13章　プログラム論証

haskellでプログラミングを論証する方法を以下の順序で紹介していきます。

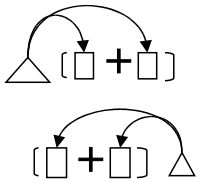
[1]等式推論の復習をする。
[2]等式推論をhaskellにどう適用するかを示す。
[3]数学的帰納法という重要な技法を説明する。
[4]連結演算子(++)を数学的帰納法と似た手法で除去して高速化する方法を示す。
　(プログラムの運算)
[5]簡単なコンパイラーの正しさを証明する。(プログラム検証論)

[1]等式推論
学校数学にて下記のような代数的性質を学びました。
数式の場合

xy	**= ys**	**---(乗算の交換法則)**
x+(y+z)	**= (x+y)+z**	**---(加算の結合法則)**
x(y+z)	**=xy+xz**	**---(左の分配法則)**
(x+y)z	**=xz+yx**	**---(右の分配法則)**

上記性質を使うと(x+a)(x+b)という形の積は次のように展開できます。

(x+a)(x+b)	
=(x+a)x+(x+a)b	(∵(左の分配法則))
=xx+ax+xb+ab	(∵(右の分配法則))
=xx+ax+bx+ab	(∵(乗算の交換法則))
=x^2+ax+bx+ab	(乗算)
=x^2+(a+b)x+ab	(∵(右の分配法則))



意味
∵　なぜならば

また、これらの性質は実際の計算に対して重要な意味を持ちます。
例えば、(a)は(乗算1回と加算1回)が必要なのに対して(b)は(乗算2回と加算1回)
が必要になります。
これらは、代数的には等しいが効率が異なり(a)の方が効率が高くなります。

x(y+z)	--(a)
xy+xz	--(b)

[2]haskellでの論証
1]の式の論証のように、haskellに対しても等式推論が使用できます。

実際の例として次の関数doubleを考えます。
double : : Int -> Int
double x = x + x　　--(*)

プログラムに対して論証する際は、次の(a)(b)の両方の置き換え方が可能です。
(a)関数の定義を左辺から右辺へ置換える『適用』
　あるプログラムに(*)を適用して *double x* を *x + x* に置き換える。
(b)関数の定義を右辺から左辺へ置換える『逆適用』
　あるプログラムに(*)を適用して ***x + x*** を *double x* に置き換える。

具体的な計算については、後にでできます。

複数の等式を用いて定義された関数に関して論証する場合は
等式が書かれた順番が重要となり、等式を適用・逆適用するには条件が必要となります。

例えば、isZero関数の定義式(a)(b)でみると、

```
isZero :: Int -> Bool
isZero 0 = True          --(a)
isZero n = False         --(b)
```

(a)は常に適用できるが、(b)はn=0以外の場合のみ適用できます。
これは、パターンマッチをするのに上から順番にマッチが試されるためです。
(a)、(b)を下記のように書き直すとマッチする順番に依存しないバターンとなります。
これを【互いに素】あるいは【重複なし】と言います。

```
isZero 0 = True          --(a)
isZero n | n /= 0  = False    --(b')
```

(b')のように定義すれば、ガードn /= 0が満たされるときに限って
isZero n は False に置き換えられ、*False は isZero n* へ変換可能となります。

[簡単な例題]
等式推論の簡単な例を見ていきます。

```
-- reverse関数
reverse :: [a] -> [a]
reverse []     = []                    --(A1)
reverse (x:xs) = reverse xs ++ [x]     --(A2)

-- reverseのある性質の証明例
reverse [x] = [x]                      --(13.3.証明1)

reverse [x]
= reverse (x:[])
= reverse [] ++ [x]                    (∵(A2)の適用)
= [] ++ [x]                            (∵(A1)の適用)
= [x]
```

よって(13.3.証明1)が成立つ。
QED.

横道:【Q. E. D. はラテン語の Quod Erat Demonstrandum(かく示された)が
略されてできた頭字語。証明や論証の末尾におかれ、議論が終わったことを示す。】

上記証明結果から、reverse [x] を [x]と置き換えても意味が変わらない。
また関数reverseの適用がなくなる分、効率的になることが分った。

等式推論は、場合分けと共に使うこともよくあります。
下に例を示します。

```
-- 否定演算子の定義
not :: Bool -> Bool
not False = True                --(B1)
not True  = False               --(B2)

-- 否定の否定をとると元に戻ることの証明
not (not b) = b                 --(13.3.証明2)

b = False のとき
not (not False)
= not True                      (∵(B1))
= False                         (∵(B2))

b = True のとき
not (not True)
= not False                     (∵(B2))
= True                          (∵(B1))
```

よって(13.3.証明2)が成立つ。
QED.

なぜ等式推論が必要か？
本章では等式推論を使用して①②を行っています。
①プログラムの検証（→コンパイラの検証）
②プログラム(アルゴリズム?)の高速化の手順化　　（→ ++演算子の削除による高速化）

①プログラムの検証について
　プログラムの信頼性を確かめることはテストにより行うが、
　ダイクストラは、『テストでバグの存在は示せるがバグのないことは示せない』と言っていた。
　（場合の数から(1-n)/nの確率で正しい動作をするとか {この場合、値の偶発的一致は避けられない}
　成長曲線からバグの推定数を割り出して大体バグが取れたとかしている。）

　一方で、TDDはテストを重ねながらプログラミングをおこない、用意したテストが全部通ったら
　プログラム完成と言うアプローチだ。と誤解しやすいが実は単なる開発プロセスに対する提案である
　欲しいのはバグはあるかも知れないが取りあえず動作する綺麗なコードではなく、
　まずバグのないコードだ。

　この点を改善するにはどうするか。
　　↓
　プログラムを証明すればよいのかもしれない？

　横道：プログラム検証ツールを使用して正しいことを証明しながらプログラミングを行う
　　PDD(Proof driven development)がある。
　　（現状ではバラ色の道ではなくイバラの道みたいですが）

