IS —Introduction to Scheme—

Yoshihiko Kakutani

はじめに

Scheme は型無しの関数型プログラミング言語の一種です。これから、この Scheme を通じて、関数型のプログラミング手法を学んでいきます。

手続き型プログラミングと関数型プログラミングの大きな違いは、想定している計算モデルの違いです。多くの場合、手続き型のような命令型プログラミングでは、プログラマはマシンへの命令を順に記述し、マシンの状態を変化させることで計算を進めます。それに対して、関数型のような宣言型プログラミングでは、プログラマはどのような計算結果が欲しいかを記述することになります。関数型プログラミングでプログラマが気にするのは、マシンの状態ではなくマシンの入出力です。

関数型プログラミングの特徴としては、以下のような点を挙げることができます。

- 関数適用の計算結果が、実行環境に依らず、引数の値のみに依存して決まる。
- 変数の値が途中で変更されない。

最初の項目については、C のポインタ渡しプログラミングと比較すると分かり易いかもしれません。通常、ポインタを引数に取る関数は、渡されるポインタの値が同じでも、メモリの状態によって関数適用の計算結果が異なります。Scheme では、そのような関数はできるだけ書かないようにしてプログラミングを行います。

C ではプログラムの実行中に 1 つの変数の値を何度も更新していくのが普通ですが、Scheme では、基本的に 1 つの変数に 2 回以上の代入を行いません。(有効範囲の異なる変数は、同じ名前でも異なる変数です。) これにより、変数はどの場所に現れても同じ値であることが保証されます。

ただし、関数型に分類されるプログラミング言語でも、手続き型のスタイルでプログラミングすることは可能です。特に、Scheme には変数の再代入のための構文も用意されています。この演習では、Scheme による手続き型プログラミングを全く推奨しないので、この資料でも手続き型の手法についてはほとんど言及しません。反対に、C で関数型のプログラミングをすることも不可能ではありません。余裕があれば、手続き型、関数型、双方のスタイルでプログラミングを行っておくとよいでしょう。

また、通常、Scheme は型無しのプログラミング言語に分類されます。「型無し」というのは型の概念がないという意味ではなく、静的型チェックを行わないということを意味します。逆に、静的型チェックを行う言語のことを、「型付き」プログラミング言語と呼びます。静的型チェックはプログラムがある種の仕様を満たすことを保証してくれるので、型付きプログラミング言語で書かれたプログラムにはバグが少なくなります。対して、型無しプログラミング言語のメリットはすぐに動作することです。不適切なコードでも、エラーが起こるところまでは簡単に実行することができます。

1　式（Expressions）

Scheme のプログラムは、式（expression）を列挙したものです。列挙された式は前から逐次的に実行されます。（式の中に式が現れることもありますが、内部に現れる式がいつ実行されるのかはそのうち分かります。）Scheme では、式の実行のことを「評価（evaluation）」と呼びます。式は評価されると何らかの値になり、処理系はその値を表示します。

式は、定数データ（数や文字列）、変数（シンボル）、関数適用（リスト）、特殊フォームのいずれかになります。このうち、関数適用と特殊フォームは、必ず「(」で始まり「)」で終わります。Scheme の括弧は構文の最も重要な要素であり、書いても書かなくても意味が変わらないという状況は決してありません。

ちなみに、コード中にコメントを書きたい場合には、「;」を使います。「;」から行末まではコメントとして無視されます。

1.1　定数データ（Constants）

定数データとは、数字や文字列など、これ以上簡単に書けない式のことです。定数データは評価してもそのままで値となります。次のような式が定数データです。

```
0
"Hello, world!"
#f
```

定数データ `#t` と `#f` はブール値と呼ばれ、それぞれ真と偽を意味します。定数データはリテラルとも呼ばれますが、リテラルには他のデータも含まれます。定数データ以外のリテラルは、元の式と評価結果が異なります。また、定数データと変数をまとめて、原子式（atom）と呼ぶことがあります。

1.2　関数適用（Applications）

話の都合上、変数よりも先に関数適用を紹介します。Scheme の関数適用は、「(関数名 引数 ...)」という形をしています。C と違って、括弧が関数名の前に置かれます。引数の数は関数によって異なります。（任意の個数の引数を受け取ることができる関数もあります。）

```
(+ 1 2)
```

