Implementing Functional Languages: a tutorial

Simon L Peyton Jones and David R Lester

2025年12月20日

目 次

序文

この本は、遅延グラフ簡約を使用して非正格な関数型言語の実装を理解するための実用的なアプローチを提供します。この本は、読者がいくつかの重要なコンパイラを開発、変更、および実験するのを支援することにより、関数型言語の実装を「生きた」ものにするのに役立つ実用的なラボワーク資料のソースとなることを目的としています。

この本の珍しい側面は、それが読むだけでなく実行されることを意図しているということです。各実装手法の抽象的な説明を提示するだけでなく、主要な各メソッドの完全に機能するプロトタイプのコードを提示し、一連の改善を行います。

本書の概要

この本の主な内容は、コア言語と呼ばれる小さな関数型言語の一連の実装です。コア言語は可能な限り小さく設計されているため、実装は簡単ですが、効率を損なうことなく最新の非厳密な関数型言語をコア言語に変換できるほどリッチです。これについては、第 1 章で詳しく説明します。この章では、コア言語用のパーサとプリティプリンタも開発しています。

Appendix B には、本書全体でテストプログラムとして使用するためのコア言語プログラムのセレクションが含まれています。

本書の本文は、コア言語の 4 つの異なる実装で構成されています。

- 第 2 章では、**テンプレートのインスタンス化**に基づいた最も直接的な実装について説明します。
- 第 3 章では、**G マシン**を紹介し、プログラムを一連の命令 (G コード) にコンパイルして、さらにマシンコードに変換する方法を示します。
- 第 4 章では、評価モデルが G マシンの評価モデルとは大きく異なる抽象マシンである **Three Instruction Machine**(TIM) についても同じ演習を繰り返します。TIM は G マシンよりも最近開発されたため、他

の文献はほとんどありません。したがって、第 4 章には、G マシンに与えられたものよりもかなり詳細な TIM の評価モデルの開発が含まれています。

- 後に、第 5 章では、**並列 G マシン**用の関数型プログラムをコンパイルする方法を示すことにより、新しい次元を追加します。

これらの実装のそれぞれについて、**コンパイラ**と**マシンインタープリタ**の 2 つの主要部分について説明します。コンパイラはコア言語プログラムを受け取り、それをマシンインタープリタによる実行に適した形式に変換します。

マシンインタープリタは、コンパイルされたプログラムの実行をシミュレートします。いずれの場合も、インタープリタは**状態遷移システム**としてモデル化されているため、マシンインタープリタと「実際の」実装の間には非常に明確な関係があります。図 1 は、実装の構造をまとめたものです。

コア言語が制限されている重要な方法の 1 つは、ローカル関数定義がないことです。**ラムダリフティング**と呼ばれるよく知られた変換があります。これは、ローカル関数定義をグローバル関数定義に変換し、ローカル関数定義を自由に記述して後で変換できるようにします。第 6 章では、適切なラムダリフターを開発します。この章は、標準的な資料の単なる再提示ではありません。**完全な遅延評価**は、以前はラムダリフティングと切り離せないと見なされていた関数型プログラムの特性です。第 6 章では、それらが実際にはまったく異なることを示し、ラムダリフティングとは別のパスで完全な遅延評価を実装する方法を示します。

この本全体を通して、Appendix A で定義されている多くのユーティリティ関数とデータ型を使用しています。

一部のセクションと演習はもう少し進んでおり、大きな損失なしに省略されています。それらは短剣で識別されます、したがって：†

プロトタイピング言語

実装を作成するためにどの言語を使用するかという問題が発生します。既存の関数型言語である Miranda [1] を使用することを選択しました。関数型言語の主な用途の 1 つは、ラピッドプロトタイピングです。これにより、管理の詳細にとらわれることなく、プロトタイプの基本的な側面を表現できるようになります。この本がこの主張を立証するための大きな例として役立つこ

[1] Miranda は、Research SoftwareLtd の商標です。

とを願っています。さらに、この本を通して作業することは、実質的な関数型プログラムを書くことの有用な経験を提供するはずです。

この本は関数型プログラミングの入門書ではありません。すでに関数型言語でプログラミングを行っていることを前提としています。(関数型プログラミングの適切な紹介には、[Bird and Wadler 1988] および [Holyer 1991] が含まれます。) それにもかかわらず、この本で開発されたプログラムは非常に充実しており、明確でモジュール式の関数型プログラムを作成する能力を伸ばし、開発するためのロールモデルとして役立つことを願っています。

Miranda コードは、「逆コメント規則」を使用してタイプライターファウントで記述されています。たとえば、リストの長さを取る関数の定義は次のとおりです。

```
> length [] = 0
> length (x : xs) = 1 + length xs
```

左マージンの

```
>
```

マークは、これが実行可能な Miranda コードの行であることを示します。

本書がカバーしていないこと

この本では、関数型コンパイラの「バックエンド」にのみ焦点を当てています。Miranda などの本格的な関数型言語で書かれたプログラムをコア言語に翻訳する方法や、そのようなプログラムの型チェック方法については議論しません。

全体の開発は非形式です。コアプログラムの意味とその実装が同等であることを正式に証明するのは良いことですが、これは非常に難しい作業です。確かに、私たちが知っているそのような同等性の唯一の完全な証明は [Lester 1988] です。

「The implementation of functional programming languages」との関係

私たちの 1 人による以前の本 [Peyton Jones 1987] は、これと同様の資料をカバーしていますが、あまり実用的ではありません。私たちの意図は、学

生が他の本を参照せずに、現在の本だけを使用して関数型言語の実装に関するコースをたどることができるようにすることです。

この本の範囲はやや控えめで、[Peyton Jones 1987] のパート 2 とパート 3 に対応しています。後者のパート 1 では、高レベルの関数型言語をコア言語に変換する方法について説明していますが、ここではまったく取り上げていません。

ソースコードの取得

間違いを見つけたら

第1章　コア言語

すべての実装は、単純なコア言語で記述されたプログラムを使用して実行します。コア言語は非常に貧弱であり、大きなプログラムを書きたくないでしょう。それでも、表現力や効率を損なうことなく、豊富な関数型言語 (Miranda など) のプログラムをコア言語に翻訳できるように慎重に選択されています。したがって、コア言語は、高水準言語構造に関係するコンパイラの「フロントエンド」と、さまざまな異なる方法でコア言語を実装することに関係する「バックエンド」との間のクリーンなインターフェースとして機能します。

まず、コア言語の非形式的な紹介から始めます (セクション 1.1)。これに続いて、次のようにしてコア言語をより形式的に定義します。

- その構文 (セクション 1.2)。
- Miranda のデータ型 coreProgram と coreExpr は、それぞれコア言語のプログラムと式です (セクション 1.3)。これらは、ビルドするコンパイラの入力データ型として機能します。
- コア言語関数の小さな標準プレリュードの定義。これは、すべてのコアプログラムで利用できるようになります (セクション 1.4)。
- コア言語プログラムを文字列に変換するプリティプリンタ。印刷すると、プログラムのフォーマットされたバージョンになります (セクション 1.5)。
- 文字列を解析してコア言語プログラムを生成するパーサ (セクション 1.6)。

この章には 2 番めの目的があります。関数型言語の実装に関与する前に、本全体で使用する Miranda の機能の多くを紹介して使用します。

1.1　コア言語の概要

これは、42 と評価されるコアプログラム[1]の例です。

[1]コアプログラムにはタイプライターフォントを使用しますが、実行可能な Miranda コードを区別する最初の > 記号はありません。

```
main = double 21
double x = x + x
```

コアプログラムは、主要なものを含む一連の**スーパーコンビネータ定義**で構成されています。プログラムを実行するために、main を評価します。スーパーコンビネータは、double の定義などの機能を定義できます。double は、1 つの引数 x の関数であり、引数の 2 倍を返します。プログラムは、関数の引数にパターンマッチングが許可されていないことを除いて、Miranda スクリプトのトップレベルと非常によく似ています。パターンマッチングは、別のコア言語構造である case 式によって実行されます。これについては、以下で説明します。各スーパーコンビネータは、引数がすべて単純な変数である単一の等式によって定義されます。

すべてのスーパーコンビネータに引数があるわけではないことに注意してください。main など一部のスーパーコンビネータは引数を取りません。引数のないスーパーコンビネータは、**定数適用形式**または CAF とも呼ばれ、後で説明するように、実装では特別な処理が必要になることがよくあります。

1.1.1　ローカル定義

スーパーコンビネータは、コア言語の let 構造を使用して、ローカル定義を持つことができます。

```
main = quadruple 20 ;
quadruple x = let twice_x = x+x
              in twice_x + twice_x
```

ここで、twice_x は quadruple の本体内でローカルに x+x と定義され、quadruple は twice_x + twice_x を返します。Miranda の where 句と同様に、ローカル定義は、中間値に名前を付けることと、同じ値を 2 回再計算することを節約することの両方に役立ちます。プログラマは、quadruple で 2 つの加算だけが実行されることを合理的に期待できます。

let 式は**非再帰的**です。再帰的定義の場合、コア言語は letrec 構造を使用します。これは、定義が再帰的である可能性があることを除いて、let とまったく同じです。例えば:

```
infinite n = letrec ns = cons n ns
             in ns
```

(letrec だけを提供するのではなく) コア言語で let と letrec を区別する理由は、let は letrec よりも実装が少し簡単であり、コードが少し良くなる可能性があるためです。

let と letrec は Miranda の where 句に似ていますが、いくつかの重要な違いがあります。

- where 句は、常に再帰スコープを定義します。非再帰的な形式はありません。
- where 句を使用して、ローカル関数を定義し、パターンマッチングを実行できます。

```
... where f x = x+y
          (p,q) = zip xs ys
```

これらの機能はどちらも、コア言語の let および letrec 式では提供されません。

関数はスーパーコンビネータとしてトップレベルでのみ定義でき、パターンマッチングは case 式によってのみ実行されます。

つまり、let または letrec バインディングの左側は単純な変数でなければなりません。

- let/letrec 構造は**式**です。したがって、次のように書くことは非常に合法です (例えば):

```
quad_plus_one x = 1 + (let tx = x+x in tx+tx)
```

対称的に、Miranda の where 句は、**定義**にのみ付加できます。(この理由の 1 つは、Miranda の where 句の定義がいくつかのガードされた右辺に及ぶことを可能にすることです。)

1.1.2 ラムダ抽象化

関数は通常、トップレベルの-スーパーコンビネータ定義を使用してコア言語で表現されます。ほとんどの本では、これが関数を表す表す表す唯一の方法です。ただし、明示的な**ラムダ抽象化**を使用して関数を指定できると便利な場合があり、コア言語はそのための構造を提供します。たとえば、次のプログラムで

```
double_list xs = map (\ x. 2*x) xs
```

ラムダ抽象化 (\　x.　2*x) は、引数を 2 倍にする関数を示します。

明示的なラムダ抽象化を含むプログラムを、トップレベルのスーパーコンビネータ定義のみを使用する同等のプログラムに変換することができます。このプロセスは**ラムダリフティング**と呼ばれ、第 6 章で詳しく説明します。他の章では、このラムダリフティングプロセスが実行されたと想定しているため、明示的なラムダ抽象化を使用していません。

コア言語の最後の主要な構成要素は, パターンマッチングを表す case 式です。パターンマッチングを処理する方法はいくつかあるため、構造化データ型の確認から始めます。

1.1.3　構造化データ型

最新の関数型プログラミング言語すべての普遍的な機能は、**代数的データ型**と呼ばれることが多い**構造化型**の提供です。たとえば、Miranda で書かれたいくつかの代数的型の定義は次のとおりです。

```
colour ::= Red | Green | Blue

complex ::= Rect num num | Polar num num

numPair ::= MkNumPair num num

tree * ::= Leaf * | Branch (tree *) (tree *)
```

各定義は、1 つ以上の**コンストラクタ** (Red, Green など) とともに、新しい**型** (colour など) を導入します。それらは次のように読むことができます:「colour 型の値は Red または Green または Blue のいずれかです」、および「complex は 2 つの num を含む Rect、または 2 つの num を含む Polar のいずれかです」。

型 tree は、**パラメータ化された**代数的データ型の例です。型 tree は、型変数*に関してパラメータ化されます。次のように読む必要があります。「*の tree」s は、*を含む Leaf、または*の 2 つの tree「s」を含む Branch のいずれかです。特定の tree には、同じ型の Leaf が必要です。たとえば、型 tree num は、Leaf に num がある tree であり、型 tree colour は、Leaf に colour がある tree です。

構造化された値は、これらのコンストラクタを使用して構築されます。たとえば、次の式は構造化された値を示します。

```
Green
Rect 3 4
Branch (Leaf num) (Leaf num)
```

構造化された値は、**パターンマッチング**を使用して**分解**されます。例えば:

```
isRed Red = True
isRed Green = False
isRed Blue = False

first (MkNumPair n1 n2) = n1

depth (Leaf n) = 0
depth (Branch t1 t2) = 1 + max (depth t1) (depth t2)
```

通常「組み込み」とみなされるいくつかのデータ型は、構造化型の特殊なケースにすぎません。例えば、ブール値は構造化型です。代数的データ型宣言によって定義できます。

```
bool ::= False | True
```

特別な構文とは別に、Miranda が提供するリストとタプルは構造化型のさらなる例です。: と [] の特別な構文ではなく、コンストラクタとして Cons と Nil を使用する場合、次のようなリストを定義できます。

```
list * ::= Nil | Cons * (list *)
```

[Peyton Jones 1987] の第 4 章では、構造化型について詳しく説明しています。従って、疑問が生じます。**小さなコア言語で構造化型をどのように表現および操作するのでしょうか。**特に、私たちの目標は、コア言語でのデータ型宣言を完全に回避することです。私たちが採用するアプローチは、2 つの部分に別れています。

- コンストラクタには、単純で統一された表現を使用します。
- パターンマッチングを単純な case 式に変換します。

1.1.4 コンストラクタの表現

コア言語で Red や Branch などのユーザ定義のコンストラクタを許可する代わりに、コンストラクタの単一ファミリ

Pack{*tag*, *arity*}

を提供します。

ここで、**tag** はコンストラクタを一意に識別する整数であり、**arity** はそれが取る引数の数を示します。たとえば、colour、complex、tree、numPair のコンストラクタを次のように表すことができます。

```
Red       = Pack{1, 0}
Green     = Pack{2, 0}
Blue      = Pack{3, 0}

Rect      = Pack{4, 2}
Polar     = Pack{5, 2}

Leaf      = Pack{6, 1}
Branch    = Pack{7, 2}

MkNumPair = Pack{8, 2}
```

したがって、コア言語では、

```
Branch (Leaf 3) (Leaf 4)
```

の代わりに

```
Pack{7,2} (Pack{6,1} 3) (Pack{6,1} 4)
```

と記述します。この tag は、異なるコンストラクタで作成されたオブジェクトを互いに区別できるようにするためです。適切に型指定されたプログラムでは、実行時に異なる型のオブジェクトを区別する必要画ないため、tag は**データ型内で**一意である必要があります。したがって、新しいデータ型ごとに 1 から tag を新たに開始して、次の表現を与えることができます。

```
Red       = Pack{1, 0}
Green     = Pack{2, 0}
Blue      = Pack{3, 0}

Rect      = Pack{1, 2}
Polar     = Pack{2, 2}

Leaf      = Pack{1, 1}
Branch    = Pack{2, 2}

MkNumPair = Pack{1, 2}
```

1.1.5 case 式

一般に、最新の関数型プログラミング言語で許可されているパターンマッチングは、複数のネストされたパターン、重複するパターン、ガードなど、かなり複雑になる可能性があります。コア言語の場合、パターンマッチングのすべての複雑な形式を非合法化することにより、これらの複雑さを排除します。これを行うには、コア言語では case 式のみを提供します。そられの正式な構文はセクション 1.2 に記載されていますが、いくつかの例を示します。

```
isRed c = case c of
                <1> -> True ;
                <2> -> False ;
                <3> -> False

depth t = case t of
                <1> n -> 0 ;
                <2> t1 t2 -> 1 + max (depth t1) (depth t2)
```

case 式で重要なことは、各選択肢がタグとそれに続くいくつかの変数 (コンストラクタのアリティと同じである必要があります) のみで構成されていることです。ネストされたパターンは許可されていません。

case 式は、多方向ジャンプのように、非常に単純な操作上の解釈を持っています。分析する式を評価し、それが構築されているコンストラクタのタグを取得して、適切な選択肢を評価します。

1.2　コア言語の構文

図 1.1 に、コア言語の構文を示します。文法では、中置二項演算子を使用できますが、（簡潔にするために）それらの優先順位については明示されていません。代わりに、次の優先順位の表を示します。優先順位が高いほど、バインディングが厳しくなります。

優先度	結合の向き	演算子
6	左結合	関数適用
5	右結合	`*`
	なし	`/`
4	右結合	`+`
	なし	`-`
3	なし	`==` `=` `>` `>=` `<` `<=`
2	右結合	`&`
1	右結合	`|`

演算子の結合法則は、演算子の繰り返しの前後で括弧を省略できる場合を決定します。たとえば、`+`は右結合であるため、`x+y+z` は `x+(y+z)` と同じ意味です。一方、`/`は結合多元環ではないため、式 `x/y/z` は不正です。

単項否定のための特別な演算子記号はありません。代わりに、通常の関数と構文的に同じように動作する `negate` 関数が提供されます。例えば：

```
f x = x + (negate x)
```

ブール否定演算子 `not` は、同じ方法で処理されます。

プログラム	*program*	→	sc_1 ; ... ; sc_n	$n \geq 1$
スーパーコンビネータ	*sc*	→	$var\ var_1 \ldots var_n\ =\ expr$	$n \geq 0$
式	*expr*	→	*expr expr*	関数適用
		|	$expr_1\ binop\ expr_2$	中置 2 項演算
		|	`let` *defns* `in` *expr*	局所定義
		|	`letrec` *defns* `in` *expr*	局所再帰定義
		|	`case` *expr* `of` *alts*	case 式
		|	\ $var_1 \ldots var_n$. *expr*	ラムダ抽象 ($n \geq 1$)
		|	*aexpr*	原子式
	aexpr	→	*var*	変数
		|	*num*	数値
		|	`Pack{`num`,`num`}`	コンストラクタ
		|	`(` *expr* `)`	括弧で囲まれた式
定義	*defns*	→	$defn_1$; ... ; $defn_n$	$n \geq 1$
	defn	→	*var* `=` *expr*	
選択肢	*alts*	→	alt_1; ... ; alt_n	$n \geq 1$
	alt	→	`<`*num*`>` $var_1 \ldots var_n$ `->` *expr*	$n \geq 0$
二項演算子	*binop*	→	*arithop* | *relop* | *boolop*	
	arithop	→	`+` | `-` | `*` | `/`	算術演算子
	relop	→	`<` | `<=` | `==` | `~=` | `>=` | `>`	比較演算子
	boolop	→	`&` | |	論理演算子
変数	*var*	→	*alpha* $varch_1$ ... $varch_n$	$n \geq 0$
	alpha	→	*analphabeticcharacter*	
	varch	→	*alpha* | *digit* | _	
数値	*num*	→	$digit_1 \ldots digit_n$	$n \geq 1$

1.3　コア言語のデータ型

この本で説明されている実装ごとに、コンパイラとマシンインタープリタを構築します。コンパイラはコアプログラムを受け取り、それをマシンインタープリタによる実行に適した形式に変換します。これを行うには、コアプログラムを表す Miranda データ型が必要です。これを、このセクションで定義します。実際、コアプログラム用の型、コア式用の型、およびその他のいくつかの補助的な型を定義します。

コア言語式 expr のデータ型は、次のように定義されます。

```
module Language where
import Utils

data Expr a
  = EVar Name                   -- Variables
  | ENum Int                    -- Numbers
  | EConstr Int Int             -- Constructor tag arity
  | EAp (Expr a) (Expr a)       -- Applications
  | ELet                        -- Let(rec) expressions
       IsRec                    --   boolean with True = recursive,
       [(a, Expr a)]            --   Definitions
       (Expr a)                 --   Body of let(rec)
  | ECase                       -- Case expression
       (Expr a)                 --   Expression to scrutinise
       [Alter a]                --   Alternatives
  | ELam [a] (Expr a)           -- Lambda abstraction
  deriving (Text)
```

バインダに関して Expr のデータ型をパラメータ化することを選択します。バインダは、変数のバインディングオカレンスで使用される名前です。つまり、let(rec) 定義の左辺、またはラムダ抽象化です。コア言語式 expr の型は、宣言は、「* の expr は、Name 含む EVar、または ...、またはタイプ * の値のリストと * の Expr を含む ELam のいずれかです」と読み取ることができます。

本書のほとんどの部分では、これらのバインディング位置で常に Name を使用しているため、**型シノニム**を使用して、通常使用するタイプである CoreExpr のタイプを定義します。

```
type CoreExpr = Expr Name
```

1.4 A small standard prelude

1.5　コア言語用のプリティプリンタ

CoreProgram 型の値を取得したら、それを表示できると便利なことがよくあります。Haskell の組み込み機能は、ここではあまり役に立ちません。たとえば、Haskell プロンプトに応答して preludeDefs と入力すると、生成される出力を理解するのはかなり困難です。(それを試してみてください。)

必要なのは、次の型が付いた「プリティプリント」関数 pprint です。

```
pprint :: CoreProgram -> String
```

次に、pprint preludeDefs と入力して、表示すると preludeDefs の適切にフォーマットされたバージョンのように見える文字のリストを取得することを期待できます。このセクションの目標は、そのような関数を作成することです。

プログラムの結果がリストの場合、Haskell は通常、リスト項目をコンマで区切り、角かっこで囲んで印刷します。ただし、プログラムの結果が [Char] 型の特殊なケースでは、Haskell は角かっことコンマを付けずに「すべて押しつぶされた」リストを表示します。たとえば、値 ‘‘Hi \nthere’’ は次のように表示されます。

```
Hi
there
```

次のようには表示されません。

```
['H', 'i', '\n', 't', 'h', 'e', 'r', 'e']
```

このようにして、pprint は出力フォーマットを完全に制御できます。

1.5.1　文字列を使ったプリティプリンティング

まず、コア言語の式に集中しましょう。プリティプリンティング関数 pprExpr が必要なように見えます。次のように定義されています。

```
pprExpr :: CoreExpr -> String
pprExpr (ENum n) = show n
pprExpr (EVar v) = v
pprExpr (EAp e1 e2) = pprExpr e1 ++ " " ++ pprAExpr e2
```

(現時点では、pprExpr のケースの多くを意図的に除外しています。) pprAExpr の型は pprExpr と同じですが、変数または数値でない限り、式を括弧で囲む点が異なります。

```
pprAExpr :: CoreExpr -> String
pprAExpr e = isAtomicExpr e | pprExpr e
pprAExpr e = otherwise | "(" ++ pprExpr e ++ ")"
```

この方法で進めることはできますが、そうすることには深刻な問題があります。プリティプリンタは、リスト連結関数 ++ を大量に使用します。次の例に示すように、これは非常に厄介なパフォーマンスをもたらす可能性があります。次の式を考えてみましょう。

```
(x1 ++ x2) ++ x3
```

内側の ++ は `length xs1` に比例する時間がかかりますが、外側の ++ は `xs1 ++ xs2` の長さに比例する時間がかかるため、合計時間は (2 * `length xs1`) + `length xs2` になります。一般に、このネストされた append にさらにリストを追加した場合、コストは結果の長さの 2 乗になる可能性があります！もちろん、式を逆に括弧でくくった場合、コストは結果の長さに比例しますが、残念ながら、プリティプリンタでこれを保証することはできません。

この効果を実証するために、最初に関数 `mkMultiAp` を記述します。これにより、特定のサイズのサンプル式を簡単に作成できます。呼び出し (`mkMultiAp` *n e1 e2*) は、式を表す `coreExpr` を生成します

$$e_1 \underbrace{e_2\ e_2 \ldots e_2}_{n}$$

```
mkMultiAp :: Int -> CoreExpr -> CoreExpr -> CoreExpr
mkMultiAp n e1 e2 = foldll EAp e1 (take n e2s)
                    where
                      e2s = e2 : e2s
```

この定義では、 `take` は、リストの最初の n 個の要素を取り、リストの残りを破棄する Miranda の標準関数です。関数 `foldll` は標準関数であり、Appendix A[2] で定義されています。二項関数 $\otimes$、値 acc およびリスト $xs = [x_1,\ \ldots,\ x_n]$ が与えられると、`foldll` $\otimes$ acc xs は acc' を計算します。

$$acc' = (\ldots((acc\ \otimes\ x_1)\ \otimes\ x_2)\ \otimes\ \ldots\ x_n)$$

これを図 1.2 に示します。`mkMultiAp` では、`foldll` を使用して `EAp` ノードの左分岐チェーンを構築します。初期のアキュムレータ acc は `e1` であり、結合関数は `EAp` コンストラクタです。最後に、`e2s` は無限のリスト `[e2, e2, ...]` です。最初の `n` 個の要素のみが `take` によって使用されます。

[2]Miranda のバージョンが異なれば foldl の定義も異なるため、Miranda の標準関数 `foldl` ではなく `foldll` を使用します。

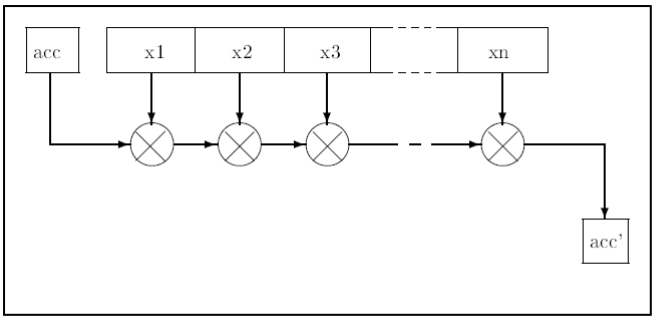


Figure 1.2: An illustration of `foldll` $\otimes$ $acc\ [x_1, \ldots, x_n]$

演習 1.1 `n` のさまざまな値に対して、計算に必要な Miranda のステップ数を測定します。

```
length (pprExpr (mkMultiAp n (EVar "f") (EVar "x")))
```

(Miranda ディレクティブ `/ count` を使用して、Miranda に実行統計を出力するように指示できます。画面が巨大なプリントアウトでいっぱいにならないように、結果の長さをとります。) 実行コストが `n` でどのように上昇するかを示すグラフをスケッチし、`n` に関してほぼ 2 乗であることを確認します。

1.5.2　プリティプリンティングのための抽象データ型

コストが表示されるプログラムのサイズの 2 倍になるプリティプリンタは、明らかに容認できないので、それを回避する方法を見つけたほうがよいでしょう。

この問題を 2 つの部分に分けることができます: **「実行したい操作は?」**と**「それらを実行する効率的な方法は?」**です。他の言語と同様に、Miranda は抽象データ型を導入することでこの区別を明確にする方法を提供します。

```
iNil     :: Iseq                 -- The empty iseq
iStr     :: String -> Iseq       -- Turn a string into an iseq
iAppend  :: Iseq -> Iseq -> Iseq -- Append two iseqs
iNewline :: Iseq                 -- New line with indentation
iIndent  :: Iseq -> Iseq         -- Indent an iseq
iDisplay :: Iseq -> String       -- Turn an iseq into a string
```

abstype キーワードは、抽象データ型 Iseq を導入します。その後に、データ型の**インターフェイス**が続きます。つまり、データ型 Iseq で実行できる操作と、各操作の型です。

このようなデータ型が与えられた場合、pprExpr を書き直して、文字のリストではなく Iseq を返すようにします。

```
pprExpr :: CoreExpr -> Iseq
pprExpr (EVar v)    = iStr v
pprExpr (EAp e1 e2) = (pprExpr e1) `iAppend` (iStr " ") `iAppend` (pprAExpr e2)
```

++ を iAppend [3] に単純に置き換え、リテラル文字列の周りに iStr を追加しました。

Iseq と文字のリストの違いは何ですか? まず、リスト追加の予想外の二次動作を持たない iAppend の実装を作成することを目指しています。次に Iseq は、インデントの制御に役立つ新しい操作 iIndent および iNewline を提供します。アイデアは、iIndent がその引数をインデントして現在の列に揃えるというものです。引数が何行にもわたっており、それ自体に iIndent の呼び出しが含まれている場合でも機能するはずです。iNewline は、改行の後に、現在のインデント レベルによって決定されるいくつかのスペースが続くことを表します。

iIndent と iNewline の使用方法の例として、pprExpr を拡張して let と letrec 式を処理してみましょう。

```
pprExpr (ELet isrec defns expr)
  = iConcat [ iStr keyword, iNewline,
              iStr " ", iIndent (pprDefns defns), iNewline,
              iStr "in ", pprExpr expr ]
    where
      keyword | not isrec = "let"
              | isrec     = "letrec"

pprDefns :: [(Name, CoreExpr)] -> Iseq
pprDefns defns = iInterleave sep (map pprDefn defns)
                 where
                   sep = iConcat [ iStr ";", iNewline ]
```

[3] Haskell では、識別子の前後にバッククォート記号を書くと、それが中置演算子に変わり、引数の前ではなく引数の間に iAppend を書くことができます。このような中置演算子は右結合です。

```
pprDefn :: (Name, CoreExpr) -> Iseq
pprDefn (name, expr) = iConcat [ iStr name, iStr " = ", iIndent (pprExpr expr) ]
```

定義をより読みやすくするために、iConcat と iInterleave の 2 つの新しい関数を使用しました。

```
iConcat :: [Iseq] -> Iseq
iInterleave :: Iseq -> [Iseq] -> Iseq
```

iConcat は iseq のリストを受け取り、iAppend を使用してそれらを単一の iseq に連結します。iInterleave は、隣接する各ペア間に指定された iseq をインターリーブする点を除いて、iConcat に似ています。

演習 1.2 iAppend と iNil に基づいて iConcat と iInterleave を定義します。

一般に、すべてのプリティプリンティング関数は Iseq を返し、表示したいもの全体を表す Iseq に iDisplay をトップレベルで 1 回だけ適用します。

演習 1.3 pprExpr にさらに式を追加して case 式とラムダ式を処理し、pprAExpr と pprProgram の定義を同じスタイルで記述します。

1.5.3 iseq の実装

Iseq 型の実装に移ります。すべてのインデントを無視する実装を作成することから始めます。抽象データ型を実装するには、型が何であるかを言う必要があります Iseq を表すために使用されます。

```
data Iseq = INil
          | IStr String
          | IAppend Iseq Iseq
```

最初の宣言は iseqRep が Iseq を表すために使用されることを示し、2 番目の宣言は iseqRep が 3 つのコンストラクター INil、IStr、および IAppend を持つ代数データ型であることを宣言します。

この特定の表現の一般的な考え方は、iDisplay の最終的なすべてまですべての作業を延期することです。操作 iNil、iStr、iAppend はすべて、関連するコンストラクタを使用するだけです。

```
iNil = INil
iStr str = IStr str
iAppend seq1 seq2 = IAppend seq1 seq2
```

インデントを無視しているので、iIndent と iNewline は自明に定義されています。次のセクションでそれらを改善します。

```
iNewline = IStr "\n"
iIndent seq = seq
```

Iseq を文字のリストに変換する操作 iDisplay にすべての関心があります。目標は、Iseq のサイズに比例した時間だけかかるようにすることです。より一般的な機能である flatten に関して iDisplay を定義すると便利であることが判明しました。

```
flatten :: [Iseq] -> String


iDisplay seq = flatten [seq]
```

関数 flatten は iseqReps のリストを受け取り、リスト内の各 iseqReps を連結した結果を返します。このリストを持つ理由は、すぐにわかるように、保留中の作業のリストを蓄積できるようにするためです。flatten は、抽象型 Iseq ではなく、表現型 iseqRep を操作することに注意してください。

ワークリストと呼ばれる引数のケース分析によって flatten を定義します。ワークリストが空の場合、完了です。

```
flatten [] = ""
```

それ以外の場合は、ワークリストの最初の要素でケース分析を行うことで作業します。INil の場合は、ワークリストからアイテムをポップするだけです。

```
flatten (INil : seqs) = flatten seqs
```

IStr の場合は、指定された文字列を追加して、残りのワークリストをフラット化することで機能します。

```
flatten (IStr s : seqs) = s ++ (flatten seqs)
```

これまでのところ、flatten がリストを取得するという事実は、私たちにはあまり役に立ちませんでした。IAppend を扱うと、リスト引数の正当性がより明確にわかります。実行する必要があるのは、もう 1 つのアイテムをワークリストの先頭にプッシュすることだけです。

```
flatten (IAppend seq1 seq2 : seqs) = flatten (seq1 : seq2 : seqs)
```

演習 1.4 Iseq のサイズに関して平坦化のコストはいくらですか? 上記のように Iseq を使用するように pprExpr を変更し、前の演習と同じ実験を使用して新しい実装の効果を測定します。pprExpr の結果に iDisplay を適用することを忘れないでください。

演習 1.5 抽象データ型を使用する主な利点は、**インターフェイス**に影響を与えずに ADT の**実装**を変更できることです。この例として、引数のいずれかが INil の場合に単純化された結果を返すように iAppend を再定義します。

1.5.4　レイアウトとインデント

これまでのところ、iIndent 操作に対してかなり些細な解釈しか与えていません。次に、それを改善します。以前と同じ精神で、最初に iseqRep 型を追加の 2 つのコンストラクタ IIndent と INewline で拡張し、これらのコンストラクターを使用するように操作を再定義します。

```
data Iseq = INil
| IStr String
| IAppend Iseq Iseq
| IIndent Iseq
| INewline

iIndent seq = IIndent seq
iNewline    = INewline
```

次に、flatten をより強力にする必要があります。まず、現在のカラムを追跡する必要があります。次に、そのワークリストは (Iseq, Int) ペアで構成されている必要があります。ここで、数値は対応する Iseq に必要なインデントを示します。

```
flatten :: Int              -- Current column; 0 for first column
           -> [(Iseq, Int)] -- Work list
           -> String        -- Result
```

flatten を適切に初期化するには、iDisplay を変更する必要があります。

```
iDisplay seq = flatten 0 [(seq, 0)]
```

flatten の興味深いケースは、INewline を扱う場合です。これは、インデントを実行する必要があるためです。

```
flatten col ((INewline, indent) : seqs)
  = '\n' : (space indent) ++ (flatten indent seqs)
```

新しい行に移動して indent スペースを追加したので、flatten の再帰呼び出しには現在の列引数 indent があることに注意してください。これは、新しい行に移動してインデント スペースを追加したためです。

IIndent ケースは、現在の列から現在のインデントを設定するだけです。

```
flatten col ((IIndent seq, indent) : seqs) = flatten col ((seq, col) : seqs)
```

演習 1.6 IAppend、IStr、および INil の flatten の定義を追加します。Elet を含む式で pprExpr を試し、レイアウトが適切に機能することを確認します。

演習 1.7 IStr に与えられた文字列に改行文字 '\n' が埋め込まれていると、プリティプリンタは正しく動作しません。これをチェックするように iStr を変更し、改行文字を INewline を使用して置き換えます。

1.5.5 演算子の優先順位

セクション 1.3 で説明したように、CoreExpr 型には中置演算子適用のための構造がありません。代わりに、そのような関数適用は、他の関数適用と同様に、前置形式で表現されます。私たちのプリティプリンタがそのような関数適用を認識し、それらを中置形式で表示できれば素晴らしいと思います。これは、次の形式の pprExpr に式を追加することで簡単に実行できます。

```
pprExpr (EAp (EAp (EVar "+") e1) e2)
  = iConcat [ pprAExpr e1, iStr " + ", pprAExpr e2 ]
```

この式はまだあまりうまく機能していません。括弧が多すぎるからです。次の式

```
x + y > p * length xs
```

という式と、次の括弧で囲まれた完全なバージョンのどちらが良いでしょうか？

```
(x + y) > (p * (length xs))
```

これを実現する最も簡単な方法は、 pprExpr にそのコンテキストの優先レベルを示す追加の引数を与え、これを使用して、生成される式の周りに括弧を追加するかどうかを決定することです。(関数 pprAExpr は冗長になります。)

演習 1.8 これらの変更を pprExpr に加えてテストします。

1.5.6　iseq のその他の便利な機能

後で Iseq で動作する関数がいくつかあると便利です。これらはすべて Iseq インターフェイス関数の観点から定義されているため、これらの定義を変更せずに実装を変更できます。

iNum は数値を Iseq にマップし、iFWNum は同じことを行いますが、結果は指定された幅までスペースで左がパディングされます。

```
iNum :: Int -> Iseq
iNum n = iStr (show n)

iFWNum :: Int -> Int -> Iseq
iFWNum width n
  = iStr (space (width - length digits) ++ digits)
    where
      digits = show n
```

(数値が必要な幅より広い場合、負の数値がスペースに渡され、空のリストが返されます。したがって、最終的な効果は、含めるのに十分な幅のフィールドを返すことです数字。) iLayn は、標準関数 layn と同じように、項目に番号を付け、それぞれの後に改行文字を付けてリストをレイアウトします。

```
iLayn :: [Iseq] -> Iseq
iLayn seqs
  = iConcat (map lay_item (zip [1..] seqs))
    where
      lay_item (n, seq)
        = iConcat [ iFWNum 4 n,
                    iStr ")",
                    iIndent seq,
                    iNewline
                  ]
```

1.5.7　概要

私たちのプリティプリンタにはまだ欠点があります。特に、優れたプリティプリンタは、収まる場合は 1 行にレイアウトし、収まらない場合は複数の行にレイアウトします。次のように Iseq データ型を精巧にすることは十分に可能です。これを行いますが、ここでは行いません。Iseq 型は、プログラム以外のデータをきれいに印刷するのに役立ちます。この本では、さまざまな

目的で使用します。このセクションから導き出したい 2 つの一般的なポイントがあります。

- 抽象データ型の**インターフェース**をその**実装**から分離することは、非常にしばしば役に立ちます。Miranda は、抽象型に対する関数が表現を検査しないようにすることで、この抽象化を直接サポートします。
- `flatten` に関する `iDisplay` の定義は、汎化と呼ばれる非常に一般的な手法の例です。より一般的な関数への単純な呼び出しという観点から、本当に必要な関数を定義することがよくあります。これは通常、より一般的な関数が、簿記をまっすぐに保つために必要ないくつかの追加の引数を持ち運ぶためです。

  一般化が適切な手法である場合について、一般的な意見を述べるのは困難です。実際、適切な一般化を行うことは、多くの場合、プログラムを作成する際の主要な創造的ステップです。ただし、この本には一般化の例がたくさんあり、アイデアを伝えるのに役立つことを願っています。

1.6　コア言語のパーサ

それぞれの実装をさまざまなコアプログラムで実行する必要があります。これは、具体的な構文で Core プログラムを含むファイルを取得し、それを CoreProgram 型の値に解析する方法が必要であることを意味します。

パーサを作成するのは一般的にかなり面倒なので、文法を受け入れてパーサを作成するツールの作成に多大な労力が費やされてきました。Unix Yacc ユーティリティは、そのようなパーサジェネレータの例です。ただし、関数型言語では、単純なパーサを作成するのは非常に簡単です。このセクションでは、コア言語について説明します。タスクを 3 つの段階に分割します。

- まず、名前付きファイルの内容を文字のリストとして取得します。これは、組み込みの Miranda 関数 read によって実行されます。
- 次に、**字句解析**関数 lex は、入力を識別子、数字、記号などの小さなチャンクのシーケンスに分割します。これらの小さなチャンクは**トークン**と呼ばれます。

  ```
  Clex :: String -> [Token]
  ```

- 最後に、**構文解析**関数の構文は、この一連のトークンを使用して CoreProgram を生成します。

  ```
  syntax :: [Token] -> CoreProgram
  ```

完全なパーサは、これら 3 つの関数の合成にすぎません。

```
parse :: String -> CoreProgram
parse = syntax . clex
-- In Gofer I propose to compose this with some function
-- CoreProgram -> String, which will illustrate some sort of
-- execution machine, and then give this composition to catWith
-- from my utils
```

記号 '.' は、Miranda の中置関数合成演算子であり、次のように定義できます。

```
(f . g) x = f (g x)
```

同様に、次のように、合成を使用せずに解析関数を定義することもできます。

```
parse filename = syntax (lex (read filename))
```

ただし、構文解析を３つの関数のパイプラインとして定義していることが特に簡単にわかるため、関数合成を使用する方が適切なスタイルです。

1.6.1 字句解析

字句解析器から始めます。トークンの種類はまだ定義されていません。最初に最も簡単なことは、トークンをまったく処理せず、(空でない) 文字列として残すことです。

```
type Token = String -- A token is never empty
```

今、字句解析自体。空白 (空白、タブ、改行) を破棄する必要があります：

```
clex (c : cs) | isWhiteSpace c = clex cs
```

数字を単一のトークンとして認識する必要があります。

```
clex (c : cs) | isDigit c = num_token : clex rest_cs
                where
                  num_token = c : takeWhile isDigit cs
                  rest_cs   = dropWhile isDigit cs
```

標準関数 `isDigit` は文字を受け取り、その文字が 10 進数の場合にのみ、`True` を返します。`takewhile` と `dropwhile` も標準機能です。`takewhile` は、述語が満たされている間、リストの先頭から要素を取得し、`dropwhile` は、述語が満たされている間、リストの先頭から要素を削除します。例えば、

```
takeWhile isDigit "123abc456"
```

の結果はリスト`"123"`です。

字句解析プログラムは、アルファベット文字で始まり、文字、数字、アンダースコアのシーケンスで続く変数も認識する必要があります。

```
clex (c : cs) | isAlpha c = var_tok : clex rest_cs
                where
                  var_tok = c : takeWhile isIdChar cs
                  rest_cs = dropWhile isIdChar cs
```

ここで、letter はアルファベット文字で True を返す digit のような標準関数であり、isIdChar は以下で定義されています。

上記の式のいずれにも当てはまらない場合、字句解析プログラムは 1 文字を含むトークンを返します。

```
clex (c : cs) = [c] : clex cs
```

最後に、入力文字列が空の場合、lex は空のトークンリストを返します。

```
clex [] = []
```

上記で使用した補助関数の定義で締めくくります。(演算子'' は Miranda のブール'or' 演算です。)

```
isIdChar, isWhiteSpace :: Char -> Bool
isIdChar c = isAlpha c || isDigit c || (c == '_')
isWhiteSpace c = c `elem` " \t\n"
```

演習 1.9 コメントと空白を無視するように字句アナライザを変更します。コメントが二重の縦棒||によって導入されるのと同じ規則を使用し、行の終わりまで延長します。

演習 1.10 字句解析プログラムは現在、<=や==などの 2 文字の演算子を単一のトークンとして認識しません。それらのリストを提供することにより、そのような演算子を定義します。

```
twoCharOps :: [String]
twoCharOps = ["==", "~=", ">=", "<=", "->"]
```

twoCharOps の member をトークンとして認識するように lex を変更します。(標準関数メンバーが役立つ場合があります。)

演習 1.11 字句解析では空白が破棄されるため、パーサは構文エラーの行番号を報告できません。この問題を解決する 1 つの方法は、各トークンに行番号を付けることです。つまり、タイプ token は次のようになります。

```
token == (num, [char])
```

これを行うように字句アナライザを変更します。これを行うには、現在の行番号であるパラメータを lex に追加する必要があります。

1.6.2 構文解析のための基本的なツール

コア言語のパーサを作成する準備として、パーサを作成するときに使用するいくつかの汎用関数を開発します。以下で説明する手法はよく知られていますが [Fairbairn1986、Wadler 1985]、関数型プログラミングで何ができるかをかなりうまく示しています。実行例として、次の小さな文法を使用します。

$$
\begin{array}{lcl}
\textit{greeting} & \rightarrow & \textit{hg person}\ \texttt{!} \\
\textit{hg} & \rightarrow & \texttt{hello} \\
 & | & \texttt{goodbye}
\end{array}
$$

ここで、*person* は文字で始まるトークンです。

関数型プログラミングで非常に一般的な私たちの一般的なアプローチは、小さなパーサを接着して大きなパーサを構築しようとすることです。重要な質問は、パーサのタイプはどうあるべきかということです。これは、トークンのリストを引数として受け取る関数であり、最初は、解析された値を返すだけでよいように見えます。しかし、これは 2 つの理由から、十分に一般的ではありません。

1. まず、トークンの残りのリストも返す必要があります。たとえば、入力から 2 つの項目を次々に解析する場合は、最初のパーサを入力に適用できますが、最初のパーサから返された残りの入力に 2 番目のパーサを適用する必要があります。

2. 次に、文法があいまいな場合があるため、入力を解析する方法は複数あります。または、入力が文法に準拠していない可能性があります。その場合、入力を正常に解析する方法はありません。これらの可能性に対応するためのエレガントな方法は、可能な解析のリストを返すことです。このリストは、入力を解析する方法がない場合は空であり、入力を解析する独自の方法がある場合は 1 つの要素を含みます。

次のように、型シノニムを使用してパーサの型を定義することにより、結論を要約できます。

```
type Parser a = [Token] -> [(a, [Token])]
```

つまり、型*の値のパーサは、トークンのリストを受け取り、解析のリストを返します。各解析は、型*の値と残りのトークンのリストを組み合わせたものです。

1.6.3　ツールを鋭利にする

これで、パーサを開発するための基本的なツールが完成しました。このセクションでは、さまざまな方法でそれらを開発します。

文法では人の名前の後に感嘆符が必要なため、上記の pGreeting の定義は完全には正しくありません。次のように問題を解決できます。

```
pGreeting = pThen keep_first
                    (pThen mk_pair pHelloOrGoodbye pVar)
                    (pLit "!")
          where
          keep_first hg_name exclamation = hg_name
          mk_pair hg name = (hg, name)
```

最終的な感嘆符は常に存在するため、解析された値の一部として返さないことを選択しました。keep_first によって破棄されます。ただし、この定義はかなり不器用です。新しい関数 pThen3 を定義すると、次のように記述できるので便利です。

```
pGreeting = pThen3 mk_greeting
                    pHelloOrGoodbye
                    pVar
                    (pLit "!")
          where
          mk_greeting hg name exclamation = (hg, name)
```

演習 1.12 pThen3 の型を指定し、その定義を書き留めて、新しいバージョンの pGreeting をテストします。同様に、後で必要になる pThen4 を記述します。

文法のもう 1 つの非常に一般的な機能は、記号の 0 回以上の繰り返しを要求することです。これを反映するために、パーサ p を受け取り、p が認識する 0 回以上の出現を認識する新しいパーサを返す関数 pZeroOrMore が必要です。成功した解析によって返される値は、p の継続的な使用によって返される値のリストである可能性があります。従って、pZeroOrMore の型は次のようになります。

```
pZeroOrMore :: Parser a -> Parser [a]
```

例えば、0 個以上の挨拶を認識するパーサは次のようになります。

```
pGreetings :: Parser [(String, String)]
pGreetings = pZeroOrMore pGreeting
```

1 つ以上の出現、または出現なしのいずれかを確認する必要があることを確認することで、pZeroOrMore になることができます。

```
pZeroOrMore p = (pOneOrMore p) `pAlt` (pEmpty [])
```

ここで、pEmpty は常に成功するパーサであり、入力から何も削除せず、最初の引数として指定された値を返します。

```
pEmpty :: a -> Parser a
```

関数 pOneOrMore は、pZeroOrMore と同じ型です。

```
pOneOrMore :: Parser a -> Parser [a]
```

演習 1.13 pOneOrMore と pEmpty の定義を記述します。(ヒント:pOneOrMore から pZeroOrMore を呼び出すと便利です。) 1 つまたは複数の挨拶を認識するパーサを定義するためにそれらを使用して、定義をテストします。

成功した解析によって返された値を処理すると便利なことがよくあります。たとえば、pGreetings が内容ではなく、挨拶の数を返すようにしたいとします。これを行うには、長さ関数 # を pZeroOrMore によって返される値に適用します。

```
pGreetingsN :: Parser Int
pGreetingsN = (pZeroOrMore pGreeting) `pApply` length
```

ここで、pApply は新しいパーサ操作関数であり、パーサと関数を受け取り、パーサによって返される値に関数を適用します。

```
pApply :: Parser a -> (a -> b) -> Parser b
```

演習 1.14 pApply の定義を記述し、それをテストします。(ヒント：リスト内包表記を使用してください。)

文法のもう 1 つの非常に一般的なパターンは、他の記号で区切られた 1 つ以上の記号の出現を探すことです。たとえば、図 1.1 のプログラムは、セミコロンで区切られた 1 つ以上のスーパーコンビネータ定義のシーケンスです。さらに別のパーサ構築関数 pOneOrMoreWithSep が必要です。この関数のタイプは、次の通りです。

```
pOneOrMoreWithSep :: Parser a -> Parser b -> Parser [a]
```

2 番目の引数は、結果の一部として返されないセパレータを認識するパーサです。そのため、型に**が 1 回だけ出現します。

演習 1.15 pOneOrMoreWithSep を定義してテストします。*program* の次の文法を考えると役立つ場合があります。

$$\begin{array}{lcl} program & \rightarrow & sc\ programRest \\ programRest & \rightarrow & ;\ program \\ & | & \epsilon \end{array}$$

ここで、ϵ は空の文字列 (pEmpty パーサに対応) です。

パーサ pLit と pVar は互いに非常に似ています。どちらも最初のトークンのプロパティをテストし、失敗するか (プロパティがない場合)、成功してトークン内の文字列を返します (トークンがある場合)。このアイデアを一般化するには、パーサ pSat('sat' は'satisfies' の略) を次のように記述します。

```
pSat :: (String -> Bool) -> Parser String
```

pSat は、トークン内の文字列に目的のプロパティがあるかどうかを通知する関数を受け取り、そのプロパティを持つトークンを認識するパーサを返します。これで、pSat[4]の観点から pLit を記述できます。

```
pLit s = pSat (== s)
```

演習 1.16 pSat を定義し、テストします。pLit と同様の方法で、pSat の観点から pVar を記述します。

pSat は、有用なレベルのモジュール性を追加します。たとえば、pVar は現在、すべてのアルファベットトークンを変数として認識しますが、最終的には、言語キーワード (let や case など) を変数として認識しないようにする必要があります。

演習 1.17 上記の pVar の定義で pSat に渡される関数を変更して、リストキーワードの文字列を変数として扱わないようにします。

```
keywords :: [String]
keywords = ["let", "letrec", "case", "in", "of", "Pack"]
```

演習 1.18 別の例として、pSat を使用して、以下に示す型の数値のパーサを定義します。

[4]式 (= s) は**セクション**と呼ばれます。これは、等式演算子=を 1 つの引数 s に部分適用し、その引数が s に等しいかどうかをテストする関数を生成します。

```
pNum :: ParserInt
```

pNum は pSat を使用して数値トークンを識別し、次に pApply を使用して文字列を数値に変換する必要があります。(Miranda は、ハードワークを実行するために使用できる [Char]-> num 型の標準関数 numval を提供します。)

pOneOrMore とそれに関連する機能に関連する興味深いパフォーマンスの問題があります。次のコアプログラムについて考えてみます。

```
f x = let x1 = x; x2 = x; ...; xn = x
      of x1
```

アイデアは、x1, x2, ..., xn の定義を持つ大きな let 式があるということです。(定義はかなり些細なものですが、目的を果たします。) このプログラムには構文エラーがあります。let 式の後に「in」ではなく「of」を記述しました。

1.6.4 コア言語の構文解析

コア言語のパーサを定義する準備が整いました。まず、「ラッパー」関数、構文を扱います。トークンのリストを取得し、coreProgram 型の結果を提供することを思い出してください。これを行うには、非終端プログラムを解析するパーサ pProgram(図 1.1) を呼び出し、それが返す最初の完全な解析を選択します。完全な解析が返されない場合、つまり、解析後に残っているトークンのシーケンスが空である場合、構文は (ひどく情報が少ない) エラーメッセージを生成します。

```
syntax = take_first_parse . pProgram
         where
           take_first_parse ((prog, []) : others) = prog
           take_first_parse (parse      : others) = take_first_parse others
           take_first_parse other                 = error "Syntax error"
```

私たちの構文解析ツールの利点は、文法を Miranda に直訳するだけでパーサを記述できることです。たとえば、図 1.1 の *program* と *sc* のプロダクションについて考えてみます。

$$\begin{array}{lcll} program & \rightarrow & sc_1 ; \ldots ; sc_n & (n \geq 1) \\ sc & \rightarrow & var\ var_1\ \ldots\ var_n = expr & (n \geq 0) \end{array}$$

これらを直接 Miranda に直訳することができます。

```
pProgram :: Parser CoreProgram
pProgram = pOneOrMoreWithSep pSc (pLit ";")

pSc :: Parser CoreScDefn
pSc = pThen4 mk_sc pVar (pZeroOrMore pVar) (pLit "=") pExpr
```

演習 1.20 関数 mk_sc を記述します。これは、pSc で使用される 4 つのパーサによって返される 4 つの引数を取り、次のタイプの値を作成します。

```
(name, [name], coreExpr)
```

関数適用と中置演算子の生成を除いて、残りの文法の定義を完了するのは簡単なことです。

演習 1.21 これらの 2 つのプロダクションを除外して、パーサを完成させます。型が Parser CoreExpr であるパーサ pAexpr には少し注意が必要です。pApply 関数は、pVar によって返される値の周りに EVar コンストラクタをラップし、pNum によって返される値の周りに ENum コンストラクタをラップするために必要です。次のプログラムでパーサをテストします。

```
f = 3 ;
g x y = let z = x in z ;
h x = case (let y = x in y) of
        <1> -> 2 ;
        <2> -> 5
```

より大きなプログラムでパーサを実行すると、出力が判読できなくなることがわかります。これを解決するには、プリティプリンティング関数 pprint を使用して、パーサの出力をフォーマットします。

演習 1.22 次のプログラムを検討します。

```
f x y = case x of
        <1> -> case y of
                 <1> -> 1;
        <2> -> 2
```

<2>で始まる選択肢は、内側の case または外側の case のどちらに取り付けられますか？答えを見つけて、パーサが期待どおりに動作するかどうかを確認します。これは「ぶら下がり else」の問題として知られています。

ここで、上記の 2 つの問題に目を向けます。

1.6.5 左再帰

関数適用の問題は比較的簡単に解決できます。関数適用のためのプロダクションは次のようになります。

$$expr \rightarrow expr\ aexpr$$

これを単に直訳すると

```
pExpr = pThen EAp pExpr pAexpr
```

残念ながら、pExpr はそれ自体を無期限に呼び出し続けるため、終了することはありません。問題は、*expr* が *expr* の生成の最初のシンボルとして表示されることです。これは**左再帰**と呼ばれます。私たちの構文解析ツールは、単に左再帰文法に対処できません。幸いなことに、通常、文法を変換して左再帰にならないようにすることができますが、結果の文法は、構築しようとしている結果の構造を反映していません。この場合、たとえば、繰り返しを使用して、問題のあるプロダクションを次のように変換できます。

$$expr \rightarrow aexpr_1 \ldots aexpr_n\ (n \geq 1)$$

これで、パーサ (pOneOrMore pAexpr) を使用できるようになりました。問題は、これが EAp コンストラクターで作成された単一の式ではなく、式のリストを返すことです。pApply を使用してこれを解決し、パーサを提供します

```
(pOneOrMore pAexpr) $ pApply mk_ap_chain
```

演習 1.23 型が [CoreExpr] -> CoreExpr の適切な関数 mk_ap_chain を定義します。関数適用のプロダクションをパーサに追加してテストします。

1.6.6 中置演算子の追加

中置演算子の最初の問題は、それらの優先順位が図 1.1 の文法に暗黙的に含まれていることです。これを明示的にする標準的な方法は、図 1.3 に示すように、いくつかの種類の式を使用することです。

$$
\begin{array}{lcll}
expr & \rightarrow & \texttt{let}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{letrec}\ defns\ \texttt{in}\ expr & \\
 & | & \texttt{case}\ expr\ \texttt{of}\ alts & \\
 & | & \backslash\ var_1 \ldots var_n\ \texttt{.}\ expr & \\
 & | & expr1 & \\
expr1 & \rightarrow & expr2\ \texttt{|}\ expr1 & \\
 & | & expr2 & \\
expr2 & \rightarrow & expr3\ \texttt{\&}\ expr2 & \\
 & | & expr3 & \\
expr3 & \rightarrow & expr4\ relop\ expr4 & \\
 & | & expr4 & \\
expr4 & \rightarrow & expr5\ \texttt{+}\ expr4 & \\
 & | & expr5\ \texttt{-}\ expr5 & \\
 & | & expr5 & \\
expr5 & \rightarrow & expr6\ \texttt{*}\ expr5 & \\
 & | & expr6\ \texttt{/}\ expr6 & \\
 & | & expr6 & \\
expr6 & \rightarrow & aexpr_1 \ldots aexpr_n & (n \geq 1)
\end{array}
$$

この文法が、| と & は右結合的であるのに対し、関係演算子は非結合的であるという事実をどのように表現しているかに注目してください。これほど多くの規則を記述するのはかなり面倒ですが、これは最初から意図していたことを明示的に示しているだけです。しかしここで 2 つ目の問題が発生します。これらの規則から直接実装されたパーサーは、恐ろしく非効率になってしまうのです！*expr*1 の生成規則を考えてみましょう。単純なパーサーは、*expr*2 を認識しようとし、次にバーチカルバー | を探します。もしバーチカルバー | が見つからなかった場合 (よくあることですが)、*expr*2 を再び探すために、元の入力を苦労して再解析することになります。さらに悪いことに、*expr*2 を解析するたびに、*expr*3 を解析する試行が 2 回必要になり、その結果、*expr*4 を解析する試行が 4 回必要になるなど、状況は続きます。

2 つの生成規則の間で *expr*2 の解析を共有したいのですが、これは *expr*1 生成規則を 2 つに分割することで簡単に実現できます。

$$
\begin{array}{lcl}
expr1 & \rightarrow & expr2\ expr1c \\
expr1c & \rightarrow & \texttt{|}\ expr1 \\
 & | & \epsilon
\end{array}
$$

ここで、ϵ は空文字列を表します。*expr*1 の生成規則では、*expr*1 はバーチカルバー | の後に *expr*1 が続くか、空であるかのいずれかであるとされています。もうすぐです！最後の質問は、*expr*1 のパーサの型は何か、という

ことです。| *expr*1 という句は式の一部に過ぎず、空文字列 ϵ も式ではないため、型 parser coreExpr にはなり得ません。いつものように、文法の変換によって構造が破壊されています。

解決策はかなり簡単です。このような新しいデータ型 partialExpr を定義します

```
data PartialExpr = NoOp
                 | FoundOp Name CoreExpr
```

これで、次のように *expr1c* のパーサを定義できます。

```
pExpr1c :: Parser PartialExpr
pExpr1c = (pThen FoundOp (pLit "|") pExpr1) `pAlt` (pEmpty NoOp)
```

expr1 のパーサは、pExpr1 によって返される中間結果を分解します：

```
pExpr1 :: Parser CoreExpr
pExpr1 = pThen assembleOp pExpr2 pExpr1c

assembleOp :: CoreExpr -> PartialExpr -> CoreExpr
assembleOp e1 NoOp = e1
assembleOp e1 (FoundOp op e2) = EAp (EAp (EVar op) e1) e2
```

演習 1.24 提案された行に沿って文法を変換し、変更を Miranda コードに直訳して、結果のパーサをテストします。

1.6.7 まとめ

パーサ構築関数のライブラリによって効率的に処理できる文法は、LL(1) 文法と呼ばれます。これは、従来の再帰降下パーサによって処理できるものとまったく同じクラスです [Aho et al. 1986]。

ライブラリを使用すると、非常に簡潔なパーサを簡単に作成できます。ほとんどすべてのプログラムには何らかの入力言語があり、プログラムによって解析される必要があるため、これは重要で便利なプロパティです。

気をつけなければならないことはたくさんありますが (左再帰、演算子の優先順位、共有)、実装されている言語に関係なく、再帰降下パーサではまったく同じ問題が発生します。

```
module Template where
import Language
import Utils
```

第2章　テンプレートのインスタンス化

この章では、関数型言語の可能な限りの単純な実装、つまりテンプレートのインスタンス化に基づくグラフ簡約器を紹介します。

初期バージョン (マーク 1) の完全なソースコードが示され、その後に基本設計の一連の改善とバリエーションが続きます。まず、グラフの簡約とテンプレートのインスタンス化のレビューから始めます。

2.1　テンプレートインスタンス化のプレビュー

テンプレートのインスタンス化かの概要から始めます。この資料は、[Peyton Jones 1987] の第 11 章と第 12 章で詳しく説明されています。

次の重要な事実を思い出します。

- 関数型プログラムは、**式を評価する**ことによって「実行」されます。
- 式は**グラフ**で表されます。
- 評価は、一連の**簡約**を実行することによって行われます。
- 簡約は、グラフ内の**簡約可能な式**をその簡約形式に置き換えます (または**更新**します)。「簡約可能な式」という用語は、しばしば「redex」と省略されます。
- redex がなくなると、評価は完了です。式は**正規形**であると言います。
- 評価される式にはいつでも複数の redex が含まれる可能性があるため、次にどれを簡約するかを選択できます。幸い、どの簡約手順を選択しても、常に同じ答え (つまり、正規形) が得られます。注意点が 1 つあります。一部の簡約手順は終了しない場合があります。
- ただし、どの redex を選択しても評価が終了する場合、常に最も外側の redex を選択するというポリシーも終了します。この簡約順序の選択は、**最外最左簡約 (normal order reduction)** と呼ばれ、常に使用されるものです。

従って、評価のプロセスは次のように説明できます。

```
until there are more redexes
      select the outermost redex
      reduce it
      update the (root of the) redex with the result
end
```

2.1.1 実例

例として、次のコア言語プログラムについて考えてみます。

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

このプログラムは、スーパーコンビネータと呼ばれる一連の定義で構成されています。square と main はどちらもスーパーコンビネータです。慣例により、評価される四季はスーパーコンビネータ main です。従って、最初に評価される式は、次のかなり些細なツリーで表されます (ツリーは単なる特別な種類のグラフであることを忘れないでください)。

```
main
```

ここで、main には引数がないため、main 自体が redex となり、main をその本体で置き換えます。

```
main            reduces to      @
                                / \
                           square   @
                                   / \
                              square   3
```

これらの図と後に続くすべての図では、関数適用は@記号で表されています。

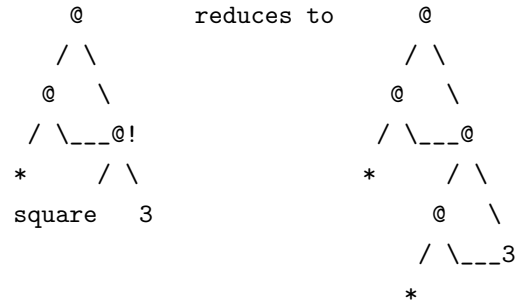
ここで、最も外側の redex は外側の関数適用 (square の適用) です。関数適用を減らすために、redex を関数本体のインスタンスに置き換え、仮引数が出現するたびに引数へのポインタを置き換えます。

```
       @!       reduces to      @!
      / \                      / \
square   @                    @   \
        / \                  / \___@
  square   3                *     / \
                            square   3
```

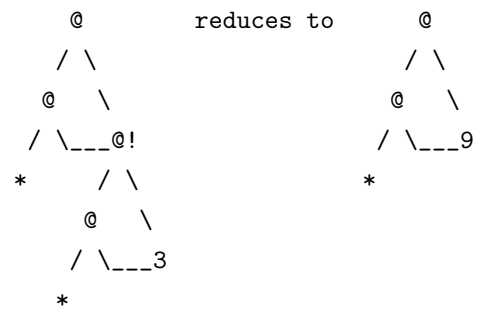
結果で上書きされる redex のルートは、!でマークされます。内側の square 3 の redex が共有され、ツリーがグラフになっていることに注意してください。

square の定義では、式 x*x(*は中置演算子として書かれています) は ((* x) x) の略で、2 つの引数に*を適用します。**カリー化**を使用して、1 引数の関数適用の形で複数の引数を持つ関数を記述します。*は、引数 p に適用されたとき、関数を返す関数となります。(* p) が返す関数は、別の引数 q に適用されると p と q の積を返すような関数です。

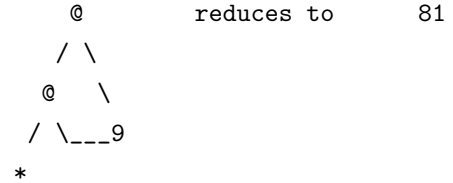
現在、唯一の redex は内側の関数適用 (square の 3 への適用) です。*の引数を評価する必要があるため、*の適用は簡約できません。内側の関数適用は次のように簡約されます。



まだ 1 つの redex、内側の乗算があります。redex を乗算結果の 9 に置き換えます。



redex のルートを結果で物理的に更新することにより、外側の乗算の両方の引数が内側の乗算の結果を「参照」することに注意してください。最終的な簡約は簡単です。



2.1.2　3 つのステップ

前に見たように、グラフ簡約は、正規形に達するまで次の 3 つのステップを繰り返すことで構成されます。

1. 次の redex を見つける。
2. それを簡約する。
3. 簡約結果で redex(のルート) を更新する。

前のセクションの例からわかるように、2 種類の redex があり、さまざまな方法で簡約されます。

スーパーコンビネータ 最も外側の関数適用がスーパーコンビネータの適用である場合、それは確かに redex でもあり、以下に説明するように簡約することができます (セクション 2.1.4)。

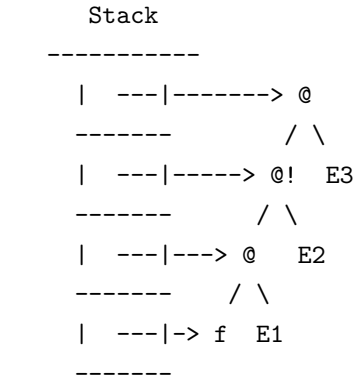
組み込みのプリミティブ 最も外側の関数適用が組み込みプリミティブの適用である場合、引数が評価されるかどうかに応じて、適用は redex である場合とそうでない場合があります。redex でない場合は、引数を評価する必要があります。これは、まったく同じプロセスを使用して行われます。引数の最も外側の redex を繰り返し見つけて、それを簡約します。これが完了すると、外側の組み込みプリミティブ適用の簡約に戻ることができます。

2.1.3　次の redex を見つけるためのスパインのアンワインド

簡約サイクルの最初のステップは、実行する次の簡約の部位を見つけることです。つまり、最も外側の簡約可能な関数適用です。次のように、最も外側の関数適用を見つけるのは簡単です（ただし、簡約できない場合があります)。

1. スーパーコンビネータまたは組み込みプリミティブに到達するまで、ルートから開始して、関数適用ノードの左側の枝をたどります。この関数適用ノードの左分岐チェーンはエクスプレッションの**スパイン**と呼ばれ、このプロセスはスパインの**アンワインド**と呼ばれます。通常、**スタック**は、途中で遭遇したノードのアドレスを記憶するために使用されます。
2. ここで、スーパーコンビネータまたはプリミティブが取る引数の数を確認し、その数の関数適用ノードに戻ります。これで、最も外側の関数適用のルートが見つかりました。