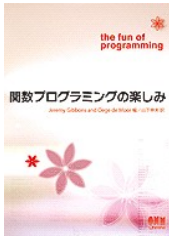
②プログラム(アルゴリズム?)の高速化の手順化
胡助教授(東大)が研究しているプログラムの運算と目的が同じだと思われます。
■プログラムの運算とは
　プログラム運算とは，プログラムを，その意味を変えない運算規則（定理）を用いて，
　その仕様から，より効率のよいプログラムに書き換えていくプログラミング手法である．

つまり、「ナイーブで非効率だけど仕様にあってることはわかりやすいプログラム」から
最終的に「仕様にあってるかどうか明らかではないけれど効率的なプログラム」までを
繋げるのが「運算」となります。

横道：　この「このプログラムとこのプログラムの挙動は等価だよね」で
　　じわじわ変形していくスタイルは、他の証明法と比べてかなり証明が理解しやすい、
　　さらに、〝二次方程式といえば平方完成〟みたいなノリで、
　　プログラムを効率化する式変形のパターン みたいなものが見いだせたら、
　　新しい問題に対する新しいアルゴリズムを系統的に見つける役にも立つかも。

　　プログラムを作成する作業を、数学の連立方程式や力学の運動方程式のように
　　方程式を立てたら、あとは手順通り自動的に解ける方法と同じように
　　素朴で分りやすいプログラムを作成したら、後は自動的に効率的なプログラムが
　　組み立てられる。
　　このような研究をおこなっているようです。

以上のテクニックのさわりの部分をプログラムの論証で行って行きます。
簡単すぎて面白くない例題かもしれませんが、プログラムの検証論は
大変で、簡単な例で勉強するのが良いのでこのような出題となっていると思います。
発展的題材は、日本語の出版物でも下記のものがあります