`+` や `-`、`*` は算術計算を行う関数です。関数適用式の評価結果が算術計算の結果になるので、この場合は 3 となります。算術計算を行う関数の他にも、様々な関数が組込みで与えられています。組込み以外に自分で関数を定義することもできますが、それについては変数の項目で述べます。

引数の部分に式を書くこともできます。（本当は、関数名の部分にも一般の式が許されます。）例えば、以下は正しい式です。

```
(+ (- (* (* 1 2) 3) (* 4 5)) (+ (* 6 7) (* 8 9)))
```

式が入れ子になっている場合、内側の部分式の評価結果は、外側の評価に再帰的に受け継がれます。Scheme は値呼びと呼ばれる評価戦略を採用しているので、単純な部分式から順に評価されていきます。上の例で言え

ば、木構造の葉に当たる (* 1 2)、(* 4 5)、(* 6 7)、(* 8 9) から評価が始まります。(正確には葉に当たるのは数字ですが、その評価は自明なので。) ただし、数学における関数と同じような副作用のない関数を使っている限り、評価順序をそれほど厳密に気にしなくても、問題はありません。[*1]

関数適用の評価結果がリストになる関数もあります。関数適用の評価結果を関数が返す値と呼びます。例えば、この場合、「リストを返す関数」などという表現をします。(プログラミングや数学全般において、常識的な用法です。) 処理系で実行すれば、次の式の評価結果が長さ 6 のリストになることが確認できます。

```
(list 1 2 3 5 7 11)
```

list に与える引数の個数はいくつでもかまいません。入れ子のリストを作ることも可能です。次の式の評価結果は、第 1 要素がリストで第 2 要素が 3 のリストになっています。

```
(list (list 1 2) 3)
```

ところで、処理系で式を評価すると、(1 2 3 5 7 11) などと表示されますが、評価結果であるリストも括弧から始まっており、見た目が式のように見えなくもないと思います。実は、Scheme では、評価結果の値を再び式とみなすことができます。これは、他の言語にはあまり見られない特徴です。ただし、特殊な関数 eval を使わない限り、通常の用法では評価結果を再び処理系内で評価する方法はありません。もちろん、プログラマが値を手で再び処理系に入力すれば評価可能です。今回の場合は、文法的には式に見えても 1 は関数ではないので、エラーになります。以下は再評価してもエラーにならない例ですが、未知の構文も含まれているので、意味については後回しにします。

```
(list (quote +) 1 2)
```

ブール値を返す関数は、特に「述語」と呼ばれることがあります。ほとんどの組込み述語は名前から挙動が推測できるようになっています。1 引数の組込み述語は、zero?、even?、odd?、boolean?、list? などです。以下は評価すると #t になる例です。

```
(even? 0)
(odd? 3)
(list? (list 1 2))
```

2 引数の組込み述語では、=、<、>、<=、>=、equal? などがよく使われます。equal? はリストの比較に使います。例を実行して、結果が期待通りであることを確かめてみてください。

```
(= 1 2)
(< 1 2)
(> 1 2)
(equal? (list 1 2) (list 1))
(equal? (list (list 1 2) 3) (list (list 1 2) 3))
```

また、not はブール値を受け取ってブール値を返すような関数で、#t と #f を入れ替えます。述語や not は、条件分岐を書く際に役立ちます。

[*1] 数学的関数の間の合成演算は結合的なので、通常、我々は関数の合成順序に注意を払いません。

通常 Scheme は処理系で実行されるので、明示的に何かを表示させる必要はあまりないのですが、文字を出力するには display 関数を使います。

```
(display "Hello, world!\n")
```

この式を処理系で評価すると、Hello, world! と表示されます。処理系によっては、それに加えて #<undef> などと表示されることがありますが、これは、関数 display が文字列の内容を出力し、その後、関数適用式の評価結果を処理系が自動的に表示しているせいです。

1.3 変数（Variables）

その他のプログラミング言語と同様、Scheme でも変数を扱うことができます。変数を定義するには、次のような式を評価します。

```
(define x (+ 0 1))
```