たとえば、式 (f E1 E2 E3) では、f は 2 つの引数を取ります。たとえば、最も外側の関数適用は (f E1 E2) です。式とスタックは次のようになります。

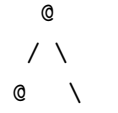


最も外側の関数適用 (のルート) は!でマークされています。

f が 2 つではなく 4 つの引数をとった場合のように、評価の結果が部分適用である可能性がある場合は、上記の手順 2 の前に、スパインに十分な関数適用ノードがあることを確認する必要があります。そうでない場合、式は Weak Head Normal Form (WHNF) に達しています。サブ式 E1、E2、および E3 にはまだ redexe が含まれている可能性がありますが、ほとんどの評価器は、サブ式も簡約しようとするのではなく、WHNF に到達すると停止します。プログラムが型チェックされており、結果が数値、たとえば、リストであることが保証されている場合、このアンダーフローチェックは省略できます。

最も外側の関数適用のルートのみが見つかったことに注意してください。それは同様に redex であるかもしれないしそうでないかもしれません。関数がスーパーコンビネータである場合、それは確かに redex になりますが、+などのプリミティブである場合、引数が評価されるかどうかによって異なります。もしスーパーコンビネータなら、私たちは最も外側の redex を見つけました。スーパーコンビネータでない場合は、まだやるべきことがあります。

プリミティブが現在評価されていない引数の値を必要とする場合、プリミティブの簡約を進める前に引数を評価する必要があります。これを行うには、以前と同じように、現在のスタックを片側に配置し、引数を減らすために新しいスタックから開始する必要があります。これは、前のセクションの例でステージに到達したときの状況でした。



```
 / \___@!
*     / \
square   3
```

新しいスタックで引数 (square 3) を評価する必要があります。この評価中に、未評価の引数を持つプリミティブに再び遭遇する可能性があるため、新しい評価を再度開始する必要があります。正しい順序でスタックに戻ることができるように、すべての「古い」スタックを追跡する必要があります。これは、ダンプと呼ばれるスタックのスタックを保持することによって便利に行われます。引数を評価する必要がある場合は、現在のスタックをダンプにプッシュします。評価が終了したら、古いスタックをダンプから取り出します。

もちろん、実際の実装では、スタック全体をコピーすることはありません。「新しい」スタックは「古い」スタックが再び必要になる前に終了するため、「新しい」スタックは「古い」スタックの上に物理的に構築できます。ダンプスタックは、「新しい」と「古い」の境界がどこにあったかを追跡するだけです。ただし、概念的には、ダンプはスタックのスタックであり、この方法でモデル化します。

2.1.4　スーパーコンビネータ redex

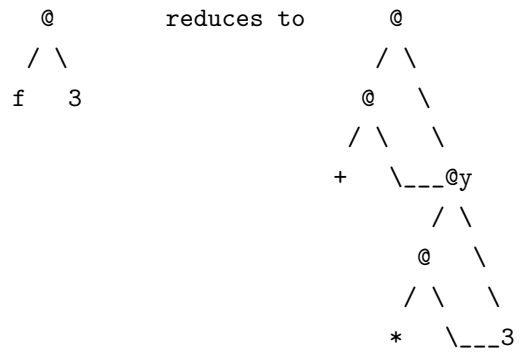
スーパーコンビネータ redex は、引数を本体に置き換えることで簡約されます。より正確には：

> スーパーコンビネータの簡約。**スーパーコンビネータ redex は、redex をスーパーコンビネータ本体のインスタンスに置き換え、実引数へのポインタで対応する仮引数の発生箇所を置き換えることによって簡約されます。引数はコピーされないことに注意してください。むしろ、それらへのポインタを使用するデバイスによって、それらは共有されます。**

スーパーコンビネータ本体には、let 式と letrec 式を含めることができます。例えば：

```
f x = let y = x*x
      in y+y
```

let および letrec 式は、**グラフのテキスト記述**として扱われます。ここでは、たとえば、f の定義の可能な使用法があります。



let 式は、y という名前の部分式 x*x を定義します。let 式の本体 y+y は、y の代わりに部分式へのポインタを使用します。したがって、通常の表現は木を表します。式に循環グラフを記述させます。および letrec 式は循環グラフを記述します。

2.1.5 更新

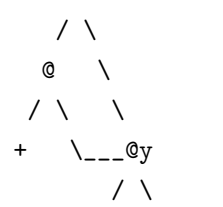
簡約を実行した後、結果で redex のルートを更新する必要があります。これにより、redex が共有されている場合 (例 (sqrt (sqrt 3)) のように)、簡約は 1 回だけ実行されます。この更新は、**遅延評価**の本質です。redex はまったく評価されない場合がありますが、評価された場合、更新により、そのコストが最大で 1 回発生することが保証されます。

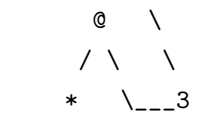
更新を省略してもエラーは発生しません。これは、一部の式が複数回評価される可能性があることを意味するだけであり、非効率的です。

更新を実行するときに少し注意が必要なケースが 1 つあります。以下のプログラムを検討します。

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

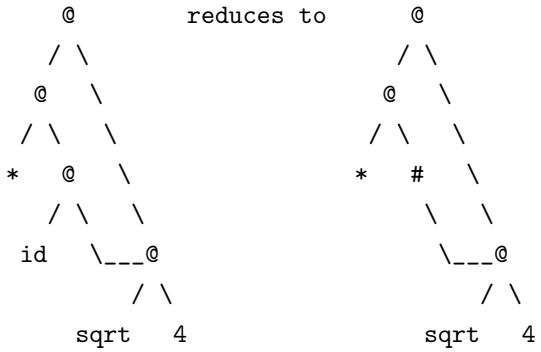
f の簡約が行われた後、グラフは次のようになります。





sqrt は、平方根を取るための組み込みプリミティブであると想定しています。ここで、次に選択された redex が * の最初の引数、つまり id の関数適用であると仮定します。(どちらの引数も通常の形式ではないため、* の 2 番目の引数でも同様に適切である可能性がありますが、最初の引数であると想定します。) id 簡約を実行した後、redex のルートを何で上書きする必要がありますか？**(sqrt 4) が 2 回評価されるため、(sqrt 4) 関数適用ノードのコピーで上書きしないでください。**

このジレンマから抜け出す最も簡単な方法は、新しい種類のグラフノードである間接ノードを追加することです。これは # 記号で示されます。間接ノードを使用して、簡約の結果を指すように redex のルートを更新できます。



[Peyton Jones 1987] のセクション 12.4 には、更新に関連する問題の詳細な説明が含まれています。

2.1.6　定作用形式 †

一部のスーパーコンビネータには引数がありません。それらは、定作用形式または CAF と呼ばれます。たとえば、以下の fac20 は CAF です。

```
fac20 = factorial 20
```

CAF の興味深い点は、スーパーコンビネータ自体が redex であるということです。fac20 が呼び出されるたびに、factorial 20 の新しいコピーをインスタンス化する必要はありません。これは、factorial 20 の計算を繰り返すことを意味するためです。むしろ、スーパーコンビネータ fac20 は fac20 簡約のルートであり、その本体をインスタンス化した結果で上書きする必要があります。

実際の結果は、スーパーコンビネータを通常の方法で更新できるようにするために、グラフノードで表す必要があるということです。これは、各実装で実際に発生することがわかります。

これでグラフ簡約のレビューは終わりです。

2.2　状態遷移システム

ここで、グラフ簡約の実装に注意を向けます。**状態遷移システム**を使用して、各実装について説明します。このセクションでは、状態遷移システムを紹介します。

状態遷移システムは、シーケンシャルマシンの動作を説明するための表記法です。いつでも、マシンは指定された**初期状態**から始まるある**状態**にあります。マシンの状態が**状態遷移ルール**の 1 つと**一致する**場合、ルールはマシンの新しい状態を指定します。一致する状態遷移ルールがない場合、実行は停止します。複数のルールが一致する場合は、1 つが任意に選択されて発火します。その場合、マシンは**非決定論的**です。すべてのマシンは決定論的です。

これは、(かなり非効率的な) 乗算器を指定するために使用される状態遷移システムの簡単な例です。状態は 4 つ組 **(n, m, d, t)** です。掛ける数は **n** と **m**、現在の合計は **t** で、マシンは状態 **(n, m, 0, 0)** に初期化されます。

マシンの動作は、2 つの遷移ルールによって指定されます。最初のルールで指定されているように、**d** は **0** に向かって繰り返しデクリメントされると同時に、**t** をインクリメントします。

	n	m	d	t
$\Longrightarrow$	n	m	d - 1	t + 1
	where d $>$ 0			

古い状態の同じ値の真下に新しい状態の各値を使用して遷移ルールを常に作成するため、どの値が変更されたかを簡単に確認できます。

d が **0** に達すると、再び **n** に初期化され、**m** が 0 に達するまで m がデクリメントされます。これは、2 番目のルールで指定されています。

	n	m	0	t
$\Longrightarrow$	n	m - 1	n	t + 1
	where m $>$ 0			

ルールが適用されない場合、マシンは終了します。この時点で、状態（**n, 0, 0, t**）になります。ここで、**t** は初期状態からの **n** と **m** の積です。

演習 2.1 初期状態（2; 3; 0; 0）から開始して、各ステップでどのルールを実行するかを指定して、マシンを手動で実行しなさい。最終状態が（2; 0; 0; 6）であることを確認しなさい。

演習 2.2 一連の状態の不変量は、すべての状態に当てはまる述語です。**n** と **m** の初期値（**N** と **M** と呼びます）と **m**、**d**、**t** の現在の値との関係を表す不変量を見つけなさい。次に、マシンが乗算を実行するという推測を**証明**しなさい。証明を行うには、以下を示す必要があります。

1. 不変量は初期状態では真になる。
2. 不変量が特定の状態で真になれば、次の状態でも真になる。
3. 不変量と終了条件 $(m = d = 0)$ が与えられると、$t = N * M$ になる。
4. マシンは終了する。

状態遷移システムは、次の理由で私たちの目的に便利です。

- それらは十分に**抽象的**であるため、非常に低レベルの詳細に巻き込まれることがない。
- それらは十分に**具体的**であるため、ルールに多くの複雑さを隠すことで「不正行為」をしていないことを確認できる。
- 状態遷移システムを直接 Miranda に書き直して、システムの実行可能な実装を提供できる。

最後のポイントを説明するために、乗算器を Miranda に書き直します。このマシンの状態の型を定義するために、型シノニムを与えることから始めます。

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

次に、関数 evalMult は状態を取得し、その状態とそれに続くすべての状態で構成されるリストを返します。

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

関数 `stepMult` は非最終状態を取り、次の状態を返します。遷移ルールごとに `stepMult` の式が 1 つあります。

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

関数 `multFinal` は状態を取得し、その状態が最終状態であるかどうかをテストします。

```
multFinal :: MultState -> Bool
```

演習 2.3 関数 multFinal を定義し、結果のマシンを初期状態 (2, 3, 0, 0) で実行し、結果リストの最後の状態が (2, 0, 0, 6) であることを確認します。標準関数 layn は、結果をより読みやすくレイアウトするのに役立つ場合があります。

```
multFinal state = m == 0 && d == 0
                  where (_, m, d, _) = state

*Main> evalMult (2, 3, 0, 0)
[(2,3,0,0),(2,2,2,0),(2,2,1,1),(2,2,0,2),
           (2,1,2,2),(2,1,1,3),(2,1,0,4),
           (2,0,2,4),(2,0,1,5),(2,0,0,6)]
```

2.3 Mark 1: 最小限のテンプレートインスタンス化グラフ簡約器

これで、かなり単純なグラフ簡約器の定義を開始する準備が整いました。シンプルですが、より洗練されたグラフ簡約器が持つ多くの部分が含まれているため、説明するのに数ページかかります。

2.3.1 グラフ簡約の遷移規則

テンプレートインスタンス化グラフ簡約マシンの状態は、4 つ組

(スタック、ダンプ、ヒープ、グローバル)

または略して **(s, d, h, f)** です。

- **スタック**は**アドレス**のスタックであり、各アドレスはヒープ内の**ノード**を識別します。これらのノードは、評価される式のスパインを形成します。 表記 $a_1 : s$ は、最上位の要素が a_1 で、残りが s であるスタックを示します。
- **ダンプ**は、厳密なプリミティブの引数を評価する前に、スパインスタックの状態を記録します。ダンプは Mark1 マシンではまったく使用されませんが、後続のバージョンでは役立ちます。
- **ヒープ**は、タグ付けされた**ノード**のコレクションです。表記 $h[a: node]$ は、ヒープ h 内のアドレス a がノード $node$ を指すことを意味します。
- 各スーパーコンビネータ (および後で各プリミティブ) について、**グローバル**はスーパーコンビネータ (またはプリミティブ) を表すヒープノードのアドレスを提供します。

ヒープノードは、次の 3 つの形式のいずれかを取ることができます (最も原始的なマシンの場合)。

- `NAp` a_1 a_2 は、アドレスが a_1 であるノードからアドレスが a_2 であるノードへの関数適用を表します。
- `NSupercomb` $args$ $body$ は、引数 $args$ と本体 $body$ を持つスーパーコンビネータを表します。
- `NNum` n は、数 n を表します。

このプリミティブテンプレートインスタンス化マシンには、2 つの状態遷移ルールしかありません。1 つ目は、単一のアプリケーションノードをスパインスタックに巻き戻す方法を説明しています。

(2.1)		$a : s$	d	$h[a : \texttt{NAp}\ a_1\ a_2]$		f
	$\Longrightarrow$	$a_1 : a : s$	d	h		f

(このルールの 2 行目のヒープコンポーネントには、アドレス a から $\texttt{NAp}\ a_1\ a_2$ へのマッピングが含まれていますが、混乱を避けるために、再度書き出すことはありません。) このルールを繰り返し適用すると、スタックの最上位のノードが NAp ノードでなくなるまで、式のスパイン全体がスタックに巻き戻されます。

2 番目のルールは、スーパーコンビネータの簡約 を実行する方法を説明しています。

(2.2)		$a_0 : a_1 : \ldots : a_n : s$	d	$h[a_0 : \texttt{NSupercomb}\ [x_1, \ldots, x_n]\ body]$	f
	$\Longrightarrow$	$a_r : s$	d	h'	f
		where $(h',\ a_r) = instantiate\ body\ h\ f[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$			

このルールへの関心のほとんどは、関数のインスタンス化の中に隠されています。その引数は次のとおりです。

- インスタンス化する式
- ヒープ
- スタックから取得したアドレスへの引数名のマッピングによって拡張された、ヒープアドレスへの名前のグローバルマッピング f

新しいヒープと、新しく構築されたインスタンス（のルート）のアドレスを返します。このような強力な操作は、各ステップが単純なアトミックアクションであることが意図されている状態遷移システムの精神とは実際には異なりますが、それがテンプレートインスタンス化マシンの性質です。後の章の実装はすべて、真にアトミックなアクションを持ちます！

redex のルート自体はこのルールの影響を受けないことに注意してください。スタック上で結果のルートに置き換えられるだけです。言い換えると、これらのルールは、グラフ簡約マシンではなく、redex のルートを更新しないツリー簡約マシンを記述しています。これについては、セクション 2.5 で後ほど改善します。

2.3.2 実装の構造

マシンの仕様がわかったので、その実装に着手する準備が整いました。

関数型言語で実装を記述しているので、たとえば、ジョブを実行するために関数 run を記述しなければなりません。そのタイプはどうあるべきですか？それは、ファイル名を取り、その中でプログラムを実行し、結果を出力する必要があります。これは、最終結果またはある種の実行トレースのいずれかである可能性があります。したがって、run のタイプは、次の型シグネチャによって指定されます。

```
runProg :: [Char] -> [Char] -- name changed to not conflict
```

これで、run がどのように構築されるかを考えることができます。プログラムの実行は、次の 4 つの段階で構成されます。

1. 指定されたファイルにある式からプログラムを解析します。parse 関数はファイル名を受け取り、解析されたプログラムを返します。

   ```
   parse :: [Char] -> CoreProgram
   ```

2. プログラムを実行に適した形式に変換します。このタスクを実行する compile 関数は、プログラムを受け取り、テンプレートインスタンス化マシンの初期状態を生成します。

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

   tiState は、テンプレートインスタンス化マシンの状態の型です。(接頭辞「ti」はテンプレートのインスタンス化の略です。)

3. 最終状態に達するまで状態遷移を繰り返し実行して、プログラムを実行します。結果は、通過したすべての状態のリストです。これから、その後、最終状態を抽出するか、すべての状態のトレースを取得できます。今のところ、結果として数値を返すプログラムに限定するので、この実行関数を eval と呼びます。

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

4. 印刷用に結果をフォーマットします。これは、印刷する情報を選択し、それを文字のリストにフォーマットする関数 showResults によって実行されます。

   ```
   showResults :: [TiState] -> [Char]
   ```

関数 run は、次の 4 つの関数を組み合わせたものです。

```
runProg = showResults . eval . compile . parse -- ‘‘run’’: name conflict
```

これらの各フェーズにサブセクションを割り当てます。

2.3.3　パーサ

パーサ関数を含むソース言語は、第 1 章で定義されている別のモジュール language で定義されています。モジュールをインポートするために%include ディレクティブを使用して利用できるようにします。

```
-- import Language
```

2.3.4　コンパイラ

このセクションでは、compile 関数を定義します。utils モジュールで定義されたデータ型と関数が必要になるため、%include を使用して使用できるようにします。

```
-- import Utils
```

次に、コンパイラが操作するデータ型の表現を検討する必要があります。

データ型

コンパイラは、型 tiState を持つマシンの初期状態を生成するため、次に行うことは、型シノニムを使用して、マシンの状態がどのように表されるかを定義することです。

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

マシンの状態は、最初の 4 つのコンポーネントがセクション 2.3.1 で指定されたコンポーネントに正確に対応し、5 番目のコンポーネントが統計の累積に使用される 5 つです。

次に、これらの各コンポーネントの表現を検討する必要があります。

- **スパインスタック**は、**ヒープアドレス**の単なるスタックです。

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

スタックをリストとして表すことを選択します。スタックの要素は、utils モジュール (付録 A.1) で定義された抽象データ型 addr のメンバーです。これらはヒープアドレスを表し、抽象化することで、utils モジュールによって提供される操作のみを使用できるようにします。したがって、たとえば誤ってアドレスに 1 つ追加することは不可能です。

- **ダンプ**はセクション 2.6 まで必要ありませんが、後で追加すると状態遷移ルールに多くの面倒な変更が必要になるため、すでに状態の一部にしています。今のところ、引数のない単一のコンストラクタで構成される簡単な型の定義を与えます。

```
data TiDump = DummyTiDump
initialTiDump = DummyTiDump
```

- **ヒープ**は、utils モジュールで定義された heap 抽象データ型によって表されます。ヒープに含まれるもの、つまり node 型のオブジェクト (まだ定義されていません) を指定する必要があります。

```
type TiHeap = Heap Node
```

ヒープ node は、次の代数的データ型宣言で表されます。これは、セクション 2.3.1 に記載されている可能性のリストに対応しています。

```
data Node = NAp Addr Addr                          -- Application
          | NSupercomp Name [Name] CoreExpr -- Supercombinator
          | NNum Int                               -- A number
```

唯一の違いは、スーパーコンビネータの名前を保持するために使用される NSupercomb コンストラクタにタイプ name のフィールドを追加したことです。これは、文書化とデバッグの目的でのみ使用されます。

- **グローバル**コンポーネントは、各スーパーコンビネータ名を、その定義を含むヒープノードのアドレスに関連付けます。

```
type TiGlobals = ASSOC Name Addr
```

assoc タイプは、その操作とともに utils モジュールで定義されます (付録 A.2)。リストの組み込み構文を使用して関連付けを操作できると非常に便利なため、実際には (抽象データ型ではなく) 型シノニムとして定義されています。ここでは、抽象化とプログラミングの容易さの間に緊張関係があります。

- 状態の tiStats コンポーネントは遷移ルールには記載されていませんが、これを使用して、マシンの動作に関する実行時のパフォーマンス統計を収集します。収集する統計を簡単に変更できるように、抽象型にします。まず、実行したステップ数のみを記録します。

```
tiStatInitial  :: TiStats
tiStatIncSteps :: TiStats -> TiStats
tiStatGetSteps :: TiStats -> Int
```

実装はかなり単純です。

```
type TiStats = Int
tiStatInitial    = 0
tiStatIncSteps s = s + 1
tiStatGetSteps s = s
```

便利な関数 applyToStats は、特定の関数を状態の統計コンポーネントに適用します。

```
applyToStats :: (TiState -> TiState) -> TiState -> TiState
applyToStats stats_fun (stack, dump, heap, sc_defs, stats)
  = (stack, dump, heap, sc_defs, stats_fun stats)
\end{itemize}
```

これで、関連するデータ型の定義が完了しました。

コンパイラ自体

コンパイラの仕事は、プログラムを取得し、そこからマシンの初期状態を作成することです。

```
compile program
  = (initial_stack, initialTiDump, initial_heap, globals, tiStatInitial)
    where
      sc_defs = program ++ preludeDefs ++ extraPreludeDefs

      (initial_heap, globals) = buildInitialHeap sc_defs

      initial_stack = [address_of_main]
      address_of_main = aLookup globals "main" (error "main is not defined")
```

where 節の各定義を順番に考えてみましょう。最初の sc_defs は、プログラムに含まれるすべてのスーパーコンビネータ定義のリストにすぎません。preludeDefs はセクション 1.4 で定義されており、すべてのプログラムに常に含まれている標準のスーパーコンビネータ定義のリストであることを思い出してください。extraPreludeDefs は、追加したい標準関数のリストです。今のところそれは空です：

```
extraPreludeDefs = []
```

2 番目の定義は、補助関数 buildInitialHeap を使用して、各スーパーコンビネータの名前をそのノードのアドレスにマップする関連付けリスト globals とともに、各スーパーコンビネータの NSupercomb ノードを含む初期ヒープを構築します。

最後に、initial_stack は、globals から取得されたスーパーコンビネータ main のノードのアドレスである 1 つの項目のみを含むように定義されています。

次に、buildInitialHeap の定義を検討する必要があります。これは少し注意が必要です。リスト sc_defs の要素ごとに何かを行う必要がありますが、厄介なのは、「何か」にヒープの割り当てが含まれることです。各ヒープ割り当ては新しいヒープを生成するため、sc_defs の 1 つの要素から次の要素にヒープを渡す方法を見つける必要があります。このプロセスは、空のヒープ hInitial(付録 A.1) から始まります。

このアイデアを高階関数 mapAccuml にカプセル化します。これは、この本で非常に多く使用します。mapAccuml は 3 つの引数を取ります：「処理関数」f、「アキュムレータ」acc、およびリスト $[x_1, \ldots, x_n]$ です。入力リストの各要素を受け取り、それと現在のアキュムレータに f を適用します。f は、結果のペア、結果リストの要素、およびアキュムレータの新しい値を返します。mapAccuml は、f の 1 つの呼び出しから次の呼び出しにアキュムレータを渡し、最終的に 2 つの結果を返します：「アキュムレータの最終値」acc' と、「結果リスト」$[y_1, \ldots, y_n]$ です。図 2.1 は、この配管を示しています。mapAccuml の定義は、付録 A.5 に記載されています。

この場合、「アキュムレータ」はヒープであり、初期値は hInitial です。リスト $[x_1, \ldots, x_n]$ はスーパーコンビネータ定義、sc_defs であり、結果リスト $[y_1, \ldots, y_n]$ は、スーパーコンビネータの名前とアドレス、sc_addrs の関連付けです。ここに、buildInitialHeap の定義があります。

```
buildInitialHeap :: [CoreScDefn] -> (TiHeap, TiGlobals)
```

```
buildInitialHeap sc_defs = mapAccuml allocateSc hInitial sc_defs
```

allocateSC と呼ぶ「処理関数」は、単一のスーパーコンビネータを割り当て、新しいヒープと sc_addrs アソシエーションリストのメンバーを返します。

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

これでコンパイラの定義は完了です。次に、評価器に注意を向けます。

2.3.5　評価器

評価器の eval は、マシンの初期状態を取得し、一度に 1 ステップずつマシンを実行して、通過したすべての状態のリストを返します。

eval は、常に現在の状態を結果の最初の要素として返します。現在の状態が最終状態の場合、それ以上の状態は返されません。それ以外の場合、eval は次の状態に再帰的に適用されます。後者は、(step を使用して) 単一のステップを実行し、次に doAdmin を呼び出して、ステップ間に必要な管理作業を実行することによって取得されます。

```
eval state = state : rest_states
             where
               rest_states | tiFinal state = []
                           | otherwise     = eval next_state
               next_state  = doAdmin (step state)

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

最終状態のテスト

関数 tiFinal は最終状態を検出します。スタックに単一のオブジェクトが含まれていて、それが数値またはデータオブジェクトのいずれかである場合にのみ、終了します。

```
tiFinal :: TiState -> Bool

tiFinal ([sole_addr], dump, heap, globals, stats)
  = isDataNode (hLookup heap sole_addr)

tiFinal ([], dump, heap, globals, stats) = error "Empty stack!"
tiFinal state = False --Stack contains more than one item
```

スタック要素はアドレスであることに注意してください。これは、数値であるかどうかを確認する前に、ヒープで検索する必要があります。スタックが空である必要がある場合 (これは決して発生しないはずです) にも、適切なエラーメッセージを生成する必要があります。

最後に、isDataNode を定義できます。

```
isDataNode :: Node -> Bool
isDataNode (NNum n) = True
isDataNode node     = False
```

ステップを進める

関数 step は、ある状態をその後続状態に写像します。

```
step :: TiState -> TiState
```

スパインスタックの最上位のノードでケース分析を行う必要があるため、ヒープからこのノードを抽出し、dispatch を使用して適切な関数を呼び出し、ノードの各形式のハードワークを実行します。

```
step state
  = dispatch (hLookup heap (hd stack))
      where
        (stack, dump, heap, globals, stats) = state
        dispatch (NNum n)                   = numStep state n
        dispatch (NAp a1 a2)                = apStep  state a1 a2  -- 状
態遷移ルール 2.1
        dispatch (NSupercomb sc args body)  = scStep  state sc args body -- 状
態遷移ルール 2.2
```

数値や関数適用のケースもほとんどトラブルなく対応できます。数値を関数として適用してはならないため、スタックの最上位に数値があるのはエラーです。(スタック上の唯一のオブジェクトである場合、実行は tiFinal によって停止されます。)

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

関数適用ノードの処理は、アンワインドルール (ルール 2.1) で説明されており、Miranda に直接変換できます。

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

スーパーコンビネータの適用

スーパーコンビネータを適用するには、その本体をインスタンス化し、引数名をスタック内の引数アドレスにバインドする必要があります (ルール 2.2)。次に、redex のルートを含む引数をスタックから破棄し、代わりにリダクションの結果 (のルート) をスタックにプッシュします。(この最初のバージョンのマシンでは、更新を実行していないことに注意してください。)

```
scStep :: TiState -> Name -> [Name] -> CoreExpr -> TiState
scStep (stack, dump, heap, globals, stats) sc_name arg_names body
  = (new_stack, dump, new_heap, globals, stats)
    where
      new_stack               = result_addr : (drop (length arg_names + 1) stack)
      (new_heap, result_addr) = instantiate body heap env
      env                     = arg_bindings ++ globals
      arg_bindings            = zip2 arg_names (getargs heap stack)
```

スーパーコンビネータとプリミティブを適用するには、補助関数が必要です。関数 getArgs は、スタック (関数適用ノードのスタックの最上位にあるスーパーコンビネータで構成されている必要があります) を受け取り、スタック上の各関数適用ノードの引数から形成されたリストを返します。

```
-- now getargs since getArgs conflicts with Gofer standard.prelude
getargs :: TiHeap -> TiStack -> [Addr]
getargs heap (sc : stack)
  = map get_arg stack
    where
      get_arg addr = arg
                     where
                       (NAp fun arg) = hLookup heap addr
```

instantiate 関数は、式、ヒープ、および名前とアドレスを関連付ける環境を取ります。ヒープ内に式のインスタンスを作成し、インスタンスのルートの新しいヒープとアドレスを返します。この環境は、instantiate によって、スーパーコンビネータとローカル変数の代わりに使用されるアドレスを指定するために使用されます。

```
instantiate :: CoreExpr           -- Body of supercombinator
               -> TiHeap          -- Heap before instantiation
               -> ASSOC Name Addr -- Association of names to addresses
               -> (TiHeap, Addr)  -- Heap after instantiation,
                                  --   and address of root of instance
```

数字の場合は、非常に簡単です。

```
instantiate (ENum n) heap env = hAlloc heap (NNum n)
```

関数適用の場合も簡単です。 2 つのブランチをインスタンス化し、関数適用ノードを構築するだけです。instantiate の再帰呼び出しを介して、ヒープを「スレッド化」する方法に注目してください。つまり、instantiate の最初の呼び出しにヒープが与えられ、新しいヒープが生成されます。後者は、instantiate の 2 番目の呼び出しに与えられ、さらに別のヒープを生成します。後者は、新しい関数適用ノードが割り当てられるヒープであり、呼び出し元に返される最終ヒープを生成します。

```
instantiate (EAp e1 e2) heap env
  = hAlloc heap2 (NAp a1 a2)
    where
      (heap1, a1) = instantiate e1 heap  env
      (heap2, a2) = instantiate e2 heap1 env
```

変数の場合、指定された環境で名前を検索するだけで、バインディングが見つからない場合は適切なエラーメッセージが生成されます。

```
instantiate (EVar v) heap env
  = (heap, aLookup env v (error ("Undefined name " ++ show v)))
```

Appendix A.2 で定義されている aLookup は、連想リストで変数を検索しますが、検索が失敗した場合は 3 番目の引数を返します。

```
instantiate (EConstr tag arity) heap env
  = instantiateConstr tag arity heap env
instantiate (ELet isrec defs body) heap env
  = instantiateLet isrec defs body heap env
instantiate (ECase e alts) heap env
  = error "Can't instantiate case exprs"
```

コンストラクタと let(rec) 式をインスタンス化する問題を延期するために、補助関数 instantiateConstr と instantiateLet を呼び出します。これらはそれぞれ、現在のエラーを生成します。後で、それらを運用定義に置き換えます。最後に、後で説明するように、テンプレートマシンは case 式をまったく処理できません。

```
instantiateConstr tag arity heap env
  = error "Can't instantiate constructors yet"
instantiateLet isrec defs body heap env
  = error "Can't instantiate let(rec) yet"
```

2.3.6　結果の書式付け

eval からの出力は状態のリストであり、全体を表示するとかなり大量になります。さらに、ヒープとスタックは抽象オブジェクトであるため、Miranda はそれらをまったく表示しません。したがって、showResults 関数は、セクション 1.5 で紹介した iseq データ型を使用して、出力をフォーマットします。

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

スタックの内容を表示するだけで状態を表示します。各ステップの後にヒープ全体を表示するのは面倒なので、スタックから直接参照されるノードの内容を表示することで満足します。状態の他のコンポーネントは変更されないため、それらも表示しません。

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

スタック上のアドレスと、それが指すノードの内容を表示することにより、スタックを最上位の要素から最初に表示します。これらのノードのほとんどは関数適用ノードであり、これらのそれぞれについて、引数ノードの内容も表示します。

```
showStack :: TiHeap -> TiStack -> Iseq
showStack heap stack
  = iConcat [ iStr "stk [",
              iIndent (iInterleave iNewline (map show_stack_item stack)),
```

```
              iStr " ]"
            ]
    where
      show_stack_item addr
        = iConcat [ showFWAddr addr,
                    iStr ": ",
                    showStkNode heap (hLookup heap addr)
                  ]

showStkNode :: TiHeap -> Node -> Iseq
showStkNode heap (NAp fun_addr arg_addr)
  = iConcat [ iStr "NAp ",
              showFWAddr fun_addr,
              iStr " ",
              showFWAddr arg_addr,
              iStr " (",
              showNode (hLookup heap arg_addr),
              iStr ")"
            ]
showStkNode heap node = showNode node
```

showNode は、ノードの値を表示します。完全な値を出力するのではなく、NSupercomb ノード内に格納されている名前のみを出力します。実際、これが名前がこれらのノード内に保存される唯一の理由です。

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

showAddr :: Addr -> Iseq
showAddr addr = iStr (show addr)

showFWAddr :: Addr -> Iseq -- Show address in field of width 4
```

```
showFWAddr addr
  = iStr (space (4 - length str) ++ str)
    where
      str = show addr
```

showStats は、累積された統計を出力する責任があります。

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

演習 2.4 これまでに与えられた実装をテストします。適切なテストプログラムは次のとおりです。

```
main = S K K 3
```

結果は 3 になります。さらにいくつかのテストプログラムを考案し、それらが機能することを確認します。算術演算はまだ定義されていないことを忘れないでください。

演習 2.5 ヒープの内容全体を出力するように showState を変更します。(ヒント：ヒープ内のすべてのノードのアドレスを検出するには、hAddresses を使用します。) このようにして、ヒープが 1 つのステップから次のステップにどのように進化するかを確認できます。

演習 2.6 スーパーコンビネータまたはプリミティブが適用される引数が少なすぎると、scStep は失敗します。このケースを検出するために、適切なチェックとエラーメッセージを scStep に追加します。

演習 2.7 より多くの実行統計を収集するようにインタープリタを変更します。たとえば、次のような統計情報を累積できます。

- 簡約の数。おそらくスーパーコンビネータの簡約とプリミティブの簡約に分けられます。
- 各種類のヒープ操作の数、特に割り当て。これを行う最も便利な方法は、heap 抽象データ型を変更してこの情報自体を累積することですが、これは結果の一部として新しいヒープを返すヒープ操作でのみ機能します。

- 最大スタック深度。

演習 2.8 scStep の定義では、instantiate に渡される環境 env は次のように定義されます。

```
env = arg\_bindings ++ globals
```

++の引数を逆にすると、どのような違いがありますか？

演習 2.9 (少し注意が必要です。) 次の eval の定義は、与えられたものよりも明白だと思うかもしれません。

```
eval state = [state],                  tiFinal state
           = state : eval next_state, otherwise
```

(next_state は以前と同じように定義されています)。なぜこれが劣った定義なのですか？(ヒント：すべての状態が showResults によってフォーマットされていて、存在しないヒープノードにアクセスしようとするなど、tiFinal state を評価するときにエラーが発生した場合にどうなるかを考えてください。エラーの原因となった状態が出力されますか？そうでない場合は、なぜですか？)

2.4　Mark 2: let(rec) 式

最初の機能強化は、マシンが let および letrec 式を処理できるようにすることです。2.1.4 で説明したように、スーパーコンビネータの本体には、グラフのテキスト記述とみなされる let(rec) 式が含まれる場合があります。

したがって、実装に加える必要がある唯一の変更は、インスタンス化を拡張することであり、その結果、ELet コンストラクの等式が得られます。

演習 2.10 **非再帰的**な let 式用の instantiate 関数定義を追加します。インスタンス化するために必要なこと (ELet nonRecursive defs body) は次のとおりです。

1. defs の各定義の右辺をインスタンス化します。
2. 環境を拡張して、defs の名前を新しく構築されたインスタンスのアドレスにバインドします。
3. 拡張した環境と式本体を渡す instantiate を呼び出します。

これはまだ let 式のみを処理します。letrec 式をインスタンス化した結果は**閉路**グラフですが、let 式は閉路グラフを生成するもとになります。

演習 2.11 非再帰的 ELet 用 instantiate 関数の定義をコピーし、再帰的な場合に機能するように変更します (または、両方を処理するように定義を変更します)。(ヒント：ステップ 1 で、既存の環境の代わりに拡張環境 (ステップ 2 で構築) を渡してインスタンス化することを除いて、let の場合とまったく同じようにすべてを実行します。)

この演習のヒントは、ステップ 2 で作成された名前とアドレスのバインディングをステップ 1 への入力として使用する必要があるため、不思議に思われます。Miranda でこれを試してみると、すべてが完全に機能します。これは、他の非厳密な関数型言語と同様に、関数を呼び出す前に関数への入力を評価する必要がないためです。実際の実装では、letrec 内の各 (ルート) ノードが割り当てられるアドレスを計算し、この情報を反映するように環境を拡張してから、定義式の右辺をインスタンス化することにより、このトリックを「手動で」実行する必要があります。

実装が機能するかどうかを確認するためのテストプログラムを次に示します。

```
pair x y f = f x y ;
fst p = p K ;
snd p = p K1 ;
```

```
f x y = letrec
            a = pair x b ;
            b = pair y a
        in
        fst (snd (snd (snd a))) ;
main = f 3 4
```

結果は 4 になります。このプログラムがどのように機能するか理解できますか？(すべてはセクション 2.8.3 で明らかにされます。)

演習 2.12 次のプログラムを検討します。

```
main = letrec f = f x in f
```

このプログラムを実行するとどうなりますか？この問題は、Miranda のような強く型付けされた言語で発生する可能性がありますか？

2.5　Mark 3: 更新機能の追加

これまでのところ簡約器は更新を実行しないため、共有部分は何度も評価される可能性があります。

セクション 2.1.5 で説明した様に、問題を解決する最も簡単な方法は、結果を指す間接ノードで redex のルートを更新することです。

(2.3)	$a_0 : a_1 : \ldots : a_n : s \quad d \quad h[a_0 : \texttt{NSupercomb}\ [x_1, \ldots, x_n]\ body]$		f
	$\Longrightarrow \qquad a_r : s \quad d \quad h'[a_n : \texttt{NInd}\ a_r]$		f
	where $(h',\ a_r) = instantiate\ body\ h\ f[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$		

違いは、**インスタンス化**関数によって返されるヒープ h' は、ノード a_n(redex のルート) を a_r(**インスタンス化**によって返される結果のルート) への間接参照で上書きすることによってさらに変更されることです。スーパーコンビネータが CAF(セクション 2.1.6 を参照) の場合、$n = 0$ であり、変更されるノードはスーパーコンビネータノード自体であることに注意してください。

さらに 1 つの変更が必要です。スパインをほどくときに間接に遭遇する可能性があるため、この場合に対処するために新しいルールを追加する必要があります。

(2.4)	$a : s \quad d \quad h[a : \texttt{NInd}\ a_1]$		f
	$\Longrightarrow \quad a_1 : s \quad d \quad h$		f

間接ノードのアドレス a は、あたかもそこになかったかのように、スタックから削除されます。

これらの新しいルールを実装するには、いくつかのことを行う必要があります。

- 新しいノードコンストラクター `NInd` を `node` データ型に追加します。これにより、次の改訂された定義が得られます。

```
data Node = NAp Addr Addr                     -- Application
          | NSupercomb Name [Name] CoreExpr -- Supercombinator
          | NNum Int                          -- Number
          | NInd Addr                         -- Indirection
```

この余分なコンストラクターを考慮に入れるために、`showNode` に新しい定義式を追加する必要があります。

- 結果への間接参照で redex のルートを更新するために hUpdate を使用するように scStep を変更します (ルール 2.3)。
- 間接化に対処するために、dispatch の定義に新な定義式を追加します (ルール 2.4)。

演習 2.13 間接ノードで更新を実行するように変更を加えます。リダクションマシンの Mark1 バージョンと Mark3 バージョンの両方で次のプログラムを実行して、変更の効果を試してください。

```
id x = x ;
main = twice twice id 3
```

(twice は preludeDefs(セクション 1.4) で定義されていることを思い出してください。) 最初に手でそれを簡約することによって何が起こるかを理解するようにしてください。main を twice twice twice id3 と定義するとどうなりますか？

2.5.1 間接参照の数を減らす

多くの場合、instantiate によって (または、後で説明するように、プリミティブによって) 新しく作成されたノードへの間接参照を使用して、redex のルートを更新します。このような状況では、間接参照を使用するのではなく、結果のルートノードを redex のルートの上に直接構築する方が安全です。結果のルートが新しく作成されるため、これを行うことで共有が失われることはなく、余分な間接ノードを構築 (およびその後トラバース) する手間が省けます。

これを行うには、新しいインスタンス化関数 instantiateAndUpdate を定義します。これは、追加の引数、結果で更新されるノードのアドレスを取り、結果のグラフのアドレスを返さないことを除いて、instantiate と同じです。

```
instantiateAndUpdate
    :: CoreExpr           -- Body of supercombinator
       -> Addr            -- Address of node to update
       -> TiHeap          -- Heap before instantiation
       -> ASSOC Name Addr -- Associate parameters to addresses
       -> TiHeap          -- Heap after instantiation
```

たとえば、式が関数適用の場合のインスタンス化と更新の定義は次のとおりです。

```
instantiateAndUpdate (EAp e1 e2) upd_addr heap env
  = hUpdate heap2 upd_addr (NAp a1 a2)
    where
    (heap1, a1) = instantiate e1 heap env
    (heap2, a2) = instantiate e2 heap1 env
```

再帰的なインスタンス化は、古いインスタンス化によって引き続き実行されることに注意してください。ルートノードのみを更新する必要があります。

演習 2.14 `instanceiateAndUpdate` の定義を完了します。次の点には少し注意が必要です。

- インスタンス化される式が単純な変数である場合でも、間接参照を使用する必要があります。なんで？
- `let(rec)` 式の方程式の再帰的なインスタンス化について慎重に検討してください。

`scStep` を変更して、インスタンス化の代わりに `instantiateAndUpdate` を呼び出し、更新するノードのアドレスとして redex のルートを渡します。`scStep` 自体から更新コードを削除します。削減の数と割り当てられたヒープノードの数に対するこの変更の影響を測定します。

2.6 Mark 4: 算術演算機能の追加

このセクションでは、算術プリミティブを追加します。これには、初めてダンプを使用することが含まれます。

2.6.1 算術演算の遷移規則

2.6.2 算術演算の実装

算術演算を実装するには、いくつかの変更を加える必要があります。まず、型 tiDump を、初期値が空のスタックのスタックに再定義する必要があります。

```
type TiDump = [TiStack]
initialTiDump = []
```

次に、新しい種類のヒープノードを追加する必要があります。NPrim n p は、名前が n で、値が p であるプリミティブを表します。ここで、p はプリミティブタイプです。NSupercomb ノードの場合と同様に、名前はデバッグと文書化の理由でのみ NPrim ノードに存在します。

```
data Node = NAp Addr Addr                       -- Application
          | NSupercomb Name [Name] CoreExpr -- Supercombinator
          | NNum Int                            -- Number
          | NInd Addr                           -- Indirection
          | NPrim Name Primitive                -- Primitive
```

いつものように、NPrim ノードを表示するには、showNode も拡張する必要があります。前のセクションで示した遷移ルールは、データ型 primitive を次のように定義する必要があることを示しています。

```
data Primitive = Neg | Add | Sub | Mul | Div
```

必要なプリミティブごとに 1 つのコンストラクターを使用します。

ここで、各スーパーコンビネータの初期ヒープに NSupercomb ノードを割り当てる必要があるのと同じように、各プリミティブの初期ヒープに NPrim ノードを割り当てる必要があります。次に、マシン状態の**グローバル**コンポーネントに追加のバインディングを追加できます。これにより、スーパーコンビネータの場合と同様に、各プリミティブの名前がそのノードのアドレスにマップされます。次のように buildInitialHeap の定義を変更することで、これを簡単に行うことができます。

```
buildInitialHeap :: [CoreScDefn] -> (TiHeap, TiGlobals)
buildInitialHeap sc_defs
  = (heap2, sc_addrs ++ prim_addrs)
    where
      (heap1, sc_addrs)   = mapAccuml allocateSc Initial sc_defs
      (heap2, prim_addrs) = mapAccuml allocatePrim heap1 primitives
```

変数名からプリミティブへのマッピングを提供する関連付けリストを定義します。したがって、次のようになります。

```
primitives :: ASSOC Name Primitive
primitives = [ ("negate", Neg),
               ("+", Add), ("-", Sub),
               ("*", Mul), ("/", Div)
             ]
```

さらにプリミティブを追加するには、primitive 型にコンストラクターを追加し、primitives 関連付けリストに要素を追加します。

次に、allocateSc を定義したのと同じように、allocatePrim を定義できます。

2.7 Mark 5: 構造化データ型の追加

このセクションでは、構造化データ型をリダクションマシンに追加します。コア言語の case 式の実装を提供するのは良いことですが、テンプレートインスタンス化マシンのフレームワーク内で実装するのはかなり難しいことがわかりました。(後の実装ではこの問題は発生しません。) 代わりに、if, casePair, caseList などの組み込み関数のコレクションを使用して、特定の構造化型を操作できるようにします。テンプレートマシンは、一般的な構造化オブジェクトを処理できません。

演習 2.18 テンプレートインスタンス化マシンに case 式を導入するのが難しいのはなぜですか？(ヒント：インスタンス化が case 式で何をするかを考えてください。)

2.7.1 構造化データの構築

構造化データは、コンストラクターのファミリー Pack{t, a} で構築されます。ここで、t はコンストラクターのタグを示し、a はそのアリティを示します（セクション 1.1.4）。したがって、グラフでこれらのコンストラクター関数を表現する必要があります。これらは実際には新しい形式のプリミティブであるため、プリミティブ型に新しいコンストラクタ PrimConstr を追加することでこれを行うことができます。これで、instantiateConstr の式で、式 EConstr t a をヒープノード NPrim "Pack"（PrimConstr t a）にインスタンス化できます。

次に、このプリミティブがどのように実装されているかという疑問が生じます。新しい補助関数 primConstr を呼び出す PrimConstr コンストラクターと一致するように、primStep にケースを追加する必要があります。これにより、十分な引数が指定されていることを確認し、指定されている場合は、ヒープ内に構造化データオブジェクトを構築します。

次に、このプリミティブがどのように実装されているかという疑問が生じます。新しい補助関数 primConstr を呼び出す PrimConstr コンストラクターと一致するように、primStep にケースを追加する必要があります。これにより、十分な引数が指定されていることを確認し、指定されている場合は、ヒープ内に構造化データオブジェクトを構築します。

これを行うには、構造化データオブジェクトを表す新しいコンストラクタ NData をノードタイプに追加する必要があります。NData コンストラクタには、オブジェクトのタグとそのコンポーネントが含まれています。

```
data Node = NAp Addr Addr                     -- Application
          | NSupercomb Name [Name] CoreExpr -- Super combinator
          | NNum Int                          -- Number
          | NInd Addr                         -- Indirection
          | NPrim Name Primitive              -- Primitive
          | NData Int [Addr]                  -- Tag, list of components
```

これで、`NPrim (PrimConstr t n)` のルールを指定できます。

$$
\begin{array}{|c|l|}
\hline
(2.10) & a_0 : a_1 : \ldots : a_n : [] \quad d \quad h \left[\begin{array}{l} a : \texttt{NPrim}\ (\texttt{PrimConstr}\ t\ n) \\ a_1 : \texttt{NAp}\ a\ b_1 \\ \ldots \\ a_n : \texttt{NAp}\ a_{n-1}\ b_n \end{array}\right] f \\
 & \Longrightarrow \quad a_n : [] \quad d \quad h[a_n : \texttt{NData}\ t\ [b_1, \ldots, b_n]] \quad f \\
\hline
\end{array}
$$

構造化オブジェクトの構築についてはこれだけです。次の質問は、それらをどのように分解するかです。これは、コア言語の case 式で表されます。すでに述べたように、case 式を直接実装するのは難しいため、ブール値から始まるいくつかの特殊なケースに満足しています。

2.7.2　条件式

ブール型は、Miranda で次のように宣言される可能性があります。

```
boolean ::= False | True
```

`True` と `False` の 2 つのコンストラクタがあります。それぞれアリティは 0 であり、タグ 1 を `False` に、タグ 2 を `True` に任意に割り当てます。したがって、次のコア言語定義を与えることができます。

```
False = Pack{1, 0}
True = Pack{2, 0}
```

一般的な case 式を使用することはできないため、次の簡約ルールを使用して条件付きプリミティブを追加するだけで十分です。

```
if Pack{2, 0} t e = t
if Pack{1, 0} t 3 = e
```

動作上、データオブジェクトであると想定される最初の引数を評価し、タグを調べ、タグが 2(`True`) または 1(`False`) のどちらであるかに応じて、2 番目または 3 番目の引数を選択します。

演習 2.19 条件付きプリミティブの状態遷移ルールを記述します。3 つのルールが必要です。ブール条件がすでに評価されている場合に削減を実行するための 2 つのルール。1 つは、条件が評価されていない場合に、古いスタックをダンプにプッシュし、条件のアドレスを新しい空のスタックにプッシュすることによって評価を開始します（ルール 2.9 を参照)。最初の 2 つのルールの更新では、間接参照を使用する必要があることに注意してください。もう 1 つのルールがありません。それは何ですか？(ヒント：条件の評価が完了すると、条件はどのように再試行されますか？)

`if` があれば、それと `False` および `True` に関して、他のブール演算子のコア言語定義を与えることができます。例えば：

```
and x y = if x y False
```

演習 2.20 `or`、`xor` および `not` のコア言語定義を指定します。これらのコア言語定義をすべて `extraPreludeDefs` に追加します。

最後に、数値を比較する何らかの方法が必要です。これには、新しいプリミティブ>、>=などが必要です。

2.7.3 構造化データの実装

構造化データオブジェクト、条件、比較操作を追加するために実装に必要な変更のリストを次に示します。

- `NData` コンストラクターを `node` データ型に追加します。`showNode` を拡張して、`NData` ノードを表示します。
- `PrimConstr`、`If`、`Greater`、`GreaterEq`、`Less`、`LessEq`、`Eq`、`NotEq` を `primitive` 型に追加します。最初のものを除くすべての場合、`primitive` の関連付けリストに適切なペアを追加して、これらのプリミティブの名前を `instantiateVar` によってそれらの値にマップできるようにします。
- `instanceiateConstr`(および必要に応じて `instantiateAndUpdateConstr`) の定義を追加します。
- `isDataNode` 関数は、`NNum` ノードだけでなく `NData` ノードも識別する必要があります。
- `step` の `dispatch` コードには、新しい補助関数 `dataStep` を呼び出す、`NData` ノードの追加のケースが必要です。

- dataStep の定義; numStep と非常によく似ています。
- primStep を拡張して、新しいプリミティブ PrimConstr、If、Greater などに対応します。PrimConstr と If の場合、新しい補助関数 primConstr と primIf を呼び出す必要があります。比較プリミティブは、primArith をほとんど使用できますが、完全には使用できません。必要なのは、primArith のわずかな一般化です。

  ```
  primDyadic :: TiState -> (Node -> Node -> Node) -> TiState
  ```

  これは、数値を組み合わせる関数ではなく、ノードを組み合わせる関数を取ります。primDyadic に関して、primArith と同様の関数 primComp を比較プリミティブとして定義するのは簡単です。そして、primArith の定義を一般化することによって primDyadic を定義します。

演習 2.21 これらすべての変更を行います。再帰を終了する条件があるので、ついに、賢明な再帰関数を書くことができます。たとえば、階乗関数を試してください

```
fac n = if (n == 0) 1 (n * fac (n-1)) ;
main = fac 3
```

2.7.4 ペア

ブール値のコンストラクタは両方ともアリティが 0 となっています。次に、ペアのデータ型を追加します。これは、Miranda で次のように宣言される可能性があります。

```
pair * ** ::= MkPair * **
```

Pack 1, 2 コンストラクタを使用してペアを作成できます。

```
MkPair = Pack{1, 2}
```

case 式を使わずに、それらを分解してみませんか？たとえば、case 式を含む次のコア言語プログラムについて考えてみます。

```
f p = case p of
        <1> a b -> b*a*a end
```

case 式がないため、代わりに次のように翻訳できます。

```
f p = casePair p f'
f' a b = b*a*a
```

ここで、f' は補助関数であり、casePair は次のように定義された組み込みのプリミティブです。

```
casePair (Pack{1, 2} a b) f = f a b
```

操作上、casePair は最初の引数を評価します。これはペアを生成することを期待しています。次に、2 番目の引数をペアの 2 つのコンポーネントに適用します。これを実装するには、プリミティブ型にさらに別のコンストラクタ PrimCasePair を追加し、それを処理するコードをさらに記述します。

たとえば、次のコア言語定義を使用して、ペアの 1 番目と 2 番目のコンポーネントを抽出する fst と snd を定義できます。

```
fst p = casePair p K
snd p = casePair p K1
```

演習 2.22 casePair の状態遷移ルールを記述します。いつものように、2 つのルールが必要になります。1 つは最初の引数が評価された場合に削減を実行し、もう 1 つは評価されなかった場合にその評価を開始します。上記のように、ペアを実装するために必要な変更を加えます。次のプログラム (および独自の他のプログラム) を使用して実装をテストします。

```
main = fst (snd (fst (MkPair (MkPair 1 (MkPair 2 3)) 4)))
```

2.7.5 リスト

ペアとブール値を作成したので、リストは簡単なはずです。リストのデータ型は、Miranda で次のように定義されている可能性があります。

```
list * ::= Nil | Cons * (list *)
```

タグ 1 を Nil に、タグ 2 を Cons に割り当てます。

唯一の問題は、リストを分解する caseList プリミティブが正確に何をすべきかということです。casePair には、ペアのコンポーネントを引数として取る関数である「継続」が 1 つあることを思い出してください。if には 2 つの「継続」があり、最初の引数の値に応じていずれかを選択します。したがって、caseList は、これら両方のアイデアを組み合わせたものにすぎません。

```
caseList Pack{1, 0}        cn cc = cn
caseList (Pack{2, 2} x xs) cn cc = cc x xs
```

3 つの引数を取り、最初の引数を評価します。それが空のリストである場合 (つまり、タグが 1 でコンポーネントがない場合)、caseList は単に 2 番目の引数 cn を選択します。それ以外の場合は、リストセル (つまり、タグが 2 でコンポーネントが 2 つ) である必要があり、caseList は 3 番目の引数をこれらのコンポーネントに適用します。

たとえば、Miranda で記述される length 関数を実装したいとします。

```
length [] = 0
length (x : xs) = 1 + length xs
```

caseList を使用すると、次のように length を記述できます。

```
length xs = caseList xs 0 length'
length' x xs = 1 + length xs
```

演習 2.23 Cons、Nil、head、tail のコア言語定義を記述します。head と tail を定義するには、新しいプリミティブ abort を導入する必要があります。これは、空リストの head または tail を取得した場合に返されます。abort は、Miranda の error プリミティブを呼び出してプログラムを停止することで便利に実装できます。

演習 2.24 caseList の状態遷移ルールを記述し、それと abort を実装して、preludeDefs に Cons、Nil、head、tail の定義を追加します。実装をテストするためのプログラムをいくつか作成します。

これで、考えたい構造化データ型に適した case プリミティブを記述できるようになります。

演習 2.25 完全な case 式を実装するのではなく、case プリミティブを使用して構造化データ型を分解することの主な欠点は何ですか？

2.7.6　リストの表示

これを説明するための状態遷移ルールを書き始めるとすぐに、状態遷移の世界で「数値を出力する」という考えをどのように表現するかを決定する必要があります。最も簡単な解決策は、出力と呼ばれる新しいコンポーネントを状態に追加し、「出力に数値を追加する」ことによって「数値を出力する」モデルを作成することです。また、Print と Stop という 2 つの新しいプリミティブを追加します。

Stop プリミティブは簡単です。スタックを空にします。(tiFinal は、現在のようにエラーを出すのではなく、空のスタックを検出したときにマシン

を停止するように変更されます。) Stop は、ダンプが空であることを期待しています。

$$(2.11)\qquad \begin{array}{lllllll} & o & a:[] & [] & h[a:\texttt{NPrim Stop}] & f \\ \Longrightarrow & o & [] & [] & h & f \end{array}$$

Print プリミティブは、最初の引数を整数に評価し、その値を出力リストに付加します。次に、結果として 2 番目の引数を返します。また、ダンプが空であることも想定しています。最初の引数がすでに評価されている場合、最初のルールが適用されます。

$$(2.12)\qquad \begin{array}{lllllll} & o & a:a_1:a_2:[] & [] & h & \left[\begin{array}{l} a:\texttt{NPrim Print} \\ a_1:\texttt{NAp } a\ b_1 \\ a_2:\texttt{NAp } a_1\ b_2 \\ b_2:\texttt{NNum } n \end{array}\right] & f \\ \Longrightarrow & o\ ++\ [n] & b_2:[] & [] & h & & f \end{array}$$

Print は、出力 o に副作用があるため、かなり奇妙なスーパーコンビネータです。Print は明らかに注意して使用する必要があります！2 番目のルールは、Print の最初の引数が評価されない場合に適用されます。つまり、通常の方法で評価を開始します。

$$(2.13)\qquad \begin{array}{lllllll} & o & a:a_1:a_2:[] & [] & h & \left[\begin{array}{l} a:\texttt{NPrim Print} \\ a_1:\texttt{NAp } a\ b_1 \\ a_2:\texttt{NAp } a_1\ b_2 \end{array}\right] & f \\ \Longrightarrow & o & b_1:[] & (a_2:[]):[] & h & & f \end{array}$$

ここで、extraPreludeDefs で次の追加関数を定義します。

```
printList xs = caseList xs stop printCons
printCons h t = print h (printList t)
```

ここで、print は primitives によって Print プリミティブにバインドされ、stop は Stop にバインドされます。

最後に、スタックに式 (printList main) のアドレスが最初に含まれるように、compile 関数を変更します。これが正しいことをしていることを自分自身に納得させるのにそれほど時間はかからないはずです。

演習 2.26 これらの変更を実装し、数値のリストを返すプログラムを作成して実装をテストします。

2.8　代替実装 †

これらの演習では、私たちが行ったことに対するいくつかの代替実装について説明します。

2.8.1　プリミティブの代替表現

プリミティブに対して行うのはそれを実行することだけなので、次のトリックを実行できます。primitive をケース分析を実行する列挙型にする代わりに、tiState を tiState に変換する以下のような関数にすることができます。

```
Type Primitive = TiState -> TiState
```

コンストラクターの Add、Sub などは完全に消滅しました。これで、primStep によって実行される「ケース分析」はかなり簡単になりました。関数を適用するだけです。primStep とプリミティブの改訂された定義は次のとおりです。

```
primStep state prim = prim state
primitives = [ ("negate", primNeg),
               ("+", primArith (+)), ("-", primArith (-)),
               ("*", primArith (*)), ("/", primArith (/))
             ]
```

これは、実際の実装では直接対応します。プリミティブを区別するために NPrim ノードに小さな整数タグを格納する代わりに、コードポインタを格納し、それにジャンプしてプリミティブを実行します。

演習 2.27 この変更を実装してテストします。

2.8.2　ダンプの代替表現

現在、ダンプをスタックのスタックとして実装していますが、実際の実装では、古いスタックの上に新しいスタックを直接構築することは間違いありません。ダンプには、スパインスタックのベースからのオフセットが含まれ、1 つのサブスタックが終了し、次のサブスタックが開始する場所を示します。

次の型宣言を使用して、これをマシンで直接モデル化できます。

```
type TiDump = Stack Num
```

演習 2.28 この変更を実装してテストします。引数の評価の開始と終了、および tiFinal の定義を処理する関数定義を変更する必要があります。

2.8.3　データ値の代替表現

2.9 ガベージコレクション

実行が進むにつれて、ヒープに割り当てられるノードが増えるため、ヒープを表す Miranda データ構造はますます大きくなります。最終的に、Miranda はスペースを使い果たします。これは実際の実装に直接対応しているため、当然のことです。ノードが割り当てられると、ヒープはどんどん大きくなり、最終的にはいっぱいになります。スペースを解放するためにガベージコレクションを実行する必要があります。

より具体的には、型も含めて関数 gc を定義する必要があります。

```
gc :: TiState -> TiState
```

その結果の状態は、ヒープが (うまくいけば) 小さいことを除いて、入力状態とまったく同じように動作します。この小さなヒープには、マシン状態の他のコンポーネントから直接または間接的にアクセスできるすべてのノードが含まれています。gc は、不要になったノードのアドレスで hFree を呼び出すことにより、ヒープを小さくします (hFree の説明については、Appendix A.1 を参照してください)。

doAdmin 関数は、各ステップの後に (hSize を使用して) ヒープサイズをチェックし、指定されたサイズよりも大きい場合はガベージコレクタを呼び出すことができます。

2.9.1 マークスキャンコレクション

まず、マークスキャンコレクタを開発します。これは 3 つのフェーズで機能します。

1. 最初のフェーズでは、すべての**ルート**を識別します。 つまり、マシンの状態に含まれるすべてのヒープアドレスです。そのようなアドレスはどこに潜んでいる可能性がありますか？addr の出現について、マシンの状態に関係するタイプを調べることで簡単に見つけることができます。答えは、アドレスがスタック、ダンプ、およびグローバルで発生する可能性があるということです。したがって、次の関数が必要です。

```
findStackRoots  :: TiStack   -> [Addr]
findDumpRoots   :: TiDump    -> [Addr]
findGlobalRoots :: TiGlobals -> [Addr]
```

2. マークフェーズでは、アドレスがマシンのステートにある各ノードがマークされます。ノードがマークされると、そのすべての子孫もマークされ、以下同様に再帰的にマークされます。markFrom 関数は、ヒープとアドレスを受け取り、そのアドレスからアクセス可能なすべてのノードがマークされた新しいヒープを返します。

```
markFrom :: TiHeap -> Addr -> TiHeap
```

3. スキャンフェーズでは、ヒープ内のすべてのノード (マークされているかどうかに関係なく) が検査されます。マークされていないノードは解放され、マークされたノードはマーク解除されます。

```
scanHeap :: TiHeap -> TiHeap
```

演習 2.30 findRoots、markFrom、scanHeap の観点から gc の定義を記述し、doAdmin から適切に呼び出します。

演習 2.31 findRoots の定義を記述します。

markFrom と scanHeap を実装する前に、ノードを**マーク**する方法が必要です。実際の実装では、これはノードのビットを使用して、ノードがマークされているかどうかを示すことによって行われます。次のように、新しいコンストラクタである node 型を追加してこれをモデル化します。

```
data Node = NAp Addr Addr -- Application
          | NSupercomb Name [Name] CoreExpr -- Supercombinator
          | NNum Int                        -- Number
          | NInd Addr                       -- Indirection
          | NPrim Name Primitive            -- Primitive
          | NData Int [Addr]                -- Tag, list of components
          | NMarked Node                    -- Marked node
```

新しい種類のノードは NMarked ノードであり、マーキングが行われる前に存在していた node がその中に含まれています。NMarked ノード内の node が別の NMarked ノードになることはありません。

これで、markFrom を定義する準備が整いました。アドレス a とヒープ h が与えられると、次のようになります。

1. h で a を検索し、ノード n を与えます。すでにマークされている場合、markFrom はすぐに戻ります。これが、ヒープ内で循環構造に遭遇したときにマーキングプロセスが永久に続くのを防ぐものです。

2. hUpdate を使用してノードをマークし、NMarked n に置き換えます。

3. n 内から任意のアドレスを抽出し (そのようなアドレスは 0 個以上ある場合があります)、それぞれに対して markFrom を呼び出します。

残っているのは scanHeap だけです。hAddresses を使用して、ヒープで使用されているすべてのアドレスのリストを抽出し、それぞれを順番に調べます。参照するノードがマークされていない場合 (つまり、NMarked ノードではない場合)、hFree を呼び出してノードを解放します。それ以外の場合は、hUpdate を使用してノードのマークを解除し、NMarked コンストラクタ内にあるノードに置き換えます。

演習 2.32 markFrom と scanHeap の定義を記述します。

これで、マークスキャンガベージコレクタは完了です。

ガベージコレクションを実行する方法はマークスキャンだけではありません。さらに調査するためのいくつかの方向性を提案します。ガベージコレクション手法の簡単な調査は、[Peyton Jones 1987] の第 17 章にあります。より包括的なレビューは [Cohen 1981] です。

2.9.2 間接参照の排除

まず、開発したばかりのコレクタの最適化から始めます。評価中に、間接ノードを導入する場合があります。図 2.2 に示すように、ポインターを再調整することにより、間接ノードを排除すると便利です。

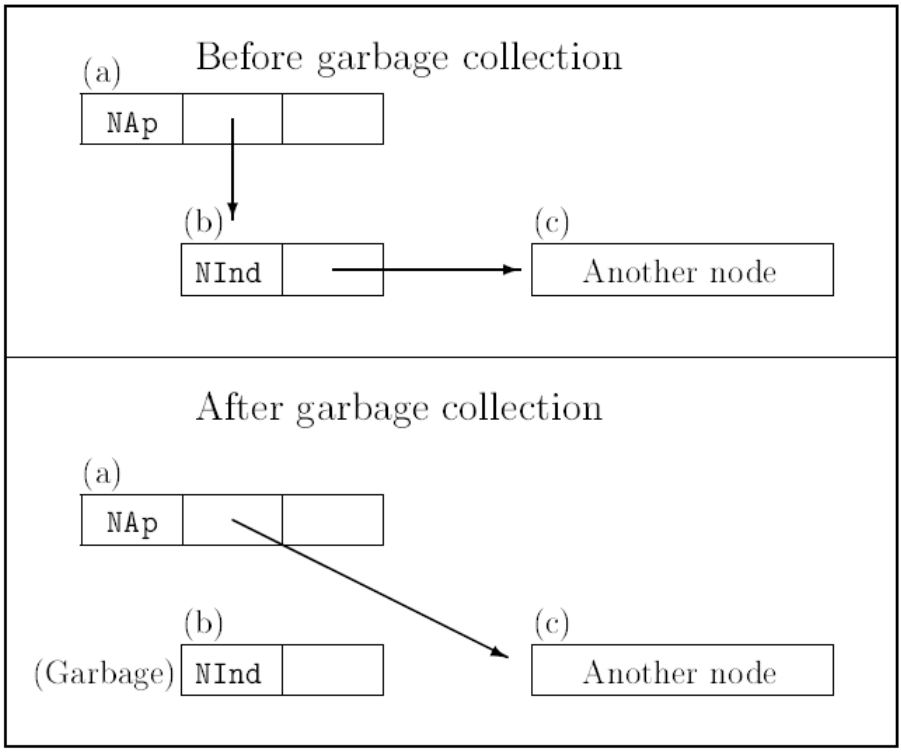


Figure 2.2: Eliminating indirections during garbage collection

これを行うには、markFrom の機能を少し変更する必要があります。これで、アドレスとヒープを取得し、そのアドレスからアクセス可能なすべてのノードにマークを付け、古いアドレスの代わりに使用する必要がある新しいアドレスとともに新しいヒープを返す必要があります。

```
markFrom :: TiHeap -> Addr -> (TiHeap, Addr)
```

図では、ノード (a) のアドレスを使用して markFrom を呼び出すと、ノード (c) をマークし (ノード (b) はマークしない)、ノード (c) のアドレスを返す必要があります。

markFrom から返されたアドレスをどのように利用しますか？markFrom が呼び出されたアドレスの代わりに挿入する必要があります。これを行う最も簡単な方法は、最初の 2 つのフェーズをマージすることです。これにより、各ルートがマシン状態で識別されると、markFrom が呼び出され、返されたアドレスを使用して、マシン状態の元のルートが置き換えられます。したがって、findStackRoots とそのコンパニオンを次のように置き換えます。

```
markFromStack   :: TiHeap -> TiStack   -> (TiHeap,TiStack)
```

```
markFromDump    :: TiHeap -> TiDump    -> (TiHeap,TiDump)
markFromGlobals :: TiHeap -> TiGlobals -> (TiHeap,TiGlobals)
```