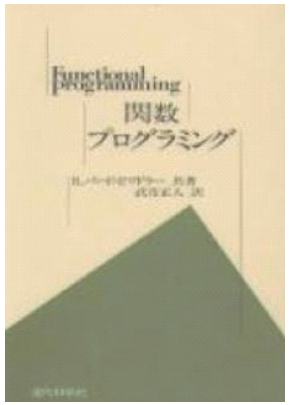
関数プログラミングの楽しみ

5章　融合変換を自動化する。

数理工学への誘い

5章　プログラム運算の数理

関数プログラミング



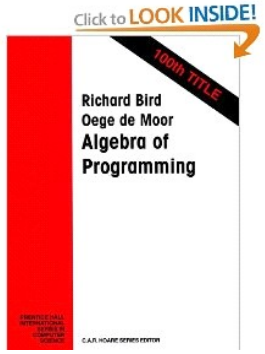
5.4　補助定理と一般化など

この本はプログラミングhaskellよりページ数があるが大体
同じレベルの論証の話が出ています。また重なっている部分
も多いです。

ソフトウエア発展

コンピュータソフトウエア　別冊

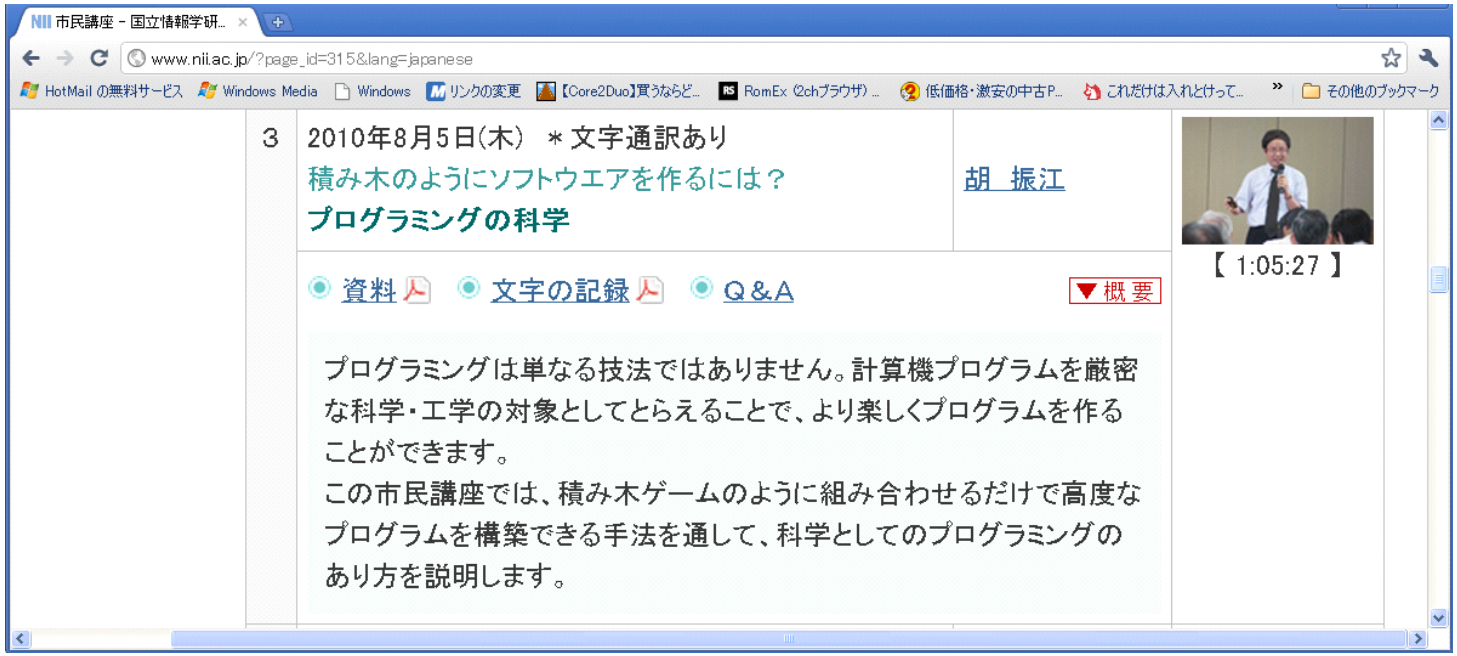
漸近的組化と融合による関数プログラムの最適化



Algebra of Programming

定番本のようです。
全く見ていません。


NII 市民講座 - 国立情報学研...
www.nii.ac.jp/?page_id=315&lang=japanese
HotMail の無料サービス
Windows Media
Windows
リンクの変更
【Core2Duo】買うならど...
RomEx (2chブラウザ) ...
低価格・激安の中古P...
これだけは入れとけって...
その他のブックマーク
NII
大学共同利用機関法人 情報・システム研究機構
国立情報学研究所
National Institute of Informatics
サイトマップ ▶ お問い合わせ ▶ アクセス ▶ English
検索
トップ
研究者紹介
サービス・事業
NIIについて
一般の方
研究者の方
学生の方
大学・図書館の方
企業の方
NIIについて
研究所の概要
知的財産室
共同研究
研究協力
国際連携
図書室利用案内
国際高等セミナーハウス
プロジェクト・研究施設
成果・刊行物一覧
ニュース
イベント
研修・講習会
軽井沢土曜懇話会
市民講座
情報学オープンフォーラム
オープンハウス
国際シンポジウム
ワークショップ
2010年度のイベント
イベント一覧
図書館総合展
情報処理学会創立50周年記念大会
NII動画チャンネルβ
図書館総合展2009
アクセス
大学院教育
問い合わせ窓口
採用情報
トップ > NIIについて > イベント > 市民講座
NII 市民講座
公開講座:＜参加費無料＞
平成22年度市民講座 「未来を変える情報学」
平成22年度市民講座 第7回
※「文字通訳」を行います。
タイトル:
「マルチメディアと検索技術 ―キーボードを使わずに検索するには？―」
概 要:
文字のみならず映像や音声を含む情報はマルチメディア情報と呼ばれ、代表的なものにテレビ放送があります。マルチメディア情報は、家庭でも大量に蓄積できるほど身近な存在になりましたが、内容を解析したり、内容に基づいて検索する手法は発展途上であり、情報学における重要なフロンティアのひとつになっています。サーチエンジンで検索するのと同じように、映像や音声を検索するにはどのような方法が考えられるのか、情報学分野での取り組みを紹介します。
講 師: 片山 紀生 (国立情報学研究所 コンテンツ科学研究系准教授)
日 時: 2011年1月19日 (水) 18:30～19:45 (講義・質疑応答)
会 場: 学術総合センター 2階中会議場
国立情報学研究所
市民講座
未来を変える情報学
初めての方へ New! 市民講座の会場や講義の様子を市民講座の"鳥"がご紹介します！
市民講座アーカイブ (平成15年～)
これまでの市民講座開催一覧
平成22年度講座のご案内
当日の (1) 映像、(2) 資料、(3) 講義内容を入力した文字の記録、(4) Q＆A (講義時のご質問への回答) を終了後に公開します。
概 要 を 全て表示 | 全て隠す
回
開催日／講演テーマ
講 師
講義映像 【収録時間】
1
2010年6月3日(木)
人間の翻訳と機械の翻訳は何が違うのか？
多言語世界の扉を開く翻訳技術
関連サイト: みんなの翻訳
＊手話通訳と文字通訳を行いました
影浦 峡
【 1:31:25 】



URL
http://www.nii.ac.jp/?page_id=315&lang=japanese

私は決して本屋の回し者ではアリマセン ‼

[3]整数に対する数学的帰納法
10章で出てきた自然数の例題を使用します。

```
-- 自然数型の定義
data Nat = Zero | Succ Nat
-- 自然数の定義
1 : Zero
2 : Succ Zero
3 : Succ (Succ Zero)
4 : Succ (Succ (Succ Zero))
～
(to be continued ...)
```

ただし、式 inf = Succ Inf で定義される無限は取り扱わない。
また、以降の再帰型でも取り扱わない。

[すべて有限な自然数に対して成り立つ性質を証明の方法]
数学的帰納法に従えば、下記(1)(2)の手順となる。
(1) n = Zeroのときに性質が成り立つ事を示す。
(2) n = x のときに性質が成り立つと仮定して、n = Succ x の時に対しても
成り立つ事を示す。

具体例としてNat型に対する足し算を定義する関数をあげます。

```
add : : Nat -> Nat -> Nat
add Zero m      = m                          --(A1)
add (Succ n) m = Succ (add n m)        --(A2)
```

(A1)は、すべての自然数mにたいして
add Zero m = m が成り立つことを意味します。
これと対となる次の性質を導く。

add n Zero = n *--(13.4.証明1)*

以下、nに対して数学的帰納法を使い証明を行う。
基底部
n = Zero のとき

```
add Zero Zero
= Zero                                    (∵(A1))
```

再帰部
n = x のときに次式が成り立つと仮定する。

```
add x Zero = x                          --(仮定1)
```

n = Succ x のとき

```
add (Succ x) Zero
=Succ (add x Zero)                  (∵(A2))
=Succ x                                  (∵(仮定1))
```

よって、(13.4.証明1)が成り立つ。
QED.

別の例として自然数の加算に対する結合法則を証明する。

add x (add y z) = add (add x y) z　　--(13.4.証明2)

x、y、zのうち、どの自然数に対して数学的帰納法を用いればよいのだろうか？

(1)関数addは、1番目の引数に対してパターンマッチを用いて定義されていることに注目する。

(2)関数addの1番目の引数として xは2回、yは1回、zは使われていない。

上記(1)(2)の理由から x に対して数学的帰納法を使うのが自然である。

(13.4.証明2)の証明

x = Zero のとき

左辺

=add Zero (add y z)

=add y z　　(∵(A1))

右辺

=add (add Zero y) z

=add y z　　(∵(A1))

[再掲]

add : : Nat -> Nat -> Nat

add Zero m　　= m　　--(A1)

add (Succ n) m = Succ (add n m)　　--(A2)

x = n のとき、次式が成り立つと仮定する。

add n (add y z) = add (add n y) z --(仮定2)

x = Succ n のとき

左辺

=add (Succ n) (add y z)

=Succ (add n (add y z))　　(∵(A2))

=Succ (add (add n y) z)　　(∵(仮定2))

右辺

=add (add (Succ n) y) z

=add (Succ (add n y)) z　　(∵(A2))

=Succ (add (add n y) z)　　(∵(A2))

よって、(13.4.証明2)が成り立つ。

QED.