これで、変数 x が定義されました。処理系で実行していれば、これ以降、ファイルにプログラムを書いているなら、この式より下の式で変数 x を利用することができます。変数 x に割り当てられた値を知りたい場合には、変数のみからなる式を評価します。

```
x
```

1 と表示されたはずです。(+ 0 1) ではないことに注意してください。define 式は、変数を定義し、第 2 引数の位置にある式を評価した結果を変数に割り当てます。

未定義の変数式を評価しようとすると、エラーになります。部分式に現れてもいけません。（本当は、現れてもいい場所があります。）先の例に続けて、次の式を評価してみるとよいでしょう。

```
(+ y 1)
(define y (+ x 1))
(+ y 1)
```

変数の値を書き換えるには、set! を使います。ただし、関数型プログラミングでは基本的に変数の値を途中で書き換えることはないので、本稿でも set! はないものとして扱います。

値を書き換えない変数の用法は、数学の変数の用法とよく似ています。数学の変数の用途は、主に不定元の表現と略記にあります。特定の値ではなく、ある集合内の任意の値で成り立つ式などを書き表す場合、変数で不定な値を表現します。また、複雑な数式には名前を与えて、変数で簡潔に表現します。どちらにしても、途中で変数の値が変わってしまうのは、都合が悪いと言えます。Scheme では、関数のパラメータ変数が前者の役割を果たし、通常の変数が後者の役割を果たします。関数型プログラミングでは代入を使わないので、このような数学との対比が明確になります。

define 式も式なので、文法的には他の式の内側に現れることも許されますが、define 式を書いてもよい場所は限られています。トップレベルで定義された変数は、いわゆる大域変数となります。変数の有効範囲を制限するには、let 式を使います。変数の有効範囲は「スコープ」と呼ばれます。

```
(let ((y (* 2 5))
```

```
      (z 100))
  (+ y 1)
  (+ x y z))
```

この例の場合、y と z という変数が定義されていて、それぞれの値は 10 と 100 になっています。ただし、この変数定義は、(+ y 1) と (+ x y z) の 2 つの式でのみ有効で、let の外側には効力が及びません。つまり、外側で未定義だった z はこの式を評価した後も、未定義のままです。また、既に定義されていて値が 2 であった変数 y の値もやはり 2 のままです。

let 式の評価結果は、let 式の末尾にある部分式の評価結果になります。上の例なら、(+ x y z) です。この部分式の評価では、let で定義されていない x の値は外側と同じ 1 となり、評価結果は 111 となります。今回は 2 つしか書いていませんが、let 式では中にいくつでも式を書くことができます。内部の式は前から順に評価されるので、実際には、まず (+ y 1) が評価された後、(+ x y z) が評価されます。ただ、評価結果として見えるのは最後の式の評価結果だけなので、今回のように複数の式を書いても普通は意味がありません。処理系は式の評価結果を自動的に表示しますが、途中の結果は表示しません。let 式を評価した場合も、途中の評価結果は表示されないので、ユーザからは (+ y 1) があってもなくても同じに見えます。

let 式を続けて入れ子にするのが便利なこともあるので、そのような式には略記が用意されています。

```
(let* ((a 2)
       (b (* 2 a))
       (c (* 2 b)))
  (* 2 c))
```

let* 式は、let 式の let を let* に置き換えた形をしています。この let* 式は、以下のような let 式を入れ子にした式と等価になります。

```
(let ((a 2))
  (let ((b (* 2 a)))
    (let ((c (* 2 b)))
      (* 2 c))))
```

define は関数を定義するのにも使われます。Scheme では、関数名と変数に本質的な違いはありません。「λ 閉包（closure、procedure）」と呼ばれる特殊なデータを値に持つ変数のことを、関数名と呼んでいるに過ぎません。λ 閉包は関数そのものを意味するデータなので、正確には、関数とは λ 閉包のことで、関数名は単なる変数ということになります。組込み関数も同じ仕組で、+ などの関数名は、その値が組込み関数になっているような変数です。

ただし、一般には、関数名のことも λ 閉包のことも、どちらも「関数」と呼ぶことがあります。本稿でも、これまでそうであったように、どちらも関数と呼び、特に区別の必要な場合には、関数名や λ 閉包と明記することにします。

```
(define square (lambda (x) (* x x)))
```