演習 2.33 markFrom の改訂版を実装し、間接をマークせずに「スキップオーバー」し、再帰的に呼び出すときに各ノード内のアドレスを更新します。次に、markFrom の観点から他のマーキング関数を実装し、それらを新しいバージョンの gc で接着します。この新しいコレクタで取得したヒープサイズを以前に取得したヒープサイズと比較して、改善を測定します。(markFrom から NInd の特別な処理を削除することで、以前の状態に簡単に戻すことができます。)

2.9.3 ポインタ反転

ヒープ内のすべての N ノードがたまたま 1 つの長いリストにリンクされている場合、markFrom はそれ自体を N 回再帰的に呼び出します。実際の実装では、これはヒープが大きいのと同じくらい深いスタックを構築します。非常にまれな状況で、ヒープと同じ大きさのスタックをカウントに割り当てる必要があるのは非常に面倒です。

ポインタ反転と呼ばれる巧妙なトリックがあります。これは、マークされているノードそのものをリンクすることでスタックを排除できます [Schorr and Waite1967]。アルゴリズムによって課せられる唯一の追加要件は、マークされたノードが数ビットの追加の状態情報を必要とすることです。これは、NMarked コンストラクターをいくらか拡張することで表現できます。

```
data Node = NAp Addr Addr                    -- Application
          | NSupercomb Name [Name] CoreExpr -- Supercombinator
          | NNum Int                         -- Number
          | NInd Addr                        -- Indirection
          | NPrim Name Primitive             -- Primitive
          | NData Int [Addr]                 -- Tag, list of components
          | NMarked MarkState Node           -- Marked node

data markState = Done       -- Marking on this node finished
               | Visits Int -- Node visited n times so far
```

markState のコンストラクターの意味については、簡単に説明します。

別の (まったく別の)) 状態遷移システムを使用して、ポインタ反転アルゴリズムを説明できます。マーキングマシンの状態には、フォワードポインタ、

バックワードポインタ、およびヒープの 3 つのコンポーネント (f, b, h) があります。markFrom を呼び出すたびに、マシンの新しい実行が開始されます。markFrom がアドレス a とヒープ h_{init} で呼び出されると、マシンは次の状態から開始されます。

$$(a, \texttt{hNull}, h_{init})$$

(hNull は、ヒープ内のオブジェクトをアドレス指定しない型 addr の識別値であり、通常のアドレスと区別できます。) マシンは、つぎの状態になると終了します。

$$(f, \texttt{hNull}, h[f : \texttt{NMarked Done} \quad n])$$

つまり、f がマークされたノードを指し、$b = \texttt{hNull}$ の場合です。(f が指すノードがすでにマークされている場合は、初期状態も最終状態である可能性があります。)

遷移規則は、マシンがマークされていないノードを処理する方法を説明することから始めます。現在のところ、NData ノードは無視されます。まず、関数適用の場合を扱います。マークされていない関数適用に遭遇すると、最初のサブグラフに「降下」し、NAp ノードの最初のフィールドに古いバックポインタを記録します。新しいフォワードポインタは最初のサブグラフをアドレス指定し、新しいバックポインタは関数適用ノード自体をアドレス指定します。状態情報 Visits 1 は、バックポインタが NAp ノードの最初のフィールドに保持されているという事実を記録します。

(x.xx)		f	b	$h[f : \texttt{NAp}\ a_1\ a_2]$
	$\Longrightarrow$	a_1	f	$h[f : \texttt{NMarked}\ (\texttt{Visits 1})\ (\texttt{NAp}\ b\ a_2)]$

それまでのフォワードポインタが指していたアドレスの内容を NMarked (Visits 1) (NAp b a2)　※ b は、それまでのバックワードポインタが指していたアドレスの値。に書き換えて、ここをバックワードポインタが指すようにする。

これを図 2.3(a) および (b) に示します。

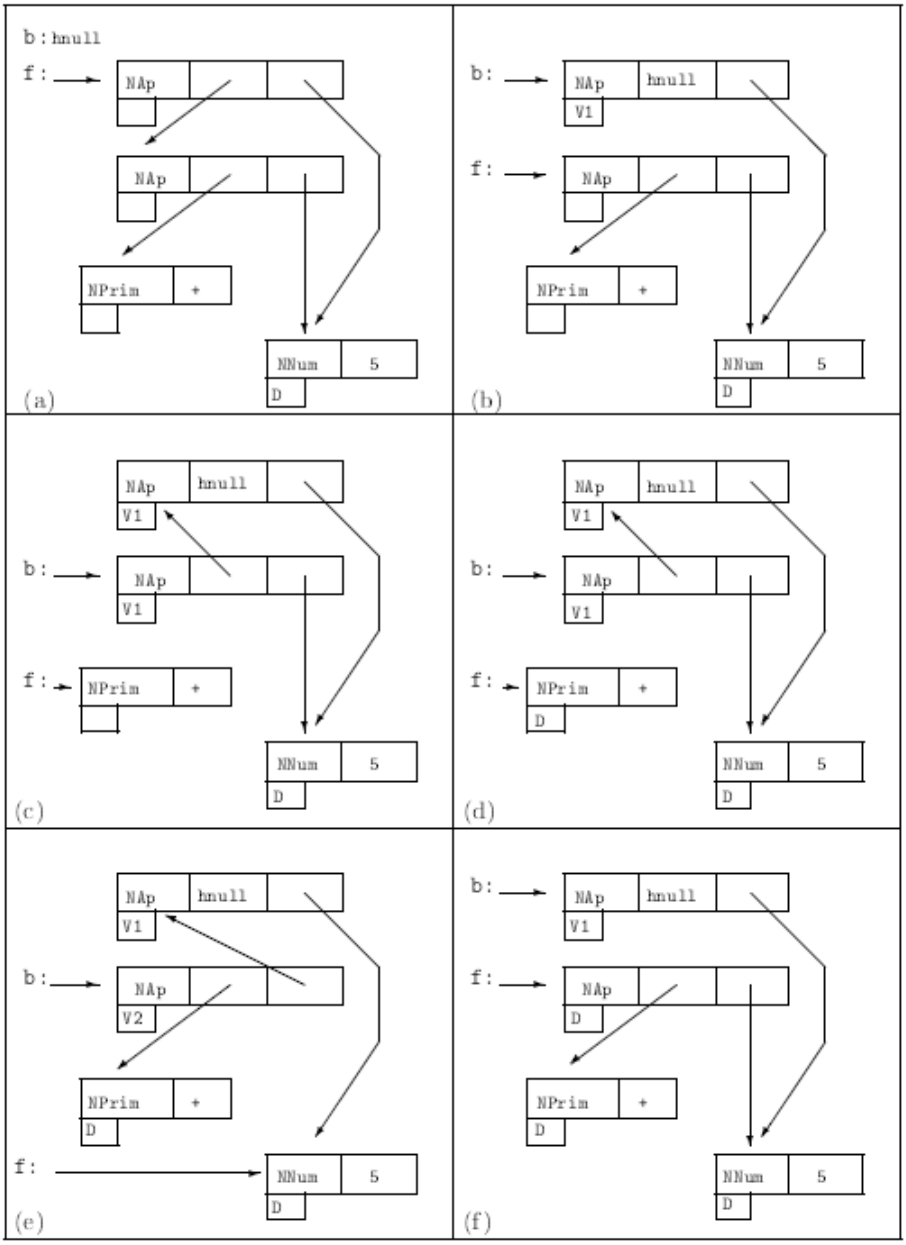


Figure 2.3: Marking a graph using pointer reversal

この図では、マークは「Visits 1」の場合は「V1」、「Visits 2」の場合は「V2」、「Done」の場合は「D」と省略されています。逆方向のポインタのチェーンが、アクセスされたアプリケーションノードに構築される方法に注意してください。

次のルールは、NMarked Done コンストラクタを使用して、マークされていない NPrim ノードを完了としてマークする必要があることを示しています (図 2.3(c))。

(x.xx)		f	b	$h[f : \texttt{NPrim}\ p]$
	$\Longrightarrow$	f	b	$h[f : \texttt{NMarked Done}\ (\texttt{NPrim}\ p)]$

NSupercomb ノードと NNum ノードは、それ以上のアドレスを含まないため、同様に扱われます。

マークされていないノードについてはこれだけです。マシンは、f がマークされたノードを指していることを検出すると、b が指しているノードを検査します。hNull の場合、マシンは終了します。(※フォワードポインタが指しているアドレスの中身が NMarked Done n でバックワードポインタが hNull の場合ということ？) それ以外の場合は、マークされた NAp ノードである必要があります。まず、状態情報が (Visits 1) の場合、ノードが 1 回訪問されたとしましょう。したがって、NAp ノードの最初のサブグラフのマーク付けが完了し、2 番目のサブグラフをマークする必要があります。これは、f をポイントにして、b を変更せずに、ノード (b') に保存されているバックポインタを最初のフィールドを 2 番目のフィールドに変更し、状態情報を変更します (図 2.3(d))。

(x.xx)		f	b	$h\left[\begin{array}{l} f : \texttt{NMarked Done}\ n \\ b : \texttt{NMarked}\ (\texttt{Visits 1})\ (\texttt{NAp}\ b'\ a_2) \end{array}\right]$
	$\Longrightarrow$	a_2	b	$h[b : \texttt{NMarked}\ (\texttt{Visits 2})\ (\texttt{NAp}\ f\ b')]$

しばらくして、マシンは 2 番目のサブグラフのマーキングを完了します。この場合、ノードを元の形式に復元し、バックポインタのチェーンを 1 段階バックアップできます (図 2.3(e))。

(x.xx)		f	b	$h\left[\begin{array}{l} f : \texttt{NMarked Done}\ n \\ b : \texttt{NMarked}\ (\texttt{Visits 2})\ (\texttt{NAp}\ a_1\ b') \end{array}\right]$
	$\Longrightarrow$	b	b'	$h[b : \texttt{NMarked Done}\ (\texttt{NAp}\ a_1\ f)]$

最後に、間接処理を扱います。f を変更することでスキップされますが、b は変更されません。ヒープは変更されないままなので、間接参照自体はマークされません。したがって、ガベージコレクションが完了すると、すべてのインダイレクションが再利用されます。ご覧のとおり、このマーキングアルゴリズムを使用すると、ガベージコレクション中に間接参照を「ショートアウト」するのは非常に簡単です。

(x.xx)		f	b	$h[f : \texttt{NInd}\ a]$
	$\Longrightarrow$	a	b	h

これで、ポインタ反転アルゴリズムの状態遷移が完了しました。

演習 2.34 NData ノードのルールを追加します。

演習 2.35 アルゴリズムを実装します。必要な主な変更は、node 型と markFrom 関数です。NMarked コンストラクタの形式が変更されたため、scan は簡単な方法で変更する必要があります。

2.9.4 2 スペースガベージコレクション

ガベージコレクションを実行するもう 1 つの非常に一般的な方法は、すべてのライブデータを 1 つのヒープから別のヒープにコピーすることです。これは、[Fenichel and Yochelson 1969] によって発明されたいわゆる 2 スペースコレクタです ([Baker 1978、Cheney 1970] も参照)。コレクタは 2 つの段階で機能します。

1. マシンの状態 (スタック、ダンプなど) が指すすべてのノードは、古いヒープ (*from-space* と呼ばれる) から最初は空だった新しいヒープ (*to-space* と呼ばれる) に退避されます。ノードは、そのコピーを to-space に割り当て、from-space のコピーを、新しいノードの to-space アドレスを含む *forwarding pointer* で上書きすることによって退避されます。markFrom と同様に、退避ルーチンは新しいノードの to-space アドレスを返します。これは、マシン状態の古いアドレスを置き換えるために使用されます。

2. 次に、to-space 内のすべてのノードが最初から線形にスキャンされ、それぞれが **scavenge されます**。ノード n は、それが指すノードを退避させ、n 内のそれらのアドレスを新しい to-space アドレスに置き換えることによってスカベンジされます。スキャンポインタが allocation pointer に追いつくと、スキャンが停止します。

これを実装するには、node 型のさらに別のバリアントを追加する必要があります。今回は、単一のアドレス (to-space アドレス) を含む NForward コンストラクタを使用します。(このコレクターには NMarked は必要ありません。) markFromStack の代わりに、次のタイプの evacuateStack が必要です。

```
evacuateStack :: TiHeap -> TiHeap -> TiStack -> (TiHeap, TiStack)
```

呼び出し (`evacuateStack` *fromheap toheap stk*) は、*stk* から *toheap* に参照される *fromheap* 内のすべてのノードを退避させ、新しい *toheap* と新しい *stk* を返します。ダンプとグローバルにも同様の関数が必要です。

最後に、関数が必要です。

```
scavengeHeap :: TiHeap-> TiHeap-> TiHeap
```

ここで、呼び出し (`scavengeHeap` *fromheap toheap*) は、*toheap* 内のノードをスカベンジし、必要に応じて *fromheap* から *toheap* にノードを退避させます。

演習 2.36 このガベージコレクターを実装します。

```
module GM where
import Language
import Utils
```

第3章　Gマシン

この章では、最初のコンパイラベースの実装である G マシンを紹介します。これは、スウェーデンのヨーテボリにあるチャルマース工科大学で、オーガストソンとジョンソンによって開発されました。この章の内容は、彼らの一連の論文 [Augustsson 1984、Johnsson 1984] に基づいており、博士論文 [Augustsson 1987、Johnsson1987] で最高潮に達します。

3.1　Gマシンの紹介

テンプレートのインスタンス化マシンの基本的な操作は、`instantiate` 関数によって実装されるスーパーコンビネータ本体のインスタンスを構築することでした。**インスタンス化が実行されるたびに**、インスタンス化はテンプレートを再帰的にトラバースする必要があるため、これはかなり遅い操作です。`instantiate` によって実行される機械語命令について考えると、テンプレートのトラバースに関係するものと、実際にインスタンスを構築することに関係するものの 2 種類があることがわかります。

G マシンおよびその他のコンパイルされた実装の「大きなアイデア」は次のとおりです。

> プログラムを実行する前に、各スーパーコンビネータ本体を一連の命令に変換し、実行時にスーパーコンビネータ本体のインスタンスを構築します。

このコードを実行すると、インスタンス化関数を呼び出すよりも高速になります。これは、すべての命令がインスタンスの構築に関係しているためです。テンプレートをトラバースするために必要な指示はありません。これは、すべてが変換プロセス中に行われたためです。したがって、プログラムの実行は 2 つの段階に分けられます。最初の段階では、コンパイラを使用してプログラムの中間形式を生成します。 これは**コンパイルタイム**と呼ばれます。第 2 段階では、中間形式が実行されます。 これは**ランタイム**と呼ばれます。

スーパーコンビネータに対して行うことは、それをインスタンス化することだけなので、変換が完了したら元のスーパーコンビネータを破棄して、コ

ンパイルされたコードのみを保持できます。

原則として、**G マシンコンパイラ**を使用して、ソース言語のプログラムを一連の**機械語命令**に変換します。多くの異なるハードウェア (68000 ベース、または VAX など) に言語を実装したい場合があるため、**抽象マシン**があると便利です。優れた抽象マシンには 2 つの特性があります。まず、具体的な機械語コード (たとえば 68000 アセンブラ) に簡単に変換できます。2 つ目は、ソースから抽象的な機械語コードを簡単に生成できることです。

ここでトレードオフに直面していることに注意してください。抽象マシンを実マシンと**同じ**にすることで、理想的には第 1 の性質 (具体的なコード生成の容易さ) を満たすことができます。しかし、これにより、2 番目のプロパティを満たすことがはるかに難しくなります。したがって、抽象マシンは、ソース言語と特定の機械語コードの間の足がかりです。

3.1.1　例

G マシンコンパイラが動作している小さな例を次に示します。次の関数を考えます。

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

これは、一連の G コード命令にコンパイルされます。

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

図 3.1 に、このコードがどのように実行されるかを示します。各ダイアグラムの左側にはスタックがあり、下に向かって成長します。各図の残りの部分はヒープです。アプリケーション ノードは @ 文字で表され、式は小文字でラベル付けされ、スーパーコンビネータは大文字でラベル付けされます。

図 3.1 の図 (a) では、`f` の一連の命令を実行する前のマシンの状態を示しています。テンプレートインスタンス化マシンと同じように、スパインがほ

どかれています。スタックの上位 2 つの項目は関数適用ノードへのポインタであり、その右側の部分は g と x にバインドされる式です。

Push 命令は、スタックのトップに相対的なアドレス指定を使用します。スーパーコンビネータノード f へのポインタを無視すると、最初のスタック アイテムは 0、次は 1 というように番号が付けられます。次の図 (b) は、Push 1 命令を実行した後の変更されたスタックを示しています。これにより、式 x へのポインターがスタックにプッシュされます。x は、スタックの 2 つ下のスタック アイテムです。

別の Push 1 の後、スタックの一番上に g へのポインタがあります。前の命令が新しいポインタをスタックにプッシュしたため、これはスタックの 2 つ下のスタックアイテムです。新しい図は (c) です。

図 (d) は、Mkap 命令が実行されるとどうなるかを示しています。スタックから 2 つのポインタを取得し、それらから関数適用ノードを作成します。スタック上の結果へのポインタを残します。

図 (e) では、Pushglobal K 命令を実行し、K スーパーコンビネータへのポインタをプッシュします。

図 (f) に示すように、別の Mkap 命令が f の本体のインスタンス化を完了します。

元の式 f g x を新しくインスタンス化されたボディK (g x) に置き換えることができます。G マシンの最初のバージョン (これは lazy ではありません) では、本体をスタックの 3 つの場所にスライドさせ、そこにあった 3 つのポインタを破棄します。これは、図 (g) に示すように、Slide 3 の指示を使用して実現されます。最終 Unwind 命令により、マシンは評価を続行します。

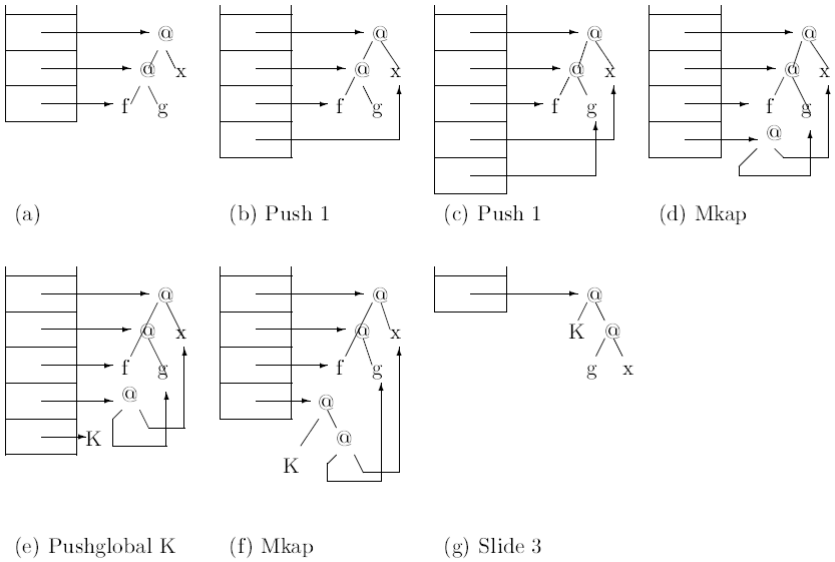


Figure 3.1: Execution of code for the `f` supercombinator

これで、G マシンの実行の概要を簡単に説明しました。

3.1.2　さらなる最適化

これまでに説明したように、テンプレートをトラバースする際の解釈のオーバーヘッドをなくすことで、適度なパフォーマンスの向上を実現できます。ただし、コンパイルは、テンプレートインスタンス化マシンでは利用できない多くのショートカットと最適化への扉を開くことも判明しています。たとえば、次の定義を考えてみましょう。

```
f x = x + x
```

テンプレートインスタンス化マシンは `x` を 2 回評価します。2 回目の場合はもちろん、すでに評価されていることがわかります。コンパイルされた実装は、コンパイル時に `x` が既に評価されていることを発見し、評価ステップを省略できます。

3.2 テンプレートを構築するためのコード シーケンス

テンプレートインスタンス化マシンは次のように動作することを思い出してください。

- スタックの一番上にある単一のアイテムが整数へのポインタである場合、マシンは終了しました。
- そうでない場合は、スーパーコンビネータノードに到達するまで、遭遇した関数適用ノードをアンワインドします。次に、スーパーコンビネータ本体のコピーをインスタンス化し、その引数を置換します。

Mark 1 テンプレートインスタンス化マシンの中心にあるのは、59 ページと 60 ページで定義されている 2 つの関数 `scStep` と `instanceiate` です。`scStep` と `instantiate` の定義を見てみると、スーパーコンビネータをインスタンス化する操作について次のように説明できます。

1. 変数名の**ローカル環境**をヒープ内のアドレスに構築します。
2. このローカル環境を使用して、ヒープ内にスーパーコンビネータ本体のインスタンスを作成します。変数はコピーされません。 代わりに、対応するアドレスが使用されます。
3. 関数適用ノードとスーパーコンビネータノードへのポインタをスタックから削除します。
4. スーパーコンビネータの新しく作成されたインスタンスのアドレスをスタックにプッシュします。

テンプレートインスタンス化では、スーパーコンビネータのインスタンスを作成するには、スーパーコンビネータの本体である式のツリー構造をトラバースする必要があります。式は再帰的に定義されるため、ツリートラバーサル関数 `instantiate` は再帰的に定義されます。たとえば、`EAp e1 e2` の場合の `instantiate` の定義 (58 ページ) を参照してください。最初に `e1` に対して、次に `e2` に対して `instanceiate` を呼び出し、各サブ式のグラフのアドレスを保持します。最後に、グラフに関数適用ノードを作成して、2 つのアドレスを結合します。

命令の線形シーケンスをコンパイルして、式をインスタンス化する操作を実行したいと考えています。

3.2.1　算術の後置評価

式をインスタンス化する命令の線形シーケンスを構築したいという欲求は、算術式の後置評価を連想させます。G マシンに戻る前に、このアナロジーをさらに調べます。

算術式の言語は、数、加算、および乗算で構成されます。この言語を aExpr 型として表すことができます。

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

言語は「明白な」意味を持つべきであることが意図されています。 関数 aInterpret を使用してこれを与えることができます。

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

あるいは、式を演算子 (または命令) の後置シーケンスにコンパイルすることもできます。式を評価するには、コンパイルされた演算子と値のスタックを使用します。たとえば、算術式 $2+3\times4$ は、以下のシーケンスとして表されます。

$$[\texttt{INum 2}, \texttt{INum 3}, \texttt{INum 4}, \texttt{IMult}, \texttt{IPlus}]$$

後置マシンの命令は、aInstruction 型として指定できます。

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

評価器の状態は、一連の演算子と数値のスタックであるペアです。コードシーケンスの意味は、次の遷移規則で与えられます。

(3.1)			
		[]	$[n]$
	$\Longrightarrow$		n

(3.2)				
		INum $n : i$		ns
	$\Longrightarrow$		i	$n : ns$

(3.3)			
	IPlus $n : i$		$n_0 : n_1 : ns$
	$\Longrightarrow$	i	$(n_1 + n_0) : ns$

(3.4)			
	IMult $n : i$		$n_0 : n_1 : ns$
	$\Longrightarrow$	i	$(n_1 \times n_0) : ns$

これらの遷移規則を Miranda に翻訳すると、次のようになります。

```
aEval :: ([AInstruction], [Int]) -> Int
aEval ([], [n]) = n
  -- 遷移規則 (3.1)
aEval (INum n : is, s) = aEval (is, n: s)
  -- 遷移規則 (3.2)
aEval (IPlus : is, n0 : n1 : s) = aEval (is, n1+n0 : s)
  -- 遷移規則 (3.3)
aEval (IMult : is, n0 : n1 : s) = aEval (is, n1*n0 : s)
  -- 遷移規則 (3.4)
```

式の後置コードのシーケンスを生成するには、コンパイラを定義する必要があります。これは式を取り、実行時に式の値を計算する一連の命令を配信します。

```
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]
```

これからの重要なアイデアは、aCompile 関数の型によって与えられます。命令のリストを返します。

> 式の後置表現は、式ツリーをフラット化または線形化する方法であり、式をフラットな一連の演算子で表すことができます。

演習 3.1. 構造帰納法を使用するか、またはその他の方法で、算術式の後置評価が式のツリー評価と同じ結果になることを証明します。つまり、型 aExpr のすべての式 e に対して、

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

これは一致証明の例です。

演習 3.2. 関数 aInterpret、aCompile、および aEval を拡張して、let 式を処理します。aExpr 型の e のすべての式について、これらの新しい関数が次の関係を満たすことを証明します。

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

letrec 式など、さらに複雑な式に言語を拡張できますか? これらの拡張機能を正しく実装したことを証明できますか?

3.2.2 後置コードを使用してグラフを作成する

同じテクニックを使用して、スーパーコンビネータ本体のインスタンスを作成できます。この場合、スタック上の「値」は、インスタンス化される式の部分のアドレスになります。

テンプレート構築命令の操作は、上記の算術例で見たものとは異なります。通常、命令にはヒープにノードを割り当てるという副作用があります。例として、Mkap 命令の導入を検討してください。この命令は、スタックの上位 2 つのアドレスからヒープ内に関数適用ノードを作成します。完了すると、この新しいノードへのポインタがスタックに残されます。

テンプレートインスタンス化マシンには既にそのようなスタックがあるため、アドレスの新しい評価スタックを創案する理由はありません。ただし、このスタックを使用する場合は、覚えておくべき重要な点があります。

> スタックからオブジェクトをポップおよびプッシュすると、変数名に対応するスタックの場所のマップが変化します。したがって、式をコンパイルするときは、これを追跡する必要があります。

スタック内のアイテムへのアクセスは、スタックの一番上を基準にしています。そのため、アイテムが追加されると、そのアイテムに到達するまでのオフセットが 1 増加します。同様に、アイテムがポップされると、オフセットは 1 だけ減少します。

3.2.3 インスタンス化が行われた後はどうなりますか?

スーパーコンビネータ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 e のヒープ内のアドレスがあります。
- 次に、$n+1$ 個のポインタがあります。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。

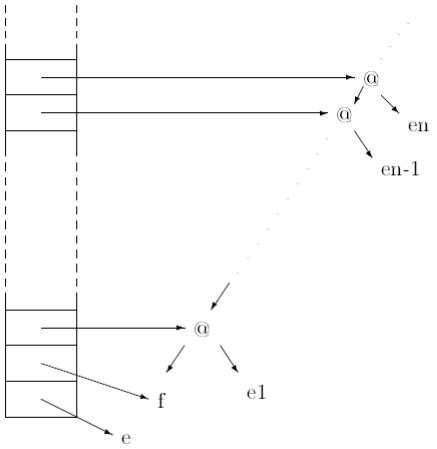


Figure 3.2: The stack layout for the Mark 1 machine

redex を新しくインスタンス化された本体に置き換え、`Slide` 命令を使用して n 個のアイテムをスタックからポップする必要があります。次のスーパーコンビネータを見つけるには、`Unwind` 命令を使用して、再びアンワインド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アンワインド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マシンをを Mark 1 G マシンに変換しました。

関数 `f x1 ... xn = e` のコードは次のとおりです。

```
<code to construct an instance of e>
Slide n+1
Unwind
```

3.3　Mark 1: 最小限の G マシン

ここで、完全な G マシンとそのコンパイラのコードを示します。更新 (セクション 3.4 で紹介) や算術演算 (セクション 3.6 で紹介) は実行しません。

3.3.1　全体構造

最上位では、G-machine はテンプレート インスタンシエーターに非常に似ています。いつものように、システム全体が実行機能と一緒に編まれています。

```
-- The function run is already defined in gofers standard.prelude
runProg :: [Char] -> [Char]
runProg = showResults . eval . compile . parse
```

アクセスする必要があるため、パーサのデータ構造と関数が含まれています。

```
-- :a language.lhs -- parser data types
```

3.3.2　データ型の定義

グラフ簡約実装手法の基本はグラフです。とりわけ、Appendix A で提供されているユーティリティの heap データ型を使用します。

```
-- :a util.lhs -- heap data type and other library functions
```

Mark 1 G マシンは、状態として 5 タプル gmState を使用します。gmState は、コンパイルされたプログラムの実行中に必要なすべての情報を保持します。

```
type GmState
  = (GmCode,    -- Current instruction stream
     GmStack,   -- Current stack
     GmHeap,    -- Heap of nodes
     GmGlobals, -- Global addresses in heap
     GmStats)   -- Statistics
```

G マシンの説明では、ステート **アクセス関数**を使用して、ステートのコンポーネントにアクセスします。このアプローチの利点は、新しいコンポーネントに対応するために状態を変更するときに、作成した元のコードのほとんどを再利用できることです。接頭辞 get を使用して状態からコンポーネントを取得するアクセス関数を示し、接頭辞 put を使用して状態内のコンポーネントを置き換えます。

状態の 5 つのコンポーネントのそれぞれの型定義と、それらのアクセス関数を順に検討します。

- 命令ストリームは gmCode 型で、単なる instruction のリストです。

```
type GmCode = [Instruction]
```

コードへの便利なアクセスを取得するために、後で状態が追加のコンポーネントで拡張されるときに、getCode と putCode の 2 つの関数を定義します。

```
getCode :: GmState -> GmCode
getCode (i, stack, heap, globals, stats) = i

putCode :: GmCode -> GmState -> GmState
putCode i' (i, stack, heap, globals, stats)
  = (i', stack, heap, globals, stats)
```

最初は 6 つの命令しかありません。これらについては、サブセクション 3.3.3 で詳しく説明します。

```
data Instruction
  = Unwind
  | Pushglobal Name
  | Pushint Int
  | Push Int
  | Mkap
  | Slide Int
  instance Eq Instruction
    where
      Unwind == Unwind = True
      Pushglobal a == Pushglobal b = a == b
      Pushint a == Pushint b = a == b
      Push a == Push b = a == b
      Mkap == Mkap = True
      Slide a == Slide b = a == b
      _ == _ = False
```

- G マシン スタック gmStack は、ヒープ内のアドレスのリストです。

```
type GmStack = [Addr]
```

スタックへの便利なアクセスを取得するために、後で状態が追加のコンポーネントで拡張されたときに、getStack と putStack の 2 つの関数を定義します。

```
getStack :: GmState -> GmStack
getStack (i, stack, heap, globals, stats) = stack

putStack :: GmStack -> GmState -> GmState
putStack stack' (i, stack, heap, globals, stats)
  = (i, stack', heap, globals, stats)
```

- テンプレート インスタンシエーターの場合と同様に、utils のヒープデータ構造を使用してヒープを実装します。

```
type GmHeap = Heap Node
```

繰り返しますが、状態のこのコンポーネントにアクセスするには、アクセス関数を定義します。

```
getHeap :: GmState -> GmHeap
getHeap (i, stack, heap, globals, stats) = heap
putHeap :: GmHeap -> GmState -> GmState
putHeap heap' (i, stack, heap, globals, stats)
  = (i, stack, heap', globals, stats)
```

最小限の G マシンには、次の 3 種類のノードしかありません。数 NNum、関数適用 NAp、そしてグローバル NGlobal です。

```
data Node
  = NNum    Int          -- Numbers
  | NAp     Addr Addr    -- Applications
  | NGlobal Int  GmCode -- Globals
```

数 ノードには関連する番号が含まれます。関数適用 ノードは、最初のアドレスの関数を 2 番目のアドレスの式に適用します。NGlobal ノードには、グローバルが期待する引数の数と、そのときに実行されるコード シーケンスが含まれています。グローバルには十分な引数があります。これは、アリティとコードの代わりにテンプレートを保持していたテンプレート インスタンシエーターの NSupercomb ノードを置き換えます。

- 後で遅延実装を作成するため、各グローバルに対して 1 つのノードのみが存在することが重要です。グローバルのアドレスは、連想リスト gmGlobals でその値を検索することによって決定できます。これは、テンプレート マシンの tiGlobals コンポーネントに対応します。

```
type GmGlobals = ASSOC Name Addr
```

使用するアクセス関数は getGlobals です。Mark 1 マシンでは、このコンポーネントは一定であるため、対応する put 関数は必要ありません。

```
getGlobals :: GmState -> GmGlobals
getGlobals (i, stack, heap, globals, stats) = globals
```

- 状態の統計コンポーネントは、抽象データ型として実装されます。

```
statInitial  :: GmStats
statIncSteps :: GmStats -> GmStats
statGetSteps :: GmStats -> Int
```

gmStats の実装が提供されるようになりました。

```
type GmStats = Int
statInitial    = 0
statIncSteps s = s + 1
statGetSteps s = s
```

このコンポーネントにアクセスするには、getStats と putStats を定義します。

```
getStats :: GmState -> GmStats
getStats (i, stack, heap, globals, stats) = stats
putStats :: GmStats -> GmState -> GmState
putStats stats' (i, stack, heap, globals, stats)
  = (i, stack, heap, globals, stats')
```

3.3.3　評価器

G マシン評価器 eval は、状態のリストを生成するように定義されています。最初のものは、コンパイラによって構築されたものです。最後の状態がある場合、評価の結果は最後の状態のスタック コンポーネントの一番上になります。

```
eval :: GmState -> [GmState]
eval state = state: restStates
  where
  restStates | gmFinal state = []
             | otherwise = eval nextState
  nextState = doAdmin (step state)
```

関数 doAdmin は statIncSteps を使用して、状態の統計コンポーネントを変更します。

```
doAdmin :: GmState -> GmState
doAdmin s = putStats (statIncSteps (getStats s)) s
```

評価器の重要な部分は、関数 gmFinal とこれから見ていく step です。

最終状態のテスト

実行中のコード シーケンスが空の場合、G マシン インタープリターは終了しています。この状態を gmFinal 関数で表現します。

```
gmFinal :: GmState -> Bool
gmFinal s = case (getCode s) of
            []        -> True
            otherwise -> False
```

ステップを進める

step 関数は、実行中の命令に基づいて状態遷移を行うように定義されています。

```
step :: GmState -> GmState
step state = dispatch i (putCode is state)
             where (i : is) = getCode state
```

現在の命令 i を dispatch し、現在のコード シーケンスをコード シーケンス is に置き換えます。これは、実機でプログラムカウンタを進めることに相当します。

```
dispatch :: Instruction -> GmState -> GmState
dispatch (Pushglobal f) = pushglobal f
dispatch (Pushint    n) = pushint n
dispatch Mkap           = mkap
dispatch (Push       n) = push n
dispatch (Slide      n) = slide n
dispatch Unwind         = unwind
```

ご覧のとおり、dispatch 関数は実行する状態遷移を選択するだけです。

後置命令の遷移規則を調べることから始めましょう。instruction 内の構文オブジェクトごとに 1 つ存在します。まず、Pushglobal 命令から始めます。この命令は、状態のグローバル コンポーネントを使用して、グローバル f を保持する heap 内の一意の NGlobal ノードを見つけます。見つからない場合は、適切なエラー メッセージが出力されます。

(3.5)							
	Pushglobal f : i		s	h	$m[f : a]$
	$\Longrightarrow$	i	$a : s$	h	m

pushglobal 関数を使用してこのルールを実装します。

```
pushglobal :: Name -> GmState -> GmState
pushglobal f state
  = putStack (a: getStack state) state
    where a = aLookup (getGlobals state) f (error ("Undeclared global " ++ f))
```

残りの遷移は、スーパーコンビネータの本体を構築するためのものです。Pushint の遷移は、整数ノードをヒープに配置します。

(3.6)						
	PushInt n : i		s	h	m
	$\Longrightarrow$	i	$a : s$	$h[a :$ NNum $n]$	m

対応する関数は pushint です。数は、アドレス a の新しいヒープ heap' に配置されます。次に、ヒープとスタックを元の状態に戻します。

```
pushint :: Int -> GmState -> GmState
pushint n state
  = putHeap heap' (putStack (a: getStack state) state)
    where (heap', a) = hAlloc (getHeap state) (NNum n)
```

Mkap 命令は、スタックの最上部にある 2 つのアドレスを使用して、ヒープ内に関数適用ノードを構築します。次の遷移規則があります。

(3.7)						
	Mkap $a_1 : a_2 : i$		s	h		m
	$\Longrightarrow$	i	$a : s$	$h[a : \texttt{NAp}\ a_1\ a_2]$		m

このトランジションが mkap になります。ここでも、heap' と a は、それぞれ新しいヒープと新しいノードのアドレスです。

```
mkap :: GmState -> GmState
mkap state
  = putHeap heap' (putStack (a:as') state)
    where (heap', a) = hAlloc (getHeap state) (NAp a1 a2)
          (a1:a2:as') = getStack state
```

Push 命令は、関数に渡された引数のコピーを取得するために使用されます。これを行うには、スタックからポイントされている関数適用ノードを「調べる」必要があります。また、スタック上にあるスーパーコンビネータ ノードをスキップすることも忘れないでください。

(3.8)						
	Push $n : i$		$a_0 : \ldots : a_{n+1} : s$	$h[a_{n+1} : \texttt{NAp}\ a_n\ a'_n]$		m
	$\Longrightarrow$	i	$a'_n : a_0 : \ldots : a_{n+1} : s$	h		m

```
push :: Int -> GmState -> GmState
push n state
  = putStack (a:as) state
    where as = getStack state
          a  = getArg (hLookup (getHeap state) (as !! (n+1)))
```

これは、補助関数 getArg を使用して、アプリケーション ノードから必要な式を選択します。

```
getArg :: Node -> Addr
getArg (NAp a1 a2) = a2
```

スタック構造のため、Push 命令のアドレッシング モードを [Peyton Jones 1987] で使用されていたものから変更しました。

次に、スーパーコンビネータがインスタンス化された後、アンワインドを続行する前に発生するスタックの整理が、Slide 命令によって実行されます。

(3.9)						
	Slide $n : i$	$a_0 : \ldots : a_n : s$	h	m
	$\Longrightarrow$	i	$a_0 : s$	h	m

```
slide :: Int -> GmState -> GmState
slide n state
= putStack (a: drop n as) state
  where (a:as) = getStack state
```

Unwind は、テンプレート インスタンス化子の外側のループを置き換えるため、最も複雑な命令です。Unwind 命令は、次のセクションで説明するように、常にシーケンスの最後の命令です。構築された newState は、スタックの一番上の項目に依存します。これは、スタックの一番上にあるアイテムにも依存する、最初の遷移ルールに依存します。

```
unwind :: GmState -> GmState
unwind state
  = newState (hLookup heap a)
    where
      (a:as) = getStack state
      heap = getHeap state
```

まず、スタックの一番上に数値がある場合を考えます。この場合、これで終了です。 G マシンが終了したことを示すために、コード コンポーネントに [] を配置します。

(3.10)					
	[Unwind]	$a : s$	$h[a :$ NNum $n]$	m
	$\Longrightarrow$ []	$a : s$	h	m

```
newState (NNum n) = state
```

スタックの一番上に関数適用ノードがある場合は、次のノードからアンワインドを続行する必要があります。

(3.11)					
	[Unwind]	$a : s$	$h[a :$ NAp $a_1\ a_2]$	m
	$\Longrightarrow$ [Unwind]	$a_1 : a : s$	h	m

```
newState (NAp a1 a2) = putCode [Unwind] (putStack (a1:a:as) state)
```

最も複雑なルールは、スタックの一番上にグローバル ノードがある場合に発生します。スーパーコンビネータの適用を減らすのに十分な引数があるかどうかに応じて、考慮すべき 2 つのケースがあります。

第 1 に、スーパーコンビネータ アプリケーションを削減するのに十分な引数がない場合、プログラムの型が正しくありません。Mark 1 G マシン については、このケースを無視します。あるいは、十分な引数がある場合、スーパーコンビネータのコードに「ジャンプ」することで、スーパーコンビネータを減らすことができます。遷移規則では、これはスーパーコンビネータ コードをマシンのコード コンポーネントに移動することによって表現されます。

$\mathcal{SC}[\![d]\!]$ は、スーパーコンビネータ定義 d の G マシン コードです。 $\mathcal{SC}[\![f\ x_1 \ldots x_n = e]\!] = \mathcal{R}[\![e]\!]\ [x_1 \mapsto 0, \ldots, x_n \mapsto n-1]\ n$
$\mathcal{R}[\![e]\!]\ \rho\ d$ は、アリティ d のスーパーコンビネータのために、 環境 ρ で式 e をインスタンス化するコードを生成し、 結果のスタックのアンワインドに進みます。 $\mathcal{R}[\![e]\!]\ \rho\ d = \mathcal{C}\rho$++[Slide $d+1$, Unwind]
$\mathcal{C}[\![e]\!]\ \rho$ は、環境 ρ で e のグラフを構築するコードを生成し、 それへのポインターをスタックの一番上に残します。 $\mathcal{C}[\![f]\!]\ \rho = [\texttt{Pushglobal}\ f]$　ここで、f はスーパーコンビネータです。 $\mathcal{C}[\![x]\!]\ \rho = [\texttt{Push}\ (\rho\ x)]$　ここで、x はローカル変数です。 $\mathcal{C}[\![i]\!]\ \rho = [\texttt{Pushint}\ i]$ $\mathcal{C}[\![e_0\ e_1]\!]\ \rho = \mathcal{C}[\![e_1]\!]\ \rho$ ++ $\mathcal{C}[\![e_0]\!]\ \rho^{+1}$ ++ [Mkap]　ここで、$\rho^{+n}\ x = (\rho\ x) + n$ です。

3.3.4　プログラムのコンパイル

一連のコンパイル スキームを使用してコンパイラを説明します。各スーパーコンビネータの定義は、コンパイル スキーム $\mathcal{SC}$ を使用してコンパイルされます。各スーパーコンビネータ用に生成されたコンパイル済みコードは、図 3.3 に定義されています。コンパイル スキーム $\mathcal{SC}$、$\mathcal{R}$、および $\mathcal{C}$ に対応するのは、**コンパイラ関数** `compileSc`、`compileR`、および `compileC` です。これらのそれぞれを順番に検討します。

`compile` 関数は、プログラムを G マシンの初期状態にします。最初のコード シーケンスは、グローバルな `main` を見つけて評価します。ヒープは、宣言された各グローバルのノードが含まれるように初期化されます。`globals` には、グローバル名からそれらに提供される `NGlobal` ノードへのマップが含まれます。

```
compile :: CoreProgram -> GmState
compile program
  = (initialCode, [], heap, globals, statInitial)
    where (heap, globals) = buildInitialHeap program
```

初期ヒープを構築し、定義された各グローバルのグローバル ノードのマップを提供するために、buildInitialHeap を使用します。これは、テンプレート マシンの場合とまったく同じです。

```
buildInitialHeap :: CoreProgram -> (GmHeap, GmGlobals)
buildInitialHeap program
  = mapAccuml allocateSc hInitial compiled
    where compiled = map compileSc (preludeDefs ++ program) ++
                     compiledPrimitives
    --where
    --  compiled = map compileSc program
```

buildInitialHeap 関数は mapAccuml を使用して、コンパイルされた各グローバルにノードを割り当てます。型 [gmCompiledSC] を持つコンパイル済みで (必要に応じて) 発生するコンパイル。

```
type GmCompiledSC = (Name, Int, GmCode)
```

関数 allocateSc は、コンパイルされたスーパーコンビネータ引数に新しいグローバルを割り当て、新しいヒープとグローバルが格納されているアドレスを返します。

```
allocateSc :: GmHeap -> GmCompiledSC -> (GmHeap, (Name, Addr))
allocateSc heap (name, nargs, instns)
  = (heap', (name, addr))
    where (heap', addr) = hAlloc heap (NGlobal nargs instns)
```

初期状態では、マシンにプログラムの値を評価してもらいたい。これは単にグローバル main の値であることを思い出してください。

```
initialCode :: GmCode
initialCode = [Pushglobal "main", Unwind]
```

各スーパーコンビネータは、図 3.3 の $\mathcal{SC}$ スキームを実装する compileSc を使用してコンパイルされます。これは、スーパーコンビネータ名、スーパーコンビネータが簡約される前に必要な引数の数、およびスーパーコンビネータに関連付けられたコード シーケンスを含むトリプルを返します。

```
compileSc :: (Name, [Name], CoreExpr) -> GmCompiledSC  -- SCスキーム
compileSc (name, env, body)
  = (name, length env, compileR body (zip2 env [0..]))
```

これは、図 3.3 の $\mathcal{R}$ スキームに対応する compileR を使用します。

```
compileR :: GmCompiler  -- R スキーム
compileR e env = compileC e env ++ [Slide (length env + 1), Unwind]
--compileR e env = compileC e env ++ [Slide (length env + 1), Unwind]
```

各コンパイラ スキームには、同じ型 gmCompiler があります。

```
type GmCompiler = CoreExpr -> GmEnvironment -> GmCode
```

コンパイルスキームから写像 ρ を連想リストとして表すことができるという事実を使用します。このリストから変数のオフセットを検索できるだけでなく、スタックにある引数の数を計算することもできます。これは、Slide 命令で絞り出すスタック要素の数を調べるために compileR で使用されます。リストのタイプは gmEnvironment で、次のように定義されています。

```
type GmEnvironment = ASSOC Name Int
```

これは、図 3.3 の $\mathcal{C}$ スキームに対応する compileC を使用して、スーパーコンビネータ本体のインスタンス化を構築します。

```
compileC :: GmCompiler  -- C スキーム
compileC (EVar v) env
  | elem v (aDomain env) = [Push n]
  | otherwise = [Pushglobal v]
  where n = aLookup env v (error "Can't happen")
compileC (ENum n) env = [Pushint n]
compileC (EAp e1 e2) env = compileC e2 env ++ compileC e1 (argOffset 1 env) ++ [M
```

関数 argOffset を使用して、スタック オフセットを変更できます。env が ρ を実装する場合、(argOffset n env) は ρ^{+n} を実装します。

```
argOffset :: Int -> GmEnvironment -> GmEnvironment
argOffset n env = [(v, n + m) | (v, m) <- env]
```

コンパイル例

K コンビネータのコンパイルを見てみましょう。この関数をコンパイルするとき、次の式を評価することから始めます。

```
compileSc ("K", ["x", "y"], EVar "x")
```

タプルの最初の要素は名前 (この場合は K) です。2 番目は引数リストです (この場合、x と y の 2 つの変数があります)。タプルの 3 番目のコンポーネントは、スーパーコンビネータの本体です (この例では、変数 x だけです)。

この式を書き直すと、次のようになります。

```
("K", 2, compileR (EVar "x") [("x", 0), ("y", 1)])
```

結果のトリプルは、名前 (K)、スーパーコンビネータを削減するために必要な引数の数 (この場合は 2 つ)、およびインスタンス化を実行するためのコード シーケンスで構成されます。この式を書き直すと、このスーパーコンビネータのコード シーケンスが生成されます。環境は式 `[("x", 0), ("y", 1)]` で表されることに注意してください。これは、本体をインスタンス化すると、x へのポインターが引数スタックの一番上にあり、y へのポインターがスタックの x のすぐ下にあることを示しています。

```
("K", 2, compileC (EVar "x") [("x", 0), ("y", 1)] ++ [Slide 3, Unwind])
```

compileR 関数は、compileC を使用して本体をコンパイルし、最後に Slide および Unwind 命令を追加するように定義されています。

本体をコンパイルするには、x を検索し、それがスタックの一番上にあることを確認します。Push 0 を使用して、スタックのトップのコピーを作成するコードを生成します。

```
("K", 2, [Push 0, Slide 3, Unwind])
```

演習 3.3 プレリュードの定義から S コンビネータの変換の同等のシーケンスを書き出します。S が次のように定義されていることを思い出してください。

```
S f g x = f x (g x)
```

Appendix B に示す単純なプログラムのいずれかでコンパイラとマシンを実行して、最終結果を確認してください (S は標準プレリュードにあります)。

プリミティブ

この最小限の G マシンにはプリミティブがないため、実装するものは何もありません。

```
compiledPrimitives :: [GmCompiledSC]
compiledPrimitives = []
```

3.3.5　結果の表示

多くの状態コンポーネントは抽象データ型である (したがって、直接出力できない) ため、マシンが生成する状態のプリティ プリンターを定義する必要があります。一度に表示すると出力が膨大になり、あまり参考にならないことも事実です。印刷は、showResults によって制御されます。スーパーコンビネータ コード シーケンス、状態遷移、および最終統計の 3 つの出力が生成されます。

```
showResults :: [GmState] -> [Char]
showResults states
  = iDisplay (iConcat [iStr "Supercombinator definitions", iNewline,
                       iInterleave iNewline (map (showSC s) (getGlobals s)),
                       iNewline, iNewline,
                       iStr "State transitions", iNewline, iNewline,
                       iLayn (map showState states),
                       iNewline, iNewline,
                       showStats (last states)])
    where (s:ss) = states
```

これらのそれぞれを順番に取り上げて、showSC から始めます。これにより、グローバルに関連付けられた一意のグローバル ヒープ ノードでスーパーコンビネータのコードが検出され、showInstructions を使用してコード シーケンスが出力されます。

```
showSC :: GmState -> (Name, Addr) -> Iseq
showSC s (name, addr)
  = iConcat [ iStr "Code for ", iStr name, iNewline,
              showInstructions code, iNewline, iNewline]
    where (NGlobal arity code) = (hLookup (getHeap s) addr)
```

次に、showInstructions を使用してコード シーケンスを出力します。

```
showInstructions :: GmCode -> Iseq
showInstructions is
  = iConcat [iStr " Code:{",
             iIndent (iInterleave iNewline (map showInstruction is)),
             iStr "}", iNewline]
```

個々の命令の出力は、showInstruction によって提供されます。

```
showInstruction :: Instruction -> Iseq
showInstruction Unwind         = iStr "Unwind"
```

```
showInstruction (Pushglobal f) = (iStr "Pushglobal ") ‘iAppend‘ (iStr f)
showInstruction (Push n)       = (iStr "Push ")       ‘iAppend‘ (iNum n)
showInstruction (Pushint n)    = (iStr "Pushint ")    ‘iAppend‘ (iNum n)
showInstruction Mkap           = iStr "Mkap"
showInstruction (Slide n)      = (iStr "Slide ")      ‘iAppend‘ (iNum n)
```

出力の次の主要な部分は状態遷移です。 これらは、showState を使用して個別に処理されます。

```
showState :: GmState -> Iseq
showState s
  = iConcat [showStack s, iNewline,
             showInstructions (getCode s), iNewline]
```

ダイアグラムに対応するために、印刷されたスタックの一番下にスタックの一番上を配置したいと考えています。この目的のために、スタックを逆にします。

```
showStack :: GmState -> Iseq
showStack s
  = iConcat [iStr " Stack:[",
             iIndent (iInterleave iNewline
                      (map (showStackItem s) (reverse (getStack s)))),
             iStr "]"]
```

スタック アイテムは、showStackItem を使用して表示されます。スタックに格納されているアドレスと、それが指すヒープ内のオブジェクトを出力します。

```
showStackItem :: GmState -> Addr -> Iseq
showStackItem s a
  = iConcat [iStr (showaddr a), iStr ": ",
             showNode s a (hLookup (getHeap s) a)]
```

関数 showNode は、グローバル名とヒープ アドレスの関連リストを逆にして、検出されたグローバル ノードを表示する必要があります。

```
showNode :: GmState -> Addr -> Node -> Iseq
showNode s a (NNum n) = iNum n
showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
                             where v = head [n | (n,b) <- getGlobals s, a==b]
showNode s a (NAp a1 a2) = iConcat [iStr "Ap ", iStr (showaddr a1),
                                    iStr " ", iStr (showaddr a2)]
```

最後に、showStats を使用して、累積された統計を出力します。

```
showStats :: GmState -> Iseq
showStats s
  = iConcat [ iStr "Steps taken = ", iNum (statGetSteps (getStats s))]
```

以上で基本的な G マシンの説明を終わります。今後は、より洗練されたものにする方法を検討していきます。

3.3.6　Mark 1 G マシンの改良

演習 3.4 プログラム `main = S K K 3` を実行します。何ステップかかりますか？テンプレートインスタンス化マシンで取得した値と異なるのはなぜですか? 取られたステップを比較することは、マシンの公平な比較だと思いますか?

演習 3.5 Appendix B の他のプログラムを実行してみてください。この単純な機械には算術がないことを思い出してください。

演習 3.6 Pushglobal で使用したのと同じトリックを使用して Pushint を実装することができます: 個別の数値ごとに、ヒープ内に一意のノードを作成します。たとえば、最初に `Pushint 2` を実行するとき、ノード `NNum 2` のヒープ内のアドレスに"2"を関連付けるように gmGlobals を更新します。

遷移規則では、n という名前のグローバルが既に存在する場合、このグローバル ノードを再利用できます。

(3.13)						
	`Pushint` $n : i$		s	h	$m[n : a]$	
	$\Longrightarrow$	i	$a : s$	h	m	

そうでない場合は、新しいノードを作成してグローバル マップに追加します。

(3.14)						
	`Pushint` $n : i$		s	h		m
	$\Longrightarrow$	i	$a : s$	$h[a :$ `NNum` $n]$		$m[n : a]$

このスキームの利点は、Pushint が実行されるたびにヒープ内の同じ番号のノードを再利用できることです。Pushint 命令のこの新しい遷移 pushint を実装します。グローバル コンポーネントのアクセス関数を定義して、putGlobals と呼ぶ必要があります。

3.4 Mark 2: 遅延評価させる

Mark 1 G マシンを遅延評価させるために、いくつかの小さな変更を加えます。巻き戻す前に元の式のルートノードを上書きしないため、Mark 1 マシンは現時点では遅延評価していません。この更新については、セクション 2.1.5 で説明します。Mark 2 マシンでは、スーパーコンビネータの本体をインスタンス化した後、元の redex のルートを、新しく構築されたインスタンスを指す間接ノードで上書きするという考え方です。その効果は、マシンが前回の redex 簡約の際にインスタンス化された値を「覚えている」ため、それを再計算する必要がないということです。

この変更を次のように実装します。Mark 1 マシンでは、各スーパーコンビネータのコードは [`Slide` $(n+1)$; `Unwind`] で終了していました。更新をキャプチャするには、これを [`Update` n; `Pop` n; `Unwind`] に置き換えます。これを次の図に示します。この図では、`#` を使用して間接ノードを表しています。

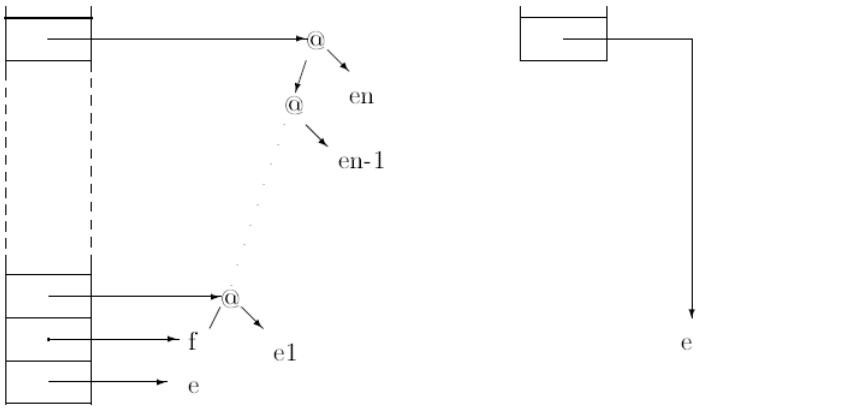


Figure 3.4: Mark 1 G-machine (executing `Slide n+1`)

図 3.4 は、Mark 1 マシンが `Slide` n +1 命令を実行する方法を示しています。

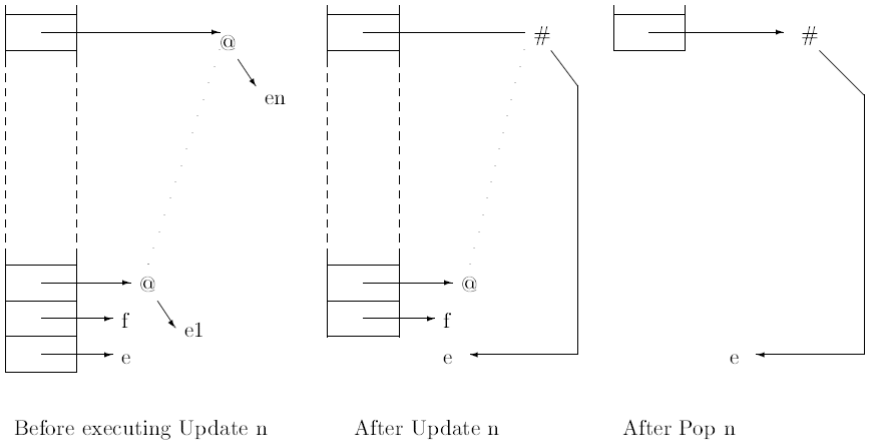


Figure 3.5: Mark 2 G-machine (executing `[Update n, Pop n]`)

図 3.5 では、Mark 2 マシンがシーケンス [`Update` n; `Pop` n;] を実行しています。これは、[`Slide` $n+1$] の遅延置換として使用することを提案するシーケンスです。`Update` 命令は、スーパーコンビネータ本体の新しく作成されたインスタンスでルートノードを上書きする役割を果たします。`Pop` 命令は、引数が不要になったため、スタックから引数を削除するために使用されます。

まず、データ構造に必要な変更を考えてみましょう。

3.4.1　データ構造

前回生成した単一の命令 `Slide` $n+1$ の代わりに、一連の命令 [`Update` n, `Pop` n] を生成します。したがって、これらの命令を新しい命令セットに含める必要があります。

```
data Instruction
 = Unwind
 | Pushglobal Name
 | Pushint Int
 | Push Int
 | Mkap
 | Update Int  -- Mark2で追加
 | Pop Int
instance Eq Instruction
  where
    Unwind       == Unwind       = True
    Pushglobal a == Pushglobal b = a == b
```

```
  Pushint    a == Pushint    b = a == b
  Push       a == Push       b = a == b
  Mkap         == Mkap         = True
  Update     a == Update     b = a == b
  _            == _            = False
```

演習 3.7 関数 showInstruction を変更して、新しい命令が表示されるようにします。

間接ノードを実装するには、ヒープに新しいノード タイプが必要です: 間接ノードに使用する NInd です。

```
data Node
  = NNum    Int          -- Numbers
  | NAp     Addr Addr    -- Applications
  | NGlobal Int  GmCode  -- Globals
  | NInd    Addr         -- Indirections  -- Mark2 で追加
instance Eq Node
  where
    NNum    a   == NNum    b   = a == b -- needed to check conditions
    NAp     a b == NAp     c d = False  -- not needed
    NGlobal a b == NGlobal c d = False  -- not needed
    NInd    a   == NInd    b   = False  -- not needed
```

ここでも、データ型の拡張を反映するように、表示関数 showNode を再定義する必要があります。

演習 3.8 showNode に必要な変更を加えます。

2 つの新しい命令のセマンティクスはまだ与えていません。これは以下で行われます。

3.4.2 評価器

Update n 命令の効果は、$n+1$ 番目のスタック項目を、スタックの一番上にある項目への間接指定で上書きすることです。このアドレッシングモードは、[Peyton Jones 1987] で使用されているものとは異なることに注意してください。この命令の意図した関数適用では、$a_1, \ldots, a_n$ はスパインを形成する n 個の関数適用ノードであり、a_0 は関数ノードです。

(3.15)	Update n : i	$a : a_0 : \ldots : a_n : s$	h		m
	$\Longrightarrow$ i	$a_0 : \ldots : a_n : s$	$h[a_n :$ NInd $a]$		m

Pop n 命令は、n 個のスタック項目を単純に削除します。繰り返しますが、Mark 2 G マシンでは、$a_1, \ldots, a_n$ は redex のスパインを形成する関数適用ノードです。

(3.16)						
	Pop $n : i$	$a_1 : \ldots : a_n : s$	h	m
	$\Longrightarrow$	i	s	h	m

スタックアイテムのトップが間接の場合、Unwind の遷移も定義する必要があります。その結果、現在のスタックアイテムが間接参照が指すアイテムに置き換えられます。

(3.17)					
	[Unwind]	$a_0 : s$	$h[a_0 : \texttt{NInd}\ a]$	m
	$\Longrightarrow$ [Unwind]	$a : s$	h	m

演習 3.9 Mark 1 マシンの dispatch 関数を変更して、新しい命令を組み込みます。新しい遷移規則を実装します。

3.4.3　コンパイラ

コンパイラに対する唯一の変更点は、$\mathcal{R}$ スキームによって生成されたコードにあります。新しい定義を図 3.6 に示します。

$\mathcal{R}[\![e]\!]\ \rho\ d$ は、アリティ d のスーパーコンビネータのために、
環境 ρ で式 e をインスタンス化するコードを生成し、
次に、結果のスタックのアンワインドに進みます。

$\mathcal{R}[\![e]\!]\ \rho\ d = \mathcal{C}[\![e]\!]\ \rho$ ++ [Update d, Pop d, Unwind]

演習 3.10 compileR を変更して、新しい $\mathcal{R}$ スキームを実装します。

演習 3.11 次のプログラムで遅延評価を実行します。

```
twice f x = f (f x)
id x = x
main = twice twice id 3
```

何ステップかかりますか？Mark1 マシンで取得した値と異なるのはなぜですか? マシンのステップ数を比較するのは公正ですか?