[再掲]

```
add : : Nat -> Nat -> Nat
add Zero m      = m                          --(A1)
add (Succ n) m = Succ (add n m)              --(A2)
```

add x (add y z) = add (add x y) z--(13.4.証明2)

これまでと同じ原理はhaskellの組み込み型である整数に対しても利用できる。
具体的に言うと、ある性質が0以上の整数すべてに対して成り立つと証明するには
下記(1)(2)を示せばよい。
(1)n = 0 のときに成り立つことを示す。
(2)n = x のときに成り立つと仮定して、n = x + 1 のときに成り立つことを示す。

以下、例としてreplicateを使用する。

```
replicate :: Int -> a -> [a]
replicate 0 _      = []                    --(B1)
replicate (n + 1) c = c:replicate n c      --(B2)
```

(replicate生成するリストの長さがnであることの証明)

length (replicate n c) = n　　*--(13.4.証明3)*

n = 0 のとき
左辺
=length (replicate 0 c)
=length []　　('.'(B1))
=0　　(length適用)

lengthの素朴な定義

```
replicate :: Int -> a -> [a]
length []     = 0
length x:xs   = c+length xs
```

ライブラリの定義

```
length = foldl (¥n _ -> n+1) 0
```

n = x のとき次式が成り立つと仮定する。

length (replicate x c) = x　　*--(仮定3)*

n = x + 1 のとき

左辺

=length (replicate (x + 1) c)

=length (c:replicate x c)　　(∵(B2))

=1 + length (replicate x c)　　(∵lengthの性質)

=1 + x　　(∵(仮定3))

=x + 1　　(∵ + の交換規則)

よって、(13.4.証明3)が成り立つ。

QED.

length (replicate n c) = n

--(13.4.証明3)

[再掲]

```
replicate :: Int -> a -> [a]
replicate 0 _      = []                          --(B1)
replicate (n + 1) c = c:replicate n c            --(B2)
```

lengthの素朴な定義

```
replicate :: Int -> a -> [a]
length []      = 0
length x:xs    = 1+length xs
```

[リストに対する数学的帰納法]
数学的帰納法は、自然数だけでなく、リストのような他の再帰型の論証にも利用できる。

(すべてのリストに対して、ある性質を証明する方法)
数学的帰納法の原理に従えば、下記(1)(2)の手順となる。
(1)その性質が、空リスト[]に対して成り立つことを示す。
(2)その性質が、リストxsに成り立つと仮定して、リストx:xsに対しても成り立つことを示す。

例として、reverseにたいして次式の性質を証明する。

reverse (reverse xs) = xs *--(13.5.証明4)*

-- reverse関数

reverse : : [a] -> [a]

reverse [] = [] **--(C1)**

reverse (x:xs) = reverse xs ++ [x] **--(C2)**

-- (++)演算子

(++) : : [a] -> [a] -> [a]

[] ++ ys = ys **--(X1)**

(x:xs) ++ ys = x:(xs ++ ys) **--(X2)**

-- reverseの分配則

reverse (xs ++ ys) = reverse ys ++ reverse xs ***--(分配則)***

(ここでは正しいとして分配則を使用し、後で分配則を証明する。)

((13.5.証明4)の証明)

基底部

xs = [] のとき

reverse (reverse [])

=reverse [] (∵(C1))

=[] (∵(C1))

再帰部
xs = ns のとき、次式が成り立つと仮定する。
reverse (reverse ns) = ns *--(仮定2)*

xs = n:ns のとき
reverse (reverse n:ns)
=reverse (reverse ns ++ [n]) (∵(C2))
=reverse [n] ++ reverse (reverse ns) (∵(分配則))
=reverse [n] ++ ns (∵(仮定2))
=[n] ++ ns (∵[2]の[簡単な例題]で示したreverse [x] = [x])
=(n:[]) ++ ns
=n:([] ++ ns) (∵(X2))
=n:ns (∵(X1))

よって、(13.5.証明4)が成り立つ。

QED.

reverse (reverse xs) = xs
--(13.5.証明4)

[再掲]
>-- reverse関数
>reverse [] = [] --(C1)
>reverse (x:xs) = reverse xs ++ [x] --(C2)
>-- reverseの分配則
>*reverse (xs ++ ys) = reverse ys ++ reverse xs*
--(分配則)
>-- (++)演算子
>[] ++ ys = ys --(X1)
>(x:xs) ++ ys = x:(xs ++ ys) --(X2)

補題

reverse (xs ++ ys) = reverse ys ++ reverse xs　　--(分配則)

-- reverse関数

reverse : : [a] -> [a]

reverse [] = []　　--(C1)

reverse (x:xs) = reverse xs ++ [x]　　--(C2)

-- (++)演算子

(++) : : [a] -> [a] -> [a]

[] ++ ys = ys　　--(X1)

(x:xs) ++ ys = x:(xs ++ ys)　　--(X2)

((分配則)の証明)

基底部

xs = [] のとき

左辺

=reverse ([] ++ ys)

=reverse ys　　(∵(X1))

右辺

=reverse ys ++ reverse []

=reverse ys ++ []　　(∵(C1))

=reverse ys　　(∵例題5の性質 xs ++ [] = xs)

再帰部

xs = ns のとき、次式が成り立つと仮定する。

reverse (ns ++ ys) = reverse ys ++ reverse ns　　--(仮定1)

xs = n:ns のとき

右辺　　***reverse (xs ++ ys) = reverse ys ++ reverse xs***

--(分配則)

=reverse ys ++ reverse (n:ns)

=reverse ys ++ (reverse ns ++ [n])　　(∵(C2))

左辺

=reverse ((n:ns) ++ ys)

=reverse (n:(ns ++ ys))　　(∵(X2))

=reverse (ns ++ ys) ++ [n]　　(∵(C2))

=(reverse ys ++ reverse ns) ++ [n]　　(∵(仮定1))

=reverse ys ++ (reverse ns ++ [n])　　(∵例題5の性質 xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
連結演算子の結合則)

以上より、(分配則)が成り立つ。

QED.

