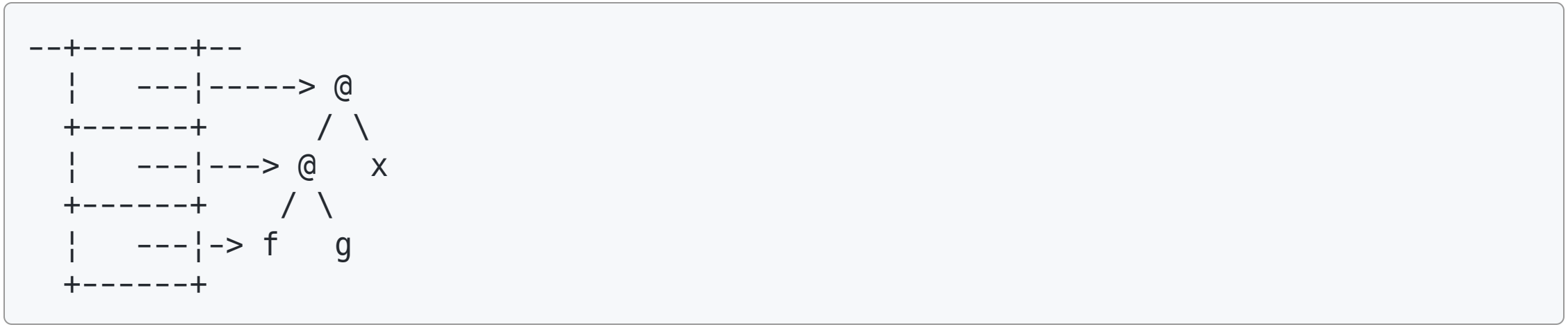
ソースコード

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

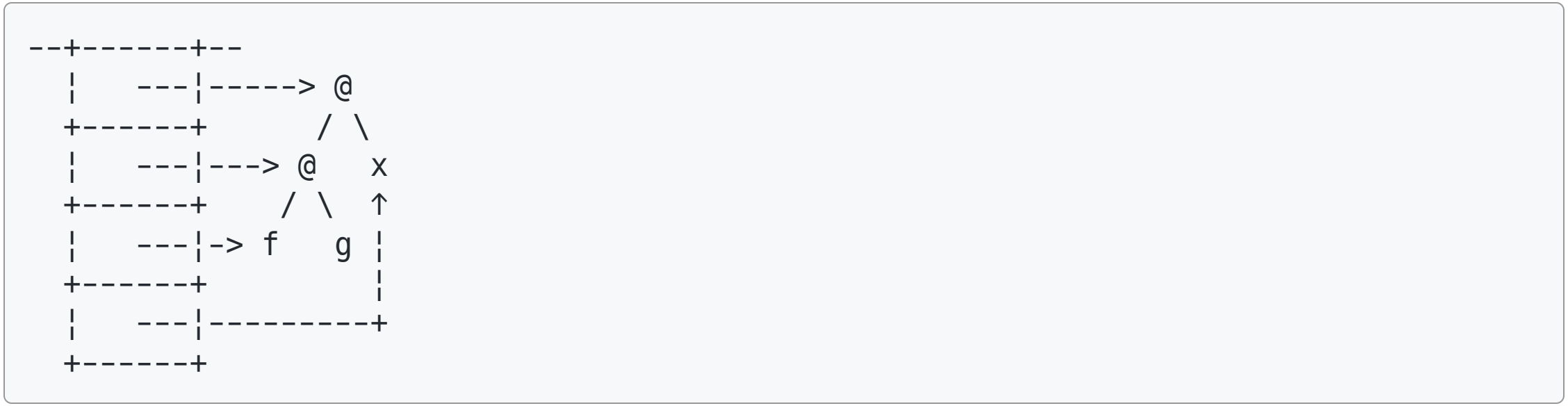
G-code