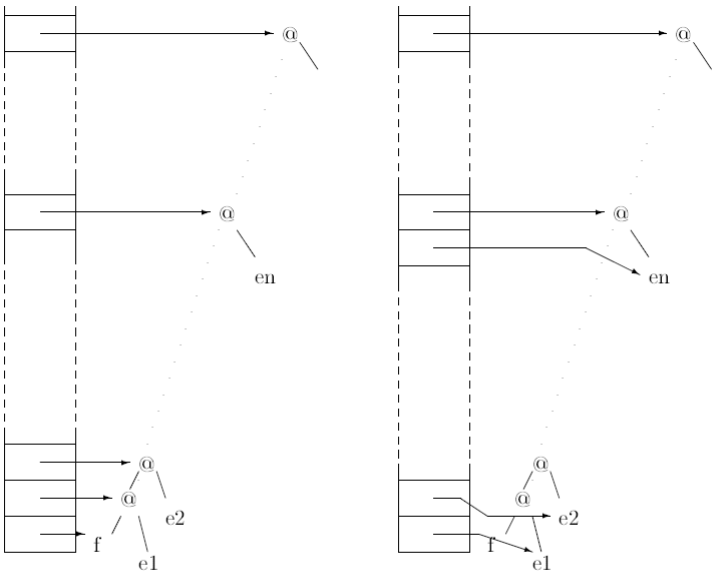
3.5 Mark 3: let(rec) 式

本体に `let(rec)` **バインド変数**を含むスーパーコンビネータをコンパイラが受け入れるように、言語を拡張します。これらは、コンストラクタ `Elet` によってデータ型 `coreExpr` で表されます。3 つの引数を取ります。定義を再帰的に処理するかどうかを示すブール フラグ、定義自体、および定義を使用する式です。

ローカル定義を追加してマシンを拡張する前に、スタックをもう一度見てみましょう。特に、変数のより効率的な**アクセス方法**を定義しようとします。効率の引数に加えて、ローカルにバインドされた変数へのアクセスを、関数パラメータのバインドに使用されるものと同じにしたいと考えています。

スタックからの引数アクセス

アンワインドプロセスがスーパーコンビネータノード `f` に到達し、スーパーコンビネータが n 個の引数を取るとします。Mark 1 マシンでは、スタックは図 3.7 の左側の図に示す状態になります。



Marks 1 and 2 Mark 3 onwards

Figure 3.7: Stack layout on entry to function `f`

スーパーコンビネータ `f` に到達したので、Mark 3 G マシンでは、スタックがわずかに変更されています。同等の Mark 3 スタックを図 3.7 の右側の

図に示します。 上位 n 要素は式 e1 ... en を直接指しています。ここで重要な点は、変数へのアクセスが高速になることです (変数が少なくとも 1 回アクセスされる場合)。これは、右側の引数を取得するために、関数適用ノードを 1 回しか見ていないためです。

これにより、スーパーコンビネータで仮パラメータに代入される式へのアクセス効率が向上します。
Mark1 マシンに関して:

- Push 命令で関数 getArg はもう必要ありません。
- しかし、十分な引数でスーパーコンビネータを Unwind するときは、スタックを再配置する必要があります。

Update を実行できるように、redex のルートへのポインターを保持していることに注意してください。

命令への影響

新しいスタックレイアウトを使用することを選択した場合、対処するために特定のマシン命令を変更する必要があります。影響を受ける命令は、Push と Unwind です。引数を取得するために関数適用ノードを「調べる」必要がないため、Push 命令を変更する必要があります。

$$
\begin{array}{|l|rlll|}
\hline
(3.18) & \texttt{Push}\ n : i & a_0 : \ldots : a_n : s & h & m \\
 & \Longrightarrow \quad i & a_n : a_0 : \ldots : a_n : s & h & m \\
\hline
\end{array}
$$

新しいスタックレイアウトに必要なその他の変更は、Unwind がスタックを再配置する必要があることです。十分な引数を持つスーパーコンビネータがスタックの一番上にある場合は常に、この再配置が必要です。Unwind の新しい遷移規則は次のとおりです。

$$
\begin{array}{|l|rlll|}
\hline
(3.19) & [\texttt{Unwind}] & a_0 : \ldots : a_n : s & h & \left[\begin{array}{l} a_0 : \texttt{NGlobal}\ n\ c \\ a_1 : \texttt{NAp}\ a_0\ a'_1 \\ \ldots \\ a_n : \texttt{NAp}\ a_{n-1}\ a'_n \end{array}\right] \quad m \\
 & \Longrightarrow \quad \mathrm{c} & a'_1 : \ldots : a'_n : a_n : s & h & m \\
\hline
\end{array}
$$

Unwind のこの定義は、n がゼロの場合に適切に機能することに注意してください。

演習 3.12 dispatch 関数と新しい命令セットの新しい遷移を書き直します。スタックを再配置するには、rearrange 関数を使用する必要があります。

```
rearrange :: Int -> GmHeap -> GmStack -> GmStack
rearrange n heap as
  = take n as' ++ drop n as
    where as' = map (getArg . hLookup heap) (tl as)
```

演習 3.13 Appendix B のいくつかのサンプルプログラムでコンパイラと新しい抽象マシンをテストし、実装が引き続き機能することを確認します。

3.5.1 ローカルにバインドされた変数

ここで、最初に非再帰的なケースを考慮して、let(rec) 式の実装に戻ります。式 let x_1 = e_1; ...; x_n = e_n in e の変数 x_1 ... x_n は、式 e_1 ... e_n が作成されると、スーパーコンビネータへの引数と同じ方法で処理できます。つまり、変数 x_1 ... x_n にオフセットを介してスタックにアクセスし、環境を使用してそれらの位置を記録します。

ローカル定義を構築するためのコードが Code であると仮定すると、図 3.8 に示す一連のアクションが必要になります。最初に、スタックにはスーパーコンビネータへの引数へのポインタが含まれます。ローカル定義を構築するコードが実行されると、スタック上に n 個の新しいポインタができます。x_i を e_i へのポインタにマップする新しい環境で、 let 式の本体の作成に進むことができます。最後に、式 e_1 ... e_n へのポインタをスタックから捨てる必要があります。

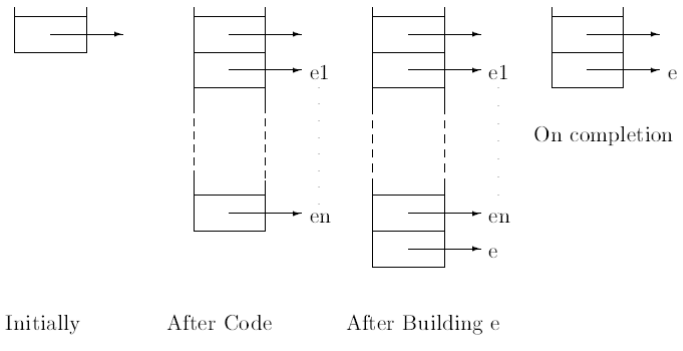


Figure 3.8: Stack usage in non-recursive local definitions

n 個の新しい変数をスタック $(x_1 \ldots x_n)$ に追加したため、e をコンパイルするために使用する変数マップでこの事実に注意する必要があります。ローカル

バインディングを構築するコード (Code と呼びます) は、単純に各式 $e_1 \ldots e_n$ のグラフを作成し、グラフの断片のアドレスをスタックに残します。

変数 $x_1 \ldots x_n$ のいずれかを使用する本体式 e を作成した後、スタックから $e_1 \ldots e_n$ へのポインタを削除する必要があります。これは、Slide 命令を使用して達成されます。非再帰的なローカル定義をコンパイルするための完全なスキームは、図 3.10 (p.109) に示されています。

再帰的なローカル定義の状況はより複雑です。変数 x_1 ... x_n がスコープ内にあるように、式 e_1 ... e_n のそれぞれをコンパイルする必要があります。これを行うには、グラフに空のノードを作成し、それらへのポインタをスタックに残します。次に、各式 e_1 ... e_n は、非再帰ケースの本体のコンパイルに使用したのと同じ変数マップを使用してコンパイルされます。各式のコンパイル済みコードの最後に、空のノードを正しいグラフで上書きする Update 命令を配置します。これを行うには、n 個の空のグラフノードを作成する新しい命令 Alloc n が 1 つ必要です。図 3.9 では、空のグラフノードは ? 記号で表されます。

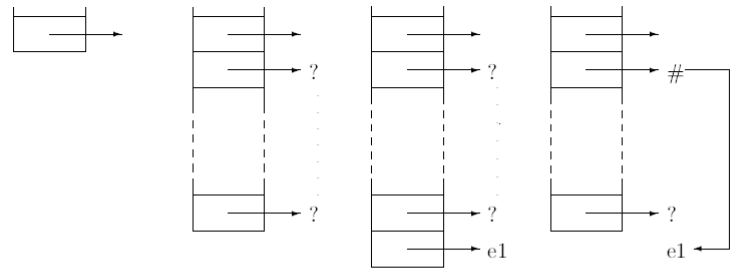


Figure 3.9: Constructing a recursively defined expression: e1

図 3.9 に示すプロセスは、各式 e_1 ... e_n が処理されるまで繰り返す必要があります。本体 e のコードのコンパイルは、非再帰的なローカル定義の前のケースと同じです。Mark3 マシン用の新しいデータ型を追加します。

3.5.2　データ構造

命令データ型には、新しい Alloc 命令と Mark1 マシンからの Slide 命令を含む、Mark2 マシンのすべての命令が含まれます。

演習 3.14 Alloc と Slide が含まれるように、データ型 instruction を変更します。また、これらの新しい命令に対応するために、関数 showInstruction を変更する必要があります。

3.5.3 評価器

Mark3 G マシンの場合、ヒープ内に n 個の場所を作成する Alloc 命令を追加する必要があります。これらの場所を使用して、ローカルにバインドされた式を保存する場所をマークします。これらのノードは、無効なヒープアドレス hNull を指す間接ノードとして最初に作成されます。Alloc によって作成されたこれらのノードは上書きされるため、それらにどの値を割り当てるかは問題ではありません。

Alloc 命令の遷移関数である alloc を実装するには、補助関数 allocNodes を使用します。必要なノード数と現在のヒープを指定すると、変更されたヒープと間接ノードのアドレスのリストで構成されるペアが返されます。

```
allocNodes :: Int -> GmHeap -> (GmHeap, [Addr])
allocNodes 0 heap = (heap, [])
allocNodes (n+1) heap = (heap2, a:as)
                        where (heap1, as) = allocNodes n heap
                              (heap2, a) = hAlloc heap1 (NInd hNull)
```

演習 3.15 新しい命令のケースを使用して、dispatch 関数を拡張します。AllocNodes を使用して、Alloc 命令の遷移関数である alloc を実装する必要があります。

3.5.4 コンパイラ

コンパイラに対する唯一の変更点は、$\mathcal{C}$ スキームがコードをコンパイルできるケースが 2 つ増えたことです。compileC への変更は簡単です。より広い範囲の coreExpr に対応できるようになりました。compileLetrec と compileLet という 2 つの新しい関数が必要です。

$\mathcal{C}[\![e]\!]\ \rho$ は、環境 ρ で e のグラフを構築するコードを生成し、
それへのポインターをスタックの一番上に残します。

$$
\begin{array}{lcll}
\mathcal{C}[\![f]\!]\ \rho & = & [\texttt{Pushglobal}\ f] & \text{ここで、} f \text{ はスーパーコンビネータです。} \\
\mathcal{C}[\![x]\!]\ \rho & = & [\texttt{Push}\ (\rho\ x)] & \text{ここで、} x \text{ はローカル変数です。} \\
\mathcal{C}[\![i]\!]\ \rho & = & [\texttt{Pushint}\ i] & \\
\mathcal{C}[\![e_0\ e_1]\!]\ \rho & = & \mathcal{C}[\![e_1]\!]\ \rho\ \texttt{++} & \\
 & & \mathcal{C}[\![e_0]\!]\ \rho^{+1}\ \texttt{++}\ [\texttt{Mkap}] & \text{ここで、} \rho^{+n}\ x = (\rho\ x) + n \text{ です。} \\
\mathcal{C}[\![\texttt{let}\ x_1\ \texttt{=}\ e_1\texttt{;}\ \ldots\texttt{;}\ x_n\ \texttt{=}\ e_n\ \texttt{in}\ e]\!]\ \rho & & & \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho^{+0}\ \texttt{++} & \\
 & & \ldots & \\
 & & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)}\ \texttt{++} & \\
 & & \mathcal{C}[\![e]\!]\ \rho'\ \texttt{++}\ [\texttt{Slide}\ n] & \text{ここで } \rho' = \rho^{+n}\,[x_1 \mapsto n-1,\ \ldots,\ x_n \mapsto 0] \text{ です。} \\
\mathcal{C}[\![\texttt{letrec}\ x_1\ \texttt{=}\ e_1\texttt{;}\ \ldots\texttt{;}\ x_n\ \texttt{=}\ e_n\ \texttt{in}\ e]\!]\ \rho & & & \\
 & = & [\texttt{Alloc}\ n]\ \texttt{++} & \\
 & & \mathcal{C}[\![e_1]\!]\ \rho'\ \texttt{++} & \\
 & & [\texttt{Update}\ n-1]\ \texttt{++} & \\
 & & \ldots & \\
 & & \mathcal{C}[\![e_n]\!]\ \rho'\ \texttt{++} & \\
 & & [\texttt{Update}\ 0]\ \texttt{++} & \\
 & & \mathcal{C}[\![e]\!]\ \rho'\ \texttt{++}\ [\texttt{Slide}\ n] & \text{ここで } \rho' = \rho^{+n}\,[x_1 \mapsto n-1,\ \ldots,\ x_n \mapsto 0] \text{ です。}
\end{array}
$$

```
compileC :: GmCompiler
compileC (EVar v) args
  | elem v (aDomain args) = [Push n]
  | otherwise             = [Pushglobal v]
  where n = aLookup args v (error "")
compileC (ENum n)    env = [Pushint n]
compileC (EAp e1 e2) env = compileC e2 env ++
                           compileC e1 (argOffset 1 env) ++
                           [Mkap]
compileC (ELet recursive defs e) args
  | recursive = compileLetrec compileC defs e args
  | otherwise = compileLet    compileC defs e args
```

compileLet の定義は、図 3.10 の仕様に従います。引数として、本体 e のコンパイルスキーム comp、定義 defs、および現在の環境 env を受け取ります。マシンの新しいバージョンでこの関数を書き直す必要がないように、コンパイラパラメータを提供しました。

```
compileLet :: GmCompiler -> [(Name, CoreExpr)] -> GmCompiler
compileLet comp defs expr env
```

```
    = compileLet' defs env ++ comp expr env' ++ [Slide (length defs)]
      where env' = compileArgs defs env
```

新しい定義のコンパイルは、関数 compileLet' によって実行されます。

```
compileLet' :: [(Name, CoreExpr)] -> GmEnvironment -> GmCode
compileLet' [] env = []
compileLet' ((name, expr):defs) env
  = compileC expr env ++ compileLet' defs (argOffset 1 env)
```

compileLet はまた、compileArgs を使用して、本体のコンパイル用のスタックへのオフセットを変更します。

```
compileArgs :: [(Name, CoreExpr)] -> GmEnvironment -> GmEnvironment
compileArgs defs env
  = zip (map first defs) [n-1, n-2 .. 0] ++ argOffset n env
    where n = length defs
```

例

この例では、不動点コンビネータ Y のコードがどのようにコンパイルされるかを示します。使用する定義は次のとおりです。

```
Y f = letrec x = f x in x
```

これは、いわゆる「結び目」不動点コンビネータです。 結果のコードを実行すると、なぜこの名前になっているのかがわかります。上記の定義がコンパイルされると、compileSc 関数はスーパーコンビネータのコードを作成する必要があります。

```
compileSc ("Y", ["f"], ELet True [("x", EAp (EVar "f") (EVar "x"))] (EVar "x"))
```

これは、変数 f の環境で compileR 関数を呼び出します。最初にスーパーコンビネータの名前 (Y) とその引数の数 (1) を作成しました。

```
("Y", 1, compileR e [("f", 0)])
where e = ELet True [("x", EAp (EVar "f") (EVar "x"))] (EVar "x")
```

便宜上、式の本体を e と呼びます。関数 compileR は compileC を呼び出し、最後に片付けコードを配置します。

```
("Y", 1, compileC e [("f", 0)] ++ [Update 1, Pop 1, Unwind])
```

図 3.10 のコンパイルスキームを参照すると、letrec をコンパイルするには、最初に新しい環境を作成することがわかります。図では、これは ρ' と呼ばれます。 この例では、p と呼びます。これは初期環境の拡張であり、ローカル変数 x にスタックの場所も与えます。

```
("Y", 1, [Alloc1] ++
         compileC (EAp (EVar "f") (EVar "x")) p ++ [Update 0] ++
         compileC (EVar "x") p ++ [Slide 1] ++
         [Update 1, Pop 1, Unwind])
         where p = [("x", 0), ("f", 1)]
```

コード生成は、コンパイルスキームと同じ方法でレイアウトされます。compileC を含む式を単純化すると、次のようになります。

```
("Y", 1, [Alloc 1] ++
         [Push 0, Push 2, Mkap] ++ [Update 0] ++
         [Push 0] ++ [Slide 1] ++
         [Update 1, Pop 1, Unwind])
```

これにより、次のコードシーケンスが得られます。

```
("Y", 1, [Alloc 1, Push 0, Push 2, Mkap, Update 0, Push 0,
          Slide 1, Update 1, Pop 1, Unwind])
```

このコードがどのように実行されるかは、図 3.11 で確認できます。Y スーパーコンビネータのこの定義は、5 番目の命令として Update 0 を実行するときにグラフに結び目を作るため、「結び目」と呼ばれます。演習 3.18 として残されているため、残りの手順は示していません。

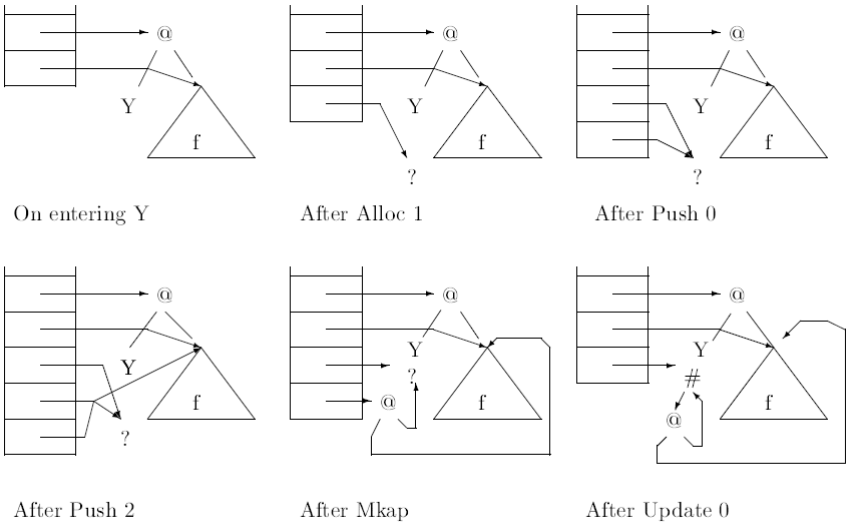


Figure 3.11: Execution of code for Y

演習 3.16 `letrec` のコンパイルは、図 3.10 で定義されています。この操作を実行するには、関数 `compileLetrec` を実装します。

演習 3.17 新しいコンパイラと命令セットが適切に動作することを示すために、どのテスト プログラムを使用しますか?

演習 3.18 スーパーコンビネータ `Y` 用に生成されたコードを実行するか、そうでない場合は、残りの状態遷移を図 3.11 のスタイルで描画します。

演習 3.19 スーパーコンビネータ `Y` の短い代替コードシーケンスを指定します。それはまだ「結び目」バージョンを構築する必要があります。

演習 3.20 言語に `letrec` 構文がない場合、不動点コンビネータ `Y` をどのように定義しますか? この定義は、例で使用した定義とどう違うのでしょうか?

3.6　Mark 4: プリミティブの追加

このセクションでは、基本的な操作を G マシンに追加します。これは便利です。基本演算とは、加算、乗算などの演算を意味します。このセクションでは、実行例として足し算を使用します。

加算命令は Add と呼ばれます。ヒープから 2 つの数値を追加し、結果をヒープ内の新しいノードに配置します。2 つの引数のアドレスはスタックの一番上にあります。これは、結果のアドレスがその後配置される場所です。次の遷移規則があります。

$$
\begin{array}{|l|rcccl|c|}
\hline
(3.21) & \texttt{Add} : i & a_0 : a_1 : s & d & h[a_0 : \texttt{NNum}\ n_0, \quad a_1 : \texttt{NNum}\ n_1] & m \\
 & \Longrightarrow \quad i & a : s & d & h[a : \texttt{NNum}\ (n_0\ +\ n_1)] & m \\
\hline
\end{array}
$$

必要な残りの操作を実装するために、他の命令で G マシンを拡張し続けることができますが、その前に、ここで何か見逃していないかどうかを検討するために一時停止しましょう。問題は、スタックの一番上にある 2 つのオブジェクトが数値である場合にのみルールが適用されることです。遅延評価をサポートするマシンで作業しているため、これが常に当てはまると想定する正当な理由はありません。テンプレートマシンで Add は、その引数が評価されることを確認しました。G マシンでは、命令を単純に保ちたいので、引数が既に評価されていることが保証されている状況でのみ Add を使用します。

代わりに、Eval 命令を使用して命令セットをさらに拡張します。これは、次の制約を満たします。

> 次の状態にあるとします。
>
> $$\boxed{\texttt{Eval} : i \quad a : s \quad h \quad m}$$
>
> 命令シーケンス i で実行が再開されるたびに、状態は次のようになります。
>
> $$\boxed{i \quad a : s \quad h' \quad m}$$
>
> スタックの一番上にあるアイテムは WHNF になります。Eval が終了しない可能性もあります。 これは、スタックの一番上から指されたノードに WHNF がない場合に当てはまります。

アドレスがスタックの一番上にあるノードが既に WHNF にある場合、Eval 命令は何もしません。実行するリダクションがある場合、Eval のアクション

は WHNF に対する評価を実行することです。この呼び出しが終了すると、ヒープ コンポーネント以外は何も変更されずに実行が再開されます。これは、プログラミング言語の実装で伝統的に使用されているサブルーチンの呼び出しと戻りの構造に似ています。この機能を実装する古典的な方法は、スタックを使用することです。スタックは、サブルーチン呼び出しが完了したときに再開できるように、マシンの現在のコンテキストを十分に保存します。

Mark 4 G マシンでは、このスタックは**ダンプ**と呼ばれ、最初のコンポーネントがコード シーケンスで、2 番目のコンポーネントがスタックであるペアのスタックです。これは、元のコード シーケンスと元のスタックを復元する必要があることを除いて、テンプレート マシンのダンプ (セクション 2.6 を参照) に似ています。したがって、両方のコンポーネントがダンプに保持されます。

3.6.1 データ構造

ダンプコンポーネントを追加して、G マシンの状態を拡張します。前に説明したように、これは評価器への再帰呼び出しを実装するために使用されます。

```
type GmState = ( GmCode, -- current Instruction
                 GmStack, -- current Stack
                 GmDump, -- current Dump
                 GmHeap, -- Heap of Nodes
                 GmGlobals, -- Global adresses in Heap
                 GmStats) -- Statistics
```

ダンプ自体は、dumpItem のスタックです。これらはそれぞれ、元の計算を再開するときに使用する命令ストリームとスタックで構成されるペアです。

```
type GmDump = [GmDumpItem]
type GmDumpItem = (GmCode, GmStack)
```

この新しいコンポーネントを追加するときは、以前に指定したすべてのアクセス機能を変更する必要があります。ダンプ用のアクセス機能も追加する必要があります。

```
getDump :: GmState -> GmDump
getDump (i, staxk, dump, heap, globals, stats) = dump

putDump :: GmDump -> GmState -> GmState
putDump dump' (i, stack, dump, heap, globals, stats)
  = (i, stack, dump', heap, globals, stats)
```

G マシンの状態でパターンマッチングを行ったのはアクセス関数のみであることに注意してください。新しいコンポーネントを状態に追加した結果としての他の関数への変更は、もはや必要ありません。

演習 3.21 他のアクセス機能に関連する変更を加えます。

状態の新しい定義に加えて、いくつかの新しい指示も必要です。Mark3 マシンからのすべての命令を再利用します。

```
data Instruction
  = Slide Int
  | Alloc Int
  | Update Int
  | Pop Int
  | Unwind
  | Pushglobal Name
  | Pushint Int
  | Push Int
  | Mkap
```

さらに、Eval 命令を含めます。

```
  | Eval
```

次の算術命令:

```
  | Add | Sub | Mul | Div | Neg
```

および次の比較命令:

```
  | Eq | Ne | Lt | Le | Gt | Ge
```

また、Cond 命令にはプリミティブ形式の条件分岐も含まれています。

```
  | Cond GmCode GmCode
```

演習 3.22 ケースを showInstruction に追加して、新しい指示をすべて出力します。

3.6.2　状態の表示

これを機に showState の定義を見直し、dump コンポーネントを表示するようにしました。

```
showState :: GmState -> Iseq
showState s
  = iConcat [showStack s, iNewline,
             showDump s, iNewline,
             showInstructions (getCode s), iNewline]
```

したがって、showDump を定義する必要があります。

```
showDump :: GmState -> Iseq
showDump s
  = iConcat [iStr " Dump:[",
             iIndent (iInterleave iNewline (map showDumpItem (reverse (getDump s)))),
             iStr "]"]
```

これには、関数 showDumpItem が必要です。

```
showDumpItem :: GmDumpItem -> Iseq
showDumpItem (code, stack)
  = iConcat [iStr "<",
             shortShowInstructions 3 code, iStr ", ",
             shortShowStack stack, iStr ">"]
```

関数 shortShowInstructions を使用して、命令ストリームの最初の 3 つの命令のみをダンプ項目に出力します。これは通常、計算が再開される場所を示すのに十分です。

```
shortShowInstructions :: Int -> GmCode -> Iseq
shortShowInstructions number code
  = iConcat [iStr "{", iInterleave (iStr "; ") dotcodes, iStr "}"]
    where codes = map showInstruction (take number code)
          dotcodes | length code > number = codes ++ [iStr "..."]
                   | otherwise            = codes
```

同様に、ダンプ アイテムのスタック コンポーネントの完全な詳細も必要ないため、shortShowStack を使用します。

```
shortShowStack :: GmStack -> Iseq
shortShowStack stack
  = iConcat [iStr "[",
             iInterleave (iStr ", ") (map (iStr . showaddr) stack),
             iStr "]"]
```

3.6.3　新しい命令遷移

評価命令

実際には、ダンプを操作する命令はほとんどありません。まず、Eval 自体があり、スタックの最上位のノードが WHNF にない場合は常に新しいダンプ項目を作成します。2 つ目は、評価が完了したときにダンプ項目をポップする Unwind 命令への変更です。

まず、新しい Unwind 命令について説明します。スタックに保持されている式が WHNF にある場合、Unwind はダンプから古いコンテキストを復元し、復元された古いスタックにスタック内の最後のアドレスを配置できます。これは、数値の場合の遷移で明確にわかります。[1]

(3.22)								
		[Unwind]	$a : s$	$\langle i', s' \rangle : d$	$h[a : \texttt{NNum}\ n]$	m
	$\Longrightarrow$	i'	$a : s'$	d	h	m

アドレス a の式は整数であるため WHNF にあるため、古い命令シーケンス i' を復元すると、スタックはアドレス a を先頭にした古いスタック s' になります。Unwind の他のすべてのトランジションは、Mark 3 マシンの場合と同じままです (ダンプ コンポーネントがその状態にあることを除いて)。

これで、Eval のルールを指定できるようになりました。 スタック s の残りと残りの命令 i をダンプ項目としてダンプに保存します。新しいコード シーケンスはちょうど巻き戻されており、新しいスタックにはシングルトン a が含まれています。

(3.23)						
		$\texttt{Eval} : i$	$a : s$	d	h	m
	$\Longrightarrow$	[Unwind]	$[a]$	$\langle i, s \rangle : d$	h	m

算術命令

すべての 2 項算術演算子には、次の一般的な遷移規則があります。実装したい算術演算子が $\odot$ であると仮定しましょう。 命令の遷移規則は次のとおりです。

(3.24)							
		$\odot : i$	$a_0 : a_1 : s$	d	$h[a_0 : \texttt{NNum}\ n_0,$	$a_1 : \texttt{NNum}\ n_1]$	m
	$\Longrightarrow$	i	$a : s$	d	$h[a : \texttt{NNum}\ (n_0\ \odot\ n_1)]$		m

[1] ルールは、ダンプが空でない場合にのみ適用されます。 ダンプが空の場合、マシンは終了しています。

何が起こったかというと、スタックの一番上にある 2 つの数値に 2 項演算子 $\odot$ が適用されています。ヒープに入力された結果のアドレスは、スタックに配置されます。Neg 命令は、スタックの一番上にある数値を否定するため、遷移規則があります。

$$\begin{array}{|c|rcccrc|}\hline (3.25) & \texttt{Neg} : i & a : s & d & h[a : \texttt{NNum}\ n] & m \\ \Longrightarrow & i & a' : s & d & h[a' : \texttt{NNum}\ (\text{-}n)] & m \\ \hline\end{array}$$

すべての二項演算がいかに類似しているかに注目してください。最初にヒープから 2 つの数値を抽出し、次に演算を実行し、最後に答えをヒープに戻します。これは、ヒープからの抽出 (値の「ボックス化解除」と呼びます) をパラメーター化する高階関数を作成する必要があることを示唆しています。そして、実行したい特定の操作とともに、ヒープに挿入します (これを値の「ボクシング」と呼びます)。

最初にボクシング操作を書きましょう。boxInteger は数値と初期状態を取り、数値がヒープに配置された新しい状態と、スタックの一番上に残されたこの新しいノードへのポインターを返します。

```
boxInteger :: Int -> GmState -> GmState
boxInteger n state
  = putStack (a: getStack state) (putHeap h' state)
    where (h', a) = hAlloc (getHeap state) (NNum n)
```

ステートからアドレス a の整数を抽出するには、unboxInteger を使用します。

```
unboxInteger :: Addr -> GmState -> Int
unboxInteger a state
  = ub (hLookup (getHeap state) a)
    where ub (NNum i) = i
          ub n        = error "Unboxing a non-integer"
```

一般的なモナド演算子は、ボックス化関数 box、ボックス化解除関数 unbox、およびボックス化されていない値に対する演算子 op に関して指定できるようになりました。

```
primitive1 :: (b -> GmState -> GmState) -- boxing function
              -> (Addr -> GmState -> a) -- unbixing function
              -> (a -> b)               -- operator
              -> (GmState -> GmState)   -- state transition
primitive1 box unbox op state
```

```
    = box (op (unbox a state)) (putStack as state)
      where (a:as) = getStack state
```

汎用二項演算子は、primitive2 を使用して同様の方法で実装できるようになりました。

```
primitive2 :: (b -> GmState -> GmState) -- boxing function
              -> (Addr -> GmState -> a) -- unbixing function
              -> (a -> a -> b)          -- operator
              -> (GmState -> GmState)   -- state transition
primitive2 box unbox op state
  = box (op (unbox a0 state) (unbox a1 state)) (putStack as state)
    where (a0:a1:as) = getStack state
```

もっと明確に言うと、arithmetic1 はすべての単項算術を実装し、arithmetic2 はすべての二項算術を実装します。

```
arithmetic1 :: (Int -> Int)            -- arithmetic operator
               -> (GmState -> GmState) -- state transition
arithmetic1 = primitive1 boxInteger unboxInteger

arithmetic2 :: (Int -> Int -> Int)     -- arithmetic operation
               -> (GmState -> GmState) -- state transition
arithmetic2 = primitive2 boxInteger unboxInteger
```

注意深い読者が予想するように、これらの関数の一般性をこの章の後半で利用します。

演習 3.23 マシンの新しい命令遷移をすべて実装します。dispatch 関数を変更して、新しい命令を処理します。高階関数の primitive1 と primitive2 を使用して、演算子を実装する必要があります。

演習 3.24 Eval 命令の完了時に間接ノードがスタックの一番上に残らないのはなぜですか?

比較命令

すべての**比較演算子**には、次の一般的な遷移規則があります。実装したい比較演算子が $\odot$ であると仮定すると、命令 $\odot$ の遷移規則は次のようになります。

(3.26)		$\odot : i$	$a_0 : a_1 : s$	d	$h[a_0 : \texttt{NNum}\ n_0,\ \ a_1 : \texttt{NNum}\ n_1]$		m
	$\Longrightarrow$	i	$a : s$	d	$h[a : \texttt{NNum}\ (n_0\ \odot\ n_1)]$		m

スタックの一番上にある 2 つの数値には、2 項演算子 $\odot$ が適用されています。ヒープに入力された結果のアドレスは、スタックに配置されます。これは算術演算とほぼ同じです

違いは、たとえば == という操作は、整数ではなくブール値を返すことです。これを修正するために、次のルールを使用してブール値を整数に変換します。

- True を整数 1 で表します。
- False は整数 0 で表します。

Primitive2 の使用を可能にするために、boxBoolean を定義します。

```
boxBoolean :: Bool -> GmState -> GmState
boxBoolean b state
  = putStack (a: getStack state) (putHeap h' state)
    where (h',a) = hAlloc (getHeap state) (NNum b')
          b' | b          = 1
             | otherwise = 0
```

この定義を使用して、comparison と呼ばれる一般的な比較関数を書くことができます。この関数は、ブール値のボックス化関数、整数のボックス化解除関数 (unboxInteger)、および比較演算子を取ります。そして状態遷移を返します。

```
comparison :: (Int -> Int -> Bool) -> GmState -> GmState
comparison = primitive2 boxBoolean unboxInteger
```

最後に、if 関数のコンパイルに使用する Cond 命令を実装します。次の 2 つの遷移規則があります。

(3.27)		Cond	i_1	$i_2 : i$	$a : s$	d	$h[a :$ NNum 1$]$	m
	$\Longrightarrow$			i_1 ++ i	s	d	h	m

最初のケース (スタックの一番上に数字の 1 がある場合) では、最初の分岐を取ります。これは、命令 i の実行を続ける前に、命令 i_1 を実行することを意味します。

(3.28)		Cond	i_1	$i_2 : i$	$a : s$	d	$h[a :$ NNum 0$]$	m
	$\Longrightarrow$			i_2 ++ i	s	d	h	m

あるいは、スタックの一番上の番号が 0 の場合、最初に命令シーケンス i_2 を実行し、次にシーケンス i を実行します。

演習 3.25 比較命令と Cond 命令の遷移を実装します。

3.6.4 コンパイラ

これらの新しい命令を利用して算術式をコンパイルするには、最終的にコンパイラを変更する必要があります。現時点では、最小限の変更のみを行います。苦労して追加した算術命令を使用できるようになります。まず、compile 関数は、初期ダンプが空で、初期コード シーケンスがこれまでに使用したものとは異なる新しい初期状態を作成する必要があります。

```
compile :: CoreProgram -> GmState
compile program
  = (initialCode, [], [], heap, globals, statInitial)
    where (heap, globals) = buildInitialHeap program

initialCode :: GmCode
initialCode = [Pushglobal "main", Eval]
```

演習 3.26 最初の命令シーケンスが変更されたのはなぜですか? 古いものを保持するとどうなりますか？

コンパイラを拡張する最も簡単な方法は、新しい組み込み関数ごとに G マシン コードを compiledPrimitives に追加することです。シーケンスの最初の 4 つの命令により、引数が整数に評価されることが保証されます。

```
compiledPrimitives :: [GmCompiledSC]
compiledPrimitives
  = [("+", 2, [Push 1, Eval, Push 1, Eval, Add, Update 2, Pop 2, Unwind]),
     ("-", 2, [Push 1, Eval, Push 1, Eval, Sub, Update 2, Pop 2, Unwind]),
     ("*", 2, [Push 1, Eval, Push 1, Eval, Mul, Update 2, Pop 2, Unwind]),
     ("/", 2, [Push 1, Eval, Push 1, Eval, Div, Update 2, Pop 2, Unwind]),
```

否定関数も追加する必要があります。これは 1 つの引数しかとらないため、1 つの引数のみを評価します。

```
     ("negate", 1, [Push 0, Eval, Neg, Update 1, Pop 1, Unwind]),
```

比較操作は次のように実装されます。

```
     ("==", 2, [Push 1, Eval, Push 1, Eval, Eq, Update 2, Pop 2, Unwind]),
     ("~=", 2, [Push 1, Eval, Push 1, Eval, Ne, Update 2, Pop 2, Unwind]),
     ("<",  2, [Push 1, Eval, Push 1, Eval, Lt, Update 2, Pop 2, Unwind]),
```

```
("<=", 2, [Push 1, Eval, Push 1, Eval, Le, Update 2, Pop 2, Unwind]),
(">",  2, [Push 1, Eval, Push 1, Eval, Gt, Update 2, Pop 2, Unwind]),
(">=", 2, [Push 1, Eval, Push 1, Eval, Ge, Update 2, Pop 2, Unwind]),
```

`if` 関数は、分岐に `Cond` を使用するようにコンパイルされます。

```
("if", 3, [Push 0, Eval, Cond [Push 1] [Push 2],
           Update 3, Pop 3, Unwind])]
```

演習 3.27 新しい命令とコンパイラが機能することを確認するために、Appendix B のどのテストプログラムを使用しますか?

3.7　Mark 5: 算術演算のより良い処理に向けて

現時点での G マシンの実装方法では、各演算子はコンパイルされたプリミティブの 1 つを経由して呼び出されます。この配置を改善するために、しばしば算術演算子を直接呼び出すことができることを観察することができます。例えば、次のような簡単なプログラムを考えてみよう。

```
main = 3+4*5
```

これにより、現在のコンパイラを使用した場合、以下のようなコードが生成されます。

```
[Pushint 5, Pushint 4, Pushglobal "*", Mkap, Mkap,
 Pushint 3, Pushglobal "+", Mkap, Mkap, Eval]
```

このコードを実行すると、33 ステップを要し、11 ヒープノードを使用することになります。最初に思いつくのは、関数 '+' と '*' の呼び出しの代わりに、Add と Mul という命令を使えばいいのでは、ということでしょう。これは次のような改善されたコードにつながります。

```
[Pushint 5, Pushint 4, Mul, Pushint 3, Add]
```

これは、わずか 5 ステップで実行され、5 つのヒープノードを使用します

3.7.1　問題点

次の例題のプログラムを考えるとき、考えられる問題が発生します。

```
main = K 1 (1/0)
```

これにより、以下のようなコードが生成されます。

```
[Pushint 0, Pushint 1, Pushglobal "/", Mkap, Mkap,
 Pushint 1, Pushglobal "K", Mkap, Mkap, Eval]
```

前の例のパターンでいくと、以下のようなコードを生成してみることができるかもしれません。

```
[Pushint 0, Pushint 1, Div,
 Pushint 1, Pushglobal "K", Mkap, Mkap, Eval]
```

問題は、K を簡約する**前に**除算演算子が適用され、その結果、**ゼロ除算のエラー**が発生することです。正しいコンパイラは、このようなエラーを出さないようなコードを生成しなければなりません。

何が起こったかというと、私たちのコードが厳しすぎるのです。その結果、評価する必要のない式が評価され、本来は発生しないはずのエラーが発生しています。同じような問題は、終端でない式が不用意に評価されたときにも起こります。

3.7.2 解決策

この問題を解決するには、ある表現が出現する文脈を記録しておくことです。ここでは、2 つのコンテキスト[2]を区別します。

Strict 式の値は WHNF で要求されます。

Lazy 式の値は、WHNF で要求されるかもしれないし、されないかもしれません。

各コンテキストに対応するコンパイルスキームがあり、式 を G マシンの命令列にコンパイルするスキームがあります。Strict なコンテキストでは、このコンパイルスキームは $\mathcal{E}$ スキームであり、Lazy なコンテキストでは、すでに見た $\mathcal{C}$ スキームを使用することになります。

できるだけ多くの Strict なコンテキストを見つけたいです。なぜなら、これらのコンテキストはより良いコードを生成することを可能にするからです。スーパーコンビネータがインスタンス化されるときは常に、その値を WHNF で評価したいからです。このことから、スーパーコンビネータの本体は常に Strict なコンテキストで評価することができると結論づけられます。また、式が WHNF で評価されれば、いくつかの部分式が WHNF で評価されることが分かっている式もあります。

Strict なコンテキストの式のクラスは、再帰的に記述することができます。

- スーパーコンビネータ定義の本体中の式は、Strict なコンテキストにある。
- $e_0 \odot e_1$ が Strict なコンテキストで出現し、ここで $\odot$ が算術演算子または比較演算子である場合、式 e_0 と e_1 も Strict なコンテキストにあります。
- もし `negate` e が Strict なコンテキストで発生した場合、e という式も Strict なコンテキストになる。

[2] より多くのコンテクストを区別することが可能です。投影法 [Wadler 1987] と評価変換器 [Burn 1991] はこのための 2 つの方法である。

- `if` e_0 e_1 e_2 という式が Strict なコンテキストで出現する場合、e_0, e_1, e_2 という式も Strict なコンテキストになる。
- もし `let(rec)` Δ `in` e が Strict なコンテキストで現れたら、式 e も Strict なコンテキストにあることになります。

例として、次のスーパーコンビネータ `f` の本体を考えてみると、このことがよくわかるでしょう。

```
f x y = (x+y) + g (x*y)
```

`(x+y)` と `g (x*y)` はどちらも Strict なコンテキストで評価されます。なぜならスーパーコンビネータの本体は Strict なコンテキストで評価されるからです。最初のケースでは、`+`が Strict なコンテキストを伝播するので、`x` と `y` は Strict なコンテキストで評価されます。2 番目の式では、ユーザー定義のスーパーコンビネータ `g` が存在するため、部分式 `x*y` は評価されないものとしてコンパイルされる。

このことは、Strict コンテキストコンパイラ $\mathcal{E}$ を再帰的な方法で実装できることを示唆しています。各スーパーコンビネータの本体は Strict コンテキストで評価されるので、$\mathcal{R}$ スキームから $\mathcal{E}$ スキームの関数を呼び出す必要があります。これは上記の第一の点を満たしています。また、コンテキスト情報を部分式に伝達するために、算術式に対して $\mathcal{E}$ スキームを再帰的に呼び出します。

新しいコンパイラのスキームは図 3.12 に定義されています。

$\mathcal{R}[\![e]\!]\ \rho\ d$ は、アリティ d のスーパーコンビネータに対して、
式 e を環境 ρ でインスタンス化し、その結果のスタックを巻き戻すコードを生成します。

$\mathcal{R}[\![e]\!]\ \rho\ d = \mathcal{E}[\![e]\!]\ \rho$ ++ [Update d, Pop d, Unwind]

$\mathcal{E}[\![e]\!]\ \rho$ は，環境 ρ において式 e を WHNF に評価するコードをコンパイルし、
スタックの先頭に式へのポインタを残します。

$$
\begin{aligned}
\mathcal{E}[\![i]\!]\ \rho &= [\texttt{Pushint}\ i] \\
\mathcal{E}[\![\texttt{let}\ x_1 = e_1;\ \ldots;\ x_n = e_n\ \texttt{in}\ e]\!]\ \rho & \\
&= \mathcal{C}[\![e_1]\!]\ \rho^{+0}\ \texttt{++} \\
&\quad \ldots \\
&\quad \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)}\ \texttt{++} \\
&\quad \mathcal{E}[\![e]\!]\ \rho'\ \texttt{++}\ [\texttt{Slide}\ n] \\
&\quad \text{ここで } \rho' = \rho^{+n}[x_1 \mapsto n-1,\ \ldots,\ x_n \mapsto 0] \text{ です。} \\
\mathcal{E}[\![\texttt{letrec}\ x_1 = e_1;\ \ldots;\ x_n = e_n\ \texttt{in}\ e]\!]\ \rho & \\
&= [\texttt{Alloc}\ n]\ \texttt{++} \\
&\quad \mathcal{C}[\![e_1]\!]\ \rho'\ \texttt{++}\ [\texttt{Update}\ n-1]\ \texttt{++} \\
&\quad \ldots \\
&\quad \mathcal{C}[\![e_n]\!]\ \rho'\ \texttt{++}\ [\texttt{Update}\ 0]\ \texttt{++} \\
&\quad \mathcal{E}[\![e]\!]\ \rho'\ \texttt{++}\ [\texttt{Slide}\ n] \\
&\quad \text{ここで } \rho' = \rho^{+n}[x_1 \mapsto n-1,\ \ldots,\ x_n \mapsto 0] \text{ です。} \\
\mathcal{E}[\![e_0\ \texttt{+}\ e_1]\!]\ \rho &= \mathcal{E}[\![e_1]\!]\ \rho\ \texttt{++}\ \mathcal{E}[\![e_0]\!]\ \rho^{+1}\ \texttt{++}\ [\texttt{Add}] \\
&\quad \text{他の算術式や比較式についても同様。} \\
\mathcal{E}[\![\texttt{negate}\ e]\!]\ \rho &= \mathcal{E}[\![e]\!]\ \rho\ \texttt{++}\ [\texttt{Neg}] \\
\mathcal{E}[\![\texttt{if}\ e_0\ e_1\ e_2]\!]\ \rho &= \mathcal{E}[\![e_0]\!]\ \rho\ \texttt{++}\ [\texttt{Cond}\ (\mathcal{E}[\![e_1]\!]\ \rho)\ (\mathcal{E}[\![e_2]\!]\ \rho)] \\
\mathcal{E}[\![e]\!]\ \rho &= \mathcal{C}[\![e]\!]\ \rho\ \texttt{++}\ [\texttt{Eval}] \\
&\quad \text{デフォルトケース}
\end{aligned}
$$

新しいスキームを使ってコンパイルできる組み込み演算子のセットを簡単に拡張できるように、builtInDyadic を採用しました。

```
builtInDyadic :: ASSOC Name Instruction
builtInDyadic
  = [("+", Add), ("-", Sub), ("*", Mul), ("div", Div),
     ("==", Eq), ("~=", Ne), (">=", Ge),
     (">", Gt), ("<=", Le), ("<", Lt)]
```

演習 3.28 既存のコンパイラ関数 compileR と compileE を、図 3.12 の $\mathcal{R}$ スキームと $\mathcal{E}$ スキームを実装するように修正します。builtInDyadic を使用する必要があります。

時折、新しいコンパイラを使うとマシンが失敗することがあります。必要

なのは、Unwind 命令に対して新しいルールを導入することです。

(3.29)		[Unwind]	$[a_0,\ \ldots,\ a_k]$	$\langle i,\ s\rangle : d$	$h[a_0 :$ NGlobal $n\ c]$	m
	$\Longrightarrow$	i	$a_k : s$	d	h	m ($k < n$ のとき)

これにより、Eval を使って数値だけでなく、あらゆるオブジェクトを WHNF に評価することができるようになりました。

演習 3.29 Unwind の新しい遷移規則を実装してください。新しい Unwind の遷移規則がないと失敗するプログラムを書いてください。

演習 3.30 冒頭で使用したサンプルプログラムの実行を、Mark4 マシンでの実行と比較してみてください。

```
main = 3+4*5
```

Appendix B の他のプログラムも試してみてください。

Strict なコンテキストをコンパイラに実装する方法は、コンパイラの理論から継承される属性の簡単な例である。Strict コンテキストを式の属性と見なすと、部分式はその親となる式から Strict コンテキストを継承する。一般的な理論は [Aho et al. 1986] で述べられている。

残念ながら、ある式を Strict なコンテキストでコンパイルすべきかどうかをコンパイル時に判断することは、一般的には不可能です。したがって、妥協点を受け入れるしかありません。本書では、限られた式の集合である算術式だけを特別な方法で扱いました。この解析をより一般的な式に拡張する研究が盛んに行われています [Burn 1991]。

3.8 Mark 6: データ構造の追加

本節では、G マシンを拡張して、任意のデータ構造を扱えるようにします。第 1 章で述べたように、データ構造を含むプログラムには、**コンストラクタ** `EConstr` と、**case 式** `ECase` という 2 つの新しいコア言語コンストラクタが必要です。Mark6 マシンの目標は、これらの式に対応したコードをコンパイルすることです。

3.8.1 概要

1.1.4 節で、タグ `t` とアリティ `a` を持つコンストラクタはコア言語では `Pack{t, a}` と表現されることを見ました。例えば、通常のリストデータ型は 2 つのコンストラクタを持っています。`Pack{1, 0}` と `Pack{2, 2}` です。これらはそれぞれ Miranda の [] と (:) に相当します。

Miranda では直接対応するものがない `case` 式は、コンストラクタに保持される値を検査するために使用されます。例えば、リストの長さを求める関数を次のように書きます。

```
length xs = case xs of
                  <1>       -> 0;
                  <2> y ys -> 1 + length ys
```

`case` 式の実行の仕方を見てみると、参考になることがあります。

1. `case` 式を評価するために、まず `xs` を WHNF に評価します。
2. この評価が行われると、どの選択肢を取るべきかがわかるようになります。評価された式のタグ（構造化データ・オブジェクトでなければなりません）が、どの選択肢を取るかを決定します。上の例では、長さについて
   - `xs` のコンストラクタのタグが `1` の場合、リストは空であり、最初の選択肢を取ります。従って、`0` を返します。
   - タグが `2` であれば、リストは空でないことになります。今回はコンストラクタのコンポーネント（`y` と `ys`）があります。リストの長さは `ys` の長さより 1 つ多くなります。

ここでは、コンストラクタを解体しようとするときはいつでも、正しい数の引数に適用されていると仮定します。このような状態のコンストラクタは**充足状態**であるといいます。例として、セクション 1.1.3 で `Cons` は 2 つの引数

を取るように定義されているので、2 つの式に適用されると充足状態になります。

また、コア言語プログラムは、構造化データオブジェクトである結果を返すことができるようになったことにも注目します。Mark5 G マシンは、構造化データオブジェクトを遅延評価スタイルで表示できるようにしなければなりません。まず、Mark5 マシンのデータ構造にどのような追加をする必要があるかを考えてみましょう。

3.8.2　データ構造

マシンが単なる数値ではない値を返すことができればいいのです。コンストラクタからなる値を返せるようにしたいのです。そのためには、構造体の構成要素を再帰的に評価し、その値を返す必要があります。そのためには、さらに別のコンポーネント、gmOutput を状態に追加する必要があります。これは、プログラムの結果を保持するためのものです。

```
type GmState =
  (GmOutput,  -- Current Output
   GmCode,    -- Current Instruction Stream
   GmStack,   -- Current Stack
   GmDump,    -- The Dump
   GmHeap,    -- Heap of Nodes
   GmGlobals, -- Global addresses in Heap
   GmStats)   -- Statistics
```

このコンポーネントは、文字列として定義されます。

```
type GmOutput = [Char]
```

アクセス関数は、当たり前のように書けばいいのです。

```
getOutput :: GmState -> GmOutput
getOutput (o, i, stack, dump, heap, globals, stats) = o

putOutput :: GmOutput -> GmState -> GmState
putOutput o' (o, i, stack, dump, heap, globals, stats)
  = (o', i, stack, dump, heap, globals, stats)
```

演習 3.31 残りのアクセス機能を適切に変更します。

ヒープ内のコンストラクタノードをサポートするために、node 型を NConstr で拡張します。これは、タグを表す正の数と、コンポーネントのリストを取ります。コンポーネントのリストはヒープ内のノードのアドレスのリストとして表されます。

```
data Node
  = NNum Int -- Numbers
  | NAp Addr Addr -- Applications
  | NGlobal Int GmCode -- Globals
  | NInd Addr
  | NConstr Int [Addr]
instance Eq Node
  where
    NNum a      == NNum b      = a == b -- needed to check conditions
    NAp a b     == NAp cd      = False  -- not needed
    NGlobal a b == NGlobal c d = False  -- not needed
    NInd a      == NInd b      = False  -- not needed
    NConstr a b == NConstr c d = False  -- not needed
```

3.8.3 結果の表示

表示したい新しい状態コンポーネントがあるため、関数 showState を再定義する必要があります。

```
showState :: GmState -> Iseq
showState s
  = iConcat [showOutput s,                iNewline,
             showStack s,                 iNewline,
             showDump s,                  iNewline,
             showInstructions (getCode s), iNewline]
```

出力コンポーネントはすでに文字列であるため、showOutput 関数は簡単です。

```
showOutput :: GmState -> Iseq
showOutput s = iConcat [iStr "Output:\"", iStr (getOutput s), iStr "\""]
```

他の唯一の変更 (新しい命令セットの showInstruction の変更を除く) は、コンストラクタノードを含めるようにデータ型を拡張したため、showNode で発生します。

```
showNode :: GmState -> Addr -> Node -> Iseq
showNode s a (NNum n)      = iNum n
showNode s a (NGlobal n g) = iConcat [iStr "Global ", iStr v]
                               where v = head [n | (n,b) <- getGlobals s, a==b]
showNode s a (NAp a1 a2)   = iConcat [iStr "Ap ", iStr (showaddr a1),
                                      iStr " ", iStr (showaddr a2)]
showNode s a (NInd a1)     = iConcat [iStr "Ind ", iStr (showaddr a1)]
showNode s a (NConstr t as)
  = iConcat [iStr "Cons ", iNum t, iStr " [",
             iInterleave (iStr ", ") (map (iStr.showaddr) as), iStr "]"]
```

3.8.4 命令セット

新しい命令セットが定義されました。Mark4 マシンに 4 つの新しい命令を追加するだけです。

```
data Instruction
  = Slide Int
  | Alloc Int
  | Update Int
  | Pop Int
  | Unwind
  | Pushglobal Name
  | Pushint Int
  | Push Int
  | Mkap
  | Eval
  | Add | Sub | Mul | Div
  | Neg
  | Eq | Ne | Lt | Le | Gt | Ge
  | Cond GmCode GmCode
```

マシンに追加された 4 つの新しい命令は次のとおりです。

```
  | Pack Int Int
  | Casejump [(Int, GmCode)]
  | Split Int
  | Print
```

演習 3.32 新しい命令セットに一致するように `showInstruction` を拡張します。

Pack 命令は単純です。 **充足したコンストラクタ**を構築するのに十分な引数がスタックにあると想定しています。存在する場合は、充足したコンストラクタの作成に進みます。 十分な引数がない場合、命令は未定義です。

$$(3.30)\quad \begin{array}{lllllll} & o & \texttt{Pack}\ t\ n : i & a_1 : \ldots : a_n : s & d & h & m \\ \Longrightarrow & o & i & a : s & d & h[a : \texttt{NConstr}\ t\ [a_1, \ldots, a_n]] & m \end{array}$$

Casejump の遷移ルールは、

(a) スタックの一番上のノードが WHNF にあること

(b) ノードが構造化データ オブジェクトであること

を想定しています。このオブジェクトのタグを使用して、選択肢の命令シーケンスの 1 つを選択すると、現在の命令ストリームの前に、選択された特定の選択肢のコードが追加されます。

$$(3.31)\quad \begin{array}{lllllll} & o & \texttt{Casejump}\ [\ldots, t\texttt{->}i', \ldots] : i & a : s & d & h[a : \texttt{NConstr}\ t\ ss] & m \\ \Longrightarrow & o & i' \texttt{ ++ } i & a : s & d & h & m \end{array}$$

これは、多方向へのジャンプと結合を指定する簡単な方法です。つまり、選択肢コード i' のコードによって現在のコード i を事前に修正することで、最初に選択肢のコードを実行し、次に主な式の残りのコードが必要とするもので再開するという効果を達成します。[3]

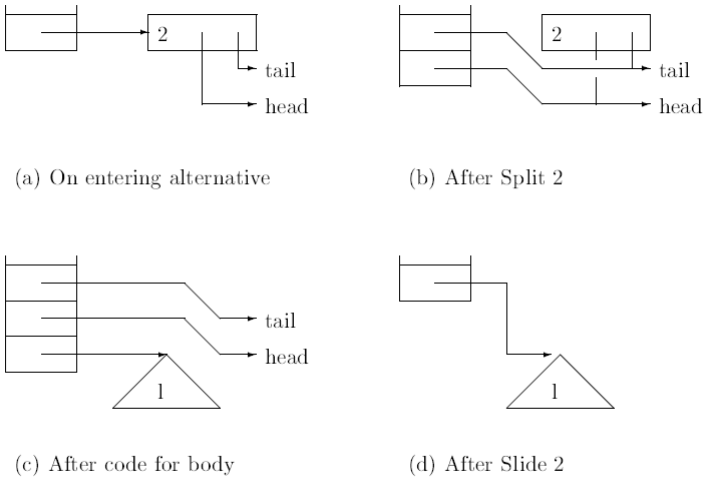
各選択肢のコードは、Split n 命令で始まり、Slide n 命令で終了します。n の値は、コンストラクタ内のコンポーネントの数によって決まります。Split 命令は、コンストラクタのコンポーネントにアクセスするために使用されます。

length 関数用に生成されたコードシーケンスを考えてみましょう。

```
[Push 0, Eval,
 Casejump [1 -> [Pushint 0]
           2 -> [Split 2, Push 1, Pushglobal "length", Mkap,
                 Eval, Pushint 1, Add, Slide 2]],
 Update 1,
 Pop 1,
 Unwind]
```

[3] Casejump を使用したコードシーケンスはフラットではないことに注意してください。ただし、各選択肢にラベルを付けて、ラベル付けされたコードアドレスにジャンプすることにより、必要なフラットコードシーケンスを構築できます。コード生成が不必要に複雑になるため、これは行っていません。

このパターンの実行を図 3.13 に示します。ここでは、現在のローカルバインディングのセットを拡張するために、Slide および Split 命令が一時的に使用されていることがわかります。



(a) On entering alternative　(b) After Split 2

(c) After code for body　(d) After Slide 2

Figure 3.13: Running compiled code for alternatives

length 関数が nil 以外のノードに適用されたと仮定すると、Casejump 命令を実行するときに、2 というラベルの付いた選択肢が取られます。これが最初の図 (a) です。

Split 2 命令は、コンストラクタノードをスタックに「unpack」します。これを図 (b) に示します。

選択肢の本体、つまりコードシーケンス

```
[Push 1, Pushglobal "length", Mkap, Eval, Pushint 1, Add]
```

を完成させた後、この length の呼び出しに対するリスト引数の長さがスタックの一番上になります。図 (c) で 1 とラベル付けされています。

実行を完了するには、head と tail へのポインタを削除します。これを図 (d) に示します。

Split の遷移規則は簡単です。

$$(3.32)\quad o\ \ \texttt{Split}\ n : i \quad a : s \quad d \quad h[a : \texttt{NConstr}\ t\ [a_1, \ldots, a_n]] \quad m$$
$$\Longrightarrow\quad o \quad i \quad a_1 : \ldots : a_n : s \quad d \quad h \quad m$$

次に、Print の遷移規則について説明します。Print には 2 つの遷移規則があります。コンストラクタと数値にそれぞれ 1 つずつです。

$$(3.33)\quad o \quad \texttt{Print} : i \quad a : s \quad d \quad h[a : \texttt{NNum}\ n] \quad m$$
$$\Longrightarrow\quad o\ \texttt{++}\ [n] \quad i \quad s \quad d \quad h \quad m$$

コンストラクタのルールは、コンストラクタの各コンポーネントをプリントするように調整する必要があるため、より複雑です。簡単にするために、コンポーネントのみをプリントします。

(3.34)								
	o	`Print` : i		$a : s$	d	$h[a :$ `NConstr` $t\ [a_1, \ldots, a_n]]$	m
	$\Longrightarrow$	o	i' `++` i	$a_1 : \ldots : a_n : s$	d	h	m

コード i' は次のとおりです。

$$\underbrace{[\texttt{Eval},\ \texttt{Print},\ \ldots,\ \texttt{Eval},\ \texttt{Print}]}_{n}$$

最後に、`Unwind` の新しいルールを追加する必要があります。これは、`NNum` のルールと同様に、`NConstr` を巻き戻すときに戻るように指示します。

(3.35)							
	o	[`Unwind`]	$a : s$	$\langle i',\ s' \rangle : d$		$h[a :$ `NConstr` $n\ as]$	m
	$\Longrightarrow$	o	i'	$a : s'$	d	h	m

演習 3.33 新しい遷移規則を実装し、`dispatch` 関数を変更します。

3.8.5 コンパイラ

図 3.14 では、$\mathcal{E}$ および $\mathcal{C}$ コンパイルスキームの新しいケースが示されています。

$\mathcal{E}[\![e]\!]\ \rho$ は、式 e を環境 ρ の WHNF に評価するコードをコンパイルし、
式へのポインタをスタックの一番上に残します。

$$\begin{aligned}
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ alts]\!]\ \rho &= \mathcal{E}[\![e]\!]\ \rho\ \texttt{++}\ [\texttt{Casejump}\ \mathcal{D}[\![alts]\!]\ \rho] \\
\mathcal{E}[\![\texttt{Pack\{}t\texttt{,}a\texttt{\}}\ e_1\ \ldots\ e_a]\!]\ \rho &= \mathcal{C}[\![e_a]\!]\ \rho^{+0}\ \texttt{++}\ \cdots\ \texttt{++}\ \mathcal{C}[\![e_1]\!]\ \rho^{+(a-1)}\ \texttt{++}\ [\texttt{Pack}\ t\ a]
\end{aligned}$$

$\mathcal{C}[\![e]\!]\ \rho$ は、環境 ρ で e のグラフを構築するコードを生成し、
それへのポインタをスタックの一番上に残します。

$$\mathcal{C}[\![\texttt{Pack\{}t\texttt{,}a\texttt{\}}\ e_1\ \ldots\ e_a]\!]\ \rho = \mathcal{C}[\![e_a]\!]\ \rho^{+0}\ \texttt{++}\ \cdots\ \texttt{++}\ \mathcal{C}[\![e_1]\!]\ \rho^{+(a-1)}\ \texttt{++}\ [\texttt{Pack}\ t\ a]$$

$\mathcal{D}[\![alts]\!]\ \rho$ は、`case` 式の選択肢のリストをコンパイルします。

$$\mathcal{D}[\![alt_1\ \ldots\ alt_n]\!]\ \rho = [\mathcal{A}[\![alt_1]\!]\ \rho\ , \ldots, \mathcal{A}[\![alt_n]\!]\ \rho]$$

$\mathcal{A}[\![alt]\!]\ \rho$ は、`case` 式の各選択肢のコードをコンパイルします。

$$\begin{aligned}
\mathcal{A}[\![\texttt{<}t\texttt{>}\ x_1\ \ldots\ x_n\ \texttt{->}\ body]\!]\ \rho &= t\ \texttt{->}\ [\texttt{Split}\ n]\ \texttt{++}\ \mathcal{E}[\![body]\!]\ \rho'\ \texttt{++}\ [\texttt{Slide}\ n] \\
&\quad \text{ここで } \rho' = \rho^{+n}\,[x_1 \mapsto 0,\ \ldots,\ x_n \mapsto n-1] \text{ です。}
\end{aligned}$$

case 式で選択される可能性のある選択肢を処理するために、補助的なコンパイルスキーム $\mathcal{D}$ および $\mathcal{A}$ が必要です。関数 compileAlts($\mathcal{D}$ スキームに対応) は、現在の環境を使用して選択肢のリストをコンパイルし、タグ付きコードシーケンスのリストを生成します。また、comp 引数 ($\mathcal{A}$ に対応) を使用して、各選択肢の本体をコンパイルします。現時点では、この引数は常に compileE' になります。

```
compileAlts :: (Int -> GmCompiler) -- compiler for alternative bodies
               -> [CoreAlt]        -- the list of alternatives
               -> GmEnvironment    -- the current environment
               -> [(Int, GmCode)]  -- list of alternative code sequences
compileAlts comp alts env
  = [(tag, comp
            (length names)
            body
            (zip names [0..] ++ argOffset (length names) env))
    | (tag, names, body) <- alts]
```

compileE' スキームは、compileE スキームを少し変更したものです。通常の compileE スキームによって生成されたコードの周りに Split と Slide を配置するだけです。

```
compileE' :: Int -> GmCompiler
compileE' offset expr env
  = [Split offset] ++ compileE expr env ++ [Slide offset]
```

演習 3.34 関連する変更を compile に加え、initialCode を変更して最後に Print 命令を追加します。

演習 3.35 新しいケースをコンパイラ関数 compileE および compileC に追加します。

演習 3.36 出力を「構造化された形式」で出力するには、どのような変更が必要ですか。これは、整数だけでなく、コンストラクタと括弧を出力コンポーネント gmOutput に配置することを意味します。

3.8.6　比較における新しい論理表現の使用

このセクションでは、構築した Mark6 マシンを変更して、ブール値の新しい表現を使用する方法を示します。最初に、ブール値を構造化データオブジェクトとして実装できることを確認します。True と False は、アリティがゼロでタグが 2 と 1 のコンストラクタとして表されます。

条件式をどのように実装しますか? これは、`if` のプログラムに新しい定義を追加することで実行できます。最初の引数に応じて、2 番目または 3 番目の引数のいずれかを返します。

```
if c t f = case c of
               <1> -> f;
               <2> -> t
```

必要な最初の変更は、比較操作にあります。これらには、次の一般的な遷移規則があります。

$$
\begin{array}{|c|rlllllll|l|}
\hline
(3.36) & & o & \odot : i & a_0 : a_1 : s & d & h[a_0 : \texttt{NNum}\ n_0, & a_1 : \texttt{NNum}\ n_1] & m \\
 & \Longrightarrow & o & i & a : s & d & h[a : \texttt{Constr}\ (n_0\ \odot\ n_1) & []] & m \\
\hline
\end{array}
$$

たとえば、`Eq` 命令では、2 つの数値 n_0 と n_1 が同じ場合は `2`(`True` のタグ) を返し、そうでない場合は `1`(`False` のタグ) を返す関数に置き換えます。

Mark4 マシン用に開発したコードの一部を再利用することで、遷移規則をすばやく実装できます。セクション 3.6.3 では、一般的な算術演算子と比較演算子を表す方法を見てきました。実際、比較関数の定義を構築した方法により、ブール値の新しい表現をほとんどすぐに使用できます。

ボックス化関数 `boxBoolean` は、比較演算と、スタックの一番上に 2 つの整数がある状態を取ります。スタックの一番上にある 2 つの整数を比較したブール値の結果がある新しい状態を返します。

```
boxBoolean :: Bool -> GmState -> GmState
boxBoolean b state
  = putStack (a: getStack state) (putHeap h' state)
    where (h',a) = hAlloc (getHeap state) (NConstr b' [])
          b' | b = 2          -- 2 is tag of True
             | otherwise = 1 -- 1 is tag of False
```

演習 3.37 Appendix B のサンプルプログラムをいくつか実行します。たとえば、階乗プログラムを試してください。

```
fac n = if (n==0) 1 (n * fac (n-1))
```

3.8.7 使用可能な言語の拡大

賢明な読者はお気づきかもしれませんが、コンパイラが失敗する `ECase` と `EConstr` に関するいくつかの正当な表現があります。コンパイルできない合

法的な表現は、次の 2 つのクラスに分類されます。

1. Strict でないコンテキスト (つまり、$\mathcal{C}$ スキームによってコンパイルされた式) での ECase の出現。
2. 適用される引数が少なすぎる式での EConstr の出現。

どちらの問題も、プログラム変換手法を使用して解決できます。ECase の解決策は、問題のある式をスーパーコンビネータにして、自由変数に適用することです。たとえば、次のプログラム

```
f x = Pack{2,2} (case x of <1> -> 1; <2> -> 2) Pack{1,0}
```

は以下の同等のプログラムに変換できます。

```
f x = Pack{2,2} (g x) Pack{1,0}
g x = case x of <1> -> 1; <2> -> 2
```

EConstr の解決策は、コンストラクタごとにスーパーコンビネータを作成することです。これは、コンストラクタを充足させるのに十分な自由変数で生成されます。ここに例があります。

```
prefix p xs = map (Pack{2,2} p) xs
```

これは次のように変換されます。

```
prefix p xs = map (f p) xs
f p x = Pack{2,2} p x
```

この問題を解決する別の方法は、Pushglobal 命令を変更して、「Pack{t,a}」という形式の名前を持つ関数に対して機能するようにすることです。そうすれば、上記の例の f のようなコンストラクタ関数を、状態のグローバルコンポーネントから探すだけでよいのです。関数がまだ存在しない場合は、その関数に関連付ける新しいグローバルノードを作成することができます。このノードは特に単純な構造持っているからです。

```
NGlobal a [Pack t a, Update 0, Unwind]
```

新しい遷移規則は、最初に、関数が既に存在する場合:

$$
\begin{array}{|l|l|}
\hline
(3.37) & o \quad \texttt{Pushglobal Pack}\{t,n\} : i \quad s \quad d \quad h \quad m[\texttt{Pack}\{t,n\} : a] \\
 & \Longrightarrow \quad o \qquad\qquad\qquad\qquad\quad i \quad a : s \quad d \quad h \quad m \\
\hline
\end{array}
$$

第二に、それがまだ存在しない場合:

(3.38)		o	Pushglobal Pack$\{t,n\}$:	i		s	d	h	m
	$\Longrightarrow$	o		i	$a:s$		d	$h[a :$ gNode$_{t,n}]$	$m[$Pack$\{t,n\} : a]$

ここで gNode$_{t,n}$ は以下の通りです。

$$\texttt{NGlobal}\ n\ [\texttt{Pack}\ t\ n,\ \texttt{Update}\ 0,\ \texttt{Unwind}]$$

その後、コンパイラは、未充足のコンストラクタノードを含む式のコードを直接生成できます。これは、不充足コンストラクタに対して次のコードを生成することによって行われます。

$$\mathcal{C}[\![\texttt{Pack}\{t,a\}]\!]\ \rho = [\texttt{Pushglobal "Pack}\{t,a\}\texttt{"}]$$

演習 3.38 Pushglobal 命令の pushglobal 関数に拡張機能を実装し、コンパイラを変更します。

3.9　Mark 7: さらなる改善

Mark 5 コンパイラの開発時に見たサンプルプログラムをもう一度考えてみましょう。

```
main = 3+4*5
```

これにより、Mark 6 コンパイラを使用した場合、以下のようなコードが生成されます。

```
[Pushint 5, Pushint 4, Mul, Pushint 3, Add]
```

このコードを実行すると、ヒープノードが 1 つ使用されます。これをさらに減らすことは可能でしょうか？

答えはイエスです。演算の中間値を表す数値のスタックを使用することで、演算のためのヒープアクセス数をさらに減らすことができるのです。Mark 7 マシンでは、これらの値は V スタックと呼ばれる新しいステートコンポーネントに保持されます。問題は、実機では数値をヒープに入れたり取り出したりするのは高くつく操作であることです。マシンのレジスタセットやスタックを利用する方がはるかに効率的です。Mark 7 G マシンではスタックを使うので、レジスタが足りなくなる心配はありません。

プログラム

```
main = 3+4*5
```

の新しいコードは、前回生成したコードと非常によく似ています。

```
[Pushbasic 5, Pushbasic 4, Mul, Pushbasic 3, Add, Mkint]
```

最初の命令 `Pushbasic 5` は、`5` を V スタックの一番上に置きます。次に `4` を V スタックにプッシュし、その後に乗算を行います。この命令は、その引数が V スタックにあることを想定しています。これは、同様に V スタックに答えを配置します。次の 2 つの命令は、V スタックの一番上にある値に `3` を足します。最後の命令 `Mkint` は、V スタックの一番上の値を取って、ヒープ内の数値ノードに入れ、この新しいノードへのポインタを S スタックの一番上に残します。

3.9.1　V スタックによる階乗関数の実行

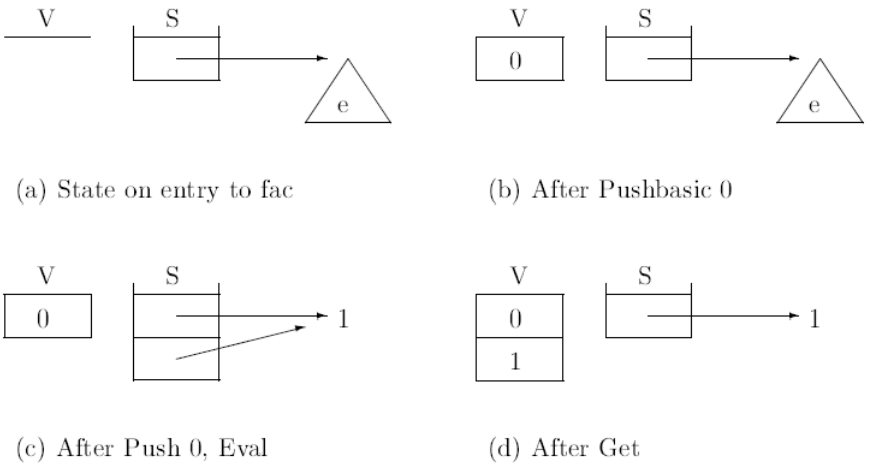
まず、Mark7 マシンを例にとって調べてみます。ここでは、次のように定義された階乗関数の実行を見ることにします。

```
fac n = if (n==0) 1 (n * fac (n-1))
```