再掲

>-- reverse関数

>reverse []　　= []　　--(C1)

>reverse (x:xs) = reverse xs ++ [x]　　--(C2)

>-- (++)演算子

>[] ++ ys　　= ys　　--(X1)

>(x:xs) ++ ys = x:(xs ++ ys)　　--(X2)

[4]連結を除去する
多くの再帰関数は、リストの連結演算子 ++ を使うと自然な形で定義できる。
しかし、この演算子を利用すると、大変効率が悪い。
この節では、数学的帰納法を使って、連結演算子を取り除き、関数の効率を向上させる方法を示す。

例として reverse関数を取り上げる。

```
-- reverse関数
reverse :: [a] -> [a]
reverse []     = []                    --(A1)
reverse (x:xs) = reverse xs ++ [x]     --(A2)

-- (++)演算子
(++) :: [a] -> [a] -> [a]
[] ++ ys      = ys                     --(X1)
(x:xs) ++ ys = x:(xs ++ ys)            --(X2)
```

関数reverseの効率は、どれくらいだろうか ?
式 xs ++ ys を評価するための簡約の回数が、リスト xs と ys が完全に評価されていると
仮定できるならば、(xs の長さ + 1)となる。
(式 xs ++ ys を評価するための簡約の回数が、(xs の長さ + 1)となることの簡略説明)
仮にxsの要素数をnとして下記のように書くことにする
「x(n)」の「(n)」がタプルにも見えるがここではnが添え字となる意味です。
xs(n) = x(n):x(n−1): ・・・ :x(2):x(1):[]
別の書き方をすると
xs(n) = x(n):xs(n−1)

xs ++ ys
=x(n):xs(n−1) ++ ys
=x(n):(xs(n−1) ++ ys)　　(∵(X2))
=x(n):(x(n−1):xs(n−2) ++ ys)
=x(n):(x(n−1):(xs(n−2) ++ ys))　　(∵(X2))
～
=x(n):(x(n−1):(x(n−2):(・・・ x(2):(x(1):[] ++ ys))・・・))
=x(n):(x(n−1):(x(n−2):(・・・ x(2):(x(1):([] ++ ys))・・・))　　(∵(X2))
=x(n):x(n−1):x(n−2): ・・・ x(2):x(1):ys　　(∵(X1))

xs ++ ys の簡約回数はx(n)～x(1)の簡約と[]の簡約の回数 つまり、(xs の長さ + 1)となる。

このため、reverseにかかる簡約の回数は1からn+1までの整数の和
すなわち (n+1)(n+2)/2である。
展開すると(n^2+3n+2)/2となるから
関数reverseの効率 は (引数の長さ)の二乗に比例する。
これは、かなり効率が悪い。

再掲
-- (++)演算子
[] ++ ys　　= ys　　--(X1)
(x:xs) ++ ys = x:(xs ++ ys)　　--(X2)

[1からn+1までの整数の和の説明]

1からn+1までの整数の和をS(n+1)と書くことにする

$$S(n+1) = 1 + 2 + 3 + \cdots\cdots + (n-1) + n + (n+1) \quad \text{--(Y1)}$$

上の式の右辺の項を順序を逆にする。

$$S(n+1) = (n+1) + n + (n-1) + \cdots\cdots + 3 + 2 + 1 \quad \text{--(Y2)}$$

(Y1+Y2)を計算すると

$$2 \times S(n+1) = (n+2) + (n+2) + (n+2) + \cdots\cdots + (n+2) + (n+2) + (n+2)$$

$\Leftrightarrow 2 \times S(N) = (n+1)(n+2)$　　(∵(1)(2)はn+1項あるから)

$\Leftrightarrow S(n+1) = (n+1)(n+2)/2$

数学的帰納法に似た手法を用いれば、関数 reverse の定義から連結演算子を除去し
効率を向上させることが可能である。
その手段は、関数 reverse と ++ の振る舞いを組み合わせた汎用的な関数を定義することである。
具体的には、次の等式を満たす関数 reverse'の定義を模索する。

reverse' xs ys = reverse xs ++ ys　　--(Z1)

基底部

reverse' [] ys
=reverse [] ++ ys　　(∵(Z1))
=[] ++ ys　　(∵(A1))
=ys　　(∵(X1))

再帰部

reverse' (x:xs) ys
=reverse (x:xs) ++ ys　　(∵(Z1))
=(reverse xs ++ [x]) ++ ys　　(∵(A2))
=reverse xs ++ ([x] ++ ys)　　(∵++の結合則)
=reverse xs ++ (x:[] ++ ys)
=reverse xs ++ (x:([] ++ ys))　　(∵(X2))
=reverse xs ++ (x:ys)　　(∵(X1))
=reverse' xs (x:ys)　　(∵(Z1)の逆適用)

再掲

-- reverse関数
reverse []　　= []　　--(A1)
reverse (x:xs) = reverse xs ++ [x]　　--(A2)

-- (++)演算子
[] ++ ys　　= ys　　--(X1)
(x:xs) ++ ys = x:(xs ++ ys)　　--(X2)

この証明から次式で示す関数 reverse'の定義を得た。

reverse' :: [a] -> [a] -> [a]
reverse' [] ys　　= ys　　--(定義1)
reverse' (x:xs) ys = reverse' xs (x:ys)　　--(定義2)

(Z1)でysを[]とすると次式を得る。

```
reverse :: [a] -> [a]
reverse xs = reverse' xs []          --(定義3)
```