これが関数定義の例です。通常の変数の定義と同じく、square という変数が定義され、その値が (lambda (x) (* x x)) を評価した結果になっています。lambda で始まる式を評価すると λ 閉包にな

ります。この square という関数は、引数として値を 1 つ受け取り、その 2 乗を返す関数です。引数を表すのに使われている x は変数の一種で、束縛変数またはパラメータ（parameter）と呼ばれます。この x のスコープは関数定義の本体部分のみで、関数定義の外で定義されている変数 x の影響を受けません。

define による関数定義は、（大して簡単になっていませんが）以下のように略記することもできます。単なる略記に過ぎないので、式としての意味は全く変わりません。

```
(define (square x) (* x x))
```

自分で定義した関数は、組込み関数と同様に関数適用で利用することができます。

```
(square 3)
```

この関数適用を評価すると、パラメータ x が引数 3 に置換されて、(* 3 3) が評価されることになります。[*2]もう少し複雑な式の場合、パラメータへの代入と部分式の評価のどちらが先に起こるのかについては、疑問が残ります。

```
(square (square 3))
```

関数適用の項目で述べたように、関数適用式は木構造の葉から根に向かって順に評価されるので、まず、(square 3) が (* 3 3) となり 9 だと分かり、その後、全体が (* 9 9) として評価され、結果が 81 になります。

$$\texttt{(square (square 3))} \Rightarrow \texttt{(square (* 3 3))} \Rightarrow \texttt{(* 9 9)} \Rightarrow 81$$

これが Scheme の本来の評価方法ですが、異なる評価方法も考えられます。まず外側の square に対して展開を行い、全体を (* (square 3) (square 3)) として評価します。部分式も更に展開されるので、結局 (* (* 3 3) (* 3 3)) を評価することになります。この式の評価結果は、やはり 81 になります。

$$\texttt{(square (square 3))} \Rightarrow \texttt{(* (square 3) (square 3))} \Rightarrow \texttt{(* (* 3 3) (* 3 3))} \Rightarrow 81$$

関数適用の項目で評価順序を気にしなくてもよいと述べたのは、このように評価順序によらず計算結果が一致するからです。評価順序については、後ほどもう少し詳しくみることにします。

また、関数名は変数だったので、変数式として評価することも可能です。組込み関数も変数なので、それ自体で式とみなせます。評価結果がどう表示されるかは処理系に依存します。自分でも確かめてみましょう。

```
square
+
```

複数の引数を受け取る関数を定義することもできます。

```
(define add (lambda (x y) (+ x y)))
```

これで、引数を 2 つとる関数 add が定義されました。この式は以下のように書くこともできます。引数の数がもっと多い場合も同様です。引数がない場合も同様に定義可能ですが、括弧は省略できません。

[*2] Scheme では式が値となり得るので、本当に置換してしまうと正しくない場合があります。今回は値が数値なので、実際に置き換えてしまっても問題ありません。

```
(define (add x y) (+ x y))
```

可変個の引数を取る関数を定義することも可能ですが、入門書としては不要なので、ここでは特に説明しません。

define と lambda は独立に使用可能なので、原理的には、関数名を宣言せずにλ閉包を扱うこともできます。

```
((lambda (x) (* x x)) 2)
```

このような式は無名関数と呼ばれることがあります。このような形で無名関数を扱う意味はほとんどありませんが、関数を返す関数などを書くと、評価結果が名前のないλ閉包になるということはよくあります。λ閉包はデータとして扱えるということを忘れないようにしましょう。

1.4 特殊フォーム（Special-Forms）

Scheme には、他のプログラミング言語と同じように、条件分岐や繰り返しのための制御構造もあります。条件分岐のための式は if から始まるリストです。普通は、条件分岐式の中で述語を使います。

```
(if (= x 0) (+ z y) (+ x y))
```

この式を評価すると、if の直後の部分式 (= x 0) を評価した結果が #f でなければ、(+ z y) が、#f なら、(+ x y) が評価されます。if 式全体の評価結果が、部分式の評価結果になります。