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

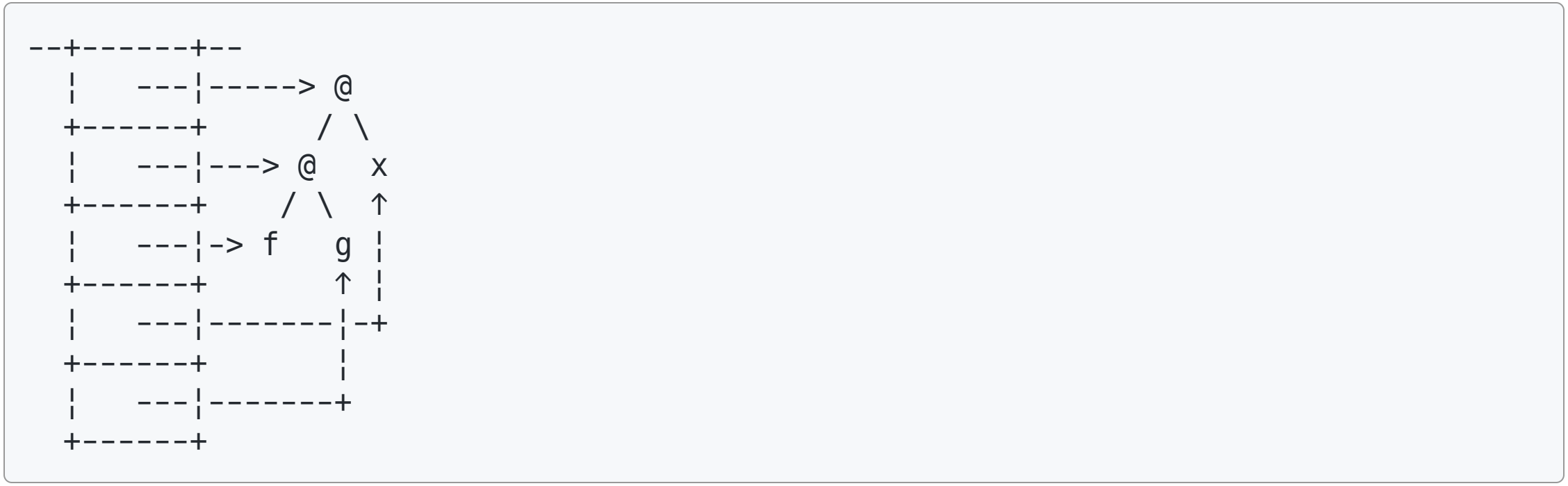
(a) 初期状態



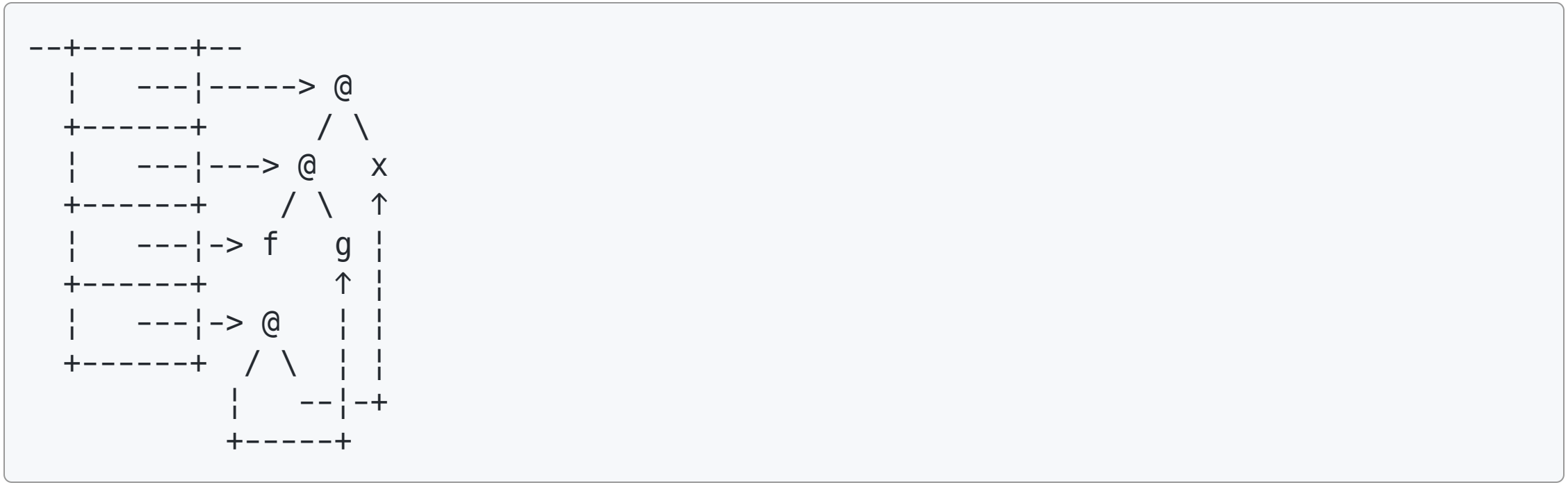
(b) Push 1 ： スタックトップから1飛した先のアプリケーションノードの引数側のポインタを Push