次にこの簡約例を示す。

```
reverse [1,2,3]
=reverse' [1,2,3] []                 ((定義3)を適用)
=reverse' [2,3] (1:[])               ((定義2)を適用)
=reverse' [3] (2:(1:[]))             ((定義2)を適用)
=reverse' [] (3:(2:(1:[])))          ((定義2)を適用)
=(3:(2:(1:[])))                      ((定義1)を適用)
```

再掲

```
reverse' xs ys = reverse xs ++ ys--(Z1)

reverse' [] ys      = ys             --(定義1)
reverse' (x:xs) ys = reverse' xs (x:ys)
                                     --(定義2)
```

すなわち、結果が補助変数に蓄えられながらリストが反転される。
この新しい reverse の定義はおそらく元の定義より分かりにくいだろう。
しかし、効率が劇的に向上している。
新しい定義を使って長さnのリストを反転されるために必要な簡約の回数は
単にn + 2回である。
したがって、引数の長さに比例する。

オーダー

O(n^2)　⇒　O(n)

次に連結演算子を除去する別例として、木構造に対して数学的帰納法に似た手法を用いてみよう。
二分木型とその木をリストに変換する関数flattenを考える。

```
data Tree = Leaf Int | Node Tree Tree
flatten :: Tree -> [Int]
flatten (Leaf n)   = [n]                    --(B1)
flatten (Node l r) = flatten l ++ faltten r  --(B2)
```

連結演算子を利用しているために、関数 flatten は効率が悪い。
つまり関数 flattenと++の振る舞いを組み合わせた汎用的な関数flatten'の定義を模索する。

```
flatten' t ns = flatten t ++ ns              --(Z2)
```

ある性質が任意の木に対して成り立つことを証明するためには、数学的帰納法の原理に従えば
まず、木 Leaf n に対して性質が成り立つことを示し、さらに木 l と r に対して性質が成り立つ
ことを示せばよい。
以下証明

基底部

```
flatten' (Leaf n) ns
=flatten (Leaf n) ++ ns          ((Z2)の適用)
=[n] ++ ns                       (∵(B1))
=n:[] ++ ns
=n:([] ++ ns)                    (∵(X2))
=n:ns                            (∵(X1))
```

再掲
-- (++)演算子

```
[] ++ ys      = ys                --(X1)
(x:xs) ++ ys = x:(xs ++ ys)       --(X2)
```

再帰部

flatten' (Node l r) ns
=flatten (Node l r) ++ ns　　((Z2)の適用)
=(flatten l ++ flatten r) ++ ns　　(∵(B2))
=flatten l ++ (flatten r ++ ns)　　(++の結合則)
=flatten l ++ (flatten' r ns)　(rに対して(Z2)の逆適用)
=flatten' l (flatten' r ns)　　(lに対して(Z2)の逆適用)

再掲

flatten' t ns = flatten t ++ ns　　--(Z2)
flatten :: Tree -> [Int]
flatten (Leaf n)　= [n]　　--(B1)
flatten (Node l r) = flatten l ++ faltten r
--(B2)

以上より、仕様を満たす関数 flatten'の定義が次のように得られた。
flatten' (Leaf n) ns　= n:ns
flatten' (Node l r) ns = flatten' l (flatten' r ns)

(Z2)でns=[]とすると
flatten' t [] = flatten t ++ []
⇔flatten t = flatten' t []

よって関数 flattenは次式のように再定義できる。
flatten :: Tree -> [Int]
flatten t = flatten' t []

おそらく元の定義よりも分かりにくいけれども、連結演算子の代わりに、結果を
蓄える補助変数を使うことで、劇的に効率が向上している。

[5]コンパイラの正しさ
10章では、整数と加算からなる数式の型と
式を直接評価して整数に直す関数evalを定義した。

```
-- eval 定義
data Expr = Val Int | Add Expr Expr
eval :: Expr -> Int
eval (Val n)   = n                          --(A1)
eval (Add x y) = eval x + eval y            --(A2)
```

上記式は、次式のようにスタックを用いて
間接的に評価する方法で定義することもできる。
スタックは整数のリストであり、コードはPUSH命令やADD命令のリストである。

```
type Stack = [Int]
type Code  = [Op]
data Op    = PUSH Int | ADD
```

上記コードの意味は、それを実行するexec関数を定義することで与えられる。

```
-- exec定義
exec :: Code -> Stack -> Stack
exec [] s           = s                     --(B1)
exec (PUSH n:c) s   = exec c (n:s)          --(B2)
exec (ADD  :c) (m:n:s) = exec c (n+m:s)     --(B3)
```

つまり、PUSH命令はスタックの上に新しい整数を置き
ADD命令はスタック上の2つの整数をその和で置き換える。

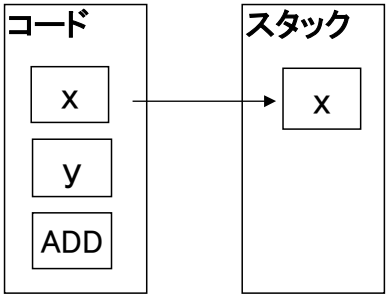
次にPUSHやADDの命令を使用して式をコードに翻訳(コンパイル)する関数compを定義する。

-- comp定義

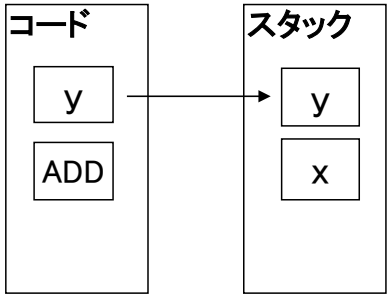
```
comp : : Expr -> Code
comp (Val n)    = [PUSH n]                          --(C1)
comp (Add x y)  = comp x ++ comp y ++ [ADD]         --(C2)
```

つまり、整数の場合はその値をPUSHする命令に翻訳し、加算の場合は
2つの引数xとyをそれぞれ翻訳し、結果として算出された整数2つをスタック上で
ADD命列で足し合わせる。
注意として、ADD命令が実行されるときは、スタックの１番上がyで2番目がxとなります。