Mark 7 コンパイラを使って、スーパーコンビネータ本体に以下のようなコード列を生成します。

```
[Pushbasic 0, Push 0, Eval, Get, Eq,
 Cond [Pushint 1, Update 1, Pop 1, Unwind]
      [Pushint 1, Push 1, Pushglobal "-", Mkap, Mkap, Pushglobal "fac", Mkap,
       Eval, Get, Push 0, Eval, Get, Mul, Mkint, Update 1, Pop 1, Unwind]
```

このコードが実行されるとき、V スタックは空であり、通常のスタックには 1 つのアイテムがあります。この 2 種類のスタックを区別するために、以後後者を S スタックと呼ぶことにします。



(a) State on entry to fac

(b) After Pushbasic 0

(c) After Push 0, Eval

(d) After Get

Figure 3.15: Mark 7 machine running `fac`

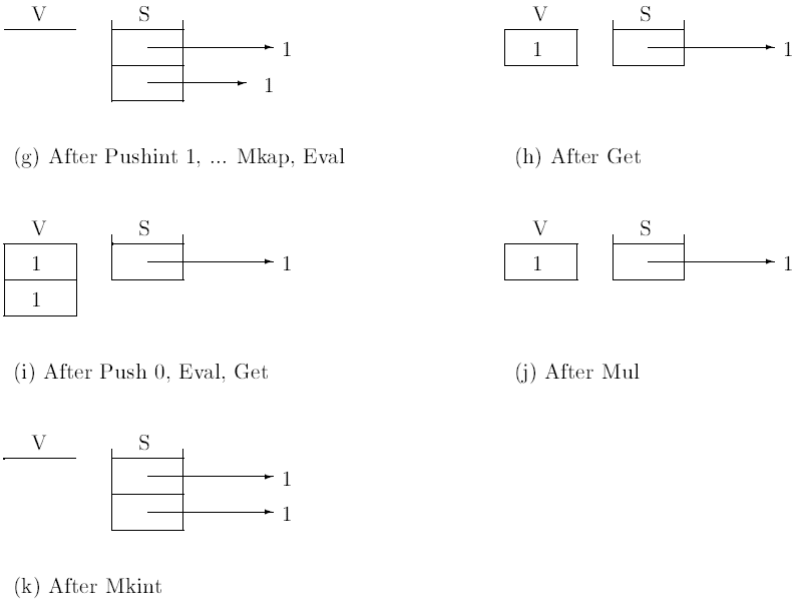
図 3.15 の (a) 図から、S スタックの上に `fac` の引数へのポインタがある初期状態を見ることができます。図 (b) では、`Pushbasic` 命令が実行されたときに、整数が V スタックにプッシュされる様子を示しています。図 (c) では、`fac` への引数が評価され、図 (d) では、`Get` 命令の効果が示されています。ヒープ内のノードから値を取り出し、V スタックに格納したものです。

(e) After Eq

(f) After Cond

Figure 3.16: Mark 7 machine running `fac`

図 (e)(図 3.16) は、Eq 命令が実行された後の状態を示しています。V スタックの 2 つの項目を比較した結果、両者が等しくないことを発見しました。Mark 7 G-machine は、V スタックのブール値 False を 1 として表現しています。図 (f) では、Cond 命令がこの値を検査し、どの分岐を実行するかを選択しています。



(g) After Pushint 1, ... Mkap, Eval

(h) After Get

(i) After Push 0, Eval, Get

(j) After Mul

(k) After Mkint

Figure 3.17: Mark 7 machine running fac

図 (g)（図 3.17）は fac (1-1) を構築し評価した後の状態です．次の命令は、新しく評価された値を V スタックにフェッチする Get です。図 (i) では、ノード 1 から評価し、V スタックに値をフェッチしていることがわかります。図 (j) では、Mul 命令によって V スタック内の 2 つの値が乗算され、その結果がスタックに戻されています。図 (k) では、Mkint 命令により、この結果を V スタックからヒープに移動し、S スタックに新しく生成されたノードのアドレスを記録しています。

ヒープにオブジェクトを作成しアクセスすることが、スタックにオブジェクトを保持することと比較して性能上のペナルティがあるマシンでは、V-stack の使用は改善されると期待されます。V スタックがどのように機能するかを見た後、Mark 7 マシンを実装するために G マシンに小さな修正を加えます。

3.9.2 データ構造

V スタックを使用することで、G マシンの各状態には、新しい状態コンポーネント gmVStack がその状態に追加されます。

```
type GmState = (GmOutput,  -- Current output
                GmCode,    -- Current instruction stream
                GmStak,    -- Current stack
                GmDump,    -- Current dump
                GmVStack,  -- Current V-stack
                GmHeap,    -- Heap of nodes
                GmGlobals, -- Global addresses in heap
                GmStats)   -- Statistics
```

すでに述べたように、この新しいコンポーネントは数のスタックとして動作します。

```
type GmVStack = [Int]
```

このコンポーネントのアクセス関数を追加します。

```
getVStack :: GmState -> GmVStack
getVStack (o, i, stack, dump, vstack, heap, globals, stats) = vstack

putVStack :: GmVStack -> GmState -> GmState
putVStack vstack' (o, i, stack, dump, vstack, heap, globals, stats)
  = (o, i, stack, dump, vstack', heap, globals, stats)
```

演習 3.39 その他のアクセス関数を変更します。

状態を表示する

関数 showState は、V-stack コンポーネントを表示するように変更されています。

```
showState :: GmState -> Iseq
showState s
  = iConcat [showOutput s, iNewline,
             showStack s, iNewline,
             showDump s, iNewline,
             showVStack s, iNewline,
             showInstructions (getCode s), iNewline]
```

そのために、関数 showVStack を使用します。

```
showVStack :: GmState -> Iseq
showVStack s
  = iConcat [iStr "Vstack:[",
             iInterleave (iStr ", ") (map iNum (getVStack s)),
             iStr "]"]
```

3.9.3　命令セット

最初の要件は、各算術演算の遷移が、通常のスタックの代わりに、V スタックから値を取得し、その結果を返すように修正されなければならないことであることは明らかです。まず、ダイアディックプリミティブの場合を考えてみましょう。この操作 $\odot$ の一般的な遷移は以下の通りです。これは V-stack から 2 つの引数を取り、演算 $\odot$ の結果を V-stack に戻します。

(3.39)

$$\begin{array}{lllllllll} & o & \odot : i & s & d & n_0 : n_1 : v & h & m \\ \Longrightarrow & o & i & s & d & n_0 \odot n_1 : v & h & m \end{array}$$

Neg 命令は、V スタックの先頭の数をその-1 倍に置き換えるだけで、次のように遷移します。

(3.40)

$$\begin{array}{lllllllll} & o & \texttt{Neg} : i & s & d & n : v & h & m \\ \Longrightarrow & o & i & s & d & (-n) : v & h & m \end{array}$$

また、ヒープと V スタック間で値を移動させる命令も必要です。まず、整数 n を V スタックにプッシュする Pushbasic から始めます。

(3.41)

$$\begin{array}{lllllllll} & o & \texttt{Pushbasic}\ n : i & s & d & v & h & m \\ \Longrightarrow & o & i & s & d & n : v & h & m \end{array}$$

V スタックからヒープに値を移動するには、2 つの命令を使用します。Mkbool と Mkint です。これらは、V スタックの上にある整数をそれぞれブール値と整数値として扱います。まず、Mkbool から。

(3.42)

$$\begin{array}{lllllllll} & o & \texttt{Mkbool} : i & s & d & t : v & h & & m \\ \Longrightarrow & o & i & a : s & d & v & h[a : \texttt{NConstr}\ t\ []] & & m \end{array}$$

Mkint の遷移規則は、ヒープに新しい整数ノードを作成することを除けば、Mkbool に似ています。

$$
\begin{array}{|l|llllllll|l|}
\hline
(3.43) & & o & \texttt{Mkint} : i & & s & d & n : v & h & & m \\
& \Longrightarrow & o & & i & a : s & d & & v & h[a : \texttt{NNum}\ n] & m \\
\hline
\end{array}
$$

逆演算を行うには、Get を使用します。これは 2 つの遷移規則で定義されています。最初の遷移規則では、Get がスタックの一番上にあるブール値をどのように扱うかを見ています。

$$
\begin{array}{|l|llllllll|l|}
\hline
(3.44) & & o & \texttt{Get} : i & a : s & d & & v & h[a : \texttt{NConstr}\ t\ []] & m \\
& \Longrightarrow & o & & i & s & d & t : v & h & & m \\
\hline
\end{array}
$$

2 つ目は、Get が数をどのように扱うかを見ることです。

$$
\begin{array}{|l|llllllll|l|}
\hline
(3.45) & & o & \texttt{Get} : i & a : s & d & & v & h[a : \texttt{NNum}\ n] & m \\
& \Longrightarrow & o & & i & s & d & n : v & h & & m \\
\hline
\end{array}
$$

最後に、V スタック上のブーリアン演算を利用するために、V スタックを検査してどの命令ストリームを使用するかを決定する単純化された Casejump 命令を使用します。この新しい命令は Cond と呼ばれ、次の 2 つの遷移規則で定義されます。最初の遷移規則では、V-stack の一番上の値が true のとき、最初のコードシーケンス t を選択します。

$$
\begin{array}{|l|lllllllll|}
\hline
(3.46) & & o & \texttt{Cond}\ t\ f : i & s & d & 2 : v & h & m \\
& \Longrightarrow & o & t \mathbin{\texttt{++}} i & s & d & v & h & m \\
\hline
\end{array}
$$

2 番目の遷移規則では、V スタックの最上位の値が false であるため、Cond はその 2 番目の符号列 f を選択することがわかります。

$$
\begin{array}{|l|lllllllll|}
\hline
(3.47) & & o & \texttt{Cond}\ t\ f : i & s & d & 1 : v & h & m \\
& \Longrightarrow & o & f \mathbin{\texttt{++}} i & s & d & v & h & m \\
\hline
\end{array}
$$

演習 3.40 命令データ型の拡張、showInstruction 関数の再定義、新しい命令遷移の実装、dispatch 関数の修正。

次に、コンパイラの改造（かなり大掛かりなもの）について考えてみましょう。

3.9.4 コンパイラ

状態コンポーネントが追加されたため、compile 関数は V スタックコンポーネントを空に初期化する必要があります。

```
compile :: CoreProgram -> GmState
compile program
  = ([], initialCode, [], [], [], heap, globals, statInitial)
    where (heap, globals) = buildInitialHeap program
```

厳密に言えば、これだけでマシンは動くのですが、Vスタックを導入したのは、算術関数を「インライン」でコンパイルできるようにするためで、私たちのコードはこれを意図しているのです。

```
buildInitialHeap :: CoreProgram -> (GmHeap, GmGlobals)
buildInitialHeap program
  = mapAccuml allocateSc hInitial compiled
    where compiled = map compileSc (preludeDefs ++ program ++ primitives)
```

プリミティブ命令の遷移規則を変更したため、コンパイルしたプリミティブごとにコードを変更する必要があります。Mark 6 マシンで行ったように、このコードを手作業でコンパイルする代わりに、コンパイラにこの仕事を任せることができます。もちろんこれは、コンパイラが生成するコードを最適化できるほど賢ければの話ですが、そうでなければ、Add 命令は生成されません。

```
primitives :: [(Name,[Name],CoreExpr)]
primitives
  = [("+", ["x","y"], (EAp (EAp (EVar "+") (EVar "x")) (EVar "y"))),
     ("-", ["x","y"], (EAp (EAp (EVar "-") (EVar "x")) (EVar "y"))),
     ("*", ["x","y"], (EAp (EAp (EVar "*") (EVar "x")) (EVar "y"))),
     ("/", ["x","y"], (EAp (EAp (EVar "/") (EVar "x")) (EVar "y"))),
```

また、ネゲーション関数を追加する必要があります。

```
     ("negate", ["x"], (EAp (EVar "negate") (EVar "x"))),
```

比較関数は、二項演算関数とほぼ同じです。

```
     ("==", ["x","y"], (EAp (EAp (EVar "==") (EVar "x")) (EVar "y"))),
     ("~=", ["x","y"], (EAp (EAp (EVar "~=") (EVar "x")) (EVar "y"))),
     (">=", ["x","y"], (EAp (EAp (EVar ">=") (EVar "x")) (EVar "y"))),
     (">",  ["x","y"], (EAp (EAp (EVar ">")  (EVar "x")) (EVar "y"))),
     ("<=", ["x","y"], (EAp (EAp (EVar "<=") (EVar "x")) (EVar "y"))),
     ("<",  ["x","y"], (EAp (EAp (EVar "<")  (EVar "x")) (EVar "y"))),
```

最後に、条件付き関数と、ブール値を表現するスーパーコンビネータを入れましょう。

```
("if", ["c","t","f"],
       (EAp (EAp (EAp (EVar "if") (EVar "c")) (EVar "t")) (EVar "f"))),
("True",  [], (EConstr 2 0)),
("False", [], (EConstr 1 0))]
```

コンパイルスキーム $\mathcal{B}$

$\mathcal{B}[\![e]\!]\ \rho$ は，環境 ρ において式 e を WHNF に評価するコードをコンパイルし、結果を V スタックに残します。

$$
\begin{aligned}
\mathcal{B}[\![i]\!]\ \rho &= [\texttt{Pushbasic}\ i] \\
\mathcal{B}[\![\texttt{let}\ x_1\ \texttt{=}\ e_1\texttt{;}\ \ldots\texttt{;}\ x_n\ \texttt{=}\ e_n\ \texttt{in}\ e]\!]\ \rho & \\
&= \mathcal{C}[\![e_1]\!]\ \rho^{+0}\ \texttt{++} \\
&\quad \ldots \\
&\quad \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)}\ \texttt{++} \\
&\quad \mathcal{B}[\![e]\!]\ \rho'\ \texttt{++}\ [\texttt{Pop}\ n] \\
&\quad \text{ここで } \rho' = \rho^{+n}\,[x_1\ \mapsto\ n-1,\ \ldots,\ x_n\ \mapsto\ 0] \text{ です。} \\
\mathcal{B}[\![\texttt{letrec}\ x_1\ \texttt{=}\ e_1\texttt{;}\ \ldots\texttt{;}\ x_n\ \texttt{=}\ e_n\ \texttt{in}\ e]\!]\ \rho & \\
&= [\texttt{Alloc}\ n]\ \texttt{++} \\
&\quad \mathcal{C}[\![e_1]\!]\ \rho'\ \texttt{++}\ [\texttt{Update}\ n-1]\ \texttt{++} \\
&\quad \ldots \\
&\quad \mathcal{C}[\![e_n]\!]\ \rho'\ \texttt{++}\ [\texttt{Update}\ 0]\ \texttt{++} \\
&\quad \mathcal{B}[\![e]\!]\ \rho'\ \texttt{++}\ [\texttt{Pop}\ n] \\
&\quad \text{ここで } \rho' = \rho^{+n}\,[x_1\ \mapsto\ n-1,\ \ldots,\ x_n\ \mapsto\ 0] \text{ です。} \\
\mathcal{B}[\![e_0\ \texttt{+}\ e_1]\!]\ \rho &= \mathcal{B}[\![e_1]\!]\ \rho\ \texttt{++}\ \mathcal{B}[\![e_0]\!]\ \rho^{+1}\ \texttt{++}\ [\texttt{Add}] \\
&\quad \text{他の算術式についても同様。} \\
\mathcal{B}[\![e_0\ \texttt{==}\ e_1]\!]\ \rho &= \mathcal{B}[\![e_1]\!]\ \rho\ \texttt{++}\ \mathcal{B}[\![e_0]\!]\ \rho^{+1}\ \texttt{++}\ [\texttt{Eq}] \\
&\quad \text{他の比較式についても同様。} \\
\mathcal{B}[\![\texttt{negate}\ e]\!]\ \rho &= \mathcal{B}[\![e]\!]\ \rho\ \texttt{++}\ [\texttt{Neg}] \\
\mathcal{B}[\![\texttt{if}\ e_0\ e_1\ e_2]\!]\ \rho &= \mathcal{B}[\![e_0]\!]\ \rho\ \texttt{++}\ [\texttt{Cond}\ (\mathcal{B}[\![e_1]\!]\ \rho)\ (\mathcal{B}[\![e_2]\!]\ \rho)] \\
\mathcal{B}[\![e]\!]\ \rho &= \mathcal{E}[\![e]\!]\ \rho\ \texttt{++}\ [\texttt{Get}] \\
&\quad \text{デフォルトケース}
\end{aligned}
$$

図 3.18 に示す $\mathcal{B}$ スキームは、別のタイプのコンテキストを構成しています。$\mathcal{B}$ スキームによってコンパイルされるには、式は WHNF への評価が必要であることが分かっているだけでなく、整数型かブール型の式でなければなりません。以下の式は $\mathcal{B}$ スキームを介して伝搬されます。

- `let(rec)` Δ `in` e が $\mathcal{B}$-strict コンテキストで現れた場合、式 e も $\mathcal{B}$-strict コンテキストになります。
- `if` $e_0\ e_1\ e_2$ という式が $\mathcal{B}$-strict コンテキストで現れたら、e_0, e_1, e_2 という式も $\mathcal{B}$-strict コンテキストで現れることになります。

- $e_0 \odot e_1$ が $\mathcal{B}$-strict のコンテキストで出現し、 $\odot$ が比較または算術演算子である場合、式 e_0 と e_1 も $\mathcal{B}$-strict のコンテキストで出現します。
- もし `negate` e が $\mathcal{B}$-strict のコンテキストで発生した場合、式 e も同様となる。

式の特殊なケースを認識できない場合、コンパイルされたコードは $\mathcal{E}$ スキームを使用して式を評価し、その後 `Get` 命令を実行します。`Get` 命令は、$\mathcal{E}$ スキームによってスタックの一番上に残された値をアンボックスし、V スタックに移動します。

このため、ある式が最初は $\mathcal{B}$-strict コンテキストにあることをどうやって知るかは未解決でした。通常の状況では、通常の $\mathcal{E}$-strict なコンテキストから、値が整数型またはブーリアン型であるという追加の知識とともに、$\mathcal{B}$-strict なコンテキストを生成します。

演習 3.41 $\mathcal{B}$ コンパイラスキームを実装する。

コンパイルスキーム $\mathcal{E}$

$\mathcal{E}[\![e]\!]\ \rho$ は，式 e を環境 ρ の WHNF に評価するコードをコンパイルし、
式へのポインタをスタックの一番上に残します。

$$
\begin{array}{rcl}
\mathcal{E}[\![i]\!]\ \rho & = & [\texttt{Pushint}\ i] \\
\mathcal{E}[\![\texttt{let}\ x_1 = e_1;\ \ldots;\ x_n = e_n\ \texttt{in}\ e]\!]\ \rho & & \\
 & = & \mathcal{C}[\![e_1]\!]\ \rho^{+0}\ \texttt{++} \\
 & & \ldots \\
 & & \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)}\ \texttt{++} \\
 & & \mathcal{E}[\![e]\!]\ \rho'\ \texttt{++}\ [\texttt{Slide}\ n] \\
 & & \text{ここで } \rho' = \rho^{+n}\,[x_1\ \mapsto\ n-1,\ \ldots,\ x_n\ \mapsto\ 0] \text{ です。} \\
\mathcal{E}[\![\texttt{letrec}\ x_1 = e_1;\ \ldots;\ x_n = e_n\ \texttt{in}\ e]\!]\ \rho & & \\
 & = & [\texttt{Alloc}\ n]\ \texttt{++} \\
 & & \mathcal{C}[\![e_1]\!]\ \rho'\ \texttt{++}\ [\texttt{Update}\ n-1]\ \texttt{++} \\
 & & \ldots \\
 & & \mathcal{C}[\![e_n]\!]\ \rho'\ \texttt{++}\ [\texttt{Update}\ 0]\ \texttt{++} \\
 & & \mathcal{E}[\![e]\!]\ \rho'\ \texttt{++}\ [\texttt{Slide}\ n] \\
 & & \text{ここで } \rho' = \rho^{+n}\,[x_1\ \mapsto\ n-1,\ \ldots,\ x_n\ \mapsto\ 0] \text{ です。} \\
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ alts]\!]\ \rho & = & \mathcal{E}[\![e]\!]\ \rho\ \texttt{++}\ [\texttt{Casejump}\ \mathcal{D}_{\mathcal{E}}[\![alts]\!]\ \rho] \\
\mathcal{E}[\![\texttt{Pack\{}t\texttt{,}a\texttt{\}}\ e_1\ \ldots\ e_a]\!]\ \rho & = & \mathcal{C}[\![e_a]\!]\ \rho\ \texttt{++} \\
 & & \ldots \\
 & & \mathcal{C}[\![e_1]\!]\ \rho\ \texttt{++} \\
 & & [\texttt{Pack}\ t\ a] \\
\mathcal{E}[\![e_0\ \texttt{+}\ e_1]\!]\ \rho & = & \mathcal{B}[\![e_0\ \texttt{+}\ e_1]\!]\ \rho\ \texttt{++}\ [\texttt{Mkint}] \\
 & & \text{他の算術式についても同様。} \\
\mathcal{E}[\![e_0\ \texttt{==}\ e_1]\!]\ \rho & = & \mathcal{B}[\![e_0\ \texttt{==}\ e_1]\!]\ \rho\ \texttt{++}\ [\texttt{Mkbool}] \\
 & & \text{他の比較式についても同様。} \\
\mathcal{E}[\![\texttt{negate}\ e]\!]\ \rho & = & \mathcal{B}[\![\texttt{negate}\ e]\!]\ \rho\ \texttt{++}\ [\texttt{Mkint}] \\
\mathcal{E}[\![\texttt{if}\ e_0\ e_1\ e_2]\!]\ \rho & = & \mathcal{B}[\![e_0]\!]\ \rho\ \texttt{++}\ [\texttt{Cond}\ (\mathcal{E}[\![e_1]\!]\ \rho)\ (\mathcal{E}[\![e_2]\!]\ \rho)] \\
\mathcal{E}[\![e]\!]\ \rho & = & \mathcal{C}[\![e]\!]\ \rho\ \texttt{++}\ [\texttt{Eval}] \\
 & & \text{デフォルトケース}
\end{array}
$$

$\mathcal{D}_{\mathcal{E}}[\![alts]\!]\ \rho\ d$ は、case 式の選択肢のリストをコンパイルします。

$$\mathcal{D}_{\mathcal{E}}[\![alt_1\ \ldots\ alt_n]\!]\ \rho \quad = \quad [\mathcal{A}_{\mathcal{E}}[\![alt_1]\!]\ \rho\ d\ ,\ldots,\mathcal{A}_{\mathcal{E}}[\![alt_n]\!]\ \rho\ d]$$

$\mathcal{A}_{\mathcal{E}}[\![alt]\!]\ \rho\ d$ は、case 式の各選択肢のコードをコンパイルします。

$$
\begin{array}{rcl}
\mathcal{A}_{\mathcal{E}}[\![\texttt{<}t\texttt{>}\ x_1\ \ldots\ x_n\ \texttt{->}body]\!]\ \rho\ d & = & t\ \texttt{->}\ [\texttt{Split}\ n]\ \texttt{++}\ \mathcal{E}[\![body]\!]\ \rho'\ \texttt{++}\ [\texttt{Slide}\ n] \\
 & & \text{ここで } \rho' = \rho^{+n}\,[x_1\ \mapsto\ 0,\ \ldots,\ x_n\ \mapsto\ n-1] \text{ です。}
\end{array}
$$

$\mathcal{E}$ スキーム（図 3.19）は、算術関数と比較関数の呼び出しを特別扱いする
ことを定めたものです。Mark6 マシンで使用したバージョンと異なるのは、

算術演算と比較演算に $\mathcal{B}$ スキームを使用するためです。また、特殊なケースがない場合、式をコンパイルするためにデフォルトのメソッドを使用しなければなりません。これは、$\mathcal{C}$ スキームを用いてグラフを構築し、コードストリームに `Eval` 命令を配置するだけです。これにより、グラフが WHNF に評価されることが保証されます。

演習 3.42 新しい $\mathcal{E}$ コンパイラスキームを実装する。

コンパイルスキーム $\mathcal{R}$

また、この機会に $\mathcal{R}$ スキームを改善します。まず、$\mathcal{B}$ スキームが使われる機会を作りたいと考えています。また，関数のコード列の末尾で実行される命令の数を減らすことも試みています．$\mathcal{R}$ スキームの新しいコンパイルスキームを図 3.20 に示します。

$\mathcal{R}[\![e]\!]\ \rho\ d$ は，アリティ d のスーパーコンビネータのために、
環境 ρ で式 e をインスタンス化するコードを生成し、
その後、結果のスタックのアンワインドに進みます。

$$
\begin{aligned}
&\mathcal{R}[\![\texttt{let } x_1 \texttt{ = } e_1\texttt{; } \ldots\texttt{; } x_n \texttt{ = } e_n \texttt{ in } e]\!]\ \rho\ d \\
&\qquad = \quad \mathcal{C}[\![e_1]\!]\ \rho^{+0}\ \texttt{++} \\
&\qquad\qquad \ldots \\
&\qquad\qquad \mathcal{C}[\![e_n]\!]\ \rho^{+(n-1)}\ \texttt{++} \\
&\qquad\qquad \mathcal{R}[\![e]\!]\ \rho'\ (n+d) \\
&\qquad\qquad \text{ここで } \rho' = \rho^{+n}\,[x_1 \mapsto n-1,\ \ldots,\ x_n \mapsto 0] \text{ です。} \\
&\mathcal{R}[\![\texttt{letrec } x_1 \texttt{ = } e_1\texttt{; } \ldots\texttt{; } x_n \texttt{ = } e_n \texttt{ in } e]\!]\ \rho\ d \\
&\qquad = \quad [\texttt{Alloc } n]\ \texttt{++} \\
&\qquad\qquad \mathcal{C}[\![e_1]\!]\ \rho'\ \texttt{++}\ [\texttt{Update } n-1]\ \texttt{++} \\
&\qquad\qquad \ldots \\
&\qquad\qquad \mathcal{C}[\![e_n]\!]\ \rho'\ \texttt{++}\ [\texttt{Update } 0]\ \texttt{++} \\
&\qquad\qquad \mathcal{R}[\![e]\!]\ \rho'\ (n+d) \\
&\qquad\qquad \text{ここで } \rho' = \rho^{+n}\,[x_1 \mapsto n-1,\ \ldots,\ x_n \mapsto 0] \text{ です。} \\
&\mathcal{R}[\![\texttt{if } e_0\ e_1\ e_2]\!]\ \rho\ d \quad = \quad \mathcal{B}[\![e_0]\!]\ \rho\ \texttt{++}\ [\texttt{Cond } [\mathcal{R}[\![e_1]\!]\ \rho\ d,\ \mathcal{R}[\![e_2]\!]\ \rho\ d]] \\
&\mathcal{R}[\![\texttt{case } e \texttt{ of } alts]\!]\ \rho\ d \quad = \quad \mathcal{E}[\![e]\!]\ \rho\ \texttt{++}\ [\texttt{Casejump } \mathcal{D}_{\mathcal{R}}[\![alts]\!]\ \rho\ d] \\
&\qquad\mathcal{R}[\![e]\!]\ \rho\ d \quad = \quad \mathcal{E}[\![e]\!]\ \rho\ \texttt{++}\ [\texttt{Update } d,\ \texttt{Pop } d,\ \texttt{Unwind}] \\
&\qquad\qquad\qquad\quad \text{デフォルトケース}
\end{aligned}
$$

$\mathcal{D}_{\mathcal{R}}[\![alts]\!]\ \rho\ d$ は、case 式の選択肢のリストをコンパイルします。

$$\mathcal{D}_{\mathcal{R}}[\![alt_1\ \ldots\ alt_n]\!]\ \rho \quad = \quad [\mathcal{A}_{\mathcal{R}}[\![alt_1]\!]\ \rho\ d\ , \ldots, \mathcal{A}_{\mathcal{R}}[\![alt_n]\!]\ \rho\ d]$$

$\mathcal{A}_{\mathcal{R}}[\![alt]\!]\ \rho\ d$ は、case 式の各選択肢のコードをコンパイルします。

$$
\begin{aligned}
&\mathcal{A}_{\mathcal{R}}[\![\texttt{<}t\texttt{>}\ x_1\ \ldots\ x_n\ \texttt{->}\ body]\!]\ \rho\ d \quad = \quad t\ \texttt{->}\ [\texttt{Split } n]\ \texttt{++}\ \mathcal{R}[\![body]\!]\ \rho'\ (n+d) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{ここで } \rho' = \rho^{+n}\,[x_1 \mapsto 0,\ \ldots,\ x_n \mapsto n-1] \text{ です。}
\end{aligned}
$$

Mark6 マシンで使用したバージョンから拡張され、コンテキストのように動作するようになりました。このコンテキストを $\mathcal{R}$-strict と呼ぶことにします。$\mathcal{R}$-strict コンテキストでコンパイルされる式は WHNF に評価され、現在の redex を上書きするために使われます。これは以下の伝搬のルールに従います。

- スーパーコンビネータ定義の本体である式は、$\mathcal{R}$-strict のコンテキストにあります。
- もし let(rec) Δ in e が $\mathcal{R}$-strict コンテキストで出現したら、式 e も $\mathcal{R}$-strict コンテキストになります。
- もし式 if e_0 e_1 e_2 が $\mathcal{R}$-strict のコンテキストで現れたら、式 e_1 と e_2

も $\mathcal{R}$-strict のコンテキストにある。(式 e_0 は今度は $\mathcal{B}$-strict のコンテキストに現れる)。

- もし `case` e `of` $alts$ が $\mathcal{R}$-strict コンテキストで出現するならば、e という式は strict なコンテキストにあることになる。さらに、各選択肢の式部分は $\mathcal{R}$-strict コンテキストで出現する。

演習 3.43 図 3.20 の $\mathcal{R}$ スキームを実装してください。`compileAlts` 関数の一般性を利用して、$\mathcal{A}_{\mathcal{R}}$ スキームと $\mathcal{A}_{\mathcal{E}}$ スキームの両方を実装できることに注意してください。

Mark7 マシンについて特筆すべき点は、現在の V スタックをダンプに保存する `Eval` 命令を定義していないことです。これは、[Peyton Jones 1987] で説明された G マシンとは対照的です。これを行うかどうかは、本書で作成した抽象的なマシンでは、本当に好みの問題です。実際のマシン用にコードをコンパイルする場合、我々はスタックの数を最小にしようとすることが多いでしょう。このような場合、`Eval` の代替遷移規則として次のようなものを使いたいです。

(3.48)									
		o	`Eval` : i	$a : s$		d	v	h	m
	$\Longrightarrow$	o	[`Unwind`]	$[a]$	$\langle i, s, v\rangle : d$		[]	h	m

演習 3.44 `Eval` のためにこの代替遷移規則を実装します。この新しい遷移規則を可能にするために変更する必要がある他の命令は何ですか？

演習 3.45 ブール演算子`&`, `|`, `not` の最適化されたコードを生成するために、どのようなコンパイルルールを変更しますか？

演習 3.46 以下の遷移規則による `Return` 命令の追加。

(3.49)									
		o	[`Return`]	$[a_0, \ldots, a_k]$	$\langle i, s\rangle : d$	v	h	m
	$\Longrightarrow$	o	i	$a_k : s$		d	v	h	m

※※※

上記の遷移規則では、WHNF にあることが分かっているスタックの一番上のアイテムを捨ててしまうことになる。
以下の様な遷移規則になるのでは？

(3.49)									
		o	[`Return`]	$[a_0, \ldots, a_k]$	$\langle i, s\rangle : d$	v	h	m
	$\Longrightarrow$	o	i	$a_0 : s$		d	v	h	m

※※※

これは、スタックの一番上のアイテムが WHNF にあることが分かっているときに、$\mathcal{R}$ スキームによって `Unwind` の代わりに使用されます。この新しい命令を生成するために `compileR` を修正します。

演習 3.47 `UpdateInt n` の遷移規則を書いてください。この命令はシーケンス `[Mkint, Update n]` と同じアクションを実行します。この遷移規則と `UpdateBool n` のための同様の遷移規則を実装してください。`compileR` を修正して、元のシーケンスの代わりにこれらの命令を生成するようにしてください。なぜ新しい命令が元のシーケンスより好ましいのでしょうか？(ヒント：いくつかのサンプルプログラムからの統計値を利用すること)

3.10　結論

この章で採用したアプローチは、大きなソフトウェアを設計する際に非常に役立ちます。最初は非常に単純なものから始め、徐々に変更を加えることで、非常に大規模で複雑なソフトウェアを作成します。これは、小さな漸進的な変更のプロセスによって常に可能であると主張するのは誤解を招く可能性があります。実際、Mark1 マシンの一部として提示された材料は、Mark7 マシンを念頭に置いて特別に設計されました。

しかし、現時点では、かなり効率的なマシンを製造しています。次の章では、TIM について説明します。

```
module Tim where
import Utils
import Language
```

第4章　TIM: 3命令マシン

3つの命令マシンである TIM は、最初は、これまでに見たものとは非常に異なる形のリダクションマシンのように見えます。それでも、一連の比較的単純な手順で G マシンを TIM に変換できることがわかりました。この章では、これらの手順について説明し、それによって TIM がどのように機能するかを示し、完全な最小 TIM コンパイラと評価器を定義します。そして、それに対する一連の改善と最適化を開発します。

TIM は Fairbairn と Wray によって発明されたもので、彼らの元の論文 [Fairbairn and Wray 1987] は読む価値があります。この章で採用したアプローチとはまったく異なる方法で TIM を説明します。ただし、この章で作成された資料は、Fairbairn と Wray の研究をはるかに超えているため、あまり知られていないアイデアが議論され、実装されている後のセクションで、詳細レベルが高められています。提示された新しいアイデアの多くは Guy Argo によるものであり、彼の FPCA 論文 [Argo 1989] と博士論文 [Argo 1991] に提示されています。

4.1　背景: TIM の仕組み

次の関数定義を検討してください。

```
f x y = g E1 E2
```

ここで、E1 と E2 は任意の (そしておそらく複雑な) 式であり、g はその他の関数です。テンプレートインスタンス化マシン (第 2 章) と G マシン (第 3 章) の両方が、次の簡約を実行します。

```
    @    reduces to    @
   / \                / \
  @   y              @   E2
 / \                / \
f   x              g   E1
```

G マシンはこれを行うためにかなりの数の (単純な) 命令を必要としますが、テンプレートマシンは 1 つの (複雑な) ステップでそれを行います。ですが、最終的な結果は同じです。

この図で、E1 と E2 は式 E1 と E2 のグラフです。たとえば、E1 が (x+y)*(x-y) の場合、g の最初の引数は (x+y)*(x-y) のグラフになります。このグラフは、(*C* コンパイル スキームによって生成されたコードによって) ヒープに苦労して構築する必要があります。残念ながら、g は最初の引数を使用せずに破棄する可能性があるため、これは無駄な作業になる可能性があります。関数への引数に対して行われるグラフ作成の量を制限する何らかの方法を見つけたいと考えています。

4.1.1　平坦化

変換のステップ 1 はまさにこれを行います。f の定義を次の新しい定義に置き換えるとします。

```
f x y = g (c1 x y) (c2 x y)
c1 x y = E1
c2 x y = E2
```

私たちは、c1 と c2 という 2 つの補助関数を発明しました。この定義は明らかに古いものと同等ですが、E1 **がどれほど大きく複雑であっても、**f **の簡約中に行われる唯一の作業は、**(c1 x y) **のグラフを作成することです。**

さらに良いことに、G マシンの実装には、自動的に得られるさらなる利点があります。最初の定義では、E1 は *C* スキームによってコンパイルされます。算術式をコンパイルするときに、*E* スキームに存在する最適化を利用することはできません。しかし、2 番目の定義では、式 E1 はスーパーコンビネーターの右辺になり、これらすべての最適化が適用されます。この方法で (x+y)*(x-y) をより効率的に評価できます。

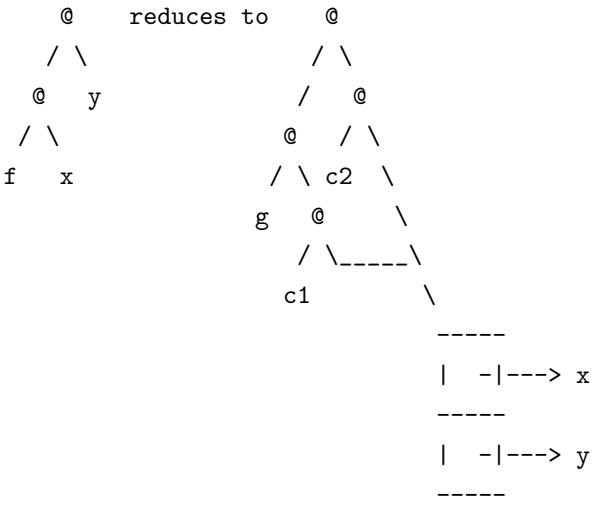
もちろん、E1 と E2 自体に *C* スキームでコンパイルされる大きな式が含まれている可能性があるため (たとえば、E2 が (h E3 E4) であるとします)、c1 と c2 の右辺に変換を再度適用する必要があります。 . その結果、ネストされた構造を持つ式がないため、**平坦化された**プログラムと呼ばれます。

4.1.2　タプル化

次の注目点は、c1 と c2 の両方が x と y の両方に適用されることです。そのため、g を呼び出す前に (c1 x y) と (c2 x y) のグラフを作成する必要があります。c1 と c2 に引数が 2 つだけではなく多数ある場合、グラフはかなり大きくなる可能性があります。2 つのグラフは互いに非常に似ているため、これらの引数グラフがいくつかの共通部分を共有できるかどうかを尋ねるのは自然なことです。重複を避け、それによってヒープ割り当てを減らします。このアイデアは、2 番目の変換で表現できます。

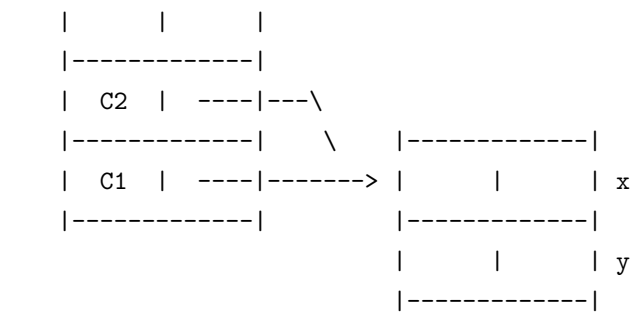
```
f x y = let tup = (x,y)
        in g (c1 tup) (c2 tup)
c1 (x,y) = E1
c2 (x,y) = E2
```

最初にその引数をタプルにパッケージ化し、次にこの単一のタプルを c1 と c2 に渡すという考え方です。f をこのように定義すると、f の簡約は次のようになります。



4.1.3 スパインレスであること

前の図を見ると、スーパーコンビネータとタプル tup の場合、スパインが指す引数は常に (c tup) の形式であることがわかります。簡約中に、これらの引数へのポインタのスタックを構築します。しかし、それらはすべて同じ形式になっているので、代わりに引数 (のルート) 自体を積み重ねることができます! したがって、f 簡約後、スタックは次のようになります。



スパインスタックの各アイテムは、コードポインタとタプルへのポインタのペアになりました。このペアは、タプルに適用される関数を定義するコー

ドである関数適用ノードと考えることができます。f へのエントリでは、引数 x と y(のルート) がスタック上にあったため、x と y のタプルは実際にはコードポインタ/タプルポインタのペアのタプルです。

コードポインタ/タプルポインタのペアは**クロージャ**と呼ばれ、そのようなクロージャのタプルは**フレーム**と呼ばれます。フレームへのポインタを**フレームポインタ**と呼びます。ヒープにスパインがなくなっていることに注意してください。スタックは、評価される式のスパインです。TIM はスパインレスマシンです。

4.1.4　例

TIM プログラムがどのように機能するかの例を示します。次のように定義された関数 compose2 を考えてみましょう。

```
compose2 f g x = f (g x x)
```

compose2 の「平坦化された」形式は次のようになります。

```
compose2 f g x = f (c1 g x)
c1 g x = g x x
```

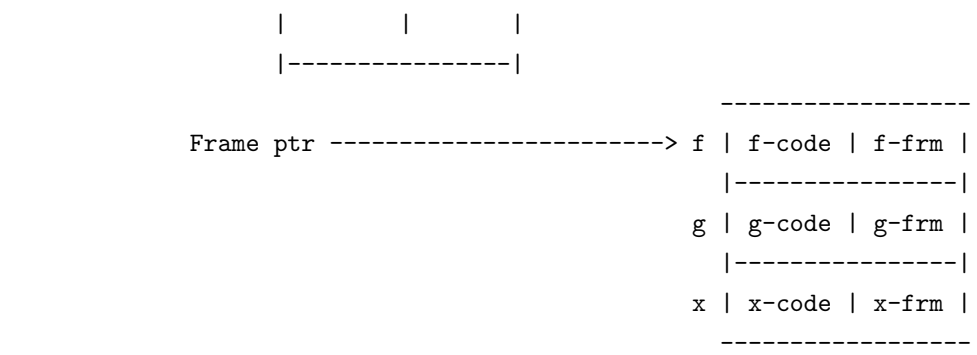
compose2 に入ると、次のように 3 つの引数がスタックの一番上に置かれます。

```
   |        |       |
   |----------------|
 x | x-code | x-frm |
   |----------------|
 g | g-code | g-frm |
   |----------------|
 f | f-code | f-frm |
   |----------------|
```

最初に行うことは、これら 3 つの引数のタプル (フレーム) をヒープに形成することです。その後、それらをスタックから削除できます。**フレームポインタ**と呼ばれる特別なレジスタに新しいフレームへのポインタを保持します。これは、命令によって行われます。

```
Take 3
```

マシンの状態は次のようになります。



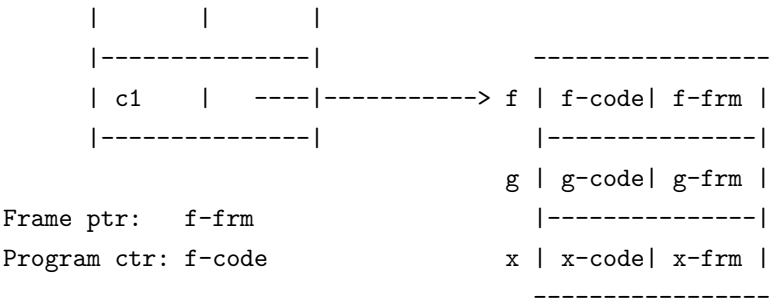
次に、f の引数を準備する必要があります。(g x x) という 1 つのみがあり、そのクロージャをスタックにプッシュします。クロージャのフレームポインタは現在のフレームポインタレジスタにすぎないため、命令はコードラベルを提供するだけで済みます。

```
Push (Label "c1")
```

最後に、f にジャンプします。f はグローバルなスーパーコンビネータではなく、compose の引数であるため、f は、現在のフレームのクロージャによって表されます。私たちがしなければならないことは、クロージャをフェッチし、そのフレームポインタをフレームポインタレジスタにロードし、そのコードポインタをプログラムカウンタにロードすることです。これは次の命令によって行われます。

```
Enter (Arg 1) -- f is argument 1
```

この命令の後、マシンの状態は次のようになります。



それだ！compose2 の本体は、次の 3 つの命令だけで構成されています。

```
compose2:   Take 3                -- 3 arguments
            Push (Label "c1")     -- closure for (g x x)
            Enter (Arg 1)         -- f is argument 1
```

ただし、ラベル c1 を処理する必要があります。(g x x) のクロージャが必要な場合は、Enter 命令で入力されるため、プログラムカウンタは c1 を指し、f、g、および x を含む元のフレームへのフレームポインタ。この時点で必要なことは、g の引数、つまり x を準備し、g を入力することだけです。

```
c1:         Push (Arg 3)          -- x is argument 3
            Push (Arg 3)          -- x again
            Enter (Arg 2)         -- g is argument 2
```

Push (Arg 3) 命令は、現在のフレームから x のクロージャのコピーをフェッチし、それをスタックにプッシュします。次に、Enter (Arg 2) 命令が、現在スタックにある引数[1]に g を適用します。

4.1.5　状態遷移規則によるマシンの定義

3 つの命令マシンと呼ばれる理由がわかります。Take、Push、Enter の 3 つの主要な命令があります。いくつかの点で、3 つの命令しかないと主張するのはかなり楽観的です。なぜなら、Push と Enter には両方ともいくつかの「アドレス指定モード」があり、さらに、やがてかなりの数の新しい命令を発明する必要があるからです。それにしても素敵な名前ですね。

いつものように、状態遷移規則を使用して、各命令の正確な作用を表現します。まず、マシンの状態を定義する必要があります。これは、

$$(instructions,\ framepointer,\ stack,\ heap,\ codestore)$$

または略して $(i,\ f,\ s,\ h,\ c)$ の五つ組です。コード ストアは、まだ説明されていない唯一の項目です。コードのコレクションが含まれており、それぞれにラベルが付いています。実際には、コードストアにはコンパイル済みのスーパーコンビネータの定義が含まれており、それぞれにスーパーコンビネータの名前が付けられていますが、原則として、有用であることが判明した場合は、他のラベル付けされたコードフラグメントも含めることができます。

次に、各命令の遷移規則を作成します。Take n は、スタックの上位 n 個の要素を新しいフレームに形成し、現在のフレームポインタがそれを指すようにします。

(4.1)

$$\begin{array}{lllllll} & \texttt{Take}\ n : i & f & c_1 : \ldots : c_n : s & h & & c \\ \Longrightarrow & i & f' & s & h[f' : \langle c_1, \ldots, c_n \rangle] & & c \end{array}$$

[1] (g x x) クロージャに入ったときにスタックが空でなかった場合、2 つ以上になる可能性があります。

次に、Push と Enter のルールについて説明します。これらの 2 つの命令はまったく同じアドレッシング モード (Arg、Label など) を持ち、それらの間には非常に明確な関係があります。

> Push/Enter の関係。Push arg 命令がクロージャ (i, f) をスタックにプッシュする場合、Enter arg は i をプログラムカウンタにロードし、f を現在のフレームポインタにロードします。

Push (Arg n) 命令は、現在のフレームから n 番目のクロージャをフェッチし、それをスタックにプッシュします。

(4.2)

$$\begin{array}{lllllll} & \texttt{Push}\ (\texttt{Arg}\ k) : i & f & & s & h[f : \langle (i_1, f_1), \ldots, (i_k, f_k), \ldots, (i_n, f_n) \rangle] & c \\ \Longrightarrow & & i & f\ (i_k,\ f_k) : s & & h & c \end{array}$$

Push (Label l) は、コードストアでラベル l を検索し、このコードポインタと現在のフレームポインタで構成されるクロージャをプッシュします。

(4.3)

$$\begin{array}{lllllll} & \texttt{Push}\ (\texttt{Label}\ l) : i & f & & s & h & c[l : i'] \\ \Longrightarrow & & i & f\ (i',\ f) : s & & h & c \end{array}$$

compose の例では、任意のラベル c1 を作成する必要がありました。これらのラベルを考え出さなければならないのは面倒です。代わりに、プッシュ命令の新しい形式、Push (Code i) を追加するだけです。これにより、ターゲットコードシーケンス i が命令自体の一部になります。したがって、

```
Push (Label "c1")
```

の代わりに

```
Push (Code [Push (Arg 3), Push (Arg 3), Enter (Arg 2)])
```

と書くことが出来ます。適切な状態遷移規則は次のとおりです。

(4.4)

$$\begin{array}{lllllll} & \texttt{Push}\ (\texttt{Code}\ i') : i & f & & s & h & c \\ \Longrightarrow & & i & f\ (i',\ f) : s & & h & c \end{array}$$

これまでのところ、Arg、Code、Label の 3 つの「アドレス指定モード」があります。整数定数用に IntConst をもう 1 つ追加する必要があります。たとえば、呼び出し (f 6) は次のコードにコンパイルされます。

```
Push (IntConst 6)
Enter (Label "f")
```

Push 命令は常にクロージャ (つまり、コードポインタ/フレームポインタのペア) をスタックにプッシュしますが、整数定数の場合は、どのクロージャをプッシュするかはまったく明確ではありません。整数自体を格納する場所が必要なので、その目的のためにフレームポインタスロットを「スチール」しましょう。[2] この決定は、次の規則につながります。ここで、intCode は整数クロージャの (まだ未定の) コードシーケンスです。

(4.5)

$$\begin{array}{lllllll} & \texttt{Push}\ (\texttt{IntConst}\ n) : i & f & & s & h & c \\ \Longrightarrow & & i & f & (\texttt{intCode}, n) : s & h & c \end{array}$$

intCode は何をすべきか? 今のところ、私たちのマシンは演算を行わないので、簡単な解決策は intCode を空のコードシーケンスにすることです。

```
intCode = []  -- Mark1 TIM を写経する際は、これを忘れずに！
```

整数クロージャが入力されると、マシンは空のコードシーケンスにジャンプし、実行が停止します。これにより、Mark 1 には十分な整数を返すプログラムを書くことができます。

Push 命令については以上です。Enter 命令のルール (アドレス指定ごとに 1 つ) モードでは、Push/Enter 関係から直接に従います。

(4.6)

$$\begin{array}{lllllll} & [\texttt{Enter}\ (\texttt{Label}\ l)] & f & s & h & c[l:i] \\ \Longrightarrow & & i & f & s & h & c \end{array}$$

(4.7)

$$\begin{array}{lllll} & [\texttt{Enter}\ (\texttt{Arg}\ k)] & f & s & h[f : \langle (i_1, f_1), \ldots, (i_k, f_k), \ldots, (i_n, f_n) \rangle] & c \\ \Longrightarrow & & i_k & f_k & s & h & c \end{array}$$

(4.8)

$$\begin{array}{lllllll} & [\texttt{Enter}\ (\texttt{Code}\ i)] & f & s & h & c \\ \Longrightarrow & & i & f & s & h & c \end{array}$$

(4.9)

$$\begin{array}{lllllll} & [\texttt{Enter}\ (\texttt{IntConst}\ n)] & f & s & h & c \\ \Longrightarrow & & \texttt{intCode} & n & s & h & c \end{array}$$

[2] 整数はフレームポインタよりも大きくないという暗黙の仮定を行っていますが、これは通常実際には当てはまります。

4.1.6 コンパイル

これで、各 TIM 命令が何を行うかについて正確な説明ができました。プログラムを TIM 命令に変換する方法を説明する必要があります。これは、以前と同様に、一連の**コンパイルスキーム**を使用して行います。各スーパーコンビネーターは、図 4.1 に示す $\mathcal{SC}$ スキームでコンパイルされます。$\mathcal{SC}$ に渡される初期環境は、各スーパーコンビネータ名をそのラベルアドレッシングモードにバインドします。$\mathcal{SC}$ スキームは単に `Take` 命令を生成し、$\mathcal{R}$ スキームを呼び出して、各引数に使用するアドレッシングモードを指定するバインドによって強化された環境を渡します。

$\mathcal{SC}[\![e]\!]\ \rho$ は、環境 ρ におけるスーパーコンビネータ定義 def の TIM コードです。 $\mathcal{SC}[\![f\ x_1\ \ldots\ x_n = e]\!]\ \rho \quad = \quad \texttt{Take}\ n\ :\ \mathcal{R}[\![e]\!]\ \rho[x_1 \mapsto \texttt{Arg}\ 1, \ldots, x_n \mapsto \texttt{Arg}\ n]$
$\mathcal{R}[\![e]\!]\ \rho$ は、環境 ρ の式 e の値をスタック上の引数に適用する TIM コードです。 $\mathcal{R}[\![e_1\ e_2]\!]\ \rho \quad = \quad \texttt{Push}\ (\mathcal{A}[\![e_2]\!]\ \rho)\ :\ \mathcal{R}[\![e_1]\!]\ \rho$ $\mathcal{R}[\![a]\!]\ \rho \quad = \quad \texttt{Enter}\ (\mathcal{A}[\![a]\!]\ \rho)$ ここで、a は整数、変数、またはスーパーコンビネータです。
$\mathcal{A}[\![e]\!]\ \rho$ は、環境 ρ における式 e の TIM アドレッシングモードです。 $\mathcal{A}[\![x]\!]\ \rho \quad = \quad \rho\ x$ ここで、x は ρ によって束縛されます。 $\mathcal{A}[\![n]\!]\ \rho \quad = \quad \texttt{IntConst}\ n$ ここで、n は整数です。 $\mathcal{A}[\![e]\!]\ \rho \quad = \quad \texttt{Code}\ (\mathcal{R}[\![e]\!]\ \rho)$ その他

$\mathcal{R}$ スキーム (図 4.1) は、変数またはスーパーコンビネータが見つかるまで、単純に引数をスタックにプッシュします。$\mathcal{A}$ 方式を使用して正しいアドレッシングモードを生成します。セクション 4.1.1 で説明した平坦化プロセスがこれらのルールによって「オンザフライ」で実行されることに注意してください。

ここでは、算術、データ構造、ケース分析、および `let(rec)` 式を省略します。それらはすべて後で追加されます。

4.1.7 更新

これまでのところ、更新についての言及はありません。これは、スパインが消滅したため、更新するスパインノードがないためです。実際、これまで説明してきたマシンはツリー簡約マシンです。共有引数は繰り返し評価される場合があります。更新を適切に行うことは、スパインレス実装のアキレス

腱です。そうしないと無制限の量の作業が複製される可能性があるため、これは絶対に必要ですが、更新しないバージョンの優雅さと速度 (複製はさておき) の一部を失う複雑さを追加します。

後でセクション 4.5 で更新に戻りますが、それまでは非更新バージョンを実装するだけで十分です。

4.2 Mark 1: 最小限の TIM

このセクションでは、演算、データ構造、または更新を使用しない、最小限で完全な TIM 実装を開発します。これらは、後続のセクションで追加されます。

4.2.1 全体構造

構造は、テンプレートインスタンス化インタープリタとほとんど同じです。run 関数は、従来通り、parse、compile、eval、showResults の 4 つの関数で構成されています。解析のタイプは第 1 章で与えられます。 他の 3 つのタイプを以下に示します。

```
runProg     :: [Char] -> [Char]
compile     :: CoreProgram -> TimState
eval        :: TimState -> [TimState]
showResults :: [TimState] -> [Char]

runProg = showResults . eval . compile . parse
```

多くの場合、すべての中間状態を表示すると便利なため、showFullResults を使用して各状態を表示する fullRun も提供します。

```
fullRun :: [Char] -> [Char]
fullRun = showFullResults . eval . compile . parse
```

language モジュールをインポートする必要があります。

4.2.2 データ型の定義

TIM 命令のデータ型は、これまでに紹介した命令に直接対応しています。

```
data Instruction = Take  Int
                 | Enter TimAMode
                 | Push  TimAMode
```

アドレッシングモードのタイプである timAMode は、Push と Enter の関係を強調するために、別個のデータタイプとして分離されています。

```
data TimAMode = Arg      Int
              | Label    [Char]
              | Code     [Instruction]
              | IntConst Int
```

TIM マシンの状態は、次の定義によって与えられます。

```
type TimState = ([Instruction], -- The current instruction stream
                 FramePtr,      -- Address of current frame
                 TimStack,      -- Stack of arguments
                 TimValueStack, -- Value stack (not used yet)
                 TimDump,       -- Dump (not used yet)
                 TimHeap,       -- Heap of frames
                 CodeStore,     -- Labelled blocks of code
                 TimStats)      -- Statistics
```

値スタックとダンプは、この章の後半でのみ必要になりますが、それらのプレースホルダーをすぐに追加する方が便利です。

これらの各コンポーネントの表現を順番に検討します。

- **現在の命令ストリーム**は、命令のリストで表されます。実際のマシンでは、これはプログラムメモリと共にプログラムカウンタになります。
- 通常、**フレームポインタ**はヒープ内のフレームのアドレスですが、他に 2 つの可能性があります。整数値を保持するために使用されるか、初期化されていない可能性があります。マシンは、これら 3 つの可能性のどれを期待するかを常に「認識」していますが、framePtr の代数データ型を使用してそれらを区別する実装が便利です。

  ```
  data FramePtr = FrameAddr Addr -- The address of a frame
                | FrameInt Int   -- An integer value
                | FrameNull      -- Uninitialised
  ```

  これを行わないと、アドレスを数値として使用しようとすると、Miranda は (正当に) 型エラーを報告します。さらに、初期化されていない状態の FrameNull のコンストラクタがあるということは、初期化されていない値を有効なアドレスとして誤って使用しようとした場合に、インタープリタが検出することを意味します。
- **スタック**には**クロージャ**が含まれており、それぞれがコード ポインタとフレーム ポインタを含むペアです。スタックをリストとして表します。

  ```
  type TimStack = [Closure]
  type Closure = ([Instruction], FramePtr)
  ```

- 値スタックとダンプは最初からまったく使用されていないため、nullary コンストラクターが 1 つだけあるダミーの代数データ型でそれぞれを表します。後で、これらの定義をより興味深いものに置き換えます。

```
data TimValueStack = DummyTimValueStack
data TimDump = DummyTimDump
```

- **ヒープ**には**フレーム**が含まれており、それぞれがクロージャのタプルです。フレームのデータ型は、独自の抽象データ型に値するほど重要です。

```
type TimHeap = Heap Frame

fAlloc  :: TimHeap -> [Closure] -> (TimHeap, FramePtr)
fGet    :: TimHeap -> FramePtr -> Int -> Closure
fUpdate :: TimHeap -> FramePtr -> Int -> Closure -> TimHeap
fList   :: Frame -> [Closure] -- Used when printing
```

これらの操作により、フレームを構築し、コンポーネントを抽出して更新できます。`fGet` と `fUpdate` のために、`fAlloc` に与えられるリストの最初の要素には 1 の番号が付けられます。これは、リストに基づく簡単な実装です。

```
type Frame = [Closure]

fAlloc heap xs = (heap', FrameAddr addr)
                 where
                 (heap', addr) = hAlloc heap xs

fGet heap (FrameAddr addr) n = f !! (n-1)
                               where
                                 f = hLookup heap addr

fUpdate heap (FrameAddr addr) n closure
  = hUpdate heap addr new_frame
    where
      frame = hLookup heap addr
      new_frame = take (n-1) frame ++ [closure] ++ drop n frame
```

```
fList f = f
```

- ラベルごとに、**コードストア**は対応するコンパイル済みコードを提供します。

```
type CodeStore = ASSOC Name [Instruction]
```

この機会に、失敗した場合にエラー メッセージを生成するラベルのルックアップ関数を提供します。

```
codeLookup :: CodeStore -> Name -> [Instruction]
codeLookup cstore l
  = aLookup cstore l (error ("Attempt to jump to unknown label "
                             ++ show l))
```

- いつものように、**統計**を簡単に追加できる抽象データ型にします。

```
statInitial :: TimStats
statIncSteps :: TimStats -> TimStats
statGetSteps :: TimStats -> Int
```

ステップ数のみをカウントする最初の実装はかなり単純です。

```
type TimStats = Int -- The number of steps
statInitial = 0
statIncSteps s = s+1
statGetSteps s = s
```

最後に、ヒープとスタックのコードが必要です。

```
-- :a util.lhs -- heap data type and other library functions
```

4.2.3　プログラムのコンパイル

compile は、与えられたプログラムから初期マシン状態を作成する、テンプレートインスタンス化のコンパイラと非常によく似た働きをします。主な違いは、コンパイル関数 compileSC にあります。これは各スーパーコンビネータに適用されます。

```
compile program
  = ([Enter (Label "main")], -- Initial instructions
     FrameNull,              -- Null frame pointer
     initialArgStack,        -- Argument stack
     initialValueStack,      -- Value stack
     initialDump,            -- Dump
     hInitial,               -- Empty heap
     compiled_code,          -- Compiled code for supercombinators
     statInitial)            -- Initial statistics
    where
      sc_defs = preludeDefs ++ program
      compiled_sc_defs = map (compileSC initial_env) sc_defs
      compiled_code = compiled_sc_defs ++ compiledPrimitives
      initial_env = [(name, Label name) | (name, args, body) <- sc_defs] ++
                    [(name, Label name) | (name, code) <- compiledPrimitives]
```

今のところ、引数スタックは空に初期化されています。

```
initialArgStack = []
```

今のところ、値スタックとダンプはダミー値に初期化されています。後でこれらの定義を変更します。

```
initialValueStack = DummyTimValueStack
initialDump = DummyTimDump
```

コンパイルされたスーパーコンビネータ compiled_sc_defs は、compileSC を使用して、プログラム内の各スーパーコンビネータをコンパイルすることによって取得されます。compileSC に渡される初期環境は、各スーパーコンビネータに適したアドレス指定モードを提供します。コードストア、compiled_code は、compiled_sc_defs と compiledPrimitives を組み合わせることによって取得されます。後者は、組み込みのプリミティブ用にコンパイルされたコードを含むことを目的としていますが、現時点では空です:

```
compiledPrimitives = []
```

テンプレートマシンや G マシンとは異なり、初期ヒープは空です。これらの場合に空でない初期ヒープの理由は、CAF(つまり、引数のないスーパーコンビネータ - セクション 2.1.6) の共有を保持するためでした。TIM マシンのこの初期バージョンでは、CAF 用にコンパイルされた TIM コードが呼び出されるたびに実行されるため、CAF を評価する作業は共有されません。この問題については、後でセクション 4.7 で説明します。

コンパイラの核心は、コンパイルスキーム $\mathcal{SC}$、$\mathcal{R}$、および $\mathcal{A}$ をそれぞれ関数 compileSC、compileR、および compileA に直接変換することです。環境 ρ は、名前をアドレッシングモードにバインドする連想リストによって表されます。G マシンのコンパイラは、名前からスタックオフセットへのマッピングを使用していましたが、アドレッシングモードを使用することで得られる特別な柔軟性はかなり有用であることがわかりました。

```
type TimCompilerEnv = [(Name, TimAMode)]
```

これで compileSC を定義する準備が整いました:

```
compileSC :: TimCompilerEnv -> CoreScDefn -> (Name, [Instruction])
compileSC env (name, args, body)
  = (name, Take (length args) : instructions)
    where
      instructions = compileR body new_env
      new_env = (zip2 args (map Arg [1..])) ++ env
```

compileR は式と環境を受け取り、命令のリストを提供します。

```
compileR :: CoreExpr -> TimCompilerEnv -> [Instruction]
compileR (EAp e1 e2) env = Push (compileA e2 env) : compileR e1 env
compileR (EVar v)    env = [Enter (compileA (EVar v) env)]
compileR (ENum n)    env = [Enter (compileA (ENum n) env)]
compileR e           env = error "compileR: can't do this yet"

compileA :: CoreExpr -> TimCompilerEnv -> TimAMode
compileA (EVar v) env = aLookup env v (error ("Unknown variable " ++ v))
compileA (ENum n) env = IntConst n
compileA e env = Code (compileR e env)
```

4.2.4　評価器

次に、評価器が実際にどのように機能するかを定義する必要があります。eval の定義は、テンプレートインスタンス化マシンとまったく同じです。

```
eval state
  = state : rest_states
    where
      rest_states | timFinal state = []
                  | otherwise      = eval next_state
      next_state = doAdmin (step state)
```

```
doAdmin state = applyToStats statIncSteps state
```

timFinal 関数は、状態が最終状態であることを示します。Stop 命令を発明することもできますが、コードシーケンスが空の場合に終了したと言うのは簡単です。

```
timFinal ([], frame, stack, vstack, dump, heap, cstore, stats) = True
timFinal state                                                  = False
```

applyToStats 関数は、状態の統計コンポーネントに関数を適用するだけです。

```
applyToStats stats_fun (instr, frame, stack, vstack,
                        dump, heap, cstore, stats)
  = (instr, frame, stack, vstack, dump, heap, cstore, stats_fun stats)
```

ステップを進める

step は、単一の命令を受け取って実行するケース分析を行います。Take 式は、対応する状態遷移規則 (4.1) を単純に訳したものです。

```
step ((Take n:instr), fptr, stack, vstack, dump, heap, cstore,stats)
  | length stack >= n = (instr, fptr', drop n stack, vstack, dump,
                         heap', cstore, stats)
  | otherwise     = error "Too few args for Take instruction"
  where (heap', fptr') = fAlloc heap (take n stack)
```

Enter と Push の方程式は、timAMode をクロージャに変換する共通関数 amToClosure を使用して、Push/Enter の関係を利用します。

```
step ([Enter am], fptr, stack, vstack, dump, heap, cstore, stats)
  = (instr', fptr', stack, vstack, dump, heap, cstore, stats)
    where (instr',fptr') = amToClosure am fptr heap cstore


step ((Push am:instr), fptr, stack, vstack, dump, heap, cstore, stats)
  = (instr, fptr, amToClosure am fptr heap cstore : stack,
     vstack, dump, heap, cstore, stats)
```

amToClosure は、最初の引数であるアドレッシングモードによってアドレス指定されたクロージャを提供します。

```
amToClosure :: TimAMode -> FramePtr -> TimHeap -> CodeStore -> Closure
amToClosure (Arg n)      fptr heap cstore = fGet heap fptr n
amToClosure (Code il)    fptr heap cstore = (il, fptr)
amToClosure (Label l)    fptr heap cstore = (codeLookup cstore l, fptr)
amToClosure (IntConst n) fptr heap cstore = (intCode, FrameInt n)
```

4.2.5 結果の表示

テンプレートインスタンス化バージョンと同様に、結果を適切な方法で出力するには、かなり退屈な関数のコレクションが必要です。スーパーコンビネータの定義を出力すると便利な場合が多いため、最初の状態の定義を使用して、showResults を出力することから始めます。

```
showFullResults states
  = iDisplay (iConcat [
      iStr "Supercombinator definitions", iNewline, iNewline,
      showSCDefns first_state, iNewline, iNewline,
      iStr "State transitions", iNewline,
      iLayn (map showState states), iNewline, iNewline,
      showStats (last states)
    ])
    where
      (first_state:rest_states) = states
```

showResults は、最後の状態といくつかの統計を表示するだけです。

```
showResults states
  = iDisplay (iConcat [
      showState last_state, iNewline, iNewline, showStats last_state
    ])
    where last_state = last states
```