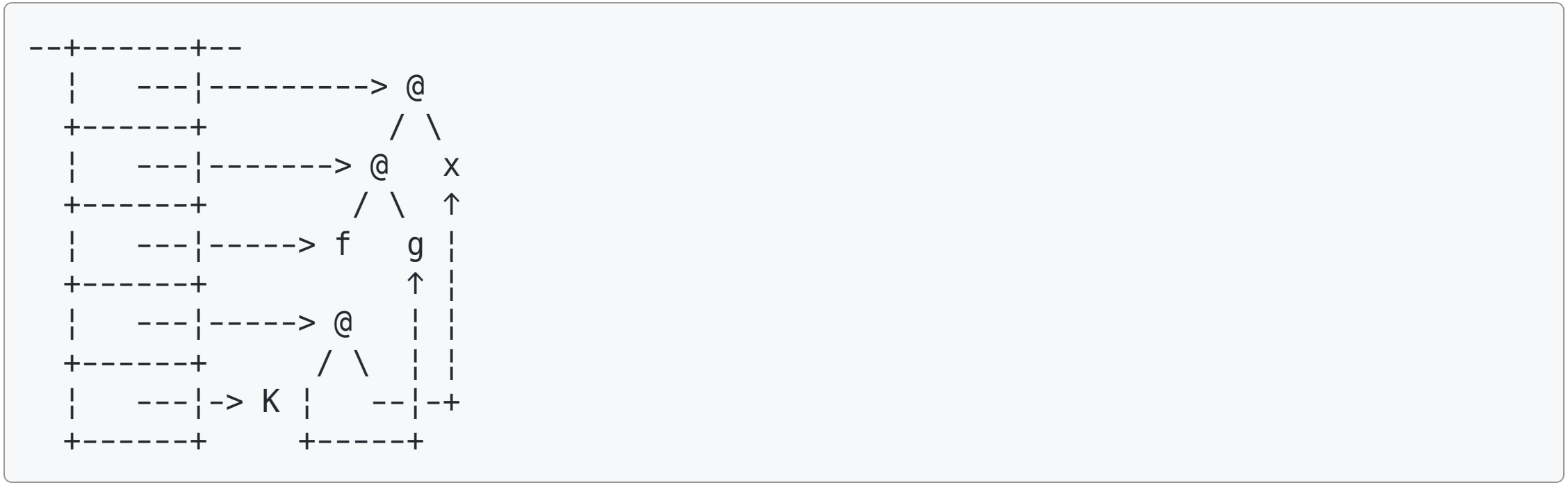
(c) Push 1 ： スタックトップから1飛した先のアプリケーションノードの引数側のポインタを Push



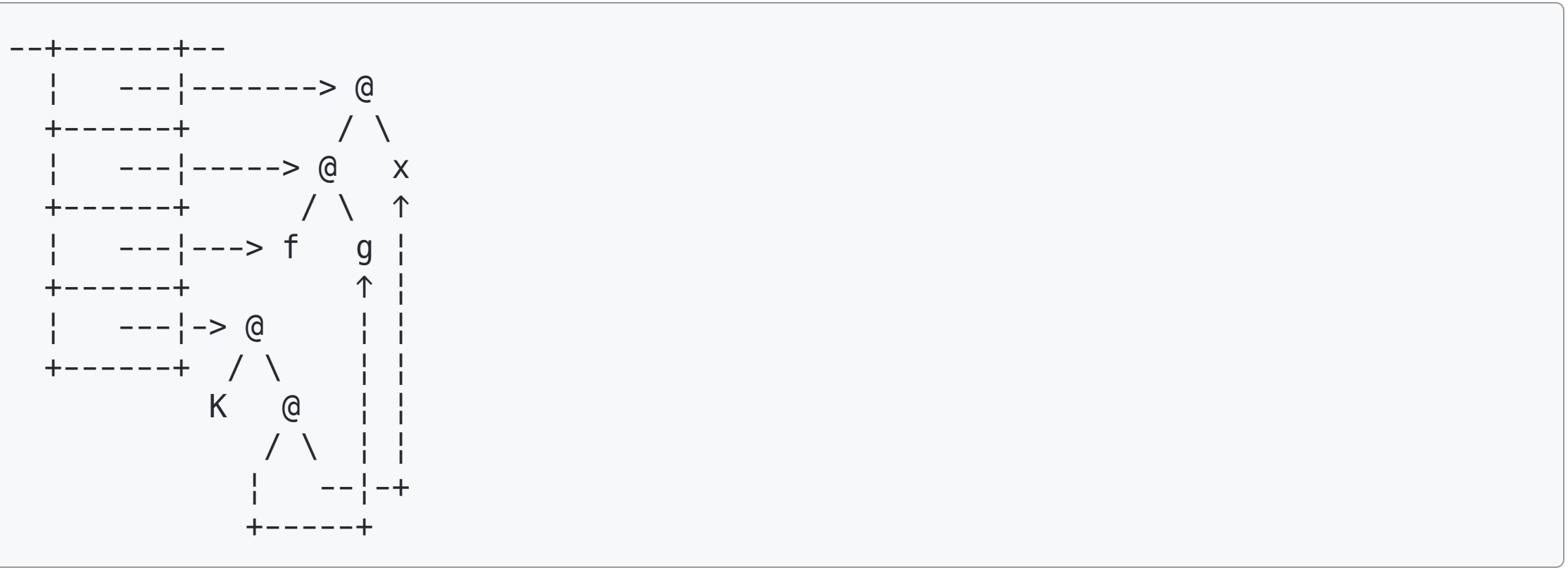
(d) Mkap ： スタックトップ 2 つのポインタをポップして、関数適用ノードをアロケートし、それへのポインタを Push