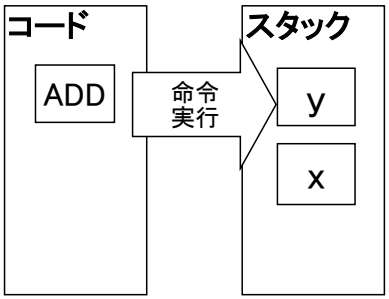
①



②



③



再掲

-- exec定義

```
exec □ s              = s                          --(B1)
exec (PUSH n:c) s     = exec c (n:s)               --(B2)
exec (ADD  :c) (m:n:s) = exec c (n+m:s)            --(B3)
```

上記で次の式を計算してみる
(式の意味：e=(2+3)+4)
e = Add (Add (Val 2) (Val 3)) (Val 4)

>eval e
9
> comp e
[PUSH 2,PUSH 3,ADD,PUSH 4,ADD]
> exec (comp e) []
[9]

上記をふまえると式コンパイラーの正しさの等式表現は次のように表現できる。
exec (comp e) [] = [eval e]

すなわち、式を翻訳し、結果のコードを空のスタックと共に実行すると
スタックの最終状態は、式を評価して結果を要素が1つのリストに変換したものに等しい。

上記の空のスタック[]を任意のスタックsに拡張して一般化すると次のようになる。
exec (comp e) s = [eval e] ++ s
exec (comp e) s = eval e : s　　--(*001)

*exec (comp e) s = eval e : s　　--(*001)*

[(*001)の証明]
(1) e = Val n のとき
(*001)の右辺
= eval (Val n) : s
= n : s　　(∵(A1))

(*001)の左辺
= exec (comp Val n) s
= exec [PUSH n] s (∵(C1))
= exec (PUSH n:[]) s
= exec [] n:s　　(∵(B2))
= n:s　　(∵(B1))

再掲
-- eval 定義
eval (Val n)　= n　　--(A1)
eval (Add x y) = eval x + eval y　　--(A2)
-- comp定義
comp (Val n)　= [PUSH n]　　--(C1)
comp (Add x y)　= comp x ++ comp y ++ [ADD]　　--(C2)
-- exec定義
exec [] s　　= s　　--(B1)
exec (PUSH n:c) s　　= exec c (n:s)　　--(B2)
exec (ADD :c) (m:n:s) = exec c (n+m:s)　　--(B3)

(2) e = Add x y のとき
e = x および e = y のとき下記が成立つと仮定する
exec (comp x) s = eval x : s　　--(仮定1)
exec (comp y) s = eval y : s　　--(仮定2)

(*001)の右辺
=eval (Add x y) : s
=(eval x + eval y) : s　　(∵(A2))

--分配法則

exec (c ++ d) s = exec d (exec c s)　　*--(分配則)*

この分配則は後で証明する。

再掲

exec (comp x) s = eval x : s　　*--(仮定1)*

exec (comp y) s = eval y : s　　*--(仮定2)*

(*001)の左辺

```
=exec (comp (Add x y)) s
=exec (comp x ++ comp y ++ [ADD]) s          (':'(C2))
=exec (comp x ++ (comp y ++ [ADD])) s        (':'(X2))
=exec (comp y ++ [ADD]) (exec (comp x) s)    (':'(分配則) d ->(comp y ++ [ADD]))
=exec (comp y ++ [ADD]) (eval x : s)         (':'(仮定1))
=exec [ADD] (exec (comp y) (eval x : s))     (':'(分配則) d ->[ADD])
=exec [ADD] (eval y : (eval x : s))          (':'(仮定2))
=exec ADD:[] (eval y : eval x : s)           (整理)
=exec [] ((eval x + eval y) : s)             (':'(B3) c->[] m->eval y n->eval x)
=(eval x + eval y ): s                       (':'(B1))
```

以上より、(*001)が成立する

QED.

exec (comp e) s = eval e : s

*--(*001)*

再掲

-- (++)演算子

```
[] ++ ys     = ys                                  --(X1)
(x:xs) ++ ys = x:(xs ++ ys)                        --(X2)
-- comp定義
comp (Val n)    = [PUSH n]                         --(C1)
comp (Add x y)  = comp x ++ comp y ++ [ADD]         --(C2)
-- exec定義
exec [] s            = s                           --(B1)
exec (PUSH n:c) s    = exec c (n:s)                --(B2)
exec (ADD :c) (m:n:s) = exec c (n+m:s)             --(B3)
```

--分配法則

exec (c ++ d) s = exec d (exec c s)　　--(分配則)

[基底部]

c = [] のとき

右辺 = exec d (exec [] s)

= exec d s　　(∵(B1))

左辺 = exec ([] ++ d):s

= exec d:s　　(∵(X1))

c = x のとき

次式が成立つと仮定する

exec (x ++ d) s = exec d (exec x s)　　--(仮定3)

再掲

-- exec定義

exec [] s　　= s　　--(B1)

exec (PUSH n:c) s　　= exec c (n:s)　　--(B2)

exec (ADD :c) (m:n:s) = exec c (n+m:s)　　--(B3)

-- (++)演算子

[] ++ ys　　= ys　　--(X1)

(x:xs) ++ ys = x:(xs ++ ys)　　--(X2)

[再起部1]

c = PUSH n:x のとき

右辺

=exec d (exec (PUSH n:x) s)

=exec d (exec x (n:s))　　(∵(B2))

左辺

=exec ((PUSH n:x) ++ d) s

=exec (PUSH n:(x ++ d)) s　　(∵(X2))

=exec (x ++ d) (n:s)　　(∵(B2))

=exec d (exec x (n:s))　　(∵(仮定3))

[再起部2]
c = ADD : x のとき
(s=m:n:s'と仮定する)
右辺
=exec d (exec (ADD:x) s)
=exec d (exec (ADD:x) (m:n:s')) (∵仮定)
=exec d (exec x (n+m:s')) (∵(B3))
左辺
=exec ((ADD:x) ++ d) s
=exec (ADD:(x ++ d)) (m:n:s') (∵(X2))
=exec (x ++ d) (n+m:s') (∵(B3))
=exec d (exec x (n+m:s')) (∵(仮定3))

再掲

-- (++)演算子
[] ++ ys = ys --(X1)
(x:xs) ++ ys = x:(xs ++ ys) --(X2)

以上より、(分配則)が成立する

QED.