残りの機能は簡単です。showSCDefns は、各スーパーコンビネータのコードを表示します。

```
showSCDefns :: TimState -> Iseq
showSCDefns (instr, fptr, stack, vstack, dump, heap, cstore, stats)
  = iInterleave iNewline (map showSC cstore)

showSC :: (Name, [Instruction]) -> Iseq
showSC (name, il)
```

```
    = iConcat [
        iStr "Code for ", iStr name, iStr ":", iNewline,
        iStr " ", showInstructions Full il, iNewline, iNewline
      ]
```

showState は、TIM マシンの状態を表示します。

```
showState :: TimState -> Iseq
showState (instr, fptr, stack, vstack, dump, heap, cstore, stats)
  = iConcat [
      iStr "Code: ", showInstructions Terse instr, iNewline,
      showFrame heap fptr,
      showStack stack,
      showValueStack vstack,
      showDump dump,
      iNewline
    ]
```

showFrame は状態のフレーム コンポーネントを表示し、showClosure を使用してその内部の各クロージャを表示します。

```
showFrame :: TimHeap -> FramePtr -> Iseq
showFrame heap FrameNull = iStr "Null frame ptr" `iAppend` iNewline
showFrame heap (FrameAddr addr)
  = iConcat [
      iStr "Frame: <",
      iIndent (iInterleave iNewline
      (map showClosure (fList (hLookup heap addr)))),
      iStr ">", iNewline
    ]
showFrame heap (FrameInt n)
  = iConcat [ iStr "Frame ptr (int): ", iNum n, iNewline ]
```

showStack は引数スタックを表示し、showClosure を使用して各クロージャを表示します。

```
showStack :: TimStack -> Iseq
showStack stack
  = iConcat [ iStr "Arg stack: [",
      iIndent (iInterleave iNewline (map showClosure stack)),
      iStr "]", iNewline
    ]
```

現時点では、値スタックとダンプを表示する showValueStack と showDump は、状態のこれらのコンポーネントを使用していないため、現時点ではスタブです。

```
showValueStack :: TimValueStack -> Iseq
showValueStack vstack = iNil

showDump :: TimDump -> Iseq
showDump dump = iNil
```

showClosure はクロージャを表示し、showFramePtr を使用してフレームポインタを表示します。

```
showClosure :: Closure -> Iseq
showClosure (i,f)
  = iConcat [ iStr "(", showInstructions Terse i, iStr ", ",
      showFramePtr f, iStr ")"
    ]

showFramePtr :: FramePtr -> Iseq
showFramePtr FrameNull     = iStr "null"
showFramePtr (FrameAddr a) = iStr (show a)
showFramePtr (FrameInt n)  = iStr "int " `iAppend` iNum n
```

showStats は、蓄積された統計を出力します。

```
showStats :: TimState -> Iseq
showStats (instr, fptr, stack, vstack, dump, heap, code, stats)
  = iConcat [ iStr "Steps taken = ", iNum (statGetSteps stats), iNewline,
              iStr "No of frames allocated = ", iNum (hSize heap),
              iNewline
    ]
```

命令の表示

命令と命令シーケンスを表示する必要があります。一連の命令が 1 つの長い行として出力されると、かなり読みにくくなるため、それらをきれいに出力するコードを作成する価値があります。

実際、命令シーケンスのコード全体 (たとえば、スーパーコンビネータの定義を出力する場合) を出力できるようにするか、一部の省略形だけを出力できるようにしたいと考えています。後者の例は、スタックの内容を表示す

るときに発生します。各クロージャのコードの一部を確認することは役に立ちますが、すべてを確認する必要はありません。したがって、各関数に追加の引数 d を与えて、どの程度完全に出力するかを伝えます。この引数の値は、Full、Terse、または None のいずれかです。

```
data HowMuchToPrint = Full | Terse | None
```

showInstructions は命令のリストを iseq に変換します。d が None の場合、省略記号のみが出力されます。d が Terse の場合、命令はすべて 1 行に出力され、ネストされた命令は d を None として出力されます。d が Full の場合、命令は 1 行に 1 つずつレイアウトされ、完全に出力されます。

```
showInstructions :: HowMuchToPrint -> [Instruction] -> Iseq
showInstructions None il = iStr "{..}"
showInstructions Terse il
  = iConcat [iStr "{", iIndent (iInterleave (iStr ", ") body), iStr "}"]
    where
      instrs = map (showInstruction None) il
      body | length il <= nTerse = instrs
           | otherwise           = (take nTerse instrs) ++ [iStr ".."]
showInstructions Full il
  = iConcat [iStr "{ ", iIndent (iInterleave sep instrs), iStr " }"]
    where
      sep = iStr "," `iAppend` iNewline
      instrs = map (showInstruction Full) il
```

showInstruction は、単一の命令を iseq に変換します。

```
showInstruction d (Take m)  = (iStr "Take ")  `iAppend` (iNum m)
showInstruction d (Enter x) = (iStr "Enter ") `iAppend` (showArg d x)
showInstruction d (Push x)  = (iStr "Push ")  `iAppend` (showArg d x)

showArg d (Arg m)      = (iStr "Arg ")      `iAppend` (iNum m)
showArg d (Code il)    = (iStr "Code ")     `iAppend` (showInstructions d il)
showArg d (Label s)    = (iStr "Label ")    `iAppend` (iStr s)
showArg d (IntConst n) = (iStr "IntConst ") `iAppend` (iNum n)
```

nTerse は、シーケンスのいくつの命令を簡潔な形式で出力する必要があるかを示します。

```
nTerse = 3
```

演習 4.1 main の次の定義を使用してマシンを実行します。

```
main = S K K 4
```

S K K は恒等関数であるため、main は 4 に減少し、マシンが停止します。もう少し手の込んだものにしてみてください。例えば

```
id = S K K ;
id1 = id id ;
main = id1 4
```

演習 4.2 さらなる性能測定を追加します。例えば：

- 実行時間を測定し、Take を除く命令ごとに 1 つの時間単位をカウントします。Take は、フレームに含まれる要素と同じ数の時間単位をカウントする必要があります。
- 実行で割り当てられたヒープの合計量を出力して、ヒープ使用量を測定します。フレームのサイズを数えて、結果をテンプレートインスタンス化バージョンの結果と直接比較できるようにします。
- 最大スタック深さを測定します。

演習 4.3 $n = 0$ の場合、Take n は何も役に立ちません。CAF の Take 命令を完全に省略することで、compileSC の定義を調整して、この最適化を見つけます。

4.2.6　ガベージコレクション †

ヒープベースのシステムと同様に、TIM にはガベージコレクタが必要ですが、少し洗練されたものも必要です。いつものように、ガベージコレクタはマシンの状態から開始して、すべてのライブデータを検出します。つまり、スタックとフレームポインタからです。スタック上の各クロージャはフレームを指しており、明確に保持する必要があります。しかし、そのフレームには、さらに別のフレームへのポインタが含まれています。問題が発生します: **特定のフレームが与えられた場合、その中のどのフレームポインタを再帰的に追跡する必要がありますか?**

安全な答えは「それらすべてに従う」ことですが、これには必要以上のデータを保持するリスクがあります。たとえば、セクション 4.1.4 の compose2 の例の (g x x) のクロージャには、f、g、および x を含むフレームへのポインタがありますが、g と x のクロージャのみが必要です。ナイーブなガベージコレクタは、f のクロージャからのフレームポインタもたどり、データを不必

要に保持する可能性があります。この不要な保持は**スペースリーク**と呼ばれ、ガベージコレクションが他の場合よりも頻繁に発生する可能性があります。

ただし、この特定のスペースリークは、かなり面倒ではありますが、解消するのは簡単です。各クロージャは、フレームポインタとペアになったコードポインタで構成されます。コードは、どのフレーム要素が必要になるかを「認識」しており、この情報をコードに記録して、ガベージコレクターが調査できるようにします。たとえば、コードポインタと呼んでいるものは、実際には、コードで使用されるスロット番号のリストとコード自体で構成されるペアを指している可能性があります。(実際の実装では、リストはビットマスクとしてエンコードされる場合があります。) 有用なスロットのリストはどのように導出できますか? それは簡単です: コンパイルされている式の自由変数を見つけ、環境を使用してそれらをスロット番号にマップするだけです。

4.3　Mark 2: 算術演算の追加

このセクションでは、マシンに算術演算を追加します。

4.3.1　概要: 算術演算の仕組み

オリジナルの Fairbairn と Wray の TIM マシンは、算術を行うためのかなり悪質なスキームを持っていました。彼らの主な動機は、マシンを**最小限**に抑えることでしたが、彼らのアプローチは非常に理解しにくく、効率的な実装を行うにはかなりの修正が必要です。

代わりに、G マシンの V スタックとまったく同じように TIM を変更します (セクション 3.9)。(評価され、ボックス化されていない) 整数のスタックである**値スタック**を導入することにより、状態を変更します。値スタックの最上位要素で算術演算 *op* を実行し、値スタックの最上位に結果を残す命令 `Op` *op* ファミリーで命令セットを拡張します。たとえば、`Op Sub` 命令は、値スタックの上位 2 つの要素を削除し、それらを減算して、結果を値スタックにプッシュします。

(4.10)

$$\begin{array}{|rrrrrrrr|}\hline & \texttt{Op Sub} : i & f & s & n_1 : n_2 : v & h & c \\ \Longrightarrow & i & f & s & (n_1 - n_2) : v & h & c \\ \hline\end{array}$$

このようにして、`Op Add`、`Op Sub`、`Op Mult`、`Op Div`、`Op Neg` などの算術命令の完全なファミリを簡単に定義できます。

次に、以下の関数 `sub` を考えてみましょう。

```
sub a b = a - b
```

`sub` 用にどのコードを生成する必要がありますか? 次の手順を実行する必要があります。

1. 引数をフレームに形成する通常の `Take 2`。
2. `b` を評価し、その値を値スタックに置きます。
3. 同様にして `a` を評価します。
4. `Op Sub` 命令を使用して `a` の値から `b` の値を引き、その結果を値スタックの一番上に残します。
5. 「呼び出し元」に「戻る」。

まず、a と b の評価を考えます。それらはクロージャによって表され、現在のフレームに保持されます。クロージャに対してできることは、それを入力することだけです。おそらく a を評価するには、a のクロージャを入力する必要がありますが、整数値のクロージャを入力するとはどういう意味でしょうか? これまで関数を入力しただけで、整数は関数ではありません。重要なアイデアは次のとおりです。

> 整数不変: **整数値のクロージャが入力されると、整数の値が計算され、それが値スタックにプッシュされ、引数スタックの一番上のクロージャが入力されます。**

引数スタックの一番上にあるクロージャは、整数の評価が完了したら次に何をすべきかを示しているため、**継続**と呼ばれます。継続は、整数の評価が完了したときに何をすべきかを示す命令シーケンスと、現在のフレームポインタ (整数の評価によって妨害された場合) で構成されます。つまり、継続は完全に通常のクロージャです。

したがって、sub のコードは次のようになります。

```
sub: Take 2
     Push (Label L1)  -- Push the continuation
     Enter (Arg 2)    -- Evaluate b

L1:  Push (Label L2)  -- Push another continuation
     Enter (Arg 1)    -- Evaluate a

L2:  Op Sub           -- Compute a-b on value stack
```

Return 命令は何をすべきですか？sub によって返される値は整数であり、Op Sub 命令の後、この整数は値スタックの一番上にあるため、Return がしなければならないことは、引数スタックの一番上にクロージャをポップして入力することだけです。

(4.11)

$$\begin{array}{lllll} & [\texttt{Return}] & f & (i', f') : s & v & h & c \\ \Longrightarrow & & i' & f' & s & v & h & c \end{array}$$

ラベルを使用して、sub のコードを記述しました。これが唯一の方法ではありません。別の方法として、Push Code 命令を使用する方法があります。これにより、面倒な新しいラベルを作成する必要がなくなります。このスタイルでは、sub のコードは次のようになります。

```
sub: Take 2
     Push (Code [ Push (Code [Op Sub, Return]),
                  Enter (Arg 1)
          ])
     Enter (Arg 2)
```

このように書かれていると、ラベルを使用するよりも何が起こっているかを確認するのが簡単ではないため、コードの断片を理解しやすくするために説明では引き続きラベルを使用しますが、コンパイラでは `Push Code` バージョンを使用します。

ここで、整数定数の問題に戻る必要があります。`(sub 4 2)` という式を考えてみましょう。以下のコードにコンパイルされます

```
Push (IntConst 2)
Push (IntConst 4)
Enter (Label "sub")
```

`sub` のコードはすぐにクロージャ`(IntConst 2)` に入り、フレームポインタに整数 2 を配置して `intCode` にジャンプします。現在、`intCode` は空のコードシーケンスです (整数を Enter するとマシンが停止します) が、これを変更する必要があります。`intCode` は今何をすべきか? 答えは整数不変式によって与えられます: 整数を値スタックにプッシュして返す必要があるため、次のようになります。

```
intCode = [PushV FramePtr, Return]
```

`PushV FramePtr` は、現在フレームポインタとして偽装されている数値を値スタックの一番上にプッシュする新しい命令です。

(4.12)

	`PushV FramePtr` : i	n	s		v	h	c
$\Longrightarrow$		i	n	s	n : v	h	c

4.3.2　実装への単純な算術演算の追加

これで、実装を変更する準備が整いました。各算術関数のコードを `compiledPrimitives` に追加することで、変更を最小限に抑えます。プログラムで (たとえば)`p-q` と書くと、パーサはそれを次のように変換することを思い出してください。

```
EAp (EAp (EVar "-") (EVar "p")) (EVar "q")
```

プリミティブ-に適したコードを作成し、このコードをコードストアに追加するだけです。コンパイラは、他のスーパーコンビネータと同じように-を扱います。最後に、必要な-のコードは、前のセクションで作成した sub とまったく同じであり、他の算術演算についても同様のコードを簡単に作成できます。

したがって、必要な手順は次のとおりです。

- 次の型定義と値スタックの初期化を追加します。

```
type TimValueStack = [Int]
initialValueStack = []
```

- 新しい命令 PushV、Return、および Op を instruction 型に追加します。この機会に、Cond という命令を 1 つ追加します。これについてはまだ説明していませんが、後の演習の対象となります。TIM はもはや 3 命令マシンではありません！

```
data Instruction = Take Int
                 | Push TimAMode
                 | PushV ValueAMode
                 | Enter TimAMode
                 | Return
                 | Op Op
                 | Cond [Instruction] [Instruction]
data Op = Add | Sub | Mult | Div | Neg
        | Gr | GrEq | Lt | LtEq | Eq | NotEq
      deriving (Eq) -- KH
```

これまでのところ、PushV 命令の引数は FramePtr のみですが、リテラル定数を値スタックにプッシュできるようにする 2 番目の形式をすぐに追加します。したがって、valueAMode の代数データ型を宣言する価値があります。

```
data ValueAMode = FramePtr
                | IntVConst Int
```

この追加構造を処理するには、showInstruction 関数を変更する必要があります。

- step 関数を変更して、追加の命令を実装します。これは、状態遷移規則を Miranda に翻訳する問題です。
- +、-などの適切な定義を compiledPrimitives に追加します。
- intCode が空でなくなったので、スタックを初期化して、main が戻るための適切な継続 (戻りアドレス) を持たせる必要があります。これを行う方法は、initialArgStack を再定義することにより、compile にクロージャ ([], FrameNull) でスタックを初期化させることです。

```
initialArgStack = [([], FrameNull)]
```

この継続には空のコードシーケンスがあるため、マシンは結果を値スタックの上に置いて停止します。

演習 4.4 これらの変更をプロトタイプに実装します。簡単な例で試してみてください。例えば

```
four = 2 * 2
main = four + four
```

演習 4.5 まだ条件式がなく、条件式がなければ再帰を使用できないため、まだ「興味深い」プログラムを実行できません。簡単な解決策は、新しい命令 Cond i1 i2 を追加することです。この命令は、値スタックの一番上から値を削除し、それがゼロかどうかをチェックし、ゼロである場合は命令シーケンス i1 を続行し、そうでない場合は i2 を続行します。その状態遷移規則は次のとおりです。

(4.13)

	[Cond	i_1	i_2]	f	s	$0 : v$	h	c
$\Longrightarrow$			i_1	f	s	v	h	c
	[Cond	i_1	i_2]	f	s	$n : v$	h	c
$\Longrightarrow$			i_2	f	s	v	h	c
	ここで $n \neq 0$								

最初のルールは、ゼロが値スタックの一番上にある場合に一致します。それ以外の場合は、2 番目の規則が適用されます。次のように動作するプリミティブ if も追加する必要があります。

```
if 0 t f = t
if n t f = f
```

Cond 命令を使用して if の TIM コードを作成し、それを compiledPrimitives に追加する必要があります。最後に、階乗関数を使用して改善されたシステムをテストできます。

```
factorial n = if n 1 (n * factorial (n-1))
main = factorial 3
```

4.3.3 算術演算のコンパイルスキーム

G マシンの場合と同様に、現在行っているマシン用のコンパイルよりもはるかに優れた仕事を行うことができます。次のような関数を考えてみましょう

```
f x y z = (x+y) * z
```

現状では、これは次のように解析されます

```
f x y z = * (+ x y) z
```

f のコードがコンパイルされ、標準関数*および+が呼び出されます。しかし、これよりもはるかに良いことができます! たとえば、(+ x y) のクロージャを作成して*に渡す代わりに、次の手順を使用してインラインで操作を行うことができます。

1. x を評価する
2. y を評価する
3. それらを加算する
4. z を評価する
5. 乗算する
6. 戻る

クロージャを構築する必要はなく、ジャンプを行う必要もありません (x、y、z を評価するために必要なものを除く)。

この改善を表現するために、値が整数である式を処理する新しいコンパイルスキーム、$\mathcal{B}$ スキームを導入します。次のように定義されます: 値が整数である任意の式 e と、任意のコード シーケンス $cont$ に対して、

($\mathcal{B}[\![e]\!]\ \rho\ cont$) は、現在のフレームが ρ で記述されているようにレイアウトされた状態で実行されると、式 e の値を値スタックにプッシュしてから、コードシーケンス $cont$ を実行するコードシーケンスです。

コンパイルスキームは**継続渡しスタイル**を使用します。このスタイルでは、$cont$ 引数が、値が計算された後に何をすべきかを示します。

図 4.2 は、改訂された $\mathcal{R}$ および $\mathcal{A}$ スキームと共に、$\mathcal{B}$ コンパイルスキームを示しています。$\mathcal{R}$ が算術式である式を見つけると、それをコンパイルするために $\mathcal{B}$ を呼び出します。$\mathcal{B}$ には、継続を明示的にプッシュすることを回避する、定数および算術演算子の適用に関する特殊なケースがあります。特別に処理できない式に遭遇した場合は、継続をプッシュして $\mathcal{R}$ を呼び出すだけです。

$\mathcal{R}[\![e]\!]\ \rho$ は、環境 ρ の式 e の値をスタック上の引数に適用する TIM コードです。

$$
\begin{array}{rcll}
\mathcal{R}[\![e]\!]\ \rho & = & \mathcal{B}[\![e]\!]\ \rho\ [\texttt{Return}] & \\
 & & \text{ここで、}e\text{ は }e_1 + e_2\text{ などの算術式、または数値です。} & \\
\mathcal{R}[\![e_1\ e_2]\!]\ \rho & = & \texttt{Push}\ (\mathcal{A}[\![e_2]\!]\ \rho)\ :\ \mathcal{R}[\![e_1]\!]\ \rho & \\
\mathcal{R}[\![a]\!]\ \rho & = & \texttt{Enter}\ (\mathcal{A}[\![a]\!]\ \rho) & \\
 & & \text{ここで、}a\text{ は整数、変数、またはスーパーコンビネータです。} &
\end{array}
$$

$\mathcal{A}[\![e]\!]\ \rho$ は、環境 ρ における式 e の TIM アドレッシングモードです。

$$
\begin{array}{rcll}
\mathcal{A}[\![x]\!]\ \rho & = & \rho\ x & \text{ここで、}x\text{ は }\rho\text{ によって束縛されます。} \\
\mathcal{A}[\![n]\!]\ \rho & = & \texttt{IntConst}\ n & \text{ここで、}n\text{ は整数定数です。} \\
\mathcal{A}[\![e]\!]\ \rho & = & \texttt{Code}\ (\mathcal{R}[\![e]\!]\ \rho) & \text{その他}
\end{array}
$$

$\mathcal{B}[\![e]\!]\ \rho$ は、環境 ρ で式 e を評価して、
その値 (整数でなければならない) を値スタックの一番上に置き、
コード シーケンス $cont$ を続行する TIM コードです。

$$
\begin{array}{rcll}
\mathcal{B}[\![e_1\texttt{+}\ e_2]\!]\ \rho\ cont & = & \mathcal{B}[\![e_2]\!]\ \rho\ (\mathcal{B}[\![e_1]\!]\ \rho\ (\texttt{Op Add}\ :\ cont)) & \\
 & & \ldots\text{および他の算術プリミティブの同様の規則} & \\
\mathcal{B}[\![n]\!]\ \rho\ cont & = & \texttt{PushV}\ (\texttt{IntVConst}\ n)\ :\ cont & \text{ここで、}n\text{ は数値です。} \\
\mathcal{B}[\![e]\!]\ \rho & = & \texttt{Push}\ (\texttt{Code}\ cont)\ :\ \mathcal{R}[\![e]\!]\ \rho & \text{その他}
\end{array}
$$

必要な新しい命令が 1 つあります。これは、$\mathcal{B}$ が定数をコンパイルするように要求されたときに使用されます。その時には、整数定数を値スタックにプッシュする命令 PushV (IntVConst n) が必要です。その遷移規則は非常に単純です。

(4.14)

	`PushV` (`IntVCont` n) : i	f	s		v	h	c
$\Longrightarrow$		i	f	s	$n : v$	h	c

演習 4.6 改善されたコンパイル スキームを実装します。実装のパフォーマンスを以前のパフォーマンスと比較します。

演習 4.7 $\mathcal{R}$ スキームに新しいルールを追加して、`if` の (完全な) アプリケーションに一致させます。`if` プリミティブを呼び出して取得するよりもはるかに優れたコードを生成できるはずです。変更を実装し、パフォーマンスの改善を測定します。

演習 4.8 次のように、より一般的な算術比較を処理するために条件を一般化するとします。

```
fib n = if (n < 2) 1 (fib (n-1) + fib (n-2))
```

必要なのは、値スタックの上位 2 つの項目をポップし、それらを比較し、比較の結果に応じて値スタックに 1 または 0 をプッシュする新しい命令 `Op Lt` です。次に、`Cond` 命令がこの結果を検査します。

このような比較命令のファミリを実装し、他の算術演算子とまったく同じ方法で、それらの特殊なケースを $\mathcal{B}$ スキームに追加します。あなたの改善をテストします。

演習 4.9 前の練習問題で、追加の比較モードを持つように `Cond` 命令を変更しなかった理由を疑問に思ったかもしれません。次に、このモードに従って値スタックの上位 2 つの項目を比較し、それに応じて動作します。なぜこれをしなかったのですか？ヒント: このようなプログラムはどうなるでしょうか?

```
multipleof3 x = ((x / 3) * 3) == x
f y = if (multipleof3 y) 0 1
```

このセクションの内容は [Argo 1989] で説明されており、[Peyton Jones 1987] の第 20 章で説明されている改良された G マシンコンパイルスキームに正確に対応しています。

4.4　Mark 3: let(rec) 式

現在、コンパイラは let(rec) 式を処理できません。これは、このセクションで解決する問題です。2 つの主要な新しいアイデアが導入されました。

- Take 命令を変更して、余分なスペースをフレームに割り当てて、let(rec) 束縛変数と仮パラメータを含めます。
- **間接クロージャ**の考え方を紹介します。

4.4.1　let 式

let 式をコンパイルするとき、定義の右辺の新しいクロージャを構築するコードを生成する必要があります。これらの新しいクロージャはどこに配置する必要がありますか? let(rec) で束縛された名前を引数名と同じように扱うには、それらを現在のフレームに配置する必要があります[3]。これには、ランタイム機構に 2 つの変更が必要です。

- Take 命令は、スーパーコンビネータの実行中に発生するすべての let 定義のクロージャを含むのに十分な大きさのフレームを割り当てる必要があります。Take 命令は、Take t n の形式に変更する必要があります。ここで、$t \geq n$ です。この命令は、サイズ t のフレームを割り当て、スタックの一番上から n 個のクロージャを取得し、それらをフレームの最初の n 個の場所に配置します。
- 新しいクロージャ a を現在のフレームのスロット i に移動するには、新しい命令 Move i a が必要です。ここで a は、Push および Enter と同様に timAMode 型です。

たとえば、次の定義です。

```
f x = let
        y = f 3
      in
        g x y
```

これは次のコードにコンパイルされます。

```
[ Take 2 1,
  Move 2 (Code [Push (IntConst 3), Enter (Label "f")]),
  Push (Arg 2),
```

[3] このセクションの資料のヒントは [Wakeling and Dix 1989] にありますが、完全には解明されていません。

```
    Push (Arg 1),
    Enter (Label "g")
  ]
```

もう少し複雑な例を次に示します。

```
f x = let
        y = f 3
      in
        g (let z = 4 in h z) y
```

これは次のコードを生成します：

```
[ Take 3 1,
  Move 2 (Code [Push (IntConst 3), Enter (Label "f")]),
  Push (Arg 2),
  Push (Code [Move 3 (IntConst 4), Push (Arg 3), Enter (Label "h")]),
  Enter (Label "g")
]
```

スーパーコンビネータ本体のクロージャが必要とする**すべての**スロットに、最初の Take がスペースを割り当てる方法に注目してください。

演習 4.10 新しい Take および Move 命令の状態遷移規則を記述します。

次に、新しい Take 命令と Move 命令を生成するようにコンパイラを変更する必要があります。let 式に遭遇した場合、フレーム内の空きスロットを各バインド変数に割り当てる必要があるため、フレーム内のどのスロットが使用中で、どのスロットが空きであるかを追跡する必要があります。これを行うには、追加のパラメータ d を各コンパイル スキームに追加して、$d+1$ 以降のフレームスロットが空いていることを記録しますが、1 から d までのスロットは占有される可能性があることを記録します。

残りの複雑な点は、最初の Take 命令で十分な大きさのフレームを割り当てることができるように、d が取り得る最大値を発見する必要があることです。これには、各コンパイルスキームが、コンパイルされたコードと d によって取得される最大値のペアを返す必要があります。新しいコンパイルスキームを図 4.3 に示します。(この図とその後で、is_1 ++ is_2 という表記を使用して、命令シーケンス is_1 と is_2 の連結を示します。)

$\mathcal{SC}[\![e]\!]\ \rho$ は、環境 ρ でコンパイルされたスーパーコンビネータ定義 def の TIM コードです。

$$\mathcal{SC}[\![f\ x_1\ \ldots\ x_n\ \texttt{=}\ e]\!]\ \rho \quad = \quad \texttt{Take}\ d'\ n\ \texttt{:}\ is$$

ここで

$(d',\ is) = \mathcal{R}[\![e]\!]\ \rho[x_1 \mapsto \texttt{Arg}\ 1,\ \ldots,\ x_n \mapsto \texttt{Arg}\ n]\ n$

です。

$\mathcal{R}[\![e]\!]\ \rho\ d$ は、ペア $(d',\ is)$ です。
ここで、is は、環境 ρ 内の式 e の値をスタック上の引数に適用する TIM コードです。
このコードは、フレームの最初の d スロットが占有されていると想定しており、
スロット $(d+1\ ...\ d')$ を使用します。

$$\mathcal{R}[\![e]\!]\ \rho\ d \quad = \quad \mathcal{B}[\![e]\!]\ \rho\ d\ [\texttt{Return}]$$

ここで、e は $e_1 + e_2$ などの算術式、または数値です。

$$\mathcal{R}[\![\texttt{let}\ x_1\ \texttt{=}\ e_1\texttt{;}\ \ldots\texttt{;}\ x_n\ \texttt{=}\ e_n\ \texttt{in}\ e]\!]\ \rho\ d$$
$$= \quad (d',\ [\texttt{Move}\ (d+1)\ am_1,\ \ldots,\ \texttt{Move}\ (d+n)\ am_n]\ \ \texttt{++}\ \ \text{is})$$

ここで

$(d_1,\ am_1) = \mathcal{A}[\![e_1]\!]\ \rho\ (d+n)$

$(d_2,\ am_2) = \mathcal{A}[\![e_2]\!]\ \rho\ d_1$

$\ldots$

$(d_n,\ am_n) = \mathcal{A}[\![e_n]\!]\ \rho\ d_{n-1}$

$\rho' = \rho\,[x_1\ \mapsto\ \texttt{Arg}\ (d+1),\ \ldots,\ x_n\ \mapsto\ \texttt{Arg}\ (d+n)]$

$(d',\ is) = \mathcal{R}[\![e]\!]\ \rho'\ d_n$

です。

$$\mathcal{R}[\![e_1\ e_2]\!]\ \rho\ d \quad = \quad (d_2,\ \texttt{Push}\ am\texttt{:}is)$$

ここで

$(d_1,\ am) = \mathcal{A}[\![e_2]\!]\ \rho\ d$

$(d_2,\ is) = \mathcal{R}[\![e_1]\!]\ \rho\ d_1$

です。

$$\mathcal{R}[\![a]\!]\ \rho\ d \quad = \quad (d',\ [\texttt{Enter}\ am])$$

ここで、a は整数、変数、またはスーパーコンビネータであり、

$(d',\ am) = \mathcal{A}[\![a]\!]\ \rho\ d$ です。

$\mathcal{A}[\![e]\!]\ \rho$ は、ペア $(d',\ am)$ です。
ここで、am は環境 ρ における式 e の TIM アドレッシングモードです。
このコードは、フレームの最初の d スロットが占有されていると想定しており、
スロット $(d+1\ ...\ d')$ を使用します。

$\mathcal{A}[\![x]\!]\ \rho\ d$	$=$	$(d,\ \rho\ x)$	ここで、x は ρ によって束縛されます。
$\mathcal{A}[\![n]\!]\ \rho\ d$	$=$	$(d,\ \texttt{IntConst}\ n)$	ここで、n は整数です。
$\mathcal{A}[\![e]\!]\ \rho\ d$	$=$	$(d',\ \texttt{Code}\ is)$	その他

ここで、$(d',\ is) = \mathcal{R}[\![e]\!]\ \rho\ d$ です。

$\mathcal{SC}$ スキームでは、スーパーコンビネータ本体のコンパイルから返された最大フレームサイズ d' を使用して、実行する Take の大きさを決定する方法を確認できます。$\mathcal{R}$ スキームの let 式の場合、定義ごとに命令 Move i a を生成します。ここで、i は現在のフレームの空きスロットの番号であり、a は e を $\mathcal{A}$ スキームでコンパイルした結果です。各右辺のコンパイルに、前の右辺によって占有されていた最後のスロットのインデックスが与えられる方法に注意してください。これにより、すべての右辺が異なるスロットを使用することが保証されます。

演習 4.11 このセクションで説明する変更を実装します。新しい命令を instruction 型に追加し、新しいケースを step と showInstruction に追加してそれらを処理し、新しいコンパイルスキームを実装します。

演習 4.12 以下のプログラムを検討します。

```
f x y z = let p = x+y in p+x+y+z
main = f 1 2 3
```

let 式がない場合は、次のように補助関数を使用して記述する必要があります。

```
f' p x y z = p+x+y+z
f x y z = f' (x+y) x y z
main = f 1 2 3
```

これら 2 つのプログラムによって生成されたコードを比較し、消費されたストアと実行されたステップの違いを測定します。let 式を直接実装することで得られる主な節約は何ですか?

4.4.2 letrec 式

letrec 式も処理するにはどうすればよいでしょうか? 最初はとても簡単に思えます。$\mathcal{R}$ スキームの letrec ケースは、am_i の定義で ρ を ρ' に置き換える必要があることを除いて、let ケースとまったく同じです。これは、x_i がそれぞれの右辺のスコープ内にあるためです。

演習 4.13 この追加のケースを compileR に実装し、プログラム上で試してみます。

```
f x = letrec p = if (x==0) 1 q ;
             q = if (x==0) p 2
      in p+q
main = f 1
```

生成されたコードを必ず理解し、テストしてください。

残念ながら、この実装には微妙なバグがあります。以下から生成されたコードを考えてみましょう。

```
f x = letrec a = b ;
             b = x
      in a
```

それは次のとおりです。

```
[Take 3 1, Move 2 (Arg 3), Move 3 (Arg 1), Enter (Arg 2)]
```

b のクロージャは、2 番目の Move によって割り当てられる前に、最初の Move によってコピーされます。

ここから抜け出す方法は 2 つあります。まず最初に、これは愚かなプログラムであると宣言します。b のバインディングのスコープ内で b を x に置き換えるだけです。しかし、後で有益になり、上記のようなばかげたプログラムでも動作できるようにする、より興味深いアプローチがあります。代わりに、最初の Move に対して次のコードを生成するとします。

```
Move 2 (Code [Enter (Arg 3)])
```

これですべてがうまくいきます。Enter (Arg 3) が実行される前にスロット 3 が割り当てられます。実際、クロージャ ([Enter (Arg 3)], f) はフレーム f のスロット 3 への**間接指定**と考えることができます。

このコードは、$\mathcal{R}$ コンパイル スキームの let(rec) ケースを変更することで取得できます。これにより、let(rec) によってバインドされた変数ごとに環境内の間接アドレス指定モードが記録されます。図 4.3 を参照すると、必要な変更は $\mathcal{R}$ スキームの let のルールに対するもので、ρ' の定義は次のようになります。

$$\begin{aligned}\rho' \quad = \quad & \rho[x_1 \mapsto \mathcal{I}[\![d+1]\!],\ \ldots,\ x_n \mapsto \mathcal{I}[\![d+n]\!]] \\ & \text{ここで } \mathcal{I}[\![d]\!] = \texttt{Code}\ [\texttt{Enter}\ (\texttt{Arg}\ d)]\end{aligned}$$

もちろん、これはかなり保守的です。多くの場合、その必要がない場合に間接参照を返しますが、結果として得られるコードは、効率は劣るものの、引き続き正常に動作します。

演習 4.14 このアイデアを実装するために compileR を変更し、上記の例に対して正しいコードが生成されることを確認します。新しい環境で間接アドレス指定モードを生成するには、compileR を変更する際に、補助関数 mkIndMode ($\mathcal{I}$ スキームに対応) を使用します。

```
mkIndMode :: Int -> TimAMode
mkIndMode n = Code [Enter (Arg n)]
```

4.4.3 フレームスロットの再利用 †

現在、スーパーコンビネータ定義の右辺にあるすべての定義は、フレーム内に独自のプライベートスロットを取得します。場合によっては、異なる let(rec) 間でスロットを安全に共有できることがわかるかもしれません。たとえば、次の定義を考えてみましょう。

```
f x = if x (let ... in ...) (let ... in ...)
```

2 つの let 式のうち 1 つだけが評価できることは明らかなので、それらの定義に同じスロットを使用しても完全に安全です。

同様に、式 e_1 + e_2 では、e_1 の評価中に使用された let(rec) スロットは、e_2 が評価されるまでに終了します (または、+ がその引数を逆の順序で評価した場合はその逆になります)。e_1 内の let(rec) バインドされた変数は、e_2 内の変数とスロットを共有できます。

演習 4.15 これを見つけて利用できるようにコンパイラを変更してください。

4.4.4 ガベージ コレクション †

セクション 4.2.6 では、スペース リークを回避できるように、コード シーケンスによってどのフレーム スロットが使用されたかを記録することが望ましいと述べました。Take が割り当てた追加のフレーム スロットを初期化しない場合、ガベージ コレクターがそのコンテンツを処理する危険があります。これらの初期化されていないスロットは有効なポインターとして扱われ、予測できない結果が生じます。最も簡単な解決策はすべてのスロットを初期化することですが、これには非常にコストがかかります。セクション 4.2.6 の

解決策を採用し、保持する必要があるスロットのリストを各コード シーケンスとともに記録する方が良いでしょう。初期化されていないスロットは、ガベージ コレクターによって参照されることはありません。

4.5 Mark 4: 更新

これまでのところ、共有 Redexes を繰り返し評価するため、グラフ簡約ではなくツリー簡約を実行してきました。これを修正する時期が来ました。TIM 更新を実行するさまざまな方法がどのように機能するかを正確に理解するのは少し難しいですが、少なくともプロトタイプの実装はあります。そのため、私たちの開発は非常に具体的なものになります。

4.5.1 基礎技術

標準的なテンプレートインスタンス化マシンと G マシンは、簡約のたびに更新を実行します。(G マシンには末尾呼び出しがいくつか最適化されていますが、原理は同じです。) TIM は**スパインレス**マシンであるため、その更新手法はかなり異なったものになる必要があります。重要なアイデアは次のとおりです。

> G マシンとは異なり、簡約のたびに更新は実行されません。代わりに、クロージャの評価が開始されるとき (つまり、クロージャが Enter されたとき)、次の手順が実行されます。
>
> - 現在のスタックと、入力されているクロージャのアドレスは、マシン状態の新しいコンポーネントである `dump` にプッシュされます。
> - クロージャの評価が完了するとトリガーされる「マウストラップ」が設定されます。
> - クロージャの評価は、空のスタックから開始して通常どおりに実行されるようになりました。
> - マウストラップがトリガーされると、クロージャは通常の形式で更新され、古いスタックがダンプから復元されます。
>
> 「マウストラップ」は次のとおりです。クロージャの評価は新しいスタック上で実行されるため、`Return` 命令が空のスタックを見つけるか、スーパーコンビネータが適用される引数が少なすぎるため、評価は最終的に停止する必要があります。この時点で、式は (頭部) 正規形に達しているため、更新を実行する必要があります。

まず、値が整数型であるクロージャの更新に焦点を当てましょう。クロージャが Enter される直前に、新しい命令 `PushMarker` x が実行されるようにします。これにより、ダンプに情報をプッシュすることで更新メカニズムが

セットアップされます。具体的には、PushMarker x はダンプに次の情報をプッシュします[4]。

- 現在のスタック。
- 現在のフレーム ポインター。更新するクロージャーを含むフレームを指します。
- フレーム内で更新されるクロージャのインデックス x 。

現在のスタックをダンプに保存したので、PushMarker は空のスタックを使用して続行します。その状態遷移規則は次のとおりです。

(4.15)

$$\begin{array}{lllllllll} & \texttt{PushMarker}\ x : i & f & s & v & & d & h & c \\ \Longrightarrow & & i & f & [] & v & (f, x, s) : d & h & c \end{array}$$

しばらくすると、クロージャの評価が完了します。その値は整数であるため、その値は値スタックの最上位にあり、スタックの最上位の継続に戻ることを期待して Return 命令が実行されます。ただし、この時点ではスタックは空になります。これが更新のトリガーです。ダンプがポップされ、更新が実行され、復元されたスタックを使用して Return 命令が再実行されます。このアクションは、次の遷移規則によって説明されます。

(4.16)

$$\begin{array}{lllllllll} & [\texttt{Return}] & f & & [] & n : v & (f_u, x, s) : d & h & c \\ \Longrightarrow & [\texttt{Return}] & f & & s & n : v & d & h' & c \\ & \text{ここで } h' = h[f_u : \langle \ldots, d_{x-1}, (\texttt{intCode}, n), d_{x+1}, \ldots \rangle] & & & & & & & \\ \hline & [\texttt{Return}] & f & (i, f') : s & & n : v & d & h & c \\ \Longrightarrow & & i & f' & s & n : v & d & h & c \end{array}$$

最初の遷移規則は、フレーム f_u 内の x 番目のクロージャをクロージャ (intCode, n) で更新することを記述します。これは、Enter されるとすぐに n を値スタックにプッシュして Return するクロージャです (セクション 4.3.1 を参照)。さらに更新を実行する必要がある場合に備えて、Return 命令が再試行されることに注意してください。これは、遷移規則の右辺のコードシーケンスが依然として [Return] であるという事実によって示されます。

2 番目の遷移規則は、遷移規則 4.11 を再度書き直したものです。実行する

[4] TIM に関する論文では、この操作は「更新マーカーのプッシュ」と呼ばれることがよくあります。これは、「マーカー」の下にある引数を使用しようとすると更新がトリガーされるようにスタックが「マーク」されているためです。

更新がない場合についても説明します。スタックの一番上の継続は、プログラムカウンタと現在のフレームポインタにロードされます。

4.5.2 PushMarker 命令のコンパイル

このように、PushMarker 命令と Return 命令の実行は非常に簡単ですが、難しい問題は、コンパイラが PushMarker 命令をどこに配置するかということです。これに答えるには、更新作業全体の動機を思い出す必要があります。評価後に Redex をその値で上書きすることで、各 Redex が 1 回だけ評価されるようにするためです。TIM では、「redex」はクロージャです。重要な洞察は次のとおりです。

> クロージャをコピーするときは非常に注意する必要があります。2 つのコピーが存在すると、それらの評価を共有することはできないからです。

たとえば、次の関数定義を考えてみましょう。

```
g x = h x x
h p q = q - p
```

現時点では、g に対して次のコードを生成します。

```
[ Take 1 1, Push (Arg 1), Push (Arg 1), Enter (Label "h") ]
```

2 つの Push Arg 命令はそれぞれ同じクロージャのコピーを取得し、その後 h はそれぞれを独立して評価します。

私たちが本当にやりたいことは、x のクロージャの**コピー**ではなく、それ**へのポインタ**をプッシュすることです。セクション 4.4.2 の**間接**クロージャのアイデアを思い出してください。これは、Push (Arg 1) を Push (Code [Enter (Arg 1)]) に置き換えることによって簡単に実行できます。

これで半分まで進みました。 クロージャをコピーしていませんが、まだ更新していません。でも今は簡単です！必要なのは、Enter (Arg 1) 命令の前に PushMarker 1 を置くことだけです。次のようになります。

```
Push (Code [PushMarker 1, Enter (Arg 1)])
```

つまり、共有クロージャを Enter する直前に、評価が完了したときに共有クロージャが更新されるようにする更新メカニズムをセットアップします。

アドレッシング モード (`Code [PushMarker n, Enter (Arg n)]`) は、フレームの n 番目のクロージャへの**間接更新**と呼ばれます。これは、更新を実行させる間接であるためです。引数を (スタックにプッシュするのではなく) Enter する場合にも、まったく同じ考慮事項が適用されます。引数自体ではなく、引数への間接更新を Enter する必要があります。`Enter (Arg 1)` は `Enter (Code [PushMarker 1, Enter (Arg 1)])` に置き換える必要があります。

コンパイルスキームの変更は簡単です。$\mathcal{SC}$ スキームと $\mathcal{R}$ スキームのみが影響を受けますが、両方とも同じように影響を受けます。つまり、環境を構築する場合、各変数を更新間接アドレス指定モードにバインドする必要があります。たとえば、$\mathcal{R}$ スキームの `let(rec)` の場合、ρ' に次の定義を使用します (図 4.3 を参照)。

$$\begin{array}{rcl} \rho' & = & \rho[x_1 \mapsto \mathcal{J}[\![d+1]\!],\ \ldots,\ x_n \mapsto \mathcal{J}[\![d+n]\!]] \\ & & \text{ここで } \mathcal{J}[\![d]\!] = \texttt{Code}\ [\texttt{PushMarker}\ d,\ \texttt{Enter}\ (\texttt{Arg}\ d)] \end{array}$$

4.5.3　更新メカニズムの実装

新しい更新メカニズムを実装するには、次の変更を加える必要があります。

- ダンプの型定義を指定します。これは単なるトリプルのスタックであり、リストとして表され、空に初期化されます。

```
type TimDump = [(FramePtr, -- The frame to be updated
                 Int,      -- Index of slot to be updated
                 TimStack) -- Old stack
                ]
initialDump = []
```

- `PushMarker` 命令を `instruction` 型に追加し、命令を表示するために適切な変更を加えます。
- `step` に `PushMarker` 命令用の定義を追加し、`Return` 命令用の定義を変更します。
- 各変数を更新間接アドレッシング モードにバインドする環境を構築するには、`compileSC` と、`compileR` の `ELet` 命令用定義を変更します。関数 `mkUpdIndMode` を使用して $\mathcal{J}$ スキームを実装します。

```
mkUpdIndMode :: Int -> TimAMode
mkUpdIndMode n = Code [PushMarker n, Enter (Arg n)]
```

演習 4.16 説明に従って更新メカニズムを実装します。いくつかのテストプログラムで新しいシステムを実行すると、PushMarker が更新情報をダンプに追加し、Return が更新を実行するのを確認できるはずです。考えられるテストプログラムの 1 つを次に示します。

```
f x = x + x
main = f (1+2)
```

(1+2) の評価は 1 回だけ行う必要があります。

演習 4.17 ここでは、実行できる簡単な最適化を示します。次のような関数の場合:

```
compose f g x = f (g x)
```

compose のコードは次のようになっていることがわかります。

```
compose: Take 3 3
         Push (Code [...])
         Enter (Code [PushMarker 1, Enter (Arg 1)])
```

ここで、[...] は (g x) のコードです。最後の命令は、f の間接更新に入ります。しかし、

Enter (Code i) が i と同等であること

は事実です。(これはルール 4.8 から直ちに導かれます。) したがって、compose の同等のコードは次のようになります。

```
compose: Take 3 3
         Push (Code [...])
         PushMarker 1
         Enter (Arg 1)
```

この最適化を実装します。これを行う最も良い方法は、コンパイラ内の [Enter *e*] 形式のすべての式を (mkEnter *e*) に置き換えることです。ここで、mkEnter は次のように定義されます。

```
mkEnter :: TimAMode -> [Instruction]
mkEnter (Code i) = i
mkEnter other_am = [Enter other_am]
```

mkEnter は Enter コンストラクタの「アクティブ」形式で、Enter 命令を生成する前に特殊なケースをチェックします。

このスキームには他にも多くの改善点があり、それについては次のセクションで検討します。

4.5.4　間接更新の問題

この更新メカニズムは非常に単純ですが、非常に非効率的です。2 つの主な問題があり、それらは Argo [Argo 1989] によって初めて識別されました。最初の問題は、**同一の更新**の問題です。前のセクションで示したプログラムを考えてみましょう。

```
f x = x+x
main = f (1+2)
```

x を使用するたびに、f はその引数 x への更新クロージャを Enter します。最初、x はその値で更新されます。2 回目では、2 回目の (完全に冗長な) 更新が行われ、x がその値で再度上書きされます。実装でサンプルを実行すると、これが起こるのを確認できるはずです。

この場合、もちろん、賢いコンパイラは x が確実に評価されることを認識し、x に間接を Enter する代わりに x をコピーするだけです。しかし、これはコンパイラを複雑にし、一般的には検出することが不可能になる可能性があります。たとえば、 f が次のように定義されたとします。

```
f x = g x x
```

f が g を分析して、さまざまな x がどの順序で評価されるかを発見しない限り (そして一般に、この質問に対する答えは 1 つだけではない可能性があります)、悲観的に考えて、間接参照の更新を引数として g にプッシュする必要があります。

2 番目の問題は、**間接チェーン**の問題です。次のプログラムを検討してみましょう。

```
g x = x+x
f y = g y
main = f (1+2)
```

ここで、f は引数 y への間接更新演算を g に渡します。ただし、 g は引数 x への間接更新を Enter します。したがって、 g は間接への間接を Enter します。つまり、**引数が別の関数の引数として渡されるたびに間接参照が追加される**ため、間接参照の連鎖が構築されます。次の末尾再帰関数で m への間接演算がどれだけ多くなるかを想像してみてください。

```
horrid n m = if (n==0) m (horrid (n-1) m)
```

これらの問題はまだ解決しません。代わりに、次のセクションでは、let(rec) バインドされた変数の更新を処理する方法と、これら 2 つの問題が発生しない理由を示します。これは、スーパーコンビネータの引数についても、より良い解決策への道を示しています。

4.5.5 let(rec) にバインドされた共有変数の更新

これまで、 let(rec) でバインドされた変数は、常に間接更新アドレス指定モードを使用することにより、スーパーコンビネータの引数とまったく同じ方法で更新されると仮定してきました。たとえば、次のスーパーコンビネータ定義を考えてみましょう。

```
f x = let y = ...
      in
      g y y
```

ここで、「...」は y の任意の右辺を表します。y を x と同じように扱って、f に対して次のコードを生成します。

```
f:    Take 2 1                                    -- Frame with room for y
      Move 2 (Code [...code for y...])            -- Closure for y
      Push (Code [PushMarker 2, Enter (Arg 2)]) -- Indirection to y
      Push (Code [PushMarker 2, Enter (Arg 2)]) -- Indirection to y
      Enter (Label "g")
```

ここで、「...code for y...」は、y の右辺から生成されたコードを表します。このコードには、前に概説した同一更新の問題があります。

しかし、はるかに優れたソリューションがすぐに利用可能です。代わりに f に対して次のコードを生成するとします。

```
f:    Take 2 1 -- Frame with room for y
      Move 2 (Code (PushMarker 2 :
                    [...code for y...])) -- Closure for y
      Push (Code [Enter (Arg 2)]) -- Non-updating indirection to y
      Push (Code [Enter (Arg 2)]) -- Indirection to y
      Enter (Label "g")
```

PushMarker 命令は、y の**使用**からその**定義**に移行しました。Move 命令によって構築された y のクロージャは、**自己更新クロージャ**になりました。つまり、Enter されると、それ自体を更新する更新メカニズムがセットアップされます。一度この状態が発生すると、PushMarker 命令を含むコードへのポインタが上書きされるため、二度と同じ問題が発生することはありません。

一般に、考え方は次のとおりです。

- コードを PushMarker 命令で始めることにより、let(rec) 束縛の右辺に自己更新クロージャを使用します。
- let(rec) で束縛された変数をスタックにプッシュする場合は、通常の (更新しない) 間接アドレッシングモードを使用します。まだ、クロージャのコピーを取得するのではなく、間接アドレッシングを使用する必要があります。これは、クロージャが更新されるまで、コピーすると重複した作業が発生するため、

このアイデアを実装するには次の変更が必要です。

- let(rec) でバインドされた変数に対して**非更新**間接参照を生成するように $\mathcal{R}$ スキームを変更します。つまり、$\mathcal{J}$ ではなく $\mathcal{I}$ スキームを使用して新しい環境を構築します。(現時点では、$\mathcal{SC}$ は スーパーコンビネータ引数に対して $\mathcal{J}$ を使用して**更新**間接を生成し続ける必要があります。)
- let(rec) 式の $\mathcal{R}$ スキームを変更して、すべての右辺のコードの先頭に PushMarker 命令を生成します。これは、新しいコンパイルスキームである $\mathcal{U}$ スキーム (図 4.4 を参照) を作成することで最も簡単に実行できます。これは、$\mathcal{R}$ スキームの let(rec) の場合に定義の右辺をコンパイルするために使用されます。$\mathcal{U}$ には、現在のフレームのどのスロットを更新する必要があるかを示す追加の引数が必要であり、この引数を使用して適切な PushMarker 命令を生成します。図 4.4 には、$\mathcal{R}$ スキームの修正された let 定義式も示されています。letrec の場合の変更もまったく同様です。

$\mathcal{U}[\![e]\!]\ u\ \rho\ d$ はペア $(d',\ am)$ です。
ここで、am は環境 ρ における式 e の TIM アドレッシングモードです。
am によってアドレス指定されたクロージャが Enter されると、
現在のフレームのスロット u が、その正規形 (※評価結果) で更新されます。
このコードは、フレームの最初の d スロットが占有されていると想定しており、
スロット $(d+1,\ \ldots,\ d')$ を使用します。

$$\begin{array}{lcl}\mathcal{U}[\![e]\!]\ u\ \rho\ d & = & (d', \texttt{Code}\ (\texttt{PushMarker}\ u\ :\ is)) \\ & & \text{ここで} \\ & & (d',\ is) = \mathcal{R}[\![e]\!]\ \rho\ d \\ & & \text{です。}\end{array}$$

$$\begin{array}{l}\mathcal{R}[\![\texttt{let}\ x_1\ \texttt{=}\ e_1\texttt{;}\ \ldots\texttt{;}\ x_n\ \texttt{=}\ e_n\ \texttt{in}\ e]\!]\ \rho\ d \\ \quad = \quad (d', [\texttt{Move}\ (d+1)\ am_1,\ \ldots,\ \texttt{Move}\ (d+n)\ am_n]\ \ \texttt{++}\ is) \\ \qquad \text{ここで} \\ \qquad (d_1,\ am_1) = \mathcal{U}[\![e_1]\!]\ (d+1)\ \rho\ (d+n) \\ \qquad (d_2,\ am_2) = \mathcal{U}[\![e_2]\!]\ (d+2)\ \rho\ d_1 \\ \qquad \ldots \\ \qquad (d_n,\ am_n) = \mathcal{U}[\![e_n]\!]\ (d+n)\ \rho\ d_{n-1} \\ \qquad \rho' = \rho\,[x_1\ \mapsto\ \mathcal{I}[\![(d+1)]\!],\ \ldots,\ x_n\ \mapsto\ \mathcal{I}[\![(d+n)]\!]] \\ \qquad \mathcal{I}[\![d]\!] = \texttt{Code}\ [\texttt{Enter}\ (\texttt{Arg}\ d)] \\ \qquad (d',\ is) = \mathcal{R}[\![e]\!]\ \rho'\ d_n \\ \qquad \text{です。}\end{array}$$

letrec の場合も同様ですが、ρ' が ρ の代わりに $\mathcal{U}[\![]\!]$ への呼び出しに渡される点が異なります。

演習 4.18 このアイデアを試して、どれだけのステップが節約されるかという点でその効果を測定してください。

演習 4.19 次の表現を考えてみる

```
let x = 3 in x+x
```

この場合、let 式の右辺はすでに正規形式になっているため、$\mathcal{U}$ スキームで PushMarker 命令を生成する意味はありません。むしろ、この場合、$\mathcal{U}$ は単純に IntConst アドレッシングモードを返すことができます。

$\mathcal{U}$ コンパイルスキームと対応する compileU 関数を変更し、変更が正しく動作することを確認します。

4.5.6　間接チェーンの排除

前のセクションの考え方は、let(rec) バインドされた変数の同一の更新を排除する方法を示しています。このセクションでは、このアイデアを拡張してスーパーコンビネータ引数の同一の更新を排除し、同時に間接チェーンの問題を根絶する方法を示します。このアイデアは Argo によって最初に提案されました [Argo 1989]。

間接チェーンから始めます。前に述べたように、スーパーコンビネータは引数クロージャをコピーしてはならないと想定する必要があるため、間接チェーンが構築されます。そのため、それらを複数回使用する場合は、間接を使用することをお勧めします。多くの場合、引数クロージャーはすでに間接的であり、それをコピーしても完全に安全であるため、これにより間接チェーンが発生します。

これは、別の戦略を示唆しています。

> すべての引数クロージャは、共有を失うことなく自由にコピー可能でなければならないという規則を採用します。

この呼び出し規約は呼び出される関数にとって明らかに便利ですが、呼び出し元はどのようにしてそれが満たされていることを確認できるのでしょうか? 引数は次のいずれかです。

- 定数。そのクロージャは自由にコピー可能です。
- スーパーコンビネータ。同じ特性があります。
- let(rec) でバインドされた変数。これも自由にコピーできます (前のセクションのアイデアを使用)。
- 現在のスーパーコンビネータへの引数。新しい規約により自由にコピーできます。
- 非アトミックな式。そのクロージャーは (現状では) 自由にコピーできません。

したがって、この新しい呼び出し規約を採用するために必要なのは、非アトミックな引数を自由にコピー可能なクロージャとして渡す何らかの方法を見つけることだけです。たとえば、次の式を考えてみましょう。

```
f (factorial 20)
```

引数 (factorial 20) を自由にコピー可能なクロージャとして渡すにはどうすればよいですか。解決策は簡単です。式を次の同等の形式に変換します。

```
let arg = factorial 20 in f arg
```

let 式は現在のフレームにクロージャ (factorial 20) 用のスロットを割り当て、その中に自己更新クロージャを置き、このクロージャへの (通常の) 間接指定が f に渡されます。(この変換は、 f に渡されるクロージャーが複数回 Enter される可能性がある場合にのみ実行する必要があることに注意してください。ここには、より効率的なコードを生成するためにグローバル共有分析を行う機会があります。)

この変換が完了すると、引数クロージャを自由にコピーできますが、let(rec) 束縛されたクロージャには (更新されない) 間接参照を使用する必要があります。間接チェーンが構築されたり、同一の更新が行われることはありません。

何が起こったのかを振り返るのは興味深いことです。スーパーコンビネータ呼び出しの割り当て全体がその引数を保持するために必要なフレームであったため、最初は TIM が G マシンよりもはるかに少ないヒープを割り当てるかのように見えました。ただし、新しい更新手法を使用すると、スーパーコンビネータ本体内のすべての部分式には、それを保持するためにフレーム内のスロットが必要であることがわかります。同様に、スーパーコンビネータの引数のほとんどが間接引数になっているため、TIM は、引数自体ではなく引数へのポインタを渡す G マシンとまったく同じように動作します。したがって、遅延更新の問題により、TIM はより G マシンらしくなる必要がありました。

この手法を、引数式ごとに let 式を導入するプログラム変換として紹介しましたが、これを行うには、新しい任意の変数名を考え出す必要があるため、少々面倒です。新しいコンパイルスキームをより直接的に記述する方が簡単です。主な変更は、図 4.5 に示す $\mathcal{R}$ スキームの関数適用ケースに対するものです。

$\mathcal{R}[\![e]\!]\ \rho$ は、環境 ρ の式 e の値をスタック上の引数に適用する TIM コードです。

$\mathcal{R}[\![e\ a]\!]\ \rho\ d$
$= \quad (d_1, \texttt{Push}\ (\mathcal{A}[\![a]\!]\ \rho) : is)$
ここで、a はスーパーコンビネータ、局所変数、
または整数であり、$d_1,\ is$ は以下の通りです。
$(d_1, is) = \mathcal{R}[\![e]\!]\ \rho\ d$

$\mathcal{R}[\![e_{fun}\ e_{arg}]\!]\ \rho\ d$
$= \quad (d_2, \texttt{Move}\ (\mathrm{d}+1)\ am_{arg} : \texttt{Push}\ (\texttt{Code}\ [\texttt{Enter}\ (\texttt{Arg}\ (\mathrm{d}+1))]) : is_{fun})$
ここで、$d_1,\ d_2,\ am_{arg},\ is_{fum}$ は以下の通りです。
$(d_1, am_{arg}) = \mathcal{U}[\![e_{arg}]\!]\ (d+1)\ \rho\ (d+1)$
$(d_2, is_{fun}) = \mathcal{R}[\![e_{fun}]\!]\ \rho\ d_1$

$\mathcal{A}[\![n]\!]\ \rho$	$=$	$\texttt{IntConst}\ n$	ここで、n は整数定数です。
$\mathcal{A}[\![x]\!]\ \rho$	$=$	$\rho\ x$	ここで、x は ρ によって束縛されます。

最初の定義式は、関数適用への引数がアトミック式 (変数または定数) である場合を扱い、以前と同様に $\mathcal{A}$ スキームを使用して適切なアドレッシングモードを生成します。2 番目の式は、引数が複合式である場合を扱います。 フレーム内の次の空きスロットを引数式の自己更新クロージャで初期化し、このクロージャへの間接参照をプッシュします。

図 4.5 にも示されている $\mathcal{A}$ スキームは、引数としてアトミック式 (変数または定数) を使用してのみ呼び出されるため、以前よりもケースが 1 つ減りました。同じ理由で、フレームスロットをまったく使用しないため、d を引数として受け取って結果として返す必要がなくなりました。

演習 4.20 この改訂されたスキームを実装し、以前のバージョンとのパフォーマンスの違いを測定します。

4.5.7　部分的な関数適用の更新

これまでのところ、値が整数であるクロージャの更新を正常に処理できました。Return 命令が空のスタックを見つけると、更新を実行し、ダンプから新しいスタックをポップします。

しかし、スタックから項目を消費する別の命令、つまり Take があります。Take 命令でスタック上に見つかったアイテムの数が必要な数よりも少ない場合はどうなるのでしょうか? たとえば、次のプログラムを考えてみましょう。

```
add a b = a+b
```

```
twice f x = f (f x)
g x = add (x*x)
main = twice (g 3) 4
```

twice が f を enter すると、間接経由で Enter され、f の更新が設定されます。この例では、f は (g 3) にバインドされ、1 つの引数への加算の部分適用として評価されます。add のコードの先頭にある Take 2 命令は、スタック上に引数が 1 つだけあることを検出します。これは、更新が行われ、(g x) のクロージャが (add (x*x)) のクロージャで上書きされることを示します。

一般的に：

> Take 命令がスタック上で見つけた引数が少なすぎる場合は、ダンプ上の先頭の項目によって識別されるクロージャの更新を実行し、現在のスタック上の項目をダンプから回復されたスタックの先頭に貼り付けて、 (別の更新が必要な場合) Take 命令を再試行する必要があります。

Take 命令はすでに十分に複雑ですが、さらに実行するタスクがあります。Take が扱いにくくなるのを避けるために、Take を 2 つの命令に分割しました。1 つは十分な引数があるかどうかのチェックを実行する UpdateMarkers で、もう 1 つは新しいフレームを実際に構築する Take です。UpdateMarkers n 命令は常に、すべての Take t n 命令の直前に配置されます。

したがって、Take の遷移規則は変更されません。UpdateMarkers の遷移規則は次のとおりです。

(4.17)

	UpdateMarkers n : i	f	$c_1 : \ldots : c_m$	s	v		d	h	c
$\Longrightarrow$		i	f	$c_1 : \ldots : c_m$	s	v		d	h	c
	ここで $m \geq n$									
	UpdateMarkers n : i	f	$c_1 : \ldots : c_m$	: []	v	(f_u, x, s)	: d	h	c
$\Longrightarrow$	UpdateMarkers n : i	f	$c_1 : \ldots : c_m$	s	v		d	h'	c
	ここで $m < n$ かつ $h' = h[f_u : \langle \ldots, d_{x-1}, (i', f'), d_{x+1}, \ldots \rangle]$									

最初の規則は、十分な引数があるため、UpdateMarkers が何も行わない場合を扱います。2 番目は、更新を行う必要があるもう 1 つのケースを扱います。適切なクロージャが更新され、現在のスタックが古いスタックの上に貼り付けられ、UpdateMarkers が再試行されます。

このルールでは、i' と f' はターゲットクロージャを上書きするコードポインタとフレームポインタですが、これまでのところ、それらがどのような

値を取るべきかについては指定されていません。それらがどうあるべきかを理解する方法は、「クロージャ (i', f') が Enter されたときに何が起こるべきか?」という質問をすることです。このクロージャは、スーパーコンビネータの引数 $c_1, \ldots, c_m$ への部分的な適用を表します。したがって、これが Enter されると、$c_1, ..., c_m$ をプッシュしてから、スーパーコンビネータのコードにジャンプする必要があります。したがって、

- f' は新しく割り当てられたフレーム $\langle c_1, \ldots, c_m \rangle$ を指している必要があります。
- i' は以下の符号列でなければなりません

  `Push (Arg` m`) : ... : Push (Arg 1) : UpdateMarkers` n `:` i

  ここで、Push 命令は部分適用の引数をスタックに配置し、UpdateMarkers 命令は実行する必要があるさらなる更新をチェックします。i はスーパーコンビネータのコードの残りの部分です。

演習 4.21 UpdateMarkers 命令を実装し、各 Take 命令の前に 1 つを配置するようにコンパイラを変更します。次のプログラムの変更の前後で実装をテストします。プログラムは高階関数を使用してペアを実装します (セクション 2.8.3)。ペア w は共有され、ペア関数の部分的な適用として評価されます。

```
pair x y f = f x y
fst p = p K
snd p = p K1
main = let w = pair 2 3
       in (fst w) * (snd w)
```

w が (pair 2 3) の部分適用で更新されていることがわかります。もう少し説得力を持たせるために、w の右辺にもう少し計算を含めることができます。たとえば、

```
main = let w = if (2*3 > 4) (pair 2 3) (pair 3 2)
       in (fst w) * (snd w)
```

演習 4.22 Take 0 0 が何もしないのと同様に、UpdateMarkers 0 も何も行いません。必要に応じてこれらの命令の両方を省略するように、compileSC を変更します。(これは演習 4.3 の単純な拡張です。)

他にも注目すべき点がいくつかあります。

- 実際の実装では、ルールにあるように、更新が行われるたびにコード i' が新しく製造されることはありません。代わりに、スーパーコンビネータ i のコードの前に一連の `Push` 命令を配置し、部分適用のコードポインタはシーケンス内の適切な場所を指すだけで済みます。

- `UpdateMarkers` ルールはクロージャ $c_1, \ldots, c_m$ を複製します。これは、スーパーコンビネータの引数が自由にコピーできるようになったことであり、セクション 4.5.6 で導入した修正です。この変更が行われる前は、そのようなコピーを作成すると redex が複製される危険があったため、代わりに `UpdateMarkers` ルールが間接化されてさらに複雑になっていたでしょう。`UpdateMarkers` の導入が遅れて放置されているのはこのためです。

- 式 (`f` e_1 e_2) のコードをコンパイルしているとします。ここで、`f` は 2 つ (またはそれ以下) の引数のスーパーコンビネーターであることがわかっています。この場合、スタックはそれを満たすのに十分な深さがあることが確実であるため、 `f` の先頭にある `UpdateMarkers` 命令は確かに何も行いません。そのため、すべての引数 (またはそれ以上) に適用されるスーパーコンビネータへの呼び出しをコンパイルするときに、`UpdateMarkers` 命令の後にそのコードを Enter できます。一般的なプログラムの関数適用の多くは、既知のスーパーコンビネータの充足した関数適用であるため、この最適化は頻繁に適用できます。

4.6　Mark 5: 構造化データ

このセクションでは、代数データ型を TIM に追加する方法を学習します。セクション 2.8.3 で説明されている高階関数を使用すると、このセクションの内容をまったく使用せずにデータ構造を実装することができます。しかし、そうするのはむしろ非効率的です。代わりに、より一般的なデータ構造を処理できるように、算術演算に使用したアプローチを開発します。

4.6.1　一般的なアプローチ

is_empty 関数について考えてみましょう。この関数は、引数が空のリストの場合は 1 を返し、そうでない場合は 0 を返します。これは、それをシングルトン リストに適用するプログラムのコンテキストで与えられます。

```
is_empty xs = case xs of
                <1>      -> 1
                <2> y ys -> 0

cons a b = Pack{2,2} a b
nil      = Pack{1,0}

main = is_empty (cons 1 nil)
```

セクション 1.1.4 でコンストラクターが Pack$\{tag,\ arity\}$ で表されることを思い出してください。リストを操作するこのプログラムでは、空リストコンストラクタ nil にはタグ 1 とアリティ 0 があり、リストコンストラクタ cons にはタグ 2 とアリティ 2 があります。パターンマッチングは case 式によってのみ実行されます。ネストされたパターンは、ネストされた case 式によって照合されます。

まず、case 式に対してどのようなコードを生成するかを検討します。算術演算子では引数を評価する必要があるのと同様に、case 式では式 (is_empty の例では xs) を評価する必要があります。この後、返されたオブジェクトのタグに応じて、多方向ジャンプを実行できます。算術演算子に使用したものと同様のアプローチを採用すると、次の規則が示唆されます。

- データオブジェクトを表すクロージャを評価するには、継続が引数スタックにプッシュされ、クロージャが Enter されます。
- (先頭) 正規形に評価されると、この継続がスタックからポップされて Enter されます。

- データオブジェクトのタグは値スタックの一番上に返されます。
- データオブジェクトのコンポーネント (存在する場合) は、新しいレジスタであるデータフレームポインタが指すフレームで返されます。

したがって、is_empty に対して生成するコードは次のようになります。[5]

```
is_empty: Take 1 1            -- One argument
          Push (Label "cont") -- Continuation
          Enter (Arg 1)       -- Evaluate xs

cont:     Switch [ 1 -> [PushV (IntVConst 1), Return]
                   2 -> [PushV (IntVConst 0), Return]
                 ]
```

Switch 命令は、値スタックの最上位項目に基づいて多方向ジャンプを実行します。この例では、case 式の両方の分岐が定数を返すだけです。

この例では、精査されたリストセルのコンポーネントは使用されませんでした。これは常に当てはまるわけではありません。たとえば、sum 関数を考えてみましょう。

```
sum xs = case xs of
               <1>      -> 0
               <2> y ys -> y + sum ys
```

sum はリストの要素の合計を計算します。新しい機能は、式 y + sum ys がリストセルのコンポーネント y と ys を使用することです。前に示したように、これらのコンポーネントは、データフレームポインタが指すフレーム内の継続、つまり新しいレジスタに返されます。(演習: なぜ通常のフレームポインタをこの目的に使用できないのですか?)