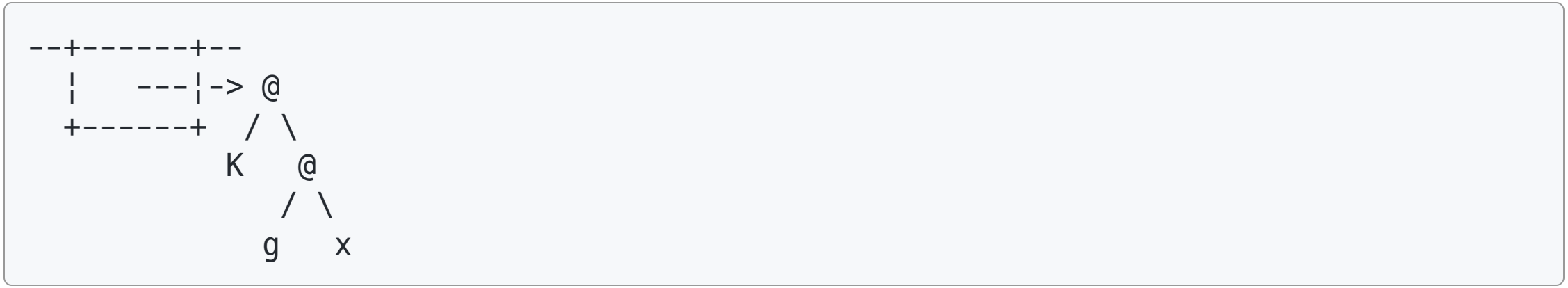
(e) Pushglobal K ： スーパーコンビネータ K をスタックにプッシュ



(f) Mkap



(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



3.1.2 その先の最適化

3.2 雛形構成のためのコード列

- `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)

$$\begin{array}{rcr} & [] & [n] \\ \Longrightarrow & & n \end{array}$$

(3.2)

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

(3.3)

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

(3.4)