再掲
-- exec定義
exec [] s = s --(B1)
exec (PUSH n:c) s = exec c (n:s) --(B2)
exec (ADD :c) (m:n:s) = exec c (n+m:s) --(B3)

exec (x ++ d) s = exec d (exec x s) --(仮定3)

再起部2]のスタックでの仮定
(s=m:n:s')の2要素が
つまれていない場合は
スタックアンダーフローとなるが、
ADD命令が実行される際には、2つの整数があることをコンパイラー(comp)が保障するので
アンダーフローは起きない。このアンダーフローの問題と分配則を使用するという問題は、
連結演算子を除去する方法で解決できる。
次にその方法を示します。

式コンパイラーの正しさの等式表現

exec (comp e) s = eval e : s
--(*001)

comp' e c = comp e ++ c
--(comp変形)

e = Val n のとき

comp' (Val n) c
=comp (Val n) ++ c (∵(comp変形))
=[PUSH n] ++ c (∵(C1))
=PUSH n:c

e = Add x y のとき

comp' (Add x y) c
=comp (Add x y) ++ c
=comp x ++ comp y ++ [ADD] ++ c (∵(C2))
=comp x ++ comp y ++ (ADD: c) ([ADD]をADD:Cに書き換えた)
=comp x ++ (comp y ++ (ADD: c)) (結合則があるので括弧を書き換えた)
=comp x ++ (comp' y (ADD: c)) (∵(comp変形))
=comp' x (comp' y (ADD: c)) (∵(comp変形))

以上より、次式を得る

comp' :: Expr -> Code -> code
comp' (Val n) c = PUSH n:c --(D1)
comp' (Add x y) c = comp' x (comp' y (ADD:c)) --(D2)

再掲

-- eval 定義
eval (Val n) = n --(A1)
eval (Add x y) = eval x + eval y --(A2)
-- comp定義
comp (Val n) = [PUSH n] --(C1)
comp (Add x y) = comp x ++ comp y ++ [ADD] --(C2)
-- exec定義
exec [] s = s --(B1)
exec (PUSH n:c) s = exec c (n:s) --(B2)
exec (ADD :c) (m:n:s) = exec c (n+m:s) --(B3)

また、(comp変形)で c =[] とすると
comp' e [] = comp e ++ []
となるから
comp e = comp' e []
と定義できる

再掲
comp' e c = comp e ++ c *--(comp変形)*
exec [] s = s --(B1)
exec (PUSH n:c) s = exec c (n:s) --(B2)

exec (comp e) [] = [eval e]
exec (comp' e []) [] = (eval e : [])
exec (comp' e []) [] = exec [] (eval e : [])
c s c s
上式を一般化([]->c,[]->s)
exec (comp' e c) s = exec c (eval e : s)

--(旧コンパイラー検証式)

(∵(B1の逆適用))
(←[]との対応)　この変形うそかもしれない？

--(新コンパイラー検証式)

[(新コンパイラー検証式)の証明]
e = Val n のとき
右辺
=exec c ((eval Val n) : s)
=exec c (n : s) (∵(A1))
左辺
=exec (comp' (Val n) c) s
=exec (PUSH n : c) s (∵(D1))
=exec c (n : s) (∵(B2))

再掲
-- eval 定義
eval (Val n) = n --(A1)
eval (Add x y) = eval x + eval y --(A2)
-- conp' 定義
comp' : : Expr -> Code -> code
comp' (Val n) c = PUSH n : c --(D1)
comp' (Add x y) c = comp' x (comp' y (ADD : c))
--(D2)

e = Add x y のとき
e = x および e = y のとき下記が成立つと仮定する

exec (comp' x c) s = exec c (eval x : s) ---(仮定3)

exec (comp' y c) s = exec c (eval y : s) ---(仮定4)

右辺

=exec c ((eval Add x y) : s)

=exec c ((eval x + eval y) : s) (∵(A2))

左辺

=exec (comp' (Add x y) c) s

=exec (comp' x (comp' y (ADD : c))) s (∵(D2))

=exec (comp' y (ADD : c)) (eval x : s) (∵(仮定3) c->(comp' y (ADD : c)))

=exec (ADD : c) (eval y : eval x : s) (∵(仮定4) c->(ADD : c) s-> (eval x : s))

=exec c ((eval x + eval y) : s) (∵(B3))

再掲

-- eval 定義

eval (Val n) = n ---(A1)

eval (Add x y) = eval x + eval y ---(A2)

-- conp' 定義

comp' : : Expr -> Code -> code

comp' (Val n) c = PUSH n : c -(D1)

comp' (Add x y) c

= comp' x (comp' y (ADD : c)) -(D2)

以上より、(新コンパイラー検証式)が成立する

QED.

再掲

exec [] s = s ---(B1)

exec (PUSH n:c) s = exec c (n:s) ---(B2)

exec (ADD :c) (m:n:s) = exec c (n+m:s) ---(B3)

exec (comp' e c) s = exec c (eval e : s) ---(新コンパイラー検証式)

comp e = comp' e []

証明過程でスタックアンダーフローの問題が避けられるのに加えて
蓄積変数を使うコンパイラーには2つの利点がある。
1つは、連結演算子(++)が除去され効率がよいこと。
2つ目は、後の証明は前の証明に比べて半分以下ですんでいること。

形式的な論証では結果をうまく一般化すると証明がきわめて簡潔になることがよくある。
数学は、効率のよいプログラムを導く為の優れた道具なのである　！