これまでのところ、すべてのローカル変数 (つまり、スーパーコンビネータ引数または let(rec) 束縛変数) には、そのクロージャを含む現在のフレーム内にスロットがあるため、このアイデアを拡張して、y と ys にさらにスロットを追加するのが論理的であると思われます。必要なのは、クロージャをリストセルフレームから現在のフレームに移動することだけです。sum のコードは次のとおりです。

[5] いつものように、継続の明示的なラベルを使用してコードを作成しますが、実際には、新しいラベルの生成を避けるために、Code アドレッシングモードを使用するようにコンパイルします。

```
sum: Take 3 1            -- One argument, two extra slots for y,ys
     Push (Label "cont") -- Continuation for case
     Enter (Arg 1)       -- Evalute xs

cont: Switch [1 -> [PushV (IntVConst 0), Return]
              2 -> [Move 2 (Data 1)
                    Move 3 (Data 2)
                    ...code to compute y + sum ys...
                   ]
             ]
```

Move 命令は、データフレームポインタが指すフレーム内のクロージャをアドレス指定する新しいアドレス指定モード Data を使用します。2 つの Move 命令は、y と ys をリストセルから現在のフレーム (xs を含むフレーム) にコピーします。

要約すると、case 式は 5 つのステップにコンパイルされます。

1. 継続を Push します。
2. 精査するクロージャを Enter します。評価されると、ステップ 1 で Push した継続に入ります。
3. 継続では、Switch 命令を使用して、値スタックの一番上に返されるタグに基づいて多方向ジャンプを実行します。
4. Switch の各ブランチは、データオブジェクトの内容を現在のフレームにコピーする Move 命令で始まります。これはクロージャをコピーするので、データオブジェクト内のすべてのクロージャが自由にコピーできるプロパティを持っていることを確認する必要があります (セクション 4.5.6)。
5. 各ブランチは、通常どおりにコンパイルされたそのブランチのコードを続行します。

最後に、式 Pack{ *tag* , *arity* } に対してどのようなコードを生成する必要があるかを尋ねることができます。たとえば、次の式を考えてみましょう。

$$\texttt{Pack\{1,2\}}\ e1\ e2$$

これによりリストセルが構築されます。最小限のアプローチは、Pack{1,2} をスーパーコンビネータとして扱うことです。そして、次のコードを生成します。[6]

[6]原理的には、使用可能なコンストラクタは無限にあるため、コードストアには、それらに対応する類似のコードフラグメントの有限ファミリーが必要と思われます。実際には、詳細なコンパイルスキームを記述するときにわかるように、これは簡単に回避できます。

```
Push (...addressing mode for e2...)
Push (...addressing mode for e1...)
Enter (Label "Pack{1,2}")
```

`Pack{1,2}` のコードは非常に単純です。

```
Pack{1,2}: UpdateMarkers 2
           Take 2 2
           ReturnConstr 1
```

最初の 2 つの命令は、他のスーパーコンビネータとまったく同じです。`UpdateMarkers` 命令は必要な更新を実行し、`Take` 命令はリストセルの 2 つのコンポーネントを含むフレームを構築し、そのフレームへのポインタを現在のフレームポインタに置きます。最後に、新しい命令 `ReturnConstr` が継続に入ります。タグ 1 を値スタックにプッシュし、現在のフレームポインタをデータフレームポインタにコピーします。`Return` と同様に、`ReturnConstr` は更新を確認し、必要に応じて実行する必要があります。

4.6.2 データ構造の遷移規則とコンパイル方式

概要が完成したので、新しい構成の遷移規則とコンパイルスキームの詳細を説明します。`Switch` の遷移規則は以下の通りです。

(4.18)

$$\begin{array}{lllllllll} & [\texttt{Switch}\ [\ldots\ t\ \texttt{->}\ i\ \ldots]] & f & f_d & s & t : v & d & h & c \\ \Longrightarrow & & i & f & f_d & s & v & d & h & c \end{array}$$

`ReturnConstr` には更新が必要になる可能性を考慮する必要があるため、2 つの遷移規則があります。最初の方法は簡単で、更新を行う必要がない場合です。

(4.19)

$$\begin{array}{lllllllll} & [\texttt{ReturnConstr}\ t] & f & f_d & (i,\ f') : s & & v & d & h & c \\ \Longrightarrow & & i & f' & f & & s & t : v & d & h & c \end{array}$$

2 番目のルールは更新を扱い、`ReturnConstr` 命令とデータフレームポインタのみを含むコードシーケンスで更新されるクロージャを上書きします。

(4.20)

$$\begin{array}{llllllllll} & [\texttt{ReturnConstr}\ t] & f & f_d & [] & v & (f_u,\ x,\ s) : d & h & c & \\ \Longrightarrow & [\texttt{ReturnConstr}\ t] & f & f_d & s & v & & d & h' & c \\ & \text{ここで } h' = h[f_u : \langle \ldots,\ d_{x-1},\ ([\texttt{ReturnConstr}\ t],\ f),\ d_{x+1},\ \ldots \rangle] & & & & & & & & \end{array}$$

コンパイルスキームに対する唯一の変更は、コンストラクタと case 式の $\mathcal{R}$ スキームに特別なケースを追加することです。後者は、case の選択肢をコンパイルする補助スキーム $\mathcal{E}$ を使用して構造化されています (図 4.6)。コンストラクタは、検出されると「インライン」でコンパイルされるため、無限の定義ファミリーをコードストアに追加する必要がなくなります。

$\mathcal{R}[\![\texttt{Pack\{}t,\ a\texttt{\}}]\!]\ \rho\ d \quad = \quad (d, [\texttt{UpdateMarkers}\ a, \texttt{Take}\ a\ a, \texttt{ReturnConstr}\ t])$

$\mathcal{R}[\![\texttt{case}\ e\ \texttt{of}\ alt_1\ \ldots\ alt_n]\!]\ \rho\ d$

$= \quad (d', \texttt{Push}\ (\texttt{Code}\ [\texttt{Switch}\ [branch_1\ \ldots\ branch_n]])\ \texttt{:}\ is_e)$

ここで、$d_1,\ \ldots,\ d_n,\ d',\ is_e$ は以下の通りです。

$(d_1,\ branch_1) = \mathcal{E}[\![alt_1]\!]\ \rho\ d)$

$\ldots$

$(d_n,\ branch_n) = \mathcal{E}[\![alt_n]\!]\ \rho\ d)$

$(d',\ is_e) = \mathcal{R}[\![e]\!]\ \rho\ max(d_1,\ \ldots,\ d_n)$

$\mathcal{E}[\![alt]\!]\ \rho\ d$ (alt は case の選択肢) はペア $(d',\ branch)$ です。
ここで $branch$ は、環境 ρ でコンパイルされた Switch の選択肢です。
このコードは、フレームの最初の d スロットが占有されていると仮定し、
スロット $(d+1,\ \ldots,\ d')$ を使用します。

$\mathcal{E}[\![\texttt{<}t\texttt{>}\ x_1\ \ldots\ x_n\ \texttt{->}\ body]\!]\ \rho\ d$

$= \quad (d', t\ \texttt{->}\ (is_{moves}\ \texttt{++}\ is_{body}))$

ここで、$d',\ is_{moves},\ is_{body}$ は以下の通りです。

$is_{moves} = [\texttt{Move}\ (\text{d} + 1)\ (\texttt{Data}\ 1), \ldots, \texttt{Move}\ (\text{d} + \text{n})\ (\texttt{Data}\ \text{n})]$

$(d',\ is_{body}) = \mathcal{R}[\![body]\!]\ \rho'\ (d+n)$

$\rho' = [x_1 \mapsto \texttt{Arg}\ (d+1),\ \ldots,\ x_n \mapsto \texttt{Arg}\ (d+n)]$

4.6.3　試してみる

次の追加のコア言語定義を使用することで、新しい機構を使用してリストとブール値を実装できます。

```
cons = Pack{2,2}
nil  = Pack{1,0}

true  = Pack{2,0}
false = Pack{1,0}
if cond tbranch fbranch = case cond of
                              <1> -> fbranch
                              <2> -> tbranch
```

`if` は、以前はコンパイルスキームに特別な命令とケースを持っていましたが、現在は他の定義と同様に単なるスーパーコンビネータ定義であることに注意してください。それでも、多くの場合、`case` よりも `if` を使用してプログラムを作成する方が明確なので、コンパイラに特別なケースを残しておきたい場合があります。ただし、今度は `Cond` 命令ではなく `Switch` 命令を生成します。(`Cond` は消える可能性があります。)

演習 4.23 新しい命令とコンパイル スキームを実装します。次のプログラムで新しい実装をテストします。

```
length xs = case xs of
                <1> -> 0
                <2> p ps -> 1 + length ps

main = length (cons 1 (cons 2 nil))
```

更新コードが正しく機能するかどうかを示す、より興味深い例は次のとおりです。

```
append xs ys = case xs of
                <1>      -> ys
                <2> p ps -> cons p (append ps ys)

main = let xs = append (cons 1 nil) (cons 2 nil)
       in
       length xs + length xs
```

ここでは `xs` が 2 回使用されていますが、`append` の作業は 1 回だけ行う必要があります。

演習 4.24 コンストラクタのアリティ a が 0 の場合、$\mathcal{R}[\![\texttt{Pack}\{t,\ a\}]\!]$ はコード `[UpdateMarkers 0, Take 0 0, ReturnConstr t]` を生成します。この場合に適したコードを生成するには、$\mathcal{R}$ スキームと `compileR` 関数を最適化します (演習 4.22 を参照)。

4.6.4 リストの表示

これまでに提案したサンプル プログラムはすべて整数を返しましたが、代わりにリストを返して出力できると便利です。

G マシンの章でこれを表現した方法は、出力の末尾に数字を追加する Print 命令とともに、出力を表す追加のコンポーネントをマシン状態に追加することでした。この場合、数値は値スタックで返されるため、Print は値スタックから数値を取り出し、それを出力に追加します。

現在、compile は継続（[], FrameNull）を使用してスタックを初期化します。これにより、この継続が Enter されるとマシンが停止します。必要なのは、この継続を変更して印刷を行うことだけです。今回は、継続ではプログラムの値がリストであることを期待しているため、処理方法を決定するためにケース分析を行う必要があります。リストが空の場合、マシンは停止する必要があるため、そのブランチには空のコードシーケンスだけを含めることができます。それ以外の場合は、リストの先頭が評価されて出力され、その後、末尾に元の継続が再度与えられる必要があります。コードは次のとおりです。

```
topCont:    Switch [ 1 -> []
                     2 -> [ Move 1 (Data 1)          -- Head
                            Move 2 (Data 2)          -- Tail
                            Push (Label "headCont")
                            Enter (Arg 1)            -- Evaluate head
                          ]
                   ]

headCont:   Print
            Push (Label "topCont")
            Enter (Arg 2)                            -- Do the same to tail
```

topCont コードには作業用ストレージとして 2 スロットフレームが必要であり、compile でそれを提供する方がよいことに注意してください。したがって、compile は次の継続でスタックを初期化します。

```
(topCont, frame)
```

ここで、topCont は上記のコードシーケンス、frame はヒープから割り当てられた 2 スロットフレームのアドレスです。

演習 4.25 説明に従ってリストの表示を実装します。唯一面倒な点は、マシンの状態に追加のコンポーネントを (再度) 追加する必要があることです。

いつものように、Push (Label "headCont") の代わりに Push (Code ...) を使用できます。実際、少し再帰を使用することで、Push (Label "topCont") に対しても同じことができます。

次のプログラムで作業をテストします。

```
between n m = if (n>m) nil (cons n (between (n+1) m))
main = between 1 4
```

演習 4.26 結果がリストであるプログラムを実行する場合、リストの要素が使用可能になったらすぐに出力できると便利です。現在の実装では、すべての状態を出力するか (showFullResults を使用する場合)、最後の状態のみを出力します (showResults を使用する場合)。前者の場合は非常に多くの出力が得られますが、後者の場合はプログラムが終了するまでまったく出力が得られません。

showResults を変更して、生成された出力を表示するようにします。これを行う最も簡単な方法は、連続する状態のペアの出力コンポーネントを比較し、ある状態と次の状態の間で出力が長くなったときに最後の要素を出力することです。

showResults に対するもう 1 つの可能な変更は、各状態 (または 10 個の状態) のドットを出力して、各出力ステップ間でどれだけの作業が行われたかを大まかに示すことです。

4.6.5 データ構造を直接使用する †

Move 以外の命令でデータアドレッシングモードを使用して、Switch 命令の選択肢内でデータ構造のコンポーネントを直接使用できないのはなぜかと疑問に思う人もいるかもしれません。理由は sum の例でわかります。これをここで繰り返します。

```
sum xs = case xs of
               <1>      -> 0
               <2> y ys -> y + sum ys
```

ここで、y + sum ys のコードをもう少し詳しく見てみましょう。このコードは最初に y を評価する必要がありますが、これには大量の計算が必要となる可能性があり、必ずデータフレームポインタレジスタを使用します。したがって、ys を評価する時点までに、データフレームポインタは変更されているため、データフレームポインタを介して ys にアクセスできなくなります。リストセルの内容を現在のフレームに移動することにより、それらをさらに評価しても保存できるようになります。

head 関数のように、それ以上の評価が行われない場合があります。

```
head xs = case xs of
```

```
<1>       -> error
<2> y ys -> y
```

この場合、最適化として、データフレームから y を直接使用できます。つまり、Switch 命令の 2 番目の分岐は単純に [Enter (Data 1)] になります。

同様に、変数が case 式の分岐でまったく使用されていない場合、その変数を現在のフレームに移動する必要はありません。

4.7 Mark 6: Constant Applicative Forms (定作用形) とコードストア †

前に述べたように (セクション 4.2.3)、コードストアを名前とコードシーケンスの関連付けリストとして表すという決定は、CAF が更新されないことを意味します。代わりに、呼び出されるたびにコードが実行されるため、作業が重複する可能性があります。この余分な作業は避けたいのですが、TIM の解決策は以前の実装ほど簡単ではありません。

テンプレートインスタンス化マシンと G マシンの場合、解決策は各スーパーコンビネータを表すためにヒープ内にノードを割り当てることでした。CAF が呼び出されるとき、redex のルートはスーパーコンビネータノード自体であるため、ノードはリダクションの結果 (つまり、スーパーコンビネータ定義式の右辺のインスタンス) で更新されます。その後スーパーコンビネータを使用すると、元のスーパーコンビネータではなく、この更新されたノードが表示されます。問題は、TIM にはヒープノードがまったくないことです。ノードに対応するのはフレーム内のクロージャです。したがって、私たちがしなければならないことは、初期ヒープに、各スーパーコンビネータのクロージャを含む単一の巨大なフレーム、つまり**グローバルフレーム**を割り当てることです。

コードストアは、スーパーコンビネータ名をフレーム内のオフセットにマッピングする関連付けリスト g とともに、グローバルフレームのアドレス f_G によって表されます。`Label` アドレッシングモードは、この関連リストを使用してオフセットを見つけ、スーパーコンビネータのクロージャをグローバルフレームからフェッチします。`Push Label` の新しい遷移規則は、これらのアイデアを形式化します。

(4.21)

	`Push (Label` l`)` $: i$	f	s	$h[f_G : \langle (i_1, f_1), \ldots, (i_n, f_n) \rangle]$	$(f_G, g[l : k])$
$\Longrightarrow$	i	f	$(i_k, f_k) : s$	h	(f_G, g)

`Enter Label` の遷移規則は、`Push`/`Enter` 関係から直接に従います。セクション 4.5.5 の `let(rec)` バインド変数のコンテキストで説明したように、グローバルフレーム内の各クロージャは自己更新クロージャです。`let(rec)` でバインドされた変数と同様に、スーパーコンビネータをスタックにプッシュするときは、(非更新の) 間接指定を使用する必要があります (セクション 4.5.5)。

4.7.1　CAF の実装

CAF の適切な更新を Mark 4 または Mark 5 TIM に追加するには、次の手順を実行する必要があります。

- マシン状態のコード ストア コンポーネントには、グローバル フレームのアドレスと、スーパーコンビネータ名とフレーム オフセット間の関連付けが含まれるようになりました。

```
type CodeStore = (Addr, ASSOC Name Int)
```

  この変更を考慮するには、showSCDefns 関数を変更する必要があります。

- 関数 amToClosure は、上で説明したように、Label アドレッシング モードに対して異なるアクションを実行する必要があります。

- compile 関数で計算された初期環境 initial_env は、スーパーコンビネータごとに間接アドレス指定モードを生成するように変更する必要があります。

- 最後の変更にはほとんどの作業が含まれます。テンプレートインスタンス化マシンと G マシンの compile 関数で行ったのと同じように、初期ヒープを構築するには compile 関数を変更する必要があります。

これらの項目の最後の部分については、もう少し議論する必要があります。compile では、空のヒープから開始するのではなく、補助関数 allocateInitialHeap を使用して初期ヒープを構築する必要があります。allocateInitialHeap には、compile 関数から compiled_code が渡されます。これは、compiled_code の各要素のクロージャを含む単一の大きなフレームを割り当て、初期ヒープと codeStore を返します。

```
allocateInitialHeap :: [(Name, [Instruction])] -> (TimHeap, CodeStore)
allocateInitialHeap compiled_code
  = (heap, (global_frame_addr, offsets))
    where
      indexed_code = zip2 [1..] compiled_code
      offsets = [(name, offset) | (offset, (name, code)) <- indexed_code]
      closures = [(PushMarker offset : code, global_frame_addr) |
                          (offset, (name, code)) <- indexed_code]
      (heap, global_frame_addr) = fAlloc hInitial closures
```

allocateInitialHeap は次のように機能します。まず、各要素を 1 から始まるフレームオフセットと組み合わせることで、compiled_code リストに

インデックスが付けられます。このリストは個別に処理されて、スーパーコンビネータ名からアドレスへのマッピングである offsets、およびグローバルフレームに配置されるクロージャのリストである closures が生成されます。最後に、グローバルフレームが割り当てられ、結果のヒープがコードストアとともに返されます。

global_frame_addr はクロージャの構築に使用されることに注意してください。各スーパーコンビネータクロージャのフレームポインタはグローバルフレームポインタそのものであるため、PushMarker 命令はグローバルフレームを参照する更新フレームをプッシュします。

演習 4.27 showSCDefns、compileA、amToClosure、および compile に必要な変更を加えます。CAF の更新が実際に行われるかどうかをテストします。

演習 4.28 allocateInitialHeap 内に追加された PushMarker 命令は CAF にのみ必要であり、1 つ以上の引数を持つスーパーコンビネータにとっては時間の無駄です。CAF に対してのみ PushMarker 命令を植えるように allocateInitialHeap を変更します。(ヒント: 非 CAF は、そのコードが Take n 命令 ($n > 0$) で始まるという事実によって識別できます。) 改善を測定します。

演習 4.29 間接アドレス指定モードは CAF にのみ必要であり、非 CAF スーパーコンビネータには必要ありません。この事実を利用するには、initial_env の構造を変更します。

4.7.2 コードストアをより忠実にモデリングする

これまでの Label の扱いには少し奇妙な点があります。それは次のとおりです。スーパーコンビネータの名前は、コンパイル時に (Label アドレッシングモードにマッピングするため) 環境内で検索され、次に実行時に (グローバルフレームのオフセットにマッピングするために) 再度検索されます[7]。これは現実的ではありません。実際のコンパイラでは、名前が検索されます。ただし、実行前にハードマシン アドレスにリンクされるため、実行時の検索は行われません。

これをモデル化するには、Label コンストラクタを 1 つではなく 2 つの引数を取るように変更します。次のようになります。

[7]前のセクションで提案した変更を CAF に対して実装したことを前提としていますが、このセクションは CAF 以前のバージョンのマシンにも適用されます。

```
timAMode ::= Label name num
           | ... 従来通り...
```

name フィールドには以前と同様にスーパーコンビネータの名前が記録されますが、num はグローバルフレームで使用するオフセットを示します。テンプレートインスタンス化マシンの NSupercomb コンストラクタと同様に、name フィールドはドキュメントとデバッグの目的でのみ存在します。Push Label の遷移規則が改訂されていることからわかるように、コードストアコンポーネントは単にグローバルフレームのアドレスになりました。

(4.22)

$$\begin{array}{llllll} & \texttt{Push (Label } l\ k\texttt{)} : i & f & s & h[g : \langle (i_1, f_1), \ldots, (i_n, f_n) \rangle] & g \\ \Longrightarrow & i & f & (i_k,\, f_k) : s & h & g \end{array}$$

Enter の遷移規則は、Push/Enter 関係に従います。

演習 4.30 このアイデアを実装します。これをするには：

- 説明に従って timAMode 型を変更します。
- codeStore 型をフレームポインタのみで構成されるように変更します。
- マシンの正しい初期状態を生成するように compile 関数を変更します。特に、適切な Label アドレッシングモードを使用して、initial_env を生成する必要があります。
- これらの変化を考慮して show 関数を調整します。

4.8 まとめ

最終的な TIM コンパイルスキームを図 4.7 と 4.8 にまとめます。

$\mathcal{SC}[\![def]\!]\ \rho$ は、環境 ρ でコンパイルされたスーパーコンビネータ定義 def の TIM コードです。

$$\mathcal{SC}[\![f\ x_1\ \ldots\ x_n = e]\!]\ \rho \quad = \quad \texttt{UpdateMarkers}\ n\ \texttt{:}\ \texttt{Take}\ d'\ n\ \texttt{:}\ is$$

ここで

$(d',\ is) = \mathcal{R}[\![e]\!]\ \rho[x_1 \mapsto \texttt{Arg}\ 1,\ \ldots,\ x_n \mapsto \texttt{Arg}\ n]\ n$

です。

$\mathcal{R}[\![e]\!]\ \rho\ d$ は、ペア $(d',\ is)$ です。
ここで、is は、環境 ρ で式 e の値をスタック上の引数に適用する TIM コードです。
このコードは、フレームの最初の d スロットが占有されていると想定しており、
スロット $(d+1\ ...\ d')$ を使用します。

$$\mathcal{R}[\![e]\!]\ \rho\ d \quad = \quad \mathcal{B}[\![e]\!]\ \rho\ d\ [\texttt{Return}]$$

ここで、e は整数または算術式です。

$$\mathcal{R}[\![a]\!]\ \rho\ d \quad = \quad (d,\ [\texttt{Enter}\ (\mathcal{A}[\![a]\!]\ \rho)])$$

ここで、a はスーパーコンビネータ
またはローカル変数です。

$$\mathcal{R}[\![e\ a]\!]\ \rho\ d \quad = \quad (d_1, \texttt{Push}\ (\mathcal{A}[\![a]\!]\ \rho)\ \texttt{:}\ is)$$

ここで、a はスーパーコンビネータ、
ローカル変数、または整数であり、
$d_1,\ is$ は以下の通りです。
$(d_1,\ is) = \mathcal{R}[\![e]\!]\ \rho\ d$
です。

$$\mathcal{R}[\![e_{fun}\ e_{arg}]\!]\ \rho\ d \quad = \quad (d_2, \texttt{Move}\ (d+1)\ am_{arg}\ \texttt{:}\ \texttt{Push}\ (\mathcal{I}[\![d+1]\!]\ \rho)\ \texttt{:}\ is_{fun})$$

ここで $d_2,\ is_{fun}$ は以下の通りです。
$(d_1,\ am_{arg}) = \mathcal{U}[\![e_{arg}]\!]\ (d+1)\ \rho\ (d+1)$
$(d_2,\ is_{fun}) = \mathcal{R}[\![e_{fun}]\!]\ \rho\ d_1$
です。

$\mathcal{R}[\![\texttt{let } x_1 \texttt{ = } e_1\texttt{; } \ldots\texttt{; } x_n \texttt{ = } e_n \texttt{ in } e]\!]\ \rho\ d$

$=\quad (d',\ [\texttt{Move } (d+1)\ am_1,\ \ldots,\ \texttt{Move } (d+n)\ am_n]\ \ \texttt{++}\ \ is)$

ここで $am_1,\ \ldots,\ am_n,\ is$ は以下の通りです。

$(d_1,\ am_1) = \mathcal{U}[\![e_1]\!]\ (d+1)\ \rho\ (d+n)$

$(d_2,\ am_2) = \mathcal{U}[\![e_2]\!]\ (d+2)\ \rho\ d_1$

$\ldots$

$(d_n,\ am_n) = \mathcal{U}[\![e_n]\!]\ \rho\ d_{n-1}$

$\rho' = \rho\,[x_1\ \mapsto\ \mathcal{I}[\![d+1]\!],\ \ldots,\ x_n\ \mapsto\ \mathcal{I}[\![d+n]\!]]$

$(d',\ is) = \mathcal{R}[\![e]\!]\ \rho'\ d_n$

です。

`letrec` の場合も同様ですが、$\mathcal{U}[\![]\!]$ の呼び出しに ρ ではなく
ρ' が渡される点が異なります。

$\mathcal{R}[\![\texttt{Pack}\{t,\ a\}]\!]\ \rho\ d$

$=\quad (d,\ [\texttt{UpdateMarkers } a,\ \texttt{Take } a\ a,\ \texttt{ReturnConstr } t])$

$\mathcal{R}[\![\texttt{case } e \texttt{ of } alt_1\ \ldots\ alt_n]\!]\ \rho\ d$

$=\quad (d',\ \texttt{Push } (\texttt{Code } [\texttt{Switch } [branch_1\ \ldots\ branch_n]])\ \texttt{:}\ is_e)$

ここで、$d_1,\ \ldots,\ d_n,\ d',\ is_e$ は以下の通りです。

$(d_1,\ branch_1) = \mathcal{E}[\![alt_1]\!]\ \rho\ d)$

$\ldots$

$(d_n,\ branch_n) = \mathcal{E}[\![alt_n]\!]\ \rho\ d)$

$(d',\ is_e) = \mathcal{R}[\![e]\!]\ \rho\ max(d_1,\ \ldots,\ d_n)$

$\mathcal{E}[\![alt]\!]\ \rho\ d$ (alt は `case` の選択肢) はペア $(d',\ branch)$ であり、
$branch$ は、環境 ρ でコンパイルされた `Switch` の選択肢です。
コードは、フレームの最初の d スロットが占有されていると想定し、
スロット $(d+1,\ \ldots,\ d')$ を使用します。

$\mathcal{E}[\![\texttt{<}t\texttt{>}\ x_1\ \ldots\ x_n\ \texttt{->}\ body]\!]\ \rho\ d$

$=\quad (d',\ t\ \texttt{->}\ (is_{moves}\ \texttt{++}\ is_{body}))$

ここで、$d',\ is_{moves},\ is_{body}$ は以下の通りです。

$is_{moves} = [\texttt{Move } (\mathrm{d}+1)\ (\texttt{Data } 1), \ldots, \texttt{Move } (\mathrm{d}+\mathrm{n})\ (\texttt{Data } \mathrm{n})]$

$(d',\ is_{body}) = \mathcal{R}[\![body]\!]\ \rho'\ (d+n)$

$\rho' = [x_1 \mapsto \texttt{Arg } (d+1),\ \ldots,\ x_n \mapsto \texttt{Arg } (d+n)]$

$\mathcal{U}[\![e]\!]\ u\ \rho\ d$ はペア $(d',\ am)$ です。
ここで、am は環境 ρ における式 e の TIM アドレッシングモードです。
am によってアドレス指定されたクロージャが Enter されると、
現在のフレームのスロット u が、そのノーマルフォーム (※評価結果) で更新されます。
このコードは、フレームの最初の d スロットが占有されていると想定しており、
スロット $(d+1,\ \ldots,\ d')$ を使用します。

$$\mathcal{U}[\![n]\!]\ u\ \rho\ d \quad = \quad (d,\ \texttt{IntConst}\ n)$$
ここで、n は整数定数です。

$$\mathcal{U}[\![e]\!]\ u\ \rho\ d \quad = \quad (d',\ \texttt{Code}\ (\texttt{PushMarker}\ u\ \texttt{:}\ is))$$
ここで e は整数定数以外であり、$d',\ is$ は以下の通りです。
$(d',\ is) = \mathcal{R}[\![e]\!]\ \rho\ d$

$\mathcal{A}[\![e]\!]\ \rho$ は、環境 ρ における式 e の TIM アドレッシングモードです。

$$\mathcal{A}[\![n]\!]\ \rho\ d \quad = \quad (d,\ \texttt{IntConst}\ n)$$
ここで、n は数値です。

$$\mathcal{A}[\![x]\!]\ \rho\ d \quad = \quad (d,\ \rho\ x)$$
ここで、x は ρ によって束縛される変数です。

$\mathcal{I}[\![d]\!]$ はオフセット d のフレームの間接アドレッシングモードです。

$$\mathcal{I}[\![d]\!] = \texttt{Code}\ [\texttt{Enter}\ (\texttt{Arg}\ d)]$$

$\mathcal{B}[\![e]\!]\ \rho\ d\ cont$ はペア $(d',\ is)$ です。
ここで、is は環境 ρ で e を評価し、
その値 (整数である必要があります) を値スタックの一番上に置き、
コードシーケンス $cont$ を続ける TIM コードです。
コードは、フレームの最初の d スロットが占有されていると想定し、
スロット $(d+1,\ldots,\ d')$ を使用します。

$$\begin{aligned}\mathcal{B}[\![e_1\texttt{+}\ e_2]\!]\ \rho\ d\ cont \quad &= \quad \mathcal{B}[\![e_2]\!]\ \rho\ d_1\ is_1 \\ &= \quad (d_1,\ is_1) = \mathcal{B}[\![e_1]\!]\ \rho\ d\ (\texttt{Op Add}\ \texttt{:}\ cont))\end{aligned}$$
... および他の算術プリミティブの同様の規則

$$\mathcal{B}[\![n]\!]\ \rho\ d\ cont \quad = \quad (d,\ \texttt{PushV}\ (\texttt{IntVConst}\ n)\ \texttt{:}\ cont)$$
ここで、n は数値です。

$$\mathcal{B}[\![e]\!]\ \rho\ d\ cont \quad = \quad (d',\ \texttt{Push}\ (\texttt{Code}\ cont)\ \texttt{:}\ is)$$
ここで、e は数値以外であり、$d',\ is$ は以下の通りです。
$(d',\ is) = \mathcal{R}[\![e]\!]\ \rho\ d$

明らかな疑問は、「TIM は G マシンより優れているのか、それとも劣っているのか?」ということです。答えるのは難しいです。私たちのプロトタイプは、設計の選択を検討するのには非常に役立ちますが、本格的なパフォーマンスの比較には全く役に立ちません。たとえば、G マシン `Mkap` と比較した `Take` 命令の相対コストをどのように確立できるでしょうか? 実際に比較できる唯一の尺度は、この 2 つのヒープ消費量です。

それでも、この章で行ったように、別の評価モデルを検討することは、TIM の側面と G マシンの側面を組み合わせた他の設計手段を示唆するため、非常に啓発的です。そうする試みの 1 つがスパインレスタグレス G マシンです [Peyton Jones and Salkild 1989, Peyton Jones 1991]。これは TIM のスパインレスと更新メカニズムを採用していますが、そのスタックは (G マシンと同様に) ヒープオブジェクトへのポインタで構成されています。(TIM のような) コードとフレームのペアではありません。

```
module ParGM where
import Utils
import Language
--import GM
```

第5章　並列Gマシン

5.1　はじめに

この章では、並列 G マシン用の抽象マシンとコンパイラを開発します。これは、ESPRIT プロジェクト 415 の一部として開発された並列 G マシンの簡易バージョンに基づいています。興味のある読者は、簡単にアクセスできる [Kingdon et al 1991] を参照してください。[Peyton Jones 1989] では、並列グラフ簡約の一般的な紹介を見つけることができます。

5.1.1　並列関数型プログラミング

並列命令型プログラムを書くのは難しいです。 その理由としては次のようなものが挙げられます。

- プログラマは、問題の仕様を満たす**並列アルゴリズム**を考え出す必要があります。
- アルゴリズムは、言語によって提供されるプログラミング言語構造に変換される必要があります。これには、同時実行タスクの識別、タスク間の**同期**と**通信**を可能にするためのインターフェースの定義が含まれる可能性があります。複数のタスクが変数に同時にアクセスすることを防ぐために、**共有データ**を特に保護する必要がある場合があります。
- プログラマは、タスクをプロセッサに割り当てる責任を負い、相互に通信する必要があるタスクが物理的に接続されているプロセッサに確実に割り当てられるようにする必要があります。
- 最後に、**スケジューリングポリシー**をプログラマが制御できないシステムでは、プログラマは、タスク操作の可能なすべての**インターリーブ**のもとで同時タスクのコレクションが正しく実行されることを証明する必要があります。

対照的に、関数型言語でプログラミングする場合は、これらの点のうち最初の点のみが適用されます。次の (人為的な) プログラム例 `psum` n を考えてみましょう。これは、数値 $1 \ldots n$ の合計を計算します。

```
psum n = dsum 1 n;
dsum lo hi = let mid = (lo + hi) / 2
             in if (hi == lo) hi ((dsum lo mid) + (dsum (mid + 1) hi))
```

dsum 関数は、問題をほぼ等しい 2 つの部分に分割することによって動作します。次に、2 つの結果を組み合わせて答えを生成します。これは古典的な**分割統治アルゴリズム**です。

関数 dsum と psum のどちらにも、言語内の並列プリミティブについての言及が含まれていないことに注意してください。では、なぜ psum は並列アルゴリズムなのでしょうか? 比較のために、逐次アルゴリズム ssum を作成できます。

```
ssum n = if (n == 1) 1 (n + ssum (n - 1))
```

この関数は、**データの依存関係**が本質的に逐次的であるため、逐次アルゴリズムです。ssum の例では、これが意味することは、ssum n の加算は ssum (n - 1) が評価された後にのみ実行できるということです。これは、ssum (n - 2) などを評価した後にのみ実行できます。この違いを次のように要約できます。

> 関数は、その本体の 2 つ以上の部分式の同時評価を許可する場合は常に、並列アルゴリズムを実装します。

他のプログラミング言語と同様に、並列アルゴリズムが不可欠です。ただし、並列命令型プログラミングとは対照的であることに注意してください。

- 並列処理、同期、通信を表現するために新しい言語構造を構築する必要はありません。同時実行性は暗黙的であり、新しいタスクが動的に作成され、マシンに空き容量があるときはいつでもマシンによって実行されます。
- 同時タスク間で共有されるデータを保護するための特別な対策は講じられていません。たとえば、mid は dsum の 2 つの同時タスク間で安全に共有されます。
- 逐次実行される関数型プログラムに使用するすべての手法が引き続き機能するため、並列プログラムについて推論するために新しい証明手法は必要ありません。また、デッドロックは次のような自己依存性の結果としてのみ発生する可能性があることにも注意してください。

```
letrec a = a + 1 in a
```

自身の値に依存する式は無意味であり、その非停止性は逐次実装で観察される動作と同じです。

- プログラムの結果は決定的です。スケジューリングアルゴリズムでは、同じプログラムの 2 つの実行間で回答が異なることはあり得ません。

要約すると、これらの機能により、多数の低レベルの問題を解決する必要なく、並列アルゴリズムを簡単に表現できることがわかります。おそらくこれは次のように特徴づけることができます。

> 並列命令型プログラムは、並列関数型プログラムでは言及すらしないリソース割り当ての決定を詳細に指定します。

これは、マシンがリソース割り当ての決定を自動的に行うことができる必要があることを意味します。これらの決定が最適でない場合には、実行効率を損なうことになり、その代償を支払うことになります。

アノテーション

関数型言語によって提供される高度な抽象化により、コンパイル時および実行時のリソースアロケータに大きな要求が課されます。すべてのリソース割り当ての決定をシステムに任せるのではなく、新しい並列スレッドを開始するアノテーション `par` を導入します。アノテーションは、プログラムテキストの意味を保持する装飾です。 `par` の場合、次の構文と意味になります。

$$\texttt{par}\ E_1\ E_2 = E_1\ E_2$$

つまり、`par` は関数適用の同義語です。異なる点は、式 E_2 が同時タスクによって評価されることを意図していることです。例として、このアノテーションを使用して `dsum` 関数を書き直します。

```
dsum lo hi = let mid = (lo + hi) / 2
             in let add x y = x+y
                in if (lo == hi) hi (par (add (dsum lo mid))
                                         (dsum (mid + 1) hi))
```

`par` によって 2 番目の引数 `+` が並列評価されることがわかります。`par` アノテーションはプログラマによって挿入できますが、原理的には賢いコンパイラによって挿入できます。ただし、そのような賢さについては本書の範囲を超えているため、`par` はすでに挿入されているものと仮定します。

5.1.2　並列グラフ簡約

この本では、グラフ簡約が逐次マシンにとって有用な実装手法であることを説明しました。 並列マシンの実装にも適していることは驚くことではありません。実際、これには多くの利点があります。

- プログラムカウンタには連続的な概念はありません。グラフの簡約は分散化され、配布されます。
- 簡約はグラフ内の多くの場所で同時に実行される可能性があり、スケジュールされた相対的な順序が結果に影響することはありません。
- すべての通信と同期はグラフを介して行われます。

並列モデル

タスクは、特定のサブグラフを WHNF に還元することを目的とした逐次計算です。いつでも、実行できるタスクが多数存在する可能性があります。このスパークされたタスクのコレクションは、**スパークプール**と呼ばれます。作業を探しているプロセッサは、スパークプールから新しいタスクをフェッチし、それを実行します。タスクは**仮想プロセッサ**と考えることができます。

最初は、プログラム全体を評価するタスクが 1 つだけあります。実行中に、メインタスクが後で必要とする式を評価するための新しいタスクが作成されることが期待されます。これらはスパークプールに配置され、仕事がなくなった場合に別のプロセッサがタスクを引き継ぐことができます。タスクをスパークプールに配置する行為を、子タスクの**スパーク**と呼びます。

親タスクと子タスク間の相互作用を考慮すると便利です。タスク管理の評価して終了するモデルでは、各グラフノードにロックビットがあります。ビットがオンの場合、ノードを評価するタスクが実行されています。 それ以外の場合、ビットはオフです。親タスクが、タスクがスパークされたサブグラフの値を必要とする場合、考慮すべきケースが 3 つあります。

- 子タスクはまだ開始されていません。
- 子タスクは開始されましたが、まだ停止していません。
- 子タスクが完了しました。

最初のケースでは、親タスクは、あたかも子タスクが存在しないかのようにグラフを評価できます。もちろん、これによりロック ビットが設定されるため、子タスクを実行しようとするとすぐに破棄されます。興味深いのは 2

番目のケースで、親と子の両方が実行されています。このような状況の場合、親タスクは子タスクが完了するまで待ってから続行する必要があります。子タスクが親タスクを**ブロック**していると言います。3 番目のケースでは、ノードは現在 WHNF にあり、ロックが解除されているため、親タスクがグラフの値をフェッチするのにそれほど時間はかかりません。

評価して終了するモデルの利点は、親と子が実際に衝突した場合にのみブロッキングが発生することです。それ以外の場合はすべて、親の実行は妨げられることなく続行されます。ブロックメカニズムがタスク間通信および同期の**唯一の**形式であることに注意してください。グラフの一部が WHNF に対して評価されると、任意の数のタスクが競合することなく同時にそれを検査できます。

例

最も単純な並列プログラムのサンプル実行から始めます。

```
main = par I (I 3)
```

後で説明するように、恒等関数の 2 つの簡約は並行して行われます。

優れたコンパイラは、`main` スーパーコンビネータに対して次のコードを生成します。[1]

```
e1 ++ par
where e1  = [Pushint 3, Pushglobal "I", Mkap, Push 0]
      par = [Par, Pushglobal "I", Mkap, Update 0, Pop 0, Unwind]
```

最初は、実行中のタスクは 1 つだけで、部分式 `(I 3)` を構築しています。コードシーケンス `e1` を実行した後、マシンは図 5.1 の図 (a) に示す状態になります。

`e1` を実行した後、マシンは `Par` 命令に遭遇します。これにより、スタックの先頭から指定されたノードを評価するための新しいタスクが作成されます。新しいタスク `Task 2` と呼びます。元のタスクは `Task 1` と呼ばれます。この状況は、図 5.1 の図 (b) に示されています。

[1] コンパイラが十分に洗練されていない場合、このコードを生成しない可能性があります。

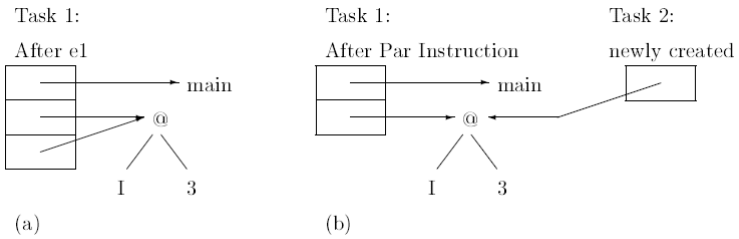


Figure 5.1: State after executing `e1` code sequence and `Par`

図 (c) (図 5.2) では、`Task 1` が評価を続行しており、`Pushglobal I` 命令を実行していることがわかります。新しく作成されたタスク (`Task 2`) は、コードシーケンス:`[Unwind]` で始まります。したがって、評価するように割り当てられたグラフのアンワインドを開始します。図 (d) は、`Task 1` が `main` の本体のインスタンス化を完了し、`Task 2` がアンワインドを完了していることを示しています。

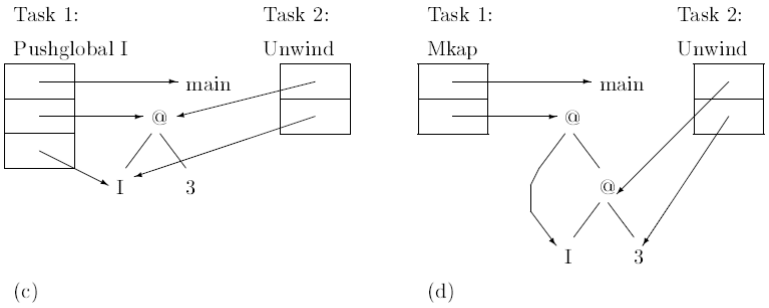


Figure 5.2: State after `Task 1` executes `[Pushglobal I, Mkap]`

本体がインスタンス化されると、`Task 1` は `main` である redex ノードを上書きします。`Task 2` は `Push 0` 命令を実行します。これは `I` スーパーコンビネータのコードの最初の命令です。これは図 (e) に示されています (図 5.3)。図 (f) では、`Task 1` がスパインをアンワインドし始め、`Task 2` が更新を実行していることがわかります。

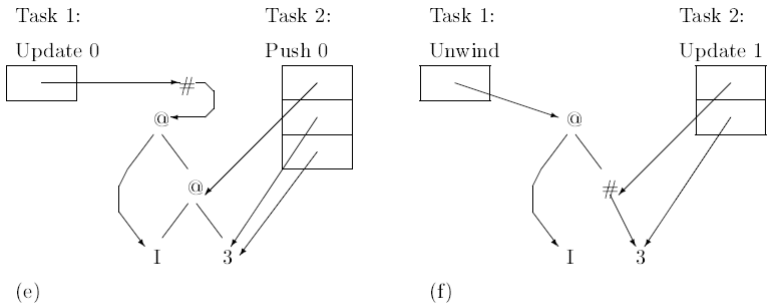


Figure 5.3: State after `Task 1` executes `[Update 0, Unwind]`

図 5.4 では、`Task 2` が完了するまで実行されています。`Task 1` の残りの実行は、シーケンシャル G マシンと同じなので省略します。

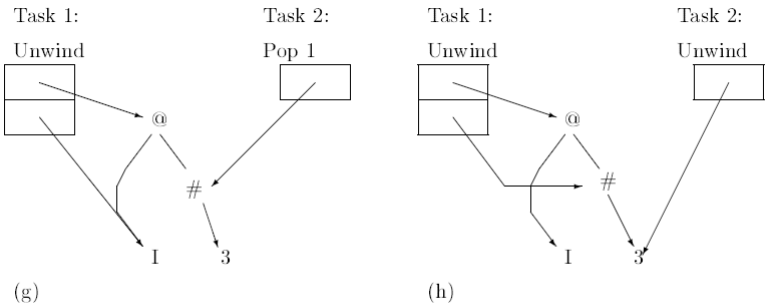


Figure 5.4: State after `Task 1` executes `[Unwind, Unwind]`

これで、並列 G マシンの同時タスク実行の概要は終了です。最小限の並列 G マシンを提供します。

5.2　Mark 1: 最小限の並列 G マシン

最初に紹介するマシンは、第 3 章の G マシンのいずれか (Mark 1 を除く) をベースにすることができます。更新を確実に行うには、この制限が必要です。この基本マシンに、並列処理のための機構を追加する必要があります。次の基本的な仮定を立てます。

1. すべてのプロセッサがアクセスできる共有グローバル グラフがあります。
2. プロセッサの数は無限です。つまり、タスクを実行できるプロセッサが常に存在します。
3. グラフ ノードはロックされません。つまり、異なるタスクが同じ式を再評価する可能性があります。

5.2.1　データ型の定義

この章では、状態アクセス関数を多用します。特に興味深いものではありませんが、完全性を保つためにこのセクションに含めています。

並列マシンでは、状態 pgmState は、グローバルコンポーネント pgmGlobalState とローカルコンポーネント pgmLocalState の 2 つのコンポーネントに分割されます。pgmLocalState には、プログラムを実行するプロセッサが含まれます。pgmGlobalState には、プロセッサによって操作されるグローバルデータ構造 (最も頻繁に使用されるのはヒープ) が含まれます。

```
type PgmState = (PgmGlobalState,   -- Current global state
                 [PgmLocalState])  -- Current states of processors
```

グローバル状態コンポーネント

並列実装のベースとして使用できる可能性のあるすべてのマシンに対応するために、グローバル状態は 5 つのコンポーネントで構成されます。プログラムの評価に対する回答として出力される出力である gmOutput、ヒープである gmHeap、各スーパーコンビネータのヒープ内の一意のノードを指すために使用される gmGlobals、タスクプールであり、タスクの実行を開始する前にタスクを保持するために使用される gmSparks、およびグローバルクロックである gmStats です。

```
type PgmGlobalState = (GmOutput,  -- output stream
                       GmHeap,    -- Heap of nodes
```

```
                    GmGlobals, -- Global addresses in heap
                    GmSparks,  -- Sparked task pool
                    GmStats)   -- Statistics
```

これらの各コンポーネントを順番に検討します。

- gmOutput コンポーネントは、Mark 6 G-machine で導入されました (セクション 3.8 を参照)。これは、構造化データを生成できるプログラムの実行結果を蓄積するために使用され、単なる文字列です。

```
type GmOutput = [Char]
```

pgmState から gmOutput を取得する関数は pgmGetOutput です。

```
pgmGetOutput :: PgmState -> GmOutput
pgmGetOutput ((o, heap, globals, sparks, stats), locals) = o
```

- ヒープ データ構造は、シーケンシャル G マシンで使用したものと同じです。

```
type GmHeap = Heap Node
```

pgmState からヒープを取得するには、pgmGetHeap 関数を使用します。

```
pgmGetHeap :: PgmState -> GmHeap
pgmGetHeap ((o, heap, globals, sparks, stats), locals) = heap
```

- ヒープ内のグローバルノードのアドレスは gmGlobals に格納されます。これもシーケンシャル G マシンで使用したのと同じ構造です。

```
type GmGlobals = ASSOC Name Addr
```

pgmState から gmGlobals を取得するには、pgmGetGlobals 関数を使用します。

```
pgmGetGlobals :: PgmState -> GmGlobals
pgmGetGlobals ((o, heap, globals, sparks, stats), locals) = globals
```

- スパークプールは、gmSparks コンポーネントによって表されます。これは、par アノテーションによって同時に評価する必要があるとマークされたグラフ内のノードのアドレスを保持します。

```
type GmSparks = [Addr]
```

このコンポーネントへのアクセスは、関数 pgmGetSparks を使用して実現されます。

```
pgmGetSparks :: PgmState -> GmSparks
pgmGetSparks ((o, heap, globals, sparks, stats), locals) = sparks
```

- 最後に、並列Gマシンでは、グローバルコンポーネントに蓄積された統計情報を保持します。これは数値のリストとして表され、マシン内の各タスクが完了するまでに実行された時間の詳細を示します。

```
type GmStats = [Int]
```

このコンポーネントへのアクセスは、pgmGetStats を使用して行われます。

```
pgmGetStats :: PgmState -> GmStats
pgmGetStats ((o, heap, globals, sparks, stats), locals) = stats
```

ローカル状態コンポーネント

並列Gマシンのローカルコンポーネントはプロセッサのリストで構成されます。プロセッサはタスクとして表されます。

ここでも、並列マシンを任意のGマシンをベースとして実行できるようにするために、各プロセッサの状態を5要素のタプルにします。

```
type PgmLocalState = (GmCode,   -- Instruction stream
                      GmStack,  -- Pointer stack
                      GmDump,   -- Stack of dump items
                      GmVStack, -- Value stack
                      GmClock)  -- Number of ticks the task has been active
```

ここで、各コンポーネントを順番に検討します。

- コードシーケンスは単なる命令のリストです。

```
type GmCode = [Instruction]
```

- シーケンシャル G マシンと同様に、スタックはヒープ内のアドレスのリストです。

```
type GmStack = [Addr]
```

- 並列実装のベースとして Mark 4 G マシン (またはそれ以降のマーク) を使用している場合は、ダンプが必要です。これは、それぞれが gmDumpItem 型のダンプ項目のスタックとして使用されます。

```
type GmDump = [GmDumpItem]
type GmDumpItem = (GmCode, GmStack)
```

- 実装のベースとして Mark 7 G マシンを使用した場合は、プロセッサごとに V スタックが必要になります。

```
type GmVStack = [Int]
```

- また、各プロセッサにクロックも提供します。これにより、タスクが実行した命令の数を記録します。

```
type GmClock = Int
```

状態アクセス関数

並列マシンの状態アクセス関数はすでにいくつか定義されていますが、さらにいくつか定義しておくと便利です。各プロセッサは 1 つの状態遷移を行い、その間はシーケンシャルマシンのように動作します。状態 gmState を、現在のマシン状態のグローバルコンポーネントと単一のプロセッサ状態からなるペアにすると、シーケンシャル G マシンの状態コンポーネントのスーパーセットが得られます。

```
type GmState = (PgmGlobalState, PgmLocalState)
```

したがって、シーケンシャル マシンで使用した状態アクセス関数を、新しいタイプの状態で動作するように単純に再定義できます。グローバル put 関数の型シグネチャは次のとおりです。

```
putOutput :: GmOutput -> GmState -> GmState
putHeap :: GmHeap -> GmState -> GmState
putSparks :: GmSparks -> GmState -> GmState
putStats :: GmStats -> GmState -> GmState
```

対応する get 関数の型シグネチャは次の通りです。

```
getOutput :: GmState -> GmOutput
getHeap :: GmState -> GmHeap
getGlobals :: GmState -> GmGlobals
getSparks :: GmState -> GmSparks
getStats :: GmState -> GmStats
```

プロセッサのローカルコンポーネントにアクセスするには、次の型シグネチャを持つ関数を配置する必要があります。

```
putCode :: GmCode -> GmState -> GmState
putStack :: GmStack -> GmState -> GmState
putDump :: GmDump -> GmState -> GmState
putVStack :: GmVStack -> GmState -> GmState
putClock :: GmClock -> GmState -> GmState
```

get 関数の型シグネチャは次の通りです。

```
getCode :: GmState -> GmCode
getStack :: GmState -> GmStack
getDump :: GmState -> GmDump
getVStack :: GmState -> GmVStack
getClock :: GmState -> GmClock
```

演習 5.1 上記の型を使用してアクセス関数を記述します。

5.2.2　評価器

評価器 eval の構造はよく知られており、G マシンで使用されるものと似ています。

```
eval :: PgmState -> [PgmState]
eval state = state: restStates
             where
               restStates | gmFinal state = []
                          | otherwise = eval (doAdmin (steps state))
```

違いは、step ではなく steps を呼び出すことです。steps 関数は、プロセッサのリストを順に実行し、それぞれに対して 1 つのステップを実行する必要があります。イベントの正確なシーケンスは次のとおりです。

1. まず、前回の steps 呼び出しで生成されたアドレスを状態から抽出します。
2. 次に、それらをプロセスに変換します。これは newtasks というラベルが付けられています。
3. 状態のスパークプールコンポーネントは空に設定されています。
4. 実行しようとしている各プロセッサのクロックをインクリメントします。
5. 最後に、mapAccuml を使用して、プロセッサごとに 1 つずつ、一連の step 遷移を実行します。

```
steps :: PgmState -> PgmState
steps state
  = mapAccuml step global' local'
    where ((out, heap, globals, sparks, stats), local) = state
          newtasks = [makeTask a | a <- sparks]
          global'  = (out, heap, globals, [], stats)
          local'   = map tick (local ++ newtasks)
```

アドレス addr のノードを評価するタスクを作成するには、makeTask 関数を定義する必要があります。Mark 2 G マシンまたは Mark 3 G マシンに基づく場合、次のようになります。

```
makeTask :: Addr -> PgmLocalState
makeTask addr = ([Unwind], [addr], [], [], 0)
```

Mark 4 以降の G マシンに基づく場合は以下を使用します。

```
makeTask addr = ([Eval], [addr], [], [], 0)
```

プロセッサのクロックコンポーネントのインクリメントは、 tick を使用して実行されます。

```
tick (i, stack, dump, vstack, clock) = (i, stack, dump, vstack, clock+1)
```

スパークプールにスパークがなくなり、タスクを実行しているプロセッサがなくなると、マシンは終了します。

```
gmFinal :: PgmState -> Bool
gmFinal s = second s == [] && pgmGetSparks s == []
```

プロセッサ上で単一のステップを実行するには、step 関数を使用します。

```
step :: PgmGlobalState -> PgmLocalState -> GmState
step global local = dispatch i (putCode is state)
                    where (i:is) = getCode state
                          state = (global, local)
```

doAdmin 関数は、実行を終了したプロセッサを削除します。コードコンポーネントが空になると、プロセッサは終了しています。この場合、プロセッサがタスクを完了するのに要した命令の数で、状態の統計コンポーネントを更新する必要があります。

```
doAdmin :: PgmState -> PgmState
doAdmin ((out, heap, globals, sparks, stats), local)
  = ((out, heap, globals, sparks, stats'), local')
    where (local', stats') = foldr filter ([], stats) local
          filter (i, stack, dump, vstack, clock) (local, stats)
            | i == [] = (local, clock:stats)
            | otherwise = ((i, stack, dump, vstack, clock): local, stats)
```

ここで、新しい命令の遷移について検討します。

Par 命令の遷移

追加する必要がある唯一の新しい命令は Par です。その効果は、マシンがノードを WHNF に評価するタスクを作成できるように、スタックの最上部のノードをマークすることです。これを行うには、命令は、ノードのアドレスをスパークプールに追加して、状態のグローバルコンポーネントを変更する必要があります。

(5.1)

$$
\begin{array}{lllllll}
 & \langle & h & m & t & \rangle & \langle \quad \texttt{Par} : i \quad a : s \quad \rangle \\
\Longrightarrow & \langle & h & m & a : t & \rangle & \langle \qquad i \qquad s \quad \rangle
\end{array}
$$

最初のタプル (*h*、*m*、*t* から構成) はグローバル状態コンポーネントで、*h*、*m*、*t* はそれぞれヒープ、グローバルアドレスマップ、スパークプールです。2 番目のタプル (命令ストリームとスタックから構成) は特定のタスクのローカル状態です。ベースとして使用した G マシンのバージョンによっては、ローカル状態に他のコンポーネントを追加する必要がある場合があります。

`Par` の効果は、アドレス *a* をスパークプールに追加することです。次のように実装されます。

```
par :: GmState -> GmState
par s = s
```

演習 5.2 `Par` が正しく処理されるように、`showInstruction`, `dispatch`, `instruction` を変更します。

5.2.3 プログラムのコンパイル

`par` 関数のコンパイル済みプリミティブを提供することで、シーケンシャルマシン (Mark 7 を除く) のいずれかに基づく並列マシン用のシンプルなコンパイラを構築できます。Mark 7 ベースのマシンでは、より広範な変更が必要です。

他に必要な変更は `compile` 関数にあります。ここでは、さまざまなコンポーネントがさまざまな場所に存在するようになり、もちろんプロセッサの数も増えました。新しい定義は次のとおりです。

```
compile :: CoreProgram -> PgmState
compile program
  = (([], heap, globals, [], []), [initialTask addr])
    where (heap, globals) = buildInitialHeap program
          addr = aLookup globals "main" (error "main undefined")
```

これにより、シーケンシャル G マシンで使用されるように、グローバルコンポーネントがヒープとグローバルマップを保持するよう設定されます。また、実行を開始するために、ローカルコンポーネントにタスクを配置します。

```
initialTask :: Addr -> PgmLocalState
initialTask addr = (initialCode, [addr], [], [], 0)
```

Mark 2 または Mark 3 シーケンシャル G マシンをベースとして使用する場合は、`initialCode` を次のように定義する必要があります。

```
initialCode :: GmCode
initialCode = [Unwind]
```

Mark 4 または Mar 5 マシンの場合、これは次のように変更されます。

```
initialCode = [Eval]
```

また、データ構造を処理するために、Mark 6 および Mark 7 マシンの initialCode は次のようになります。

```
initialCode = [Eval, Print]
```

ここで、マシンの基本機能に par を追加する方法を検討します。まず、シーケンシャル G マシン Mark 2 から 6 に基づくマシンを検討します。

Mark 2-6 G マシンをベースとして

compiledPrimitives の定義には、次の内容を含める必要があります。

```
("par", 2, [Push 1, Push 1, Mkap, Push 2, Par, Update 2, Pop 2, Unwind])
```

このかなり難解なコードは、関数 par が 2 つの引数 E_1 と E_2 に適用されたときに、次のタスクを実行します。

1. まず、E_1 を E_2 に適用します。これがシーケンスの役割です。

   ```
   [Push 1, Push 1, Mkap]
   ```

2. 次に、Push 2 は E_2 へのポインタのコピーを作成します。次に、Par 命令がこのアドレスをスパークプールに追加します。

3. 最後に、インスタンス化後に通常の更新と整理を実行します。

Mark 7 G マシンをベースに

上で述べたように、これは少しトリッキーな操作です。par を含む特殊なケースを認識するために、コンパイラ関数 compileR と compileE を変更する必要があります。まず、compileR に次のケースを追加する必要があります。

```
compileR (EAp (EAp (EVar "par") e1) e2) args
  = compileC e2 args ++ [Push 0, Par] ++
    compileC e1 (argOffset 1 args) ++ [Mkap, Update n, Pop n, Unwind]
    where n = #args
```

これは $\mathcal{C}$ スキームを使用して e2 をコンパイルし、その後スパークします。式 e1 は、アプリケーションノードを作成する前に $\mathcal{C}$ スキームを使用してコンパイルされます。最後に、スタックの更新と整理を実行します。

次に、compileE を修正して、次のケースを作成します。

```
compileE (EAp (EAp (EVar "par") e1) e2) args
  = compileC e2 args ++ [Push 0, Par] ++
    compileC e1 (argOffset 1 args) ++ [Mkap, Eval]
```

これは、作成されるアプリケーションノードを WHNF に強制するために Eval を使用するという点でのみ、compileR の場合と異なります。

コンパイラにこれら 2 つの変更を加えたら、プリミティブに次のコードを追加するだけで十分です。

```
("par", ["x","y"], (EAp (EAp (EVar "par") (EVar "x")) (EVar "y")))
```

このアプローチは、Mark 5 または Mark 6 ベースのマシンでも使用できることに注意してください。

演習 5.3 par 関数が定義されるようにコンパイラを変更します。

演習 5.4 compileR および compileE の特殊なケースで、最初の引数を構築する前に 2 番目の引数のグラフのスパークを実行するのはなぜですか?

5.2.4 結果の表示

マシンの状態を計算したら、showResults を使用してそれらの表示を制御します。これは、スーパーコンビネータのコード、状態遷移、および統計を出力します。型は次のとおりです。

```
showResults :: [PgmState] -> [Char]
```

スーパーコンビネータコードを表示するには、G マシンと同じ showSC 関数を使用します。その型は次のとおりです。

```
showSC :: PgmState -> (Name, Addr) -> Iseq
```

showState 関数は、遷移中のローカルプロセスの状態を表示するために使用されます。並列マシンがあるため、複数のタスクが同時に実行される可能性があります。その型は次のとおりです。

```
showState :: PgmState -> Iseq
```

他に 2 つの関数、showStats と showOutput を変更する必要があります。これらの関数の型は次の通りです。

```
showStats :: PgmState -> Iseq
showOutput :: GmOutput -> Iseq
```

演習 5.5 関数 showResults、showSC、showState、showStats、showOutput を変更します。次の型の新しい表示関数 showSparks を定義します。

```
showSparks :: GmSparks -> Iseq
```

演習 5.6 次のプログラムを並列 G マシンで実行してみてください。

```
main = par (S K K) (S K K 3)
```

マシンサイクルではどれくらい時間がかかりますか？同等のシーケンシャルプログラムにはどれくらい時間がかかりますか？

```
main = S K K (S K K 3)
```

演習 5.7 次のプログラムを実行すると何が起きますか？

```
main = par I (I 3)
```

このプログラムで par を使用することは正当でしょうか?