いま、変数 x と y が定義されていて、x の値が 0 でなく、z が未定義だと仮定してみます。（ここまでに掲載されているコードを順に実行しているなら、そうなっているはずです。）そのとき、この式を評価してもエラーにはなりません。実際に評価されるのが、(= x 0) と (+ x y) だけだからです。式に現れる変数は事前に定義されていなければならないと述べましたが、それは正確ではなく、条件分岐で実際に起こらない分岐部分には未定義の変数があってもかまわないということになります。（ただし、条件式の真偽が分かっているのなら、分岐先を直接書けばよいので、普通はそのようなコードを書く必要がありません。）

if 式は普通の式と同じく評価結果を持つので、他の関数の引数として使うことができます。関数を返す場合には、関数適用の関数部分に書いてもかまいません。

```
((if (even? x) + *) 1 2)
```

評価結果がブール値になる場合には、条件式の部分に if 式を書くこともできます。

```
(if (if (= x 1) (= y 2) #f) "yes" "no")
```

このような用法は複雑な条件を書くのに有用なので、等価な構文として and と or が用意されています。上の例にある (if (= x 1) (= y 2) #f) は以下の式と等価です。

```
(and (= x 1) (= y 2))
```

以下の 2 つの式も等価になります。[*3]not と違い、and と or は関数ではないことに注意してください。

[*3] 評価結果がブール値でない式に対しても or や and を使うことができます。その場合、この書き換えはもう少し複雑になります。

```
(or (= x 1) (= y 2))
(if (= x 1) #t (= y 2))
```

条件分岐後の式を入れ子にすれば、より複雑な分岐を表現することができますが、分岐の片側のみが入れ子になっていくような場合には少し便利な構文があります。

```
(cond ((< x 0) "minus")
      ((= x 0) "none")
      ((= x 1) "one")
      ((= x 2) "two")
      (else "many"))
```

これは、以下の式と同じ意味を持ちます。一般に、cond 式は if 式を入れ子にしたもので表現可能です。

```
(if (< x 0) "minus"
    (if (= x 0) "none"
        (if (= x 1) "one"
            (if (= x 2) "two"
                "many"))))
```

他に、C の while 文のように分岐の一方を繰り返しにする while 式もあります。しかしながら、繰り返しを止めるには変数への再代入を行う必要があるので、関数型プログラミングでは while はほとんど使用されず、同様のことは再帰で実現されます。再帰については次の章で説明します。

特殊フォームに分類される式は条件分岐だけではありません。これまでに登場した式の中にも特殊フォームはあります。define 式や let 式、lambda 式がそれです。lambda で始まる式を評価した結果は、λ 閉包という特別なものになることを思い出してください。これは、関数適用式の評価規則とは違います。

Scheme では、リストの形をした式は関数適用か特殊フォームのどちらかに分類されます。関数適用なのか特殊フォームなのかは、リストの先頭要素（関数適用の関数に当たる部分）によって決まります。先頭要素が if や lambda といった特別な形のとき、式は特殊フォームに分類されます。それ以外は全て関数適用なので、例えば、以下のような式も関数適用です。括弧の数に注意してください。

```
((lambda () 0))
```

関数適用のところでは見なかった例ですが、実は、関数名に当たる部分にも一般の式を書くことが許されています。（if 式の例でも使っています。）先頭の部分の評価結果が λ 閉包であれば、エラーにならずに関数適用が実行されます。

2　関数（Procedures）

簡単な関数（procedure）の定義については、既に述べました。この章では、より複雑な関数を定義するための手法を紹介します。

2.1 再帰（Recursion）

関数の定義では、内部で自分自身を参照することができます。このような定義方法を「再帰（recursion）」と呼びます。定義の書き方は基本的な場合と特に変わりなく、単に関数定義の内部で自分自身を使用するだけです。次の例は階乗を計算する関数を再帰的に定義します。

```
(define (fact n)
  (if (< n 1)
      1
      (* n (fact (- n 1)))))
```

この関数が階乗を計算することは、実際に関数を適用してみることで確認できます。適用方法と評価方法も普通の関数と同じです。

(fact 4) ⇒ (if (< 4 1) 1 (* 4 (fact (- 4 1)))) ⇒ (* 4 (fact (- 4 1)))