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

```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)
```

```
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。

3.2.2 後置コードを使ってグラフを構成する

後置評価の技法によりスーパーコンビネータの本体を構成できる。

- スタックに積むのは具体化された式の各部へのポインタ
- 雛形を構成する命令にはヒープ上にノードをアロケートするという副作用がある

スタックに関しては、雛形具体化機械で用意したものでよいが、

> スタック操作のたびに、変数名に対応するアドレスのスタック上での位置が変化することに注意が必要 式をコンパイルする際その変化を追跡する必要がある

ここではスタック上の項にアクセスするのにスタックトップからのオフセットを使う。したがって、push、pop のたびに各項のオフセットが1増減する。

3.2.3 具体化のあとに起こること

スーパーコンビネーター本体のインスタンス化が完了したら、スタックを整理し、評価プロセスの継続を手配する必要があります。 n 個の引数を持つスーパーコンビネータの後置シーケンスの評価が完了すると、スタックは次の形式になります。

・一番上には、新しくインスタンス化された本体のヒープ内のアドレスがあります。・次に、n + 1 ポインターがあります。 これらから、インスタンス化プロセスで使用される引数にアクセスできます。・n +1 個のポインターの最後は、インスタンス化したばかりの式のルートを指します。

これを図 3.2 に示します。

スライド命令を使用して、redex を新しくインスタンス化された本体に置き換え、スタックから n 個の項目をポップする必要があります。 次のスーパーコンビネーターを見つけるには、Unwind 命令を使用して、再び巻き戻しを開始する必要があります。 後置演算子シーケンスに整理と巻き戻しを行う操作を追加することで、テンプレート インスタンス化子を Mark 1 G-machine に変換しました。

3.3 Mark 1: ミニマル G-machine

- 更新なし
- 算術演算なし

3.3.1 全体構造

`Gmachine.Mark1.Machine` モジュール

3.3.2 データ型定義

- `Gmachine.Mark1.State` モジュール
- `Gmachine.Mark1.Code` モジュール
- `Gmachine.Mark1.Node` モジュール

3.3.3 評価器

`Gmachine.Mark1.Machine` モジュール

3.3.4 プログラムのコンパイル

`Gmachine.Mark1.Machine` モジュール

練習問題 3.3

プレリュードにある S コンビネータに対応する変換列を書け。コンパイラとGmachineを走らせて、付録Bの単純なプログラムを実行して最終結果を確認せよ。

結果の表示

`Gmachine.Mark1.PPrint` モジュール

3.3.6 Mark1 Gmachine の改良

練習問題 3.4

`main = S K K 3` というプログラムを走らせたとき、ステップはどうなるか。雛形マシンと、どのくらい違うか。Gマシンと雛形マシンとをステップ数で比較するのは公平だと思うか。

練習問題 3.5

付録B にある他のプログラムも試してみよ。まだ、算術演算には対応していないことに注意せよ。

練習問題 3.6

`pushint` を実装するとき、`pushglobal` で使った方法がそのまま使える。つまり、`Pushint 2` を最初に実行したときに、`"2"` をヒープ内ノード `NNum 2` のアドレスに結びつけるのである。

n というグローバル変数があれば、そのノードを再利用する

$$\begin{array}{rcccccc} & \mathbf{Pushint}\ n : i & & s & h & m[n : a] \\ \Longrightarrow & & i & a : s & h & m \end{array}$$

なければ、新しいノードを生成して登録する

$$\begin{array}{rcccccc} & \mathbf{Pushint}\ n : i & & s & h & m \\ \Longrightarrow & & i & a : s & h[a : \mathbf{NNum}\ n] & m[n : a] \end{array}$$

この変更を反映した `pushint` を実装せよ。

3.4 Mark 2：Lazyにする

Mark 1 は、巻き戻しに先立って、元の式のルートノードを上書きしないので、Lazy にはなっていない。 Mark 2 では、元の式のルートノードを具体化したスーパーコンビネータの本体を指す間接参照ノードで上書きする。 こうすることで、前回、簡約可能項を簡約して得た値を記憶し、再計算を排除できる。

3.4.1 データ構造

`Instruction`

- `Update Int` と `Pop Int` を追加
- `Slide Int` を削除

練習問題 3.7

`showInstruction` を対応させよ

`Node`

- `NInd Addr` を追加

練習問題 3.8

`showNode` を対応させよ

3.4.2 評価器

`Update` n はスタック上の n+1 番目の項目をスタックトップの項目への間接参照で上書きする

(3.15)

$$\begin{array}{rccccc} & \mathtt{Update}\ n : i & a : a_0 : \ldots : a_n : s & h & & m \\ \Longrightarrow & i & a_0 : \ldots : a_n : s & h[a_n : \mathtt{NInd}\ a] & & m \end{array}$$

`Pop` n は単純にn個の項目をスタックから取り除く

(3.16)

$$\begin{aligned} & \mathbf{Pop}\, n : i \quad a_0 : \ldots : a_n : s \quad h \quad m \\ \Longrightarrow \quad & i \qquad\qquad\qquad\qquad\quad s \quad h \quad m \end{aligned}$$

スタックトップの項目が間接参照ノードであった場合は、`Unwind` は、間接参照ノードが指す先に置き換える

(3.17)

$$\begin{array}{rcccc} & [\mathtt{Unwind}] & a_0 : s & h[a_0 : \mathtt{NInd}\ a] & m \\ \Longrightarrow & [\mathtt{Unwind}] & a : s & h & m \end{array}$$

練習問題 3.9

Mark 1 の `dispatch` 関数を新しく導入したインストラクションと遷移規則に対応できるよう変更せよ。