5.3 Mark 2: 評価して終了するモデル

Mark 1 マシンの問題は、ヒープ内の同じノードを簡約するために多くのタスクを作成し、その結果マシンによって実行される作業が重複するリスクがあることです。これを防ぐ方法は、アンワインド中にノードを**ロック**することです。これにより、同じノードに遭遇する他のすべてのタスクが**ブロック**されます。また、ノードが再び解放されたら、ノードの**ロックを解除**することを忘れないようにしてください。これにより、ブロックされたタスクが再開されます。

ロックされていないノードをロック状態にする唯一の命令は `Unwind` です。これは、スパイン上の各ノードに発生します。`Eval` ではなくこの命令を選択する理由は、スパインをロック解除する途中でロックされたノードに遭遇する可能性があるためです。`Eval` を使用すると、このケースをキャッチできません。同様に、タスクをブロックする唯一の命令は `Unwind` です。結局のところ、ノードの値を検査する必要があるのは `Unwind` だけです。

逆に、ロックされたノードのロックを解除する唯一の命令は `Update` です。以前にロックされたノードは、WHNF であることがわかっている場合にロック解除されます。しかし、Redex を更新するときには、Redex の下のスパイン内のすべてのノードが WHNF であることがわかっているため、Redex のルートの下のすべてのノードのロックを解除する必要があります。

5.3.1 ノードデータ構造

私たちが取り入れようとしている改善には、マシンのデータ構造にほとんど変更を加える必要がありません。まず、ノードデータ型に 2 種類の新しいノードを追加する必要があります。これらは、ロックされたアプリケーションノードである `NLAp` と、ロックされたスーパーコンビネータノードである `NLGlobal` です。新しい命令遷移のセクションで、これらがどのように使用されるかを見ていきます。

```
data Node = NNum Int             -- Numbers
          | NAp Addr Addr        -- Applications
          | NGlobal Int GmCode   -- Globals
          | NInd Addr            -- Indirections
          | NConstr Int [Addr]   -- Constructors
          | NLAp Addr Addr       -- Locked applications
          | NLGlobal Int GmCode -- Locked globals
```

演習 5.8 新しくロックされたノードを処理するために、showNode 関数を書き直します。

5.3.2　命令セット

必要な変更は、適切な場所でノードをロックおよびロック解除することだけです。アンワインド時に、関数適用ノードとスーパーコンビネータを引数なしでロックする必要があります。ノードが更新されると、そのノードの下のスパインにあるすべてのノードのロックを解除する必要があります。ヒープノードのロックとロック解除を実行するには、2 つの関数を使用します。lock 関数は、ロックされていないが更新可能な可能性のあるノードをロックされたノードに変換します。ロックする必要があるノードは、関数適用ノードと引数のないグローバルノードです。

```
lock :: Addr -> GmState -> GmState
lock addr state
  = putHeap (newHeap (hLookup heap addr)) state
    where
      heap = getHeap state
      newHeap (NAp a1 a2) = hUpdate heap addr (NLAp a1 a2)
      newHeap (NGlobal n c) | n == 0 = hUpdate heap addr (NLGlobal n c)
                            | otherwise = heap
```

ロックされた関数適用ノードを unlock する場合、そのノードが指すスパインもロック解除されていることを確認する必要があります。したがって、unlock は再帰的です。

```
unlock :: Addr -> GmState -> GmState
unlock addr state
  = newState (hLookup heap addr)
    where
      heap = getHeap state
      newState (NLAp a1 a2)
        = unlock a1 (putHeap (hUpdate heap addr (NAp a1 a2)) state)
      newState (NLGlobal n c)
        = putHeap (hUpdate heap addr (NGlobal n c)) state
      newState n = state
```

Unwind と Update の新しいステップ遷移は、lock と unlock の観点から定義する必要があります。これらの遷移は現在定義されています。遷移規則 5.2

では、関数適用ノード (*NAp で表される) をロックすること以外は、Unwind は Mark 1 マシンと同じ遷移規則を持つことがわかります。

(5.2)

$$\begin{array}{rcccccccccc} & \langle & h[a : \texttt{NAp}\ a_1\ a_2] & m & t & \rangle & \langle & [\texttt{Unwind}] & a : s & \rangle \\ \Longrightarrow & \langle & h[a : \texttt{*NAp}\ a_1\ a_2] & m & t & \rangle & \langle & [\texttt{Unwind}] & a_1 : a : s & \rangle \end{array}$$

同じことは、スタックの一番上にアリティ0 のスーパーコンビネータがある場合の Unwind の遷移規則にも当てはまります。この場合、*NGlobal はロックされたグローバルノードです。

(5.3)

$$\begin{array}{rcccccccccc} & \langle & h[a : \texttt{NGlobal}\ 0\ c] & m & t & \rangle & \langle & [\texttt{Unwind}] & a : s & \rangle \\ \Longrightarrow & \langle & h[a : \texttt{*NGlobal}\ 0\ c] & m & t & \rangle & \langle & c & a : s & \rangle \end{array}$$

Update の新しい遷移規則では、アドレスが a である redex のルートを更新するときに、a から派生するスパイン内のすべてのノードのロックを解除する必要があります。したがって、遷移規則は次の様になります。

(5.4)

$$\begin{array}{rcccccccccc} & \langle & h & \left[\begin{array}{l} a'_1 : \texttt{NGlobal}\ n\ c \\ a'_2 : \texttt{*NAp}\ a'_1\ a_1 \\ \cdots \\ a'_{n-1} : \texttt{*NAp}\ a'_{n-2}\ a_{n-2} \\ a_n : \texttt{*NAp}\ a'_{n-1}\ a_{n-1} \end{array}\right] & m & t & \rangle & \langle & \texttt{Update}\ n : i & a : a_1 : \ldots : a_n : s & \rangle \\ \Longrightarrow & \langle & h & \left[\begin{array}{l} a'_1 : \texttt{NGlobal}\ n\ c \\ a'_2 : \texttt{NAp}\ a'_1\ a_1 \\ \cdots \\ a'_{n-1} : \texttt{NAp}\ a'_{n-2}\ a_{n-2} \\ a_n : \texttt{NInd}\ a \end{array}\right] & m & t & \rangle & \langle & i & a_1 : \ldots : a_n : s & \rangle \end{array}$$

演習 5.9 遷移関数 unwind と update の定義を変更します。コードでは lock 関数と unlock 関数を使用する必要があります。また、スタックを再配置するときにロックされたノードを「調べる」必要があるため、getArg は次のようになります。

```
getArg (NLAp a1 a2) = a2
```

演習 5.10 あなたのマシンで次のプログラムを実行してみてください。

```
main = twice' (twice' (twice' (S K K))) 3
```

```
twice' f x = par f (f x)
```

演習 5.11 分割統治プログラムは、実行中の各プロセスで 30 命令ごとに `par` 命令を実行します。プロセスは終了しません。宇宙の電子 1 つにつき 1 つのタスクが実行されるまでに、シミュレートされたクロックティックが何回経過しますか? (ヒント: 宇宙には約 10^{85} 個の電子があります。)

演習 5.12 プロセッサのコストが £0.01 の場合、誰もそのマシンを購入できなくなるまで、演習 5.11 のプログラムはどのくらい実行できますか? (ヒント: 米国の連邦予算は約 £5×10^{12} です。)

5.4 Mark 3: 現実的な並列 G マシン

演習 5.11 と 5.12 で示したように、現実世界では並列処理の量には物理的および経済的な制限があります。使用している並列処理モデルはあまり現実的ではありません。各並列タスクを実行するために新しいプロセッサを作成していますが、現実世界ではすぐにリソースが不足します。

5.4.1 スケジューリングポリシー

より現実的なモデルでは、並列処理の量をハードウェアによって提供される量に制限する必要があります。これは、一度に実行できるプロセッサの数に上限を設定することで簡単に実現できます。その結果、タスクを実行するために使用できるプロセッサがない場合、タスクは変更されません。

タスクよりもプロセッサの数が多い場合、一部のプロセッサはアイドル状態になります。一方、逆の場合は、次にどのタスクを実行するかを決定する問題に直面します。この決定は、**スケジューリングポリシー**と呼ばれます。

5.4.2 保守的な並列性と投機的な並列性

いくつかのタスクは他のタスクよりも重要になります。タスクは、次の 2 つのグループに分類できます:

- 結果が確実に必要になるタスク
- 結果が必要になる可能性があるタスク。

最初のカテゴリのタスクを**保守的タスク**と呼び、2 番目のカテゴリのタスクを**投機的タスク**と呼びます。

投機的並列タスクを許可することを選択した場合、タスクのスケジュール設定における優先順位の問題に対処する必要があります。つまり、タスクを重要度順にランク付けする必要があります。また、同じ優先順位を持つ異なるタスクに同じ計算時間を許可する必要があります。これが望ましい理由を理解するには、判別式 `e0` の評価と並行してケース式の 2 つの分岐 (`e1` と `e2`) を評価することを検討してください。

```
case e0 of
     <1> -> e1
     <2> -> e2
```

e0 の評価が完了するまで、e1 と e2 のどちらが必要になるかはわかりません。したがって、それぞれを同量評価するようにするのが理にかなっています。このようなスケジューリングポリシーは、**公平なスケジューリング**と呼ばれます。この例では、e0 が完了すると、タスクの 1 つ (e1 または e2) が必要になり、もう 1 つは強制終了する必要があることに注意してください。したがって、タスクの実行中にタスクの優先順位を調整する必要があります。

この本では、投機的並列処理に適したマシンを実装する試みは行いません。これは「難しい問題」であるという言い訳をして、この問題はそのままにしておきます。今後は、 par プリミティブの使用はすべて、保守的な並列処理のみを生み出す状況で発生するものと想定します。

Mark 3 並列 G マシンでは、マシン内のアクティブなタスクの数は限られています。これらはマシン内のプロセッサによって実行されます。プロセッサの数は固定されています。この固定数は machineSize で、現在は 4 に設定されています。

```
machineSize :: Int
machineSize = 4
```

評価ツールの主な変更点は steps 関数にあります。これは、作成されたすべてのタスクをマシンに追加するだけではありません。代わりに、scheduler を使用して、続行できないタスクの代替タスクを選択します。

1. まず、グローバル状態コンポーネントからタスクプール内のスパークを抽出します。
2. 次に、スパークに対して新しいタスクが作成され、すでに実行中のタスクに追加されます。
3. 次に、scheduler 関数が実行するタスクを選択します。

steps をコード化する方法は次のとおりです。

```
steps :: PgmState -> PgmState
steps state
  = scheduler global' local'
    where ((out, heap, globals, sparks, stats), local) = state
          newtasks = [makeTask a | a <- sparks]
          global' = (out, heap, globals, [], stats)
          local' = local ++ newtasks
```

スケジューリングポリシーは非常にシンプルです。最初の machineSize タスクを選択して実行します。これらのタスクは、スケジューリングキューの最後に配置されます。このスケジューリングポリシーは通常、**ラウンドロビン**スケジューリングポリシーと呼ばれます。

```
scheduler :: PgmGlobalState -> [PgmLocalState] -> PgmState
scheduler global tasks
  = (global', nonRunning ++ tasks')
    where running = map tick (take machineSize tasks)
          nonRunning = drop machineSize tasks
          (global', tasks') = mapAccuml step global running
```

演習 5.13 実行されたタスクが次のステップのスケジュールキューの最後に配置されていない場合、何が起こるでしょうか。(ヒント: 試してみてください!)

演習 5.14 改善できる点の 1 つは、アイドル状態のプロセッサがある場合にのみ、スパークプールからタスクを作成することです。これを行うには、steps を変更します。

演習 5.15 もう 1 つの改善点は、続行可能なタスクのみをスケジュールすることです。現時点では、すでに WHNF にあるノードを評価するタスクをスケジュールします。ロックされたノードをアンワインドしようとしているためにブロックされているタスクもスケジュールします。これが発生しないように scheduler を変更します。

演習 5.16 他のスケジューリング戦略の使用を調査します。たとえば、タスクプールの最後のタスクをスケジュールしてみます。スケジューリング戦略によってプログラムの実行時間に違いが生じますか?

5.5　Mark 4: ブロックを処理するためのより良い方法

これまで、ブロックされたタスクはマシンのローカル状態に残し、scheduler 関数で実行可能なタスクを選択する必要がありました。つまり、scheduler 関数は、実行可能なタスクを見つけるまでに、かなりの数のブロックされたタスクをスキップしなければならない可能性があります。

これよりも良い方法があります! ブロックされたタスクを、ブロックの原因となったノードにアタッチする方がはるかに良いでしょう。ロックされたノードは任意の数のタスクをブロックする可能性があるため、ロックされたノードにタスクのリストを配置できるようにする必要があります。これを**保留リスト**と呼びます。

保留リストはどのように使用しますか?

1. ノードがロックされると、保留リストは [] に設定されます。
2. タスクがロックされたノードに遭遇すると、タスクはロックされたノードの保留リストに自身を配置します。
3. ロックされたノードがロック解除されると、保留リスト内のすべてのタスクが実行可能になり、マシンのローカル状態に転送されます。

Mark 4 マシンを実装するには、マシンのデータ構造に次の変更を加える必要があります。

5.5.1　データ構造

まず、ロックされた各ノードには保留リストが必要です。つまり、node のデータ型は次のようになります。

```
data Node = NNum Int                              -- Numbers
          | NAp Addr Addr                         -- Applications
          | NGlobal Int GmCode                    -- Globals
          | NInd Addr                             -- Indirections
          | NConstr Int [Addr]                    -- Constructors
          | NLAp Addr Addr PgmPendingList         -- Locked applications
          | NLGlobal Int GmCode PgmPendingList -- Locked globals
```

保留リストは単なるタスクのリストです。ロックされたノードの保留リスト内のタスクは、ロックされたノードでブロックされているタスクになります。

```
type PgmPendingList = [PgmLocalState]
```

演習 5.17 node の新しい定義で動作するように showNode を変更します。

もう 1 つの変更は、gmSparks の型です。以前の並列 G マシンのようにアドレスのリストではなく、タスクのリストになりました。

```
type GmSparks = [PgmLocalState]
```

演習 5.18 par の遷移規則を変更して、タスクをアドレスではなくスパークプールに配置するようにします。showSparks 関数を変更してスパークプール内のタスクの数を出力する必要があります。また、スパークプール内の項目をタスクに変換する必要がなくなったため、steps 関数を変更する必要があります。

5.5.2 ロックとロック解除

ノードのロックとロック解除を処理する新しい方法を構築してきました。まず、ロックされたノードが更新されるときに何が起こるかを考えてみましょう。Update 命令は、unlock の呼び出しを使用して実装されます。上書きされる式のスパイン内の各ノードでは、保留リスト内のタスクがスパークプールに転送される必要があります。これを行うには、タスクをスパークプールに転送する次の関数を定義します。

```
emptyPendingList :: [PgmLocalState] -> GmState -> GmState
emptyPendingList tasks state
  = putSparks (tasks ++ getSparks state) state
```

演習 5.19 unlock 関数を変更して、保留リスト内のタスクをスパークプールに移して空にします。

ロック操作は、Unwind 命令の一部として行われます。以前と同様に、ロック操作を実行するには lock 関数を使用します。ここで、新しくロックされた各ノードに空の保留リストを与える必要があります。

演習 5.20 ロックされたノードに空の保留リストを与えるように lock 関数を変更します。

最後に、タスクがロックされたノードを解除しようとしたときに何が起こるかについて説明します。明らかに、タスクをノードの保留リストに配置します。しかし、タスクを何に置き換えるのでしょうか? step の型は次のようになります。

```
step :: gmState -> gmState
```

私たちが採用した解決策は、タスクを emptyTask に置き換えることです。

```
emptyTask :: PgmLocalState
emptyTask = ([], [], [], [], 0)
```

したがって、Unwind には 2 つの新しい遷移規則が必要です。まず、ロックされた関数適用ノードの遷移規則から始めます。この遷移では、タスクがノードの保留リストに配置され、タスクが emptyTask に置き換えられます。

(5.5)

$$\begin{array}{lllllllll} & \langle & h[a : \texttt{*NAp}\ a_1\ a_2\ pl] & m & t & \rangle & \langle & [\texttt{Unwind}] & a : s & \rangle \\ \Longrightarrow & \langle & h[a : \texttt{*NAp}\ a_1\ a_2\ \langle[\texttt{Unwind}],\ a{:}s\rangle{:}pl] & m & t & \rangle & & \texttt{emptyTask} & & \end{array}$$

ロックされたグローバルノードのルールも同様です。タスクはノードの保留リストに配置され、それ自体が emptyTask に置き換えられます。

(5.6)

$$\begin{array}{lllllllll} & \langle & h[a : \texttt{*NGlobal}\ 0\ c\ pl] & m & t & \rangle & \langle & [\texttt{Unwind}] & a : s & \rangle \\ \Longrightarrow & \langle & h[a : \texttt{*NGlobal}\ 0\ c\ \langle[\texttt{Unwind}],\ a{:}s\rangle{:}pl] & m & t & \rangle & & \texttt{emptyTask} & & \end{array}$$

演習 5.21 Unwind 関数を変更して、Unwind 命令の新しい遷移を実装します。getArg 関数も作成する必要があります。

```
getArg (NLAp a1 a2 pl) = a2
```

演習 5.22 実行されていないタスクをスパークプールに配置するように scheduler 関数を変更します。

演習 5.23 doAdmin 関数を変更して、ローカル状態から emptyTask を除外します。

5.6　結論

この章では、原則として、遅延関数型言語の共有メモリ実装は簡単であることを示しました。もちろん、Mark 1 マシンで最初に使用した単純なスキームの最適化を慎重に検討することで得られる利益があることもわかりました。すべての並列マシンにおいて、グラフは通信および同期媒体として機能します。また、Mark 2 および Mark 3 マシンでは、ヒープ内のロックされたノードにアクセスしようとすると、個々のプロセスがブロックされます。

それでは、遅延関数型言語の並列実装における現在の課題はどこにあるのでしょうか? 本書に含まれる並列処理のメカニズムでは、プロセスの削除は処理されません。投機的な並列処理を使用する場合は、現実的な実装でこの問題に対処する必要があります。一方、非投機的並列処理を見つけることは多くの場合困難であり、大規模なプログラムではこれが困難になる場合もあります。この目的のために抽象的な解釈を使用する試みがなされており、その結果は有望に見えますが、暫定的なものとみなされるべきです。

私たちがカバーしていない最後の領域の 1 つは、分散メモリ並列マシンの領域です。繰り返しますが、原理的には共有メモリマシンに似ていますが、実際の機能はまったく異なります。デッドロックを回避するようにメッセージの受け渡しを調整するのは、一種の黒魔術です。

第6章　ラムダリフティング

6.1　はじめに

この章では、本書で以前に見たマシンに受け入れられる一連のプログラムを拡張する方法を見ていきます。導入する拡張機能は、**ローカル関数定義**を許可することです。次に、代替アプローチに直面します。

- 環境に対処するためのメカニズムをマシンに追加します。
- ローカル関数定義がないようにプログラムを変換します。代わりに、すべての関数がスーパーコンビネータとして定義されます。

この本では、常に 2 番目のアプローチが取られると想定してきました。

この章は、**完全な遅延評価**の概念を紹介する適切なポイントでもあります。ここでも、関数型言語のこの望ましい最適化は、プログラム変換を使用して実現されます。

6.2　`expr` データ型の改善

プログラムを本格的に始める前に、言語モジュールとユーティリティモジュールをインポートする必要があります。

```
module Lambda where
import Utils
import Language
```

残念ながら、そこで定義されているデータ型 (CoreExpr、CoreProgram など) は、この章でのニーズを満たすには柔軟性が不十分なので、まずこの問題に取り組みます。多くのコンパイラパスは抽象構文木に情報を追加するため、この情報を表す体系的な方法が必要です。この種の情報を生成する分析の例としては、自由変数分析、バインディングレベル分析、型推論、厳密性分析、共有分析などがあります。

このような情報を追加する最も明白な方法は、Expr データ型に注釈用の新しいコンストラクタを追加することです。

```
Expr a = EVar Name
       | ...
       | EAnnot Annotation (Expr a)
```

必要に応じて拡張できる注釈用の補助データ型とともに:

```
Annotation ::= FreeVars (Set Name)
             | Level Int
```

これにより、構文ツリー全体に注釈を自由に表示できるようになり、非常に柔軟に見えます。実際には、2 つの大きな欠点があります。

- 先ほど説明した形式で注釈情報を**追加する**のは簡単ですが、前のパスで配置された情報を**使用する**コンパイラパスを記述するのは実に厄介です。たとえば、パスが、前のパスでツリーの各ノードに残された自由変数情報を使用したいとします。おそらくこの情報は各ノードのすぐ上に添付されますが、データ型によって各ノードの上に複数の注釈ノードが表示されることが許可され、さらに悪いことに、自由変数注釈がまったく存在しない (または 1 つ以上する) 可能性があります。

  プログラマが、各ツリーノードの上に 1 つの注釈ノードがあり、それが自由変数注釈であることを証明する準備ができている場合でも、実装は注釈を抽出するときにパターンマッチングを実行してこれらのアサーションをチェックします。これらの問題、つまりチェックできないプログラマのアサーションの必要性と実装のオーバーヘッドは、注釈が付けられたすべてのツリーが、どの注釈が存在するかについて何も言わない、あまり情報のない型 Expr を持っているという事実に直接起因します。

- 2 つ目の大きな問題は、さらに実験を進めると、2 つの異なる形式の注釈が必要であることがわかったことです。1 つ目は上記のように式に注釈を付けますが、2 つ目は変数のバインディング出現に注釈を付けます。

つまり、let(rec) 定義の左辺の出現と、ラムダ抽象化または case 式内のバインドされた変数です。これらの出現を**バインダ**と呼びます。バインダに注釈を付ける必要がある例として、型推論が挙げられます。型推論では、コンパイラが各バインダと各部分式の型を推論します。

式注釈を使用してバインダに注釈を付けることもできますが、これは扱いにくく不便です。

簡単な解決策があるため、最初に 2 番目の問題に対処します。セクション 1.3 から、Expr 型がバインダの型に関して**パラメータ化された**ことを思い出してください。ここでは、リマインダーとしてその定義を繰り返します。

```
data Expr a
  =  EVar Name                    -- Variables
   | ENum Int                     -- Numbers
   | EConstr Int Int              -- Constructor tag arity
   | EAp (Expr a) (Expr a)        -- Applications
   | ELet                         -- Let(rec) expressions
        IsRec                     --   boolean with True = recursive,
        [(a, Expr a)]             --   Definitions
        (Expr a)                  --   Body of let(rec)
   | ECase                        -- Case expression
        (Expr a)                  --   Expression to scrutinise
        [Alter a]                 --   Alternatives
   | ELam [a] (Expr a)            -- Lambda abstraction
    deriving (Text)
```

CoreExpr 型は、バインダが Name 型の特殊な形式の Expr です。これは、型シノニムを使用して表現されます (セクション 1.3 からも繰り返されます)。

```
type CoreExpr = Expr Name
```

パラメータ化された Expr の利点は、他の特殊な形式を定義できることです。たとえば、typedExpr は、バインダが型で注釈された名前であるデータ型です。

```
typedExpr = expr (name, typeExpr)
```

ここで、typeExpr は型式を表すデータ型です。

式の注釈に戻ると、注釈型に関して式のデータ型をパラメーター化することにより、同じ手法を再利用できます。ツリーのすべてのノードに注釈を付

けたいので、1 つの可能性は、注釈情報を使用してすべてのコンストラクタに追加のフィールドを追加することです。たとえば、ルートノードでケース分析を実行せずに特定の式の上部に自由変数情報を抽出したい場合、これは不便です。これは、次のアイデアにつながります。

> ツリーの各レベルはペアで、最初のコンポーネントが注釈であり、
> 2 番目のコンポーネントは抽象的な構文ツリーノードです。

対応する Haskell のデータ型定義は次のとおりです。

```
type AnnExpr a b = (b, AnnExpr' a b)

data AnnExpr' a b = AVar Name
                  | ANum Int
                  | AConstr Int Int
                  | AAp (AnnExpr a b) (AnnExpr a b)
                  | ALet Bool [AnnDefn a b] (AnnExpr a b)
                  | ACase (AnnExpr a b) [AnnAlt a b]
                  | ALam [a] (AnnExpr a b)

type AnnDefn a b = (a, AnnExpr a b)

type AnnAlt a b = (Int, [a], (AnnExpr a b))

type AnnProgram a b = [(Name, [a], AnnExpr a b)]
```

AnnExpr と AnnExpr' 間の相互再帰によって、ツリー内のすべてのノードがアノテーションを持つことが保証されていることに注目してください。式が持つアノテーションの種類は、式の型に明示されます。例えば、自由変数でアノテーションされた式の型は AnnExpr Name (Set Name)) です。

AnnExpr' と Expr が本質的に同一のコンストラクタ群を二つ定義しなければならないというのは実に厄介な問題です。Hindley-Milner 型システムでは、この問題を回避する方法はなさそうです。Expr a は AnnExpr a b とほぼ同型なので、Expr 型を完全に廃止することも可能ですが、そうしないことにしたのには 2 つの理由があります。第一に、後者 (AnnExpr a b) は (() [1] , ⊥) と ⊥ を区別しますが、前者 (Expr a) は区別しないため、この 2 つの型は完全に同型ではありません。第二に (そしてより深刻なことですが)、AnnExpr a b 型のツリーを構築したりパターンマッチングしたりする際に、すべての () を書くのは非常に面倒です。

[1] () はユニット。

これで、中心となるデータ型の開発は完了です。これまでの議論を通して、現代の関数型プログラミング言語が提供する代数的データ型の長所と短所がいくつか明らかになりました。

演習 6.1 1.5 節で定義した現在の整形出力関数 pprint は、coreProgram のみを出力できます。ラムダリフターの中間段階を出力するには、program * 型の値を表示できる関数 pprintGen が必要です。('Gen' は 'generic' の略です。) pprintGen には、バインダの出力方法を指定するための追加の引数が必要です。

```
pprintGen :: (String -> Iseq)  -- function from binders to iseq
             -> Program Name   -- the program to be formatted
             -> [Char]         -- result string
```

例えば、pprintGen を記述したら、それを使って pprint を定義できます。

```
pprint prog = pprintGen iStr prog
```

pprintGen とそれに関連する関数 pprExprGen などの定義を記述しなさい。

演習 6.2 annProgram * ** 型の値を出力するためにも同様の処理を行います。ここでは、バインダのフォーマット用とアノテーションのフォーマット用の 2 つの追加引数が必要になります。

```
pprintAnn :: (a -> Iseq)        -- function from binders to iseq
             -> (b -> Iseq)     -- function from annotations to iseq
             -> annProgram a b  -- program to be displayed
             -> [Char]          -- result string
```

pprintAnn とそれに関連する補助関数の定義を記述しなさい。

6.3 Mark 1: シンプルなラムダリフター

レキシカルスコープのプログラミング言語の実装は、関数や手続きが**自由変数**を持つ可能性があるという事実に対処する必要があります。何らかの方法で自由変数を削除しない限り、環境ベースの実装ではリンクされた環境フレームを操作する必要があり、簡約ベースのシステムは置換時に名前変更を

行う必要があるため、大幅に複雑になります。特にグラフ簡約の実装において、これらの問題を回避する一般的な方法は、ラムダリフティングと呼ばれる変換によって関数定義からすべての自由変数を削除することです。ラムダリフティングという用語は [Johnsson 1985] によって造られましたが、この変換は [Hughes 1983] によって独自に開発されました。

ここでの文脈では、ラムダリフティングはコア言語のプログラムを、埋め込まれたラムダ抽象化のない同等のプログラムに変換します。簡単な例として、次のプログラムを考えてみましょう。

```
f x = let g = \y. x*x + y in (g 3 + g 4)
main = f 6
```

\y 抽象化は、x を追加パラメータとして受け取り、その本体が問題となっている抽象化である新しいスーパーコンビネータ $g を定義し、\y 抽象化を $g の適用に置き換えることで削除できます。これにより、次のスーパーコンビネータ定義のセットが得られます。

```
$g x y = x*x + y
f x = let g = $g x in (g 3 + g 4)
main = f 6
```

なぜ x だけを $g の追加パラメータにすることにしたのでしょうか? x は抽象化 \y. x*x + y の**自由変数**であるため、このようになりました。

> **定義** 式 e 内の変数 v の出現が e 内のその変数を囲むラムダ式または let(rec) 式によって束縛されていない場合、その出現は e 内で**自由**であると言われる。

一方、y は (\y. x*x + y) では自由ではありません。これは、y の出現が、それを束縛するラムダ抽象が囲むスコープ内にあるためです。

再帰が絡んでも、問題はそれほど複雑ではありません。g が再帰的であると仮定すると、次のようになります。

```
f x = letrec g = \y. cons (x*y) (g y) in g 3
main = f 6
```

これで、x と g は両方とも \y 抽象化内で自由になったため、ラムダリフターはそれらを両方とも $g の追加パラメーターにして、次のスーパーコンビネータのセットを生成します。

```
$g g x y = cons (x*y) (g y)
f x = letrec g = $g g x in g 3
main = f 6
```

g の定義は依然として再帰的ですが、ラムダリフターによって局所的なラムダ抽象化が排除されていることに注意してください。このプログラムは、ほとんどのコンパイラバックエンド、特に本書で紹介するすべての抽象マシンによって直接実装できるようになりました。

最後にもう一つ補足しておきます。他のトップレベル関数を追加の引数として扱う必要はありません。例えば、次のプログラムを考えてみましょう。

```
h p q = p*q
f x = let g = \y. (h x x) + y in (g 3 + g 4)
main = f 6
```

ここでは、h を \y 抽象化の自由変数として扱いたくありません。なぜなら、h は新しい $g スーパーコンビネータの本体から直接参照できる定数だからです。もちろん、+ 関数と * 関数にも同じことが当てはまります。つまり、スーパーコンビネータの引数、およびラムダ抽象化や let(rec) 式で束縛された変数だけが、自由変数の候補となります。

ついでに言えば、レキシカルスコープの問題は関数型言語に限った話ではありません。例えば、Pascal では関数を別の関数内でローカルに宣言することが可能で、その関数は外側のスコープによって束縛された自由変数を持つことができます。一方、C 言語ではこのようなローカルな関数定義は許可されていません。副作用がなければ、ローカルな関数定義をグローバルなものにするのは簡単です。必要なのは、自由変数を追加のパラメータとして追加し、すべての呼び出しにこれらのパラメータを追加することだけです。まさにこれがラムダリフティングの目的です。

6.3.1 シンプルなラムダリフターの実装

これで、シンプルなラムダリフターを開発する準備が整いました。coreProgram を受け取り、ELam コンストラクターが出現しない同等の coreProgram を返します。

```
lambdaLift :: CoreProgram -> CoreProgram
```

ラムダリフターは 3 つのパスで動作します。

- まず、式内のすべてのノードに自由変数をアノテーションします。これは、次のパスでラムダ抽象化に追加するパラメータを決定するために使用されます。`freeVars` 関数の型は以下の通りです。

```
freeVars :: CoreProgram -> AnnProgram Name (Set Name)
```

  `set *` 型は集合の標準的な抽象データ型であり、その定義は付録 A.4 に示されています。

- 次に、関数 `abstract` は各ラムダ抽象化 $\backslash x_1 \ldots x_n . e$ から自由変数 $v_1 \ldots v_m$ を抽象化し、ラムダ抽象化を次の形式の式に置き換えます。

$$\texttt{(let sc = } \backslash v_1 \ldots v_m \; x_1 \ldots x_n . e \texttt{in sc)} \; v_1 \ldots v_m$$

  ラムダ抽象化を自由変数に直接適用することもできますが、新しいスーパーコンビネータに名前を付ける必要があるため、ここでは最初のステップとして常に `sc` という名前を付けます。例えば、ラムダ抽象化

```
(\x. y*x + y*z)
```

  は

```
(let sc= (\y z x. y*x + y*z) in sc) y z
```

  に変換されます。`abstract` の型シグネチャは次のとおりです。

```
abstract :: AnnProgram Name (Set Name) -> CoreProgram
```

  型シグネチャから、`abstract` によって不要になった自由変数情報が削除されていることに注意してください。

- ここで、プログラムを走査し、各変数に一意の名前を付けます。これにより、前のパスで導入されたすべての `sc` 変数がユニークになります。実際、最初に追加の `let` 式を導入した唯一の目的は、各スーパーコンビネータにユニークな名前を与えることでした。副作用として、プログラム内の他のすべての名前もユニークになりますが、これは問題ではなく、後で役立つことがわかります。

```
rename :: CoreProgram -> CoreProgram
```

- 最後に、collectSCs はすべてのスーパーコンビネータの定義を 1 つのリストに収集し、プログラムの最上位レベルに配置します。

```
collectSCs :: CoreProgram -> CoreProgram
```

ラムダ リフター自体は、次の 3 つの関数を合成したものです。

```
lambdaLift = collectSCs . rename . abstract . freeVars
```

何が起こっているかを簡単に確認できるように、パーサとプリンタを統合する関数 runS ('S' は 'simple' の略) を定義します。

```
runS = pprint . lambdaLift . parse
```

もちろん、すべての作業を 1 つのパスで実行することも可能ですが、パスを分離することで提供されるモジュール性には、いくつかの利点があります。個々のパスは理解しやすくなり、パスは再利用可能になり (たとえば、以下では freeVars を再利用します)、モジュール性によってアルゴリズムを多少変更しやすくなります。

最後の点の例として、一部のコンパイラは collectSCs パスを省略することでより良いコードを生成できます。これは、スーパーコンビネータが適用されるコンテキストについてより多くの情報がわかるためです。 [Peyton Jones 1991] 例えば、abstract パスによって生成される可能性のある次の式を考えてみましょう。

```
let f = (\v x. v-x) v
in ...f...f...
```

ここで abstract は、自由変数である v を \x 抽象化から削除しています。[2] スーパーコンビネータをそのコンテキストから独立してコンパイルするのではなく、コンパイラは f のクロージャを構築し、そのコードでクロージャから v に直接アクセスし、スタックから x にアクセスすることができます。したがって、f の呼び出しで v をスタックに移動する必要はありません。自由変数が多ければ多いほど、この利点は大きくなります。また、定義がローカルなものであるため、f の呼び出しの効率が低下することもありません。コンパイラは f の束縛を認識し、そのコードに直接ジャンプできます。

以下の節では、これらの各パスの定義を示します。演習 6.4 に示されている case 式の等式は省略します。

[2] スーパーコンビネータに名前を付けるために abstract が導入する let 式は無視します。

6.3.2　自由変数

自由変数パスの中核は、関数 freeVars_e です。型シグネチャは以下の通りです。

```
freeVars_e :: (Set Name)                 -- Candidates for free variables
              -> CoreExpr                -- Expression to annotate
              -> AnnExpr Name (Set Name) -- Annotated result
```

最初の引数はスコープ内のローカル変数の集合です。これらは自由変数として利用可能なものです。2 番目の引数は注釈を付ける式であり、結果は注釈を付けられた式です。メイン関数 freeVars は、スーパーコンビネータ定義のリストを順に実行し、それぞれに freeVars_e を適用します。

```
freeVars prog = [ (name, args, freeVars_e (setFromList args) body)
                | (name, args, body) <- prog
                ]
```

freeVars_e 関数は式を再帰的に実行します。数値の場合は自由変数がないため、これが注釈付き式で返されます。

```
freeVars_e lv (ENum k) = (setEmpty, ANum k)
```

変数の場合は、候補集合内にあるかどうかを確認して、空集合を返すか、単一要素集合を返すかを決定します。

```
freeVars_e lv (EVar v) | setElementOf v lv = (setSingleton v, AVar v)
                       | otherwise         = (setEmpty, AVar v)
```

関数適用の場合は簡単です。まず、式 e1 をその自由変数で注釈付けし、次に e2 を注釈付けして、2 つの自由変数集合の和集合を EAp e1 e2 の自由変数として返します。

```
freeVars_e lv (EAp e1 e2)
  = (setUnion (freeVarsOf e1') (freeVarsOf e2'), AAp e1' e2')
    where e1' = freeVars_e lv e1
          e2' = freeVars_e lv e2
```

ラムダ抽象化の場合、渡されたローカル変数に引数を追加し、渡された自由変数から引数を減算する必要があります。

```
freeVars_e lv (ELam args body)
  = (setSubtraction (freeVarsOf body') (setFromList args), ALam args body')
    where body'  = freeVars_e new_lv body
          new_lv = setUnion lv (setFromList args)
```

let(rec) 式の等式は、かなり複雑な構造になっていますが、非常に単純です。本体に渡されるスコープ内のローカル変数は body_lv です。各右辺に渡されるローカル変数の集合は rhs_lv です。次に、各右辺に自由変数集合 rhss' をアノテーションします。これにより、アノテーション付き定義 defns' を構築できます。let(rec) のアノテーション付き本体は body' です。定義の自由変数は defnsFree として計算され、本体の自由変数は bodyFree として計算されます。

```
freeVars_e lv (ELet is_rec defns body)
  = (setUnion defnsFree bodyFree, ALet is_rec defns' body')
    where binders      = bindersOf defns
          binderSet    = setFromList binders
          body_lv      = setUnion lv binderSet
          rhs_lv | is_rec    = body_lv
                 | otherwise = lv
          rhss'        = map (freeVars_e rhs_lv) (rhssOf defns)
          defns'       = zip2 binders rhss'
          freeInValues = setUnionList (map freeVarsOf rhss')
          defnsFree | is_rec    = setSubtraction freeInValues binderSet
                    | otherwise = freeInValues
          body'        = freeVars_e body_lv body
          bodyFree     = setSubtraction (freeVarsOf body') binderSet
```

defns' の定義にある関数 zip2 は、2 つのリストを受け取り、引数リストの対応する要素のペアからなるリストを返す標準関数です。setUnion、setSubtraction などの集合演算は utilities モジュールで定義されており、そのインターフェースは付録 A.4 に示されています。

case とコンストラクタ式の処理は保留します。

```
freeVars_e lv (ECase e alts) = freeVars_case lv e alts
freeVars_e lv (EConstr t a) = error "freeVars_e: no case for constructors"
freeVars_case lv e alts = error "freeVars_case: not yet written"
```

freeVarsOf と freeVarsOf_alt は単純な補助関数です。

```
freeVarsOf :: AnnExpr Name (Set Name) -> Set Name
freeVarsOf (free_vars, expr) = free_vars


freeVarsOf_alt :: AnnAlt Name (Set Name) -> Set Name
freeVarsOf_alt (tag, args, rhs)
  = setSubtraction (freeVarsOf rhs) (setFromList args)
```

6.3.3　スーパーコンビネータの生成

次のパスでは、自由変数で注釈が付けられた各ラムダ抽象化を、その自由変数に適用された新しい抽象化 (スーパーコンビネータ) に置き換えるだけです。

```
abstract prog = [ (sc_name, args, abstract_e rhs)
                | (sc_name, args, rhs) <- prog
                ]
```

いつものように、ほとんどの作業を実行する補助関数 abstract_e を定義します。

```
abstract_e :: AnnExpr Name (Set Name) -> CoreExpr
```

これは、自由変数情報で注釈が付けられた式を受け取り、各ラムダ抽象化を自由変数に適用した新しい抽象化に置き換えた式を返します。最初の 4 つのケースについてはあまり説明する必要はありません。これらは単に各式を再帰的に抽象化するだけです。

```
abstract_e (free, AVar v) = EVar v
abstract_e (free, ANum k) = ENum k
abstract_e (free, AAp e1 e2) = EAp (abstract_e e1) (abstract_e e2)
abstract_e (free, ALet is_rec defns body)
  = ELet is_rec [(name, abstract_e body) | (name, body) <- defns]
                (abstract_e body)
```

関数 foldll は標準関数で、付録 A.5 で定義されています。2 項関数 $\oplus$、値 b、リスト $xs = [x_1, \ldots, x_n]$ が与えられると、foldll b xs は $(\ldots((b \oplus x_1) \oplus x_2) \oplus \ldots x_n)$ を計算します。自由変数情報は不要になったため、パスによって破棄されることに注意してください。

最後に示すケースは、abstract_e 関数の核心です。まず、自由変数のリスト fvList を作成します。集合には暗黙的な順序付けが存在しないことを思い出してください。関数 setToList は要素に順序付けを誘導しますが、その順序はあまり重要ではありません。次に、新しいスーパーコンビネータを作成します。これには、次の処理が含まれます。

1. abstract_e をラムダ式の本体に適用する。
2. 元の引数リストの前に自由変数リストを追加することで、引数リストを拡張する。

次に、collectSCs が新しいスーパーコンビネータを検出できるように、それを let 式で囲みます。最後に、新しいスーパーコンビネータを各自由変数に順番に適用します。

```
abstract_e (free, ALam args body)
  = foldll EAp sc (map EVar fvList)
    where
      fvList = setToList free
      sc     = ELet nonRecursive [("sc",sc_rhs)] (EVar "sc")
      sc_rhs = ELam (fvList ++ args) (abstract_e body)
```

case 式とコンストラクタについては保留します。

```
abstract_e (free, AConstr t a) = error "abstract_e: no case for Constr"
abstract_e (free, ACase e alts) = abstract_case free e alts
abstract_case free e alts = error "abstract_case: not yet written"
```

abstract_e が (ELam args1 (ELam args2 body)) と (ELam (args1++args2) body) という 2 つの式を異なる方法で扱うことに注目すべきです。前者の場合、2 つの抽象化は別々に扱われ、2 つのスーパーコンビネータが生成されますが、後者の場合、スーパーコンビネータは 1 つしか生成されません。ラムダリフティングを実行する前に、直接ネストされた ELam をマージすることは明らかに有利です。これは、[Hughes 1983] で指摘された η 抽象化最適化と同等です。

6.3.4 すべての変数をユニークにする

次に、abstract によって導入されたすべての sc 変数が一意になるように、各変数を定義する必要があります。補助関数 rename_e は、古い名前を新しい名前にマッピングする環境、名前の供給、および式を受け取ります。rename_e は、使い果たした名前の供給と新しい式を返します。

```
rename_e :: ASSOC Name Name           -- Binds old names to new
            -> NameSupply             -- Name supply
            -> CoreExpr               -- Input expression
            -> (NameSupply, CoreExpr) -- Depleted supply and result
```

これで、各スーパーコンビネータ定義に rename_e を適用し、mapAccuml とともに名前の供給を配管することで、rename を rename_e の観点から定義できるようになりました。

```
rename prog
  = second (mapAccuml rename_sc initialNameSupply prog)
    where
      rename_sc ns (sc_name, args, rhs)
        = (ns2, (sc_name, args', rhs'))
          where
            (ns1, args', env) = newNames ns args
            (ns2, rhs') = rename_e env ns1 rhs
```

関数 newNames は、名前の供給と名前のリストを引数として受け取ります。この関数は、名前の供給から古い名前ごとに新しい名前を割り当て、使い果たした名前の供給、新しい名前のリスト、そして古い名前と新しい名前をマッピングする連想リストを返します。

```
newNames :: NameSupply -> [Name] -> (NameSupply, [Name], ASSOC Name Name)
newNames ns old_names
  = (ns', new_names, env)
    where
      (ns', new_names) = getNames ns old_names
      env = zip2 old_names new_names
```

rename_e の定義は、退屈ではありますが、分かりやすくなりました。変数に遭遇すると、環境からその変数を検索します。トップレベル関数と組み込み関数 (+ など) の場合、環境内でその変数の置換は見つからないため、既存の名前を使用します。

```
rename_e env ns (EVar v) = (ns, EVar (aLookup env v v))
```

数値も関数適用も簡単です。

```
rename_e env ns (ENum n) = (ns, ENum n)


rename_e env ns (EAp e1 e2)
  = (ns2, EAp e1' e2')
    where
      (ns1, e1') = rename_e env ns e1
      (ns2, e2') = rename_e env ns1 e2
```

ELam に遭遇したときは、newNames を使用して引数に新しい名前を作成し、newNames によって返されるマッピングを使用して環境を拡張する必要があります。

```
rename_e env ns (ELam args body)
  = (ns1, ELam args' body')
    where
      (ns1, args', env') = newNames ns args
      (ns2, body') = rename_e (env' ++ env) ns1 body
```

let(rec) 式も同様に動作します。

```
rename_e env ns (ELet is_rec defns body)
  = (ns3, ELet is_rec (zip2 binders' rhss') body')
    where
      (ns1, body') = rename_e body_env ns body
      binders = bindersOf defns
      (ns2, binders', env') = newNames ns1 binders
      body_env = env' ++ env
      (ns3, rhss') = mapAccuml (rename_e rhsEnv) ns2 (rhssOf defns)
      rhsEnv | is_rec = body_env
             | otherwise = env
```

case 式は練習問題として残しておきます。

```
rename_e env ns (EConstr t a) = error "rename_e: no case for constructors"
rename_e env ns (ECase e alts) = rename_case env ns e alts
rename_case env ns e alts = error "rename_case: not yet written"
```

6.3.5 スーパーコンビネータの収集

最後に、スーパーコンビネータに名前を付けて、それらをまとめて収集する必要があります。したがって、メイン関数 collectSCs_e は、見つかったスーパーコンビネータのコレクションと、変換された式を返す必要があります。

```
collectSCs_e :: CoreExpr -> ([CoreScDefn], CoreExpr)
```

collectSCs は、すべての配管を実行するために mapAccuml を使用して定義されます。

```
collectSCs prog
  = concat (map collect_one_sc prog)
    where
      collect_one_sc (sc_name, args, rhs)
        = (sc_name, args, rhs') : scs
          where
            (scs, rhs') = collectSCs_e rhs
```

collectSCs_e のコードは簡単に書けるようになりました。collectSCs_e を部分式に再帰的に適用し、生成されたスーパーコンビネータをまとめていくだけです。

```
collectSCs_e (ENum k) = ([], ENum k)

collectSCs_e (EVar v) = ([], EVar v)

collectSCs_e (EAp e1 e2) = (scs1 ++ scs2, EAp e1' e2')
                           where
                             (scs1, e1') = collectSCs_e e1
                             (scs2, e2') = collectSCs_e e2

collectSCs_e (ELam args body) = (scs, ELam args body')
                                where
                                  (scs, body') = collectSCs_e body
collectSCs_e (EConstr t a) = ([], EConstr t a)

collectSCs_e (ECase e alts)
  = (scs_e ++ scs_alts, ECase e' alts')
    where
      (scs_e, e') = collectSCs_e e
      (scs_alts, alts') = mapAccuml collectSCs_alt [] alts
      collectSCs_alt scs (tag, args, rhs) = (scs ++ scs_rhs, (tag, args, rhs'))
                                            where
                                              (scs_rhs, rhs') = collectSCs_e rhs
```

let(rec) の場合が興味深いです。定義を再帰的に処理し、それを 2 つのグループに分割する必要があります。$v = \backslash args.e$ という形式のもの (スーパーコンビネータ) と、それ以外のもの (非スーパーコンビネータ) です。スーパーコンビネータはスーパーコンビネータリストの一部として返され、残りの非スーパーコンビネータから新しい let(rec) が生成されます。

```
collectSCs_e (ELet is_rec defns body)
  = (rhss_scs ++ body_scs ++ local_scs, mkELet is_rec non_scs' body')
    where
      (rhss_scs, defns') = mapAccuml collectSCs_d [] defns
      scs'     = [(name,rhs) | (name,rhs) <- defns', isELam rhs]
      non_scs' = [(name,rhs) | (name,rhs) <- defns', not (isELam rhs)]
      local_scs = [(name,args,body) | (name,ELam args body) <- scs']
      (body_scs, body') = collectSCs_e body
```

```
collectSCs_d scs (name,rhs) = (scs ++ rhs_scs, (name, rhs'))
                              where
                                (rhs_scs, rhs') = collectSCs_e rhs
```

補助関数 isELam は ELam コンストラクタをテストします。これはスーパーコンビネータを識別するために使用されます。

```
isELam :: Expr a -> Bool
isELam (ELam args body) = True
isELam other = False
```

mkELet 関数は ELet 式を構築するだけです。

```
mkELet is_rec defns body = ELet is_rec defns body
```

6.4 Mark 2: シンプルなラムダリフターの改良

これで、単純なラムダリフターの定義は完了です。次に、いくつかの簡単な改良について検討します。

6.4.1 シンプルな拡張機能

演習 6.3 単純なラムダリフターは、collectSCs が abstract によって導入された各スーパーコンビネータ let 式から単一のバインディングを削除するため、バインディングのリストが空である let 式を多数生成します。これらの冗長な let 式を省略するように mkELet を変更します。

演習 6.4 freeVars_case、abstract_case、collectSCs_case の定義を与え、テストします。

6.4.2 冗長なスーパーコンビネータの排除

次のコア言語プログラムを検討します。

```
f = \x. x+1
```

これは lambdaLift によって次のように変換されます。

```
f = $f
$f x = x+1
```

冗長な定義の導入は避けた方が良いでしょう。この改善は、完全な遅延評価を考慮すると、この形式のスーパーコンビネータが多数導入されるため、より顕著になります。

演習 6.5 collectSCs 内の関数 collect_one_sc に特別なケースを追加し、rhs がラムダ抽象化の場合に異なる動作をするようにしなさい。この状況では、新しいスーパーコンビネータの導入を回避できるはずです。

6.4.3　冗長なローカル定義の削除

ローカル定義でも同様の状況が発生する可能性があります。次のコア言語プログラムについて考えてみましょう。

```
f x = let g = (\y. y+1) in g (g x)
```

ラムダリフターは次のプログラムを生成します。

```
f x = let g = $g in g (g x)
$g y = y+1
```

演習 6.6 collectSCs_d の定義 (ELet の collectSCs_e の場合) を改良し、右辺がラムダ抽象である定義に特別な処理を与えるようにしなさい。上記の例では、以下を生成すべきです。

```
f x = g (g x)
g y = y+1
```

6.5 Mark 3: ジョンソンスタイルのラムダリフティング

ラムダリフティング技法には興味深い変種があり、[Johnsson 1985] によって発見されました。現在の技法には、引数として渡される関数への呼び出しが多く含まれるプログラムが生成されてしまうという小さな問題があります。例えば、6.3 節の再帰例を考えてみましょう。

```
f x = letrec g = \y. cons (x*y) (g y) in g 3
main = f 6
```

現在のラムダリフターは、次のスーパーコンビネータの集合を生成します。

```
$g g x y = cons (x*y) (g y)
f x = letrec g = $g g x in g 3
main = f 6
```

`$g` が引数 `g` を呼び出すことに注意してください。実装によっては、次のように `$g` を直接再帰的に実行する方が効率的です。

```
$g x y = cons (x*y) ($g x y)
f x = $g x 3
main = f 6
```

内部の `letrec` は完全に消え、スーパーコンビネータ g は直接再帰的になりました。

ジョンソンスタイルのラムダリフティングのやり方をより詳しく理解するために、もう少し複雑な例を見てみましょう。

```
f x y = letrec
          g = \p. ...h...x...
          h = \q. ...g...y...
        in
        ...g...h...
```

ここで、`g` は `h` を呼び出し、変数 `x` に言及する関数です。同様に、`h` は `g` を呼び出し、変数 `y` に言及します。最初のステップは、定義を次のように変換することです。

```
f x y = letrec
          g = \x y p. ...(h x y)...x...
          h = \x y q. ...(g x y)...y...
        in
        ...(g x y)...(h x y)...
```

この変換は**抽象化ステップ**と呼ばれていますが、少し複雑です。以下の処理が行われます。

- letrec の右辺の自由変数、つまり g、h、x、y を取ります。
- バインドされている変数 (g と h) を除外して、x と y だけを残します。
- これらの変数を各右辺の追加引数にします。
- そして、g をすべて（g x y）に置き換え、h についても同様です。

y は g の右辺に直接出現しませんが、g が h に渡すために y を必要とするため、y を g の追加パラメータにすることが重要です。一般に、相互再帰グループの各メンバーは、**すべての**メンバーの自由変数をまとめて追加引数として取る必要があります。

ここで必要なのは、g と h の定義をトップレベルにフロートして、次のコードを残すだけです。

```
f x y = ...(g x y)...(h x y)...
g x y p = ...(h x y)...x...
h x y q = ...(g x y)...y...
```

最後にもう 1 点。このプロセスを実行する前に、すべてのバインダーが一意であることが重要です。そうでない場合、2 つの理由で名前の衝突が発生する可能性があります。明らかなのは、2 つのスーパーコンビネータが同じ名前を持つことです。あまり明らかではない例として、同じ例の次のバリエーションを示します。

```
f x y = letrec
           g = \p. ...h...x...
           h = \x. ...g...y...
        in
        ...g...h...
```

すると、h は引数に f と同じ名前を使用するようになり、g の自由変数 x を h の追加引数にしようとすると問題が発生します。全体として、作業を開始する前にプログラムの名前を変更する方がはるかに簡単です。

6.5.1　実装

ジョンソンスタイルのラムダリフターは、複数のパスで実装できます。

- 最初のパスでは、すべてのバインダーが一意になるようにプログラムの名前を変更します。この目的のために、rename 関数を再利用できます。
- 次に、既存の関数 freeVars を使用して、プログラムに自由変数情報を注釈付けします。
- ここで、上で説明した主要な抽象化のステップに進みます。

```
abstractJ :: AnnProgram Name (Set Name) -> CoreProgram
```

- 最後に、collectSCs を使用してスーパーコンビネータを収集できます。

完全なジョンソンスタイルのラムダリフターは、次のステージの組み合わせです。

```
lambdaLiftJ = collectSCs . abstractJ . freeVars . rename
runJ = pprint . lambdaLiftJ . parse
```

6.5.2 関数内の自由変数の抽象化

必要な新しい関数は abstractJ だけです。抽象化プロセスは、処理が進むにつれて g を $g\ v_1 \ldots v_n$ に置き換えます。ここで g は新しいスーパーコンビネータの 1 つであり、$v_1, \ldots, v_n$ はその宣言グループの自由変数です。したがって、補助関数 abstractJ_e は、各スーパーコンビネータ g をそのグループ $v_1, \ldots, v_n$ の自由変数にマッピングする環境を受け取る必要があります。

```
abstractJ_e :: ASSOC Name [Name]          -- Maps each new SC to
                                          --   the free vars of its group
               -> AnnExpr Name (Set Name) -- Input expression
               -> CoreExpr                -- Result expression
```

公平に言えば、最初の引数は ASSOC Name CoreExpr 型になるように見えますが、後で説明するように、環境を別の方法でも使用する必要があり、それがここで提案する型につながります。

各トップレベルの定義に abstractJ_e を適用することで、abstractJ を abstractJ_e で簡単に定義できます。

```
abstractJ prog = [ (name,args,abstractJ_e [] rhs)
                 | (name, args, rhs) <- prog]
```

さて、abstractJ_e について見てみましょう。定数と関数適用のケースは簡単です。

```
abstractJ_e env (free, ANum n) = ENum n
abstractJ_e env (free, AConstr t a) = EConstr t a
abstractJ_e env (free, AAp e1 e2) = EAp (abstractJ_e env e1)
                                        (abstractJ_e env e2)
```

変数 g に遭遇すると、環境内でその変数を検索し、変数のリスト $v_1, \ldots, v_n$ を取得します。その後、適用結果 $g\ v_1 \ldots v_n$ を返します。g が環境に現れない場合は、環境検索から空のリストを返します。したがって、構築する「適用結果」は単に変数 g になります。

```
abstractJ_e env (free, AVar g)
  = foldll EAp (EVar g) (map EVar (aLookup env g []))
```

場合によっては、ラムダ抽象化自体が見つかることもあります。たとえば、次のようになります。

```
f xs = map (\x. x+1) xs
```

\x 抽象化は let(rec) 定義の右辺ではないので、単純なラムダリフターの場合と同じように扱います (abstract の ELam ケースを参照)。

重要な違いが 1 つだけあります。abstractJ_e は式に対して同時に置換を実行するため、自由変数情報は置換後の状態を反映しません。むしろ、結果の中でどの変数が自由であるかを調べるために、自由変数集合に対しても置換を実行する必要があります。これは、このセクションの最後で定義されている関数 actualFreeList によって行われます。自由変数情報に対してこの操作を実行する必要性が、環境表現の選択を導きました。

```
abstractJ_e env (free, ALam args body)
  = foldll EAp sc (map EVar fv_list)
    where
      fv_list = actualFreeList env free
      sc = ELet nonRecursive [("sc",sc_rhs)] (EVar "sc")
      sc_rhs = ELam (fv_list ++ args) (abstractJ_e env body)
```

最後に、let(rec) 式を扱います。ラムダ抽象に束縛された各変数はスーパーコンビネータに変換されますが、ラムダ抽象に束縛されていない他の変数はスーパーコンビネータに変換されません。したがって、これら 2 種類の定義を別々に扱う必要があり、それぞれ「関数定義」と「変数定義」と呼びます。

```
abstractJ_e env (free, ALet isrec defns body)
  = ELet isrec (fun_defns' ++ var_defns') body'
```

```
    where
      fun_defns = [(name,rhs) | (name,rhs) <- defns, isALam rhs]
      var_defns = [(name,rhs) | (name,rhs) <- defns, not (isALam rhs)]
```

関数定義と変数定義を分離したので、関数から抽象化する変数の集合を計算できます。関数定義の自由変数の和集合を取り、この集合から束縛される関数名を取り除き、actualFreeList (ELam 定義式と同じ理由) を使用して結果を取得します。

```
fun_names = bindersOf fun_defns
free_in_funs = setSubtraction
                    (setUnionList [freeVarsOf rhs | (name,rhs)<-fun_defns])
                    (setFromList fun_names)
vars_to_abstract = actualFreeList env free_in_funs
```

次に、let(rec) の右辺と本体で使用される新しい環境を計算します。

```
body_env = [(fun_name, vars_to_abstract) | fun_name <- fun_names] ++ env
rhs_env | isrec = body_env
        | otherwise = env
```

最後に、適切な環境で abstractJ_e を再帰的に使用して、新しい関数定義、変数定義、および本体を計算します。

```
fun_defns' = [ (name, ELam (vars_to_abstract ++ args)
                           (abstractJ_e rhs_env body))
             | (name, (free, ALam args body)) <- fun_defns
             ]
var_defns' = [(name, abstractJ_e rhs_env rhs) | (name, rhs) <- var_defns]
body' = abstractJ_e body_env body
```

関数 actualFreeList は環境と自由変数の集合を受け取り、その集合に環境を適用し、置換後の自由変数のリスト (重複なし) を返します。

```
actualFreeList :: ASSOC Name [Name] -> Set Name -> [Name]
actualFreeList env free
  = setToList (setUnionList [ setFromList (aLookup env name [name])
                            | name <- setToList free
                            ])
```

関数 isALam は ALam コンストラクタを識別します。

```
isALam :: AnnExpr a b -> Bool
isALam (free, ALam args body) = True
isALam other                  = False
```

これでジョンソンスタイルのラムダリフターは完了です。

演習 6.7 関数 abstractJ_e に case 式のケースを追加します。

6.5.3　難しい点 †

letrec に関数定義と変数定義が混在している場合、設計したラムダリフターによって冗長なパラメータが導入される可能性があります。例えば、次の定義を考えてみましょう。

```
f x y = letrec
          g = \p. h p + x ;
          h = \q. k + y + q;
          k = g y
        in
        g 4 ;
```

g / h グループの自由変数は x、y、k なので、次のように変換します。

```
f x y = letrec
           k = g x y k y
        in
        g 4


g x y k p = h x y k p + x ;
h x y k q = k + y + q;
```

ここで、追加のパラメータ x は実際には h では使用されません。したがって、g と h のより良い定義は次のようになります。

```
g x y k p = h y k p + x ;
h y k q = k + y + q;
```

演習 6.8 † このより洗練された変換を実行するには、abstractJ を修正してください。警告: これはかなり難しい作業です。

6.6 Mark 4: 完全な遅延処理パスの分離

ここで、関数型プログラムの重要な特性である**完全遅延性**に注目します。完全遅延性に関するこれまでの説明では、常にラムダリフティングが関連付けられており、「完全遅延ラムダリフティング」と説明されていますが、これはかなり複雑なプロセスであることがわかります。[Hughes 1983] はアルゴリズムを示していますが、非常に微妙で、`let(rec)` 式を扱っていません。一方、[Peyton Jones 1987] は `let(rec)` 式を扱っていますが、その説明は非公式なもので、アルゴリズムは示されていません。

このセクションでは、完全な遅延処理とラムダ式による処理を明確に分離する方法を示します。これは、`let` 式を用いた変換によって行われます。ある方法で単純化したものが、別の方法で複雑化しただけだと誤解されないよう、`let(rec)` 式を使用せずに完全な遅延処理のラムダ式処理を実行すると、予期せぬ遅延処理の損失が発生するリスクがあることも示します。さらに、ほとんどのコンパイラでは、`let(rec)` 式に対して、同等のラムダ式よりもはるかに効率的なコードを後の段階で生成できます。

6.6.1 完全な遅延処理のレビュー

まず、完全遅延処理の概念を簡単に確認してみましょう。6.3 節で示した例をもう一度考えてみましょう。

```
f x = let g = \y. x*x + y in (g 3 + g 4)
main = f 6
```

シンプルなラムダリフターは次のプログラムを生成します。

```
$g x y = x*x + y
f x = let g = $g x in (g 3 + g 4)
main = f 6
```

`f` の本体には `g` と `$g` が 2 回呼び出されています。しかし (`$g x`) は簡約可能な式ではないため、`x*x` は 2 回計算されます。しかし、`x` は `f` の本体で固定されているため、一部の作業が重複しています。`x*x` の計算を 2 回の `$g` の呼び出しで共有する方が適切です。これは次のように実現できます。`x` を `$g` の引数にする代わりに、`x*x` を引数にします。

```
\$g p y = p + y
f x = let g = \$g (x*x) in (g 3 + g 4)
```

(`main` の定義は変更されないため、以降は省略します。) したがって、完全遅延評価ラムダリフターは、ラムダ抽象の各自由変数ではなく、各最大自由部分式を、対応するスーパーコンビネータの引数に変換します。ラムダ抽象の最大自由式 (または MFE) とは、抽象化によって束縛される変数の出現を含まない式であり、この特性を持つより大きな式の部分式ではありません。

完全な遅延評価は、ループ不変式をループの外側に移動する操作に正確に対応し、ループの反復ごとに 1 回ではなく、開始時に 1 回だけ計算されます。

完全な遅延性は「実際の」プログラムにとってどれほど重要なのでしょうか？この問題については、まだ本格的な研究は行われていませんが、私たちはそうする予定です。しかしながら、Holst による最近の研究は、完全な遅延性の重要性は当初考えられていたよりも大きい可能性があることを示唆しています [Holst 1990]。彼は、完全な遅延性の効果を自動的に高める変換を実行する方法を示しており、得られる最適化は部分評価 [Jones et al. 1989] によって得られる最適化に匹敵しますが、その労力ははるかに少なくなります。

6.6.2 Full-lazy lambda lifting in the presence of `let(rec)`s

6.6.3 Full laziness without lambda lifting

6.6.4 A full lazy lambda lifter

6.6.5 Separating the lambdas

6.6.6 Adding level numbers

6.6.7 Identifying MFEs

6.6.8 Renaming variables

6.6.9 Floating `let(rec)` expressions

6.7 Mark 5: Improvements to full laziness

6.7.1 Adding `case` expressions

6.7.2 Eliminating redundant supercombinators

6.7.3 Avoiding redundant full laziness

6.8 Mark 6: Dependency analysis †

6.8.1 Strongly connected components

6.8.2 Implementing a strongly connected component algorithm

6.8.3 A dependency analysis

6.9 結論

完全な遅延評価に対する私たちのアプローチを、同様の問題を扱った Bird の非常に優れた論文 [Bird 1987] と比較するのは興味深いことです。Bird の目的は、初期仕様を連続的に変換することによって、効率的な完全遅延ラムダリフターを形式的に開発することです。結果として得られるアルゴリズムはかなり複雑で、直接書き出すのは難しいため、形式的な開発の努力は十分に正当化されます。

対照的に、私たちはアルゴリズムを多数の非常に単純なフェーズの構成として表現しており、各フェーズは簡単に指定して直接書き出すことができます。結果として得られるプログラムは、式を何度も走査するため、定数因数の非効率性を伴います。これは、連続するパスを 1 つの関数に折りたたんで、中間データ構造を排除することで簡単に削除できます。Bird の変換とは異なり、これは簡単なプロセスです。

私たちのアプローチにはモジュール式であるという大きな利点があります。例えば：

- いくつかの場合 (`freeVars`、`rename`、`collectSCs` など) に既存の関数を再利用することができました。
- マルチパスアプローチは、各パスに明確な単純な目的があることを意味し、変更が容易になります。たとえば、完全な遅延評価が導入される場所についてより選択的になるように、`identifyMFEs` アルゴリズムを変更しました (セクション 6.7.3)。
- 主要フェーズをさまざまに組み合わせて使用し、さまざまな変換を「スナップ」することができます。たとえば、トップレベルで適切な関数を構成するだけで、依存関係分析と完全な遅延処理を実行するかどうか、およびどのラムダリフターを使用するかを選択できます。

私たちのアプローチの主な欠点は、Hughes が提案した 1 つの最適化、つまりパラメータをスーパーコンビネータに順序付けして MFE の数を減らすことができないことです。その理由は、最適化ではラムダリフティングを MFE 識別のプロセスに組み込むことが絶対に必要である一方で、これらのアクティビティを慎重に分離しているためです。私たちにとって幸いなことに、この最適化によって作成されたより大きな MFE は常に部分的なアプリケーションであり、作業が共有されないため、おそらく MFE として識別されるべきではありません (セクション 6.7.3)。それでも、事態はそれほど偶然に起こったわけではないかもしれません。関心事の分離により、ある種の変革がかなり困難になっているのは確かです。

付 録A　ユーティリティモジュール

この Appendix では、本全体で使用されるさまざまな便利な型と関数の定義を示します。

```
module Utils where

-- The following definitions are used to make some synonyms for routines
-- in the Gofer prelude to be more Miranda compatible
shownum n = show n
hd :: [a] -> a
hd = head -- in Gofer standard prelude
tl :: [a] -> [a]
tl = tail -- in Gofer standard prelude
zip2 :: [a] -> [b] -> [(a,b)]
zip2 = zip -- in Gofer standard prelude
-- can't do anything about # = length, since # not binary.
```

A.1　The heap type

抽象データ型の heap と address は、実装ごとにガベージコレクションされたノードのヒープを表すために使用されます。

A.1.1　Specification

A.1.2　Representation

A.2 The association list type

A.3 Generating unique names

A.3.1 Representation

A.4 Sets

A.4.1 Representation

A.5 Other useful function definitions

付 録B　コア言語プログラムの例

この Appendix では、この本で開発された実装のいくつかをテストするのに役立ついくつかのコア言語プログラムを示します。それらは、プレリュード (セクション 1.4) で定義された関数が定義されていると想定しています。

B.1　基本プログラム

このセクションのプログラムでは、整数定数と関数の適用のみが必要です。

B.1.1　超基本テスト

このプログラムは、かなり早く値 3 を返すはずです!

```
main = I 3
```

次のプログラムでは、3 を返す前にさらにいくつかの手順が必要です。

```
id = S K K ;
main = id 3
```

これはかなりの数の id の関数適用を作成します (いくつ?)。

```
id = S K K ;
main = twice twice twice id 3
```

B.1.2　更新のテスト

このプログラムは、更新を行うシステムと行わないシステムの違いを示すはずです。更新が発生した場合、(I I I) の評価は 1 回だけ行う必要があります。更新しないと、2 回行われます。

```
main = twice (I I I) 3
```

B.1.3 もっと興味深い例

この例では、リストの関数表現 (セクション 2.8.3 を参照) を使用して 4 の無限リストを作成し、次にその 2 番目の要素を取ります。head と tail の関数 (hd と tl) は、引数が空のリストの場合、abort を返します。abort スーパーコンビネータは、無限ループを生成するだけです。

```
cons a b cc cn = cc a b ;
nil      cc cn = cn ;
hd list = list K abort ;
tl list = list K1 abort ;
abort = abort ;
infinite x = cons x (infinite x) ;
main = hd (tl (infinite 4))
```

B.2 let と letrec

更新が実装されている場合、id1 の評価が共有されるため、このプログラムは実装されていない場合よりも少ないステップで実行されます。

```
main = let id1 = I I I
       in id1 id1 3
```

ネストされた let 式もテストする必要があります。

```
oct g x = let h = twice g
          in let k = twice h
          in k (k x) ;
main = oct I 4
```

次のプログラムは、以前の cons や nil などの定義に基づく「関数リスト」を使用して、letrec をテストします。

```
infinite x = letrec xs = cons x xs
             in xs ;
main = hd (tl (tl (infinite 4)))
```

B.3 算術演算

B.3.1 条件分岐なし

まずは、条件式を必要としない簡単なテストから始めます。

```
main = 4*5+(2-5)
```

この次のプログラムは、正しく動作させるために関数呼び出しが必要です。`twice twice` を `twice twice twice` または `twice twice twice twice` に置き換えてみてください。どのような結果になるかを予想してください。

```
inc x = x+1;
main = twice twice inc 4
```

再び関数型リストを使って、length 関数を書きます。

```
length xs = xs length1 0 ;
length1 x xs = 1 + (length xs) ;
main = length (cons 3 (cons 3 (cons 3 nil)))
```

B.3.2 条件分岐あり

条件分岐ができれば、ようやく「面白い」プログラムが書けるようになります。例えば、階乗。

```
fac n = if (n==0) 1 (n * fac (n-1)) ;
main = fac 5
```

次のプログラムは、ユークリッドのアルゴリズムを用いて、2 つの整数の最大公約数を計算するものです。

```
gcd a b = if (a==b)
                a
          if (a<b) (gcd b a) (gcd b (a-b)) ;
main = gcd 6 10
```

`nfib` 関数は、その結果（整数）が、実行中に何回関数が呼び出されたかをカウントしてくれるから面白いです。そのため、結果を実行時間で割ると、1 秒あたりの関数呼び出し回数で性能を表すことができます。その結果、`nfib` はベンチマークとしてかなり広く使われています。しかし、特定の実装の「nfib-number」は、様々な特殊な最適化に決定的に依存しているため、非常に大きな塩梅で受け取る必要があります。

```
nfib n = if (n==0) 1 (1 + nfib (n-1) + nfib (n-2)) ;
main = nfib 4
```

B.4 データ構造

このプログラムは、降順の整数のリストを返します。評価者は、プログラムの結果としてリストを期待する必要があります。cons と nil は、それぞれ Pack{2,2} と Pack{1,0} としてプレリュードに実装されることが期待されています。

```
downfrom n = if (n == 0)
                 nil
                 (cons n (downfrom (n-1))) ;
main = downfrom 4
```

次のプログラムは、エラトステネスのふるいを実装して素数の無限リストを生成し、結果リストの最初のいくつかの要素を取ります。出力が生成されたときにインクリメンタルに出力されるように調整する場合は、take の呼び出しを削除して、無限リストを出力するだけで済みます。

```
main = take 3 (sieve (from 2)) ;

from n = cons n (from (n+1)) ;

sieve xs = case xs of
               <1> -> nil ;
               <2> p ps -> cons p (sieve (filter (nonMultiple p) ps)) ;

filter predicate xs
        = case xs of
               <1> -> nil ;
               <2> p ps -> let rest = filter predicate ps
                           in
                           if (predicate p) (cons p rest) rest ;

nonMultiple p n = ((n/p)*p) ~= n ;

take n xs = if (n==0)
               nil
               (case xs of
                      <1> -> nil ;
                      <2> p ps -> cons p (take (n-1) ps))
```