計算の途中で再帰的に関数が出てきたら、再び同じ方法で評価されます。

(fact (- 4 1)) ⇒ (if (< 3 1) 1 (* 3 (fact (- 3 1)))) ⇒ (* 3 (fact (- 3 1)))
(fact (- 3 1)) ⇒ (if (< 2 1) 1 (* 2 (fact (- 2 1)))) ⇒ (* 2 (fact (- 2 1)))
(fact (- 2 1)) ⇒ (if (< 1 1) 1 (* 1 (fact (- 1 1)))) ⇒ (* 1 (fact (- 1 1)))
(fact (- 1 1)) ⇒ (if (< 0 1) 1 (* 0 (fact (- 0 1)))) ⇒ 1

必要な計算結果が得られたら、呼び出された場所の計算に戻っていきます。この場合、最終的には (* 4 3 2 1 1) と同じ計算結果が得られます。

(fact (- 2 1)) ⇒ (* 1 (fact (- 1 1))) ⇒ 1
(fact (- 3 1)) ⇒ (* 2 (fact (- 2 1))) ⇒ 2
(fact (- 4 1)) ⇒ (* 3 (fact (- 3 1))) ⇒ 6
(fact 4) ⇒ (* 4 (fact (- 4 1))) ⇒ 24

ここで、関数の定義に戻って、define 式の評価方法を思い出してみると、変数 fact の値は以下の式の評価結果ということになります。

```
(lambda (n)
  (if (< n 1)
      1
      (* n (fact (- n 1)))))
```

lambda 式を評価するとλ閉包になるのですが、内部に未定義の変数 fact が含まれていることになります。一般に、lambda 式の中には未定義の変数を書くことができるので、以下のような順で定義することも可能です。

```
(define (add-const x) (+ x a))
```

```
(define a 10)
(add-const 1)
```

また、定義の内部で相互に参照し合うような複数の関数を定義することも可能です。以下の例では、my-odd? と my-even? はお互いの定義に現れていて、実行時にはお互いを呼び出し合います。

```
(define (my-odd? n)
  (cond ((= n 0) #f)
        ((< n 0) (my-even? (+ n 1)))
        (else (my-even? (- n 1)))))
(define (my-even? n)
  (cond ((= n 0) #t)
        ((< n 0) (my-odd? (+ n 1)))
        (else (my-odd? (- n 1)))))
```

■演習問題　フィボナッチ数を計算する関数を定義せよ。

2.2　末尾再帰（Tail-Recursion）

再帰は、C の while 文のような繰り返しを書く場合にも使われます。次の関数定義を見てください。これは、(not (good-enough-sqrt? x guess)) の評価結果が #t の間、guess を更新し続け、最後に guess の値を返します。

```
(define (sqrt-loop x)
  (let ((guess 1.0))
    (while (not (good-enough-sqrt? x guess))
           (set! guess (improve-sqrt x guess)))
    guess))
```

この定義を関数型プログラミングで書きなおすことを考えます。if で分岐した先で再帰的に関数適用を行うことで、while を表現することができます。ただし、関数型では変数に値を代入するという操作がないので、代わりに関数適用を用います。関数適用を使うと、関数のパラメータに引数を代入することができるからです。今回の例なら、局所変数 guess を関数のパラメータに移しておきます。

```
(define (sqrt-iter x guess)
  (if (good-enough-sqrt? x guess)
      guess
      (sqrt-iter x (improve-sqrt x guess))))
```

guess の初期値がなくなっていますが、初期値は最初に関数を呼び出す際に指定することにします。実際、次のように書けば、関数 sqrt は、平方根をニュートン法で計算する関数になります。ここで、abs は絶対値を計算する組込み関数です。

```
(define (square t) (* t t))
(define (good-enough-sqrt? x guess) (< (abs (- (square guess) x)) 0.0000001))
(define (improve-sqrt x guess) (/ (+ guess (/ x guess)) 2.0))
(define (sqrt x) (sqrt-iter x 1.0))
```

ところで、上のような繰り返しを表現する再帰には、一般の再帰と異なる特徴があります。先ほど定義した fact では、(fact 100) を計算する際、n を 100 として、(fact (- n 1)) の評価結果が必要になります。このとき、(fact (- n 1)) の結果が分かるまで、(* n (fact (- n 1))) の部分は計算できません。つまり、再帰的に呼び出された (fact (- n 1)) を計算している間、処理系は後で計算する部分を覚えておく必要があります。これに対して、sqrt-iter では、内部の (sqrt-iter x (improve guess x)) の計算結果をそのまま使えばよいので、処理系にそのような負荷はかかりません。このように、再帰的な関数適用の評価結果がそのまま全体の評価結果になっている再帰を「末尾再帰（tail-recursion）」と呼びます。

通常の再帰的関数適用の場合、Scheme の処理系は、計算の残りの部分をメモリ上のスタックに退避させます。再帰の呼び出し回数が深くなると、消費するスタックの量も増加します。メモリは有限なので、スタックの限界を超える量のデータを退避させようとすると、スタックオーバフローというエラーが発生してしまいます。実際、fact に適当に大きな数を引数として与えるとオーバフローが起こります。（どのぐらいの大きさでオーバフローするかは、処理系やマシンの性能に依存します。）しかしながら、末尾再帰の場合は、どれだけ階層が深くても、スタックオーバフローは起こりません。

一般の再帰を末尾再帰に書き換えることはできませんが、ある程度のものは末尾再帰で実現可能です。例えば、fact の場合、以下のようにパタメータを 1 つ増やした関数を末尾再帰で定義することができます。

```
(define (fact-iter acc n)
  (if (< n 1)
      acc
      (fact-iter (* n acc) (- n 1))))
```

この新しく加わったパラメータ acc は、途中の計算結果を蓄えておく役割を果たすので、累積器（accumulator）と呼ばれることもあります。計算の流れは次のようになるので、スタックの消費はありません。

(fact-iter 1 4) ⇒ (fact-iter (* 4 1) (- 4 1)) ⇒ (fact-iter (* 3 4) (- 3 1))
⇒ (fact-iter (* 2 12) (- 2 1)) ⇒ (fact-iter (* 1 24) (- 1 1)) ⇒ 24

■演習問題　フィボナッチ数を計算する関数を末尾再帰で定義せよ。

2.3　局所変数（Local Variables）

上の fact-iter は引数が多いので、本当に fact と同じ働きの関数が欲しい場合は、次のように書くことができます。こうすると、inner-fact-iter のスコープは fact2 の定義中に限定されます。

```
(define (fact2 n)
  (define (inner-fact-iter acc n)
    (if (< n 1)
```

```
      acc
      (inner-fact-iter (* n acc) (- n 1))))
  (inner-fact-iter n 1))
```

この fact2 の定義の中では、define 式が使われています。これまでの例にはなかったのですが、lambda 式の関数本体部分には複数の式を書くことができます。let の場合と同様、実質的には複数の式を書く意味はないのですが、define 式だけは便利です。lambda 式の中に書かれた define 式で定義される変数は、その lambda 式の中でのみ有効です。このような局所変数は let で作る変数とよく似ているので、以下のように書いてもよいと思えるかも知れません。

```
(define (fact2 n)
  (let ((inner-fact-iter (lambda (acc n)
                           (if (< n 1)
                               acc
                               (inner-fact-iter (* n acc) (- n 1))))))
    (inner-fact-iter 1 n)))
```

しかしながら、この定義はうまく機能しません。let の局所変数宣言が再帰を許さないからです。局所変数を再帰的に定義したい場合には、letrec を使います。let との違いは再帰を許すかどうかだけで、構文は全く同じです。

```
(define (fact2 n)
  (letrec ((inner-fact-iter (lambda (acc n)
                              (if (< n 1)
                                  acc
                                  (inner-fact-iter (* n acc) (- n 1))))))
    (inner-fact-iter 1 n)))
```

この定義は define を使ったものと同じ関数を定義します。トップレベル以外の define 式は全て letrec で代替可能で、代替可能な場所にのみ define 式を書いてもよいということになっています。トップレベルの define 式は letrec で代替できないこともありますが、define 式をプログラムの先頭部分に集めて書いてしまえば、letrec で代替することができます。つまり、define 構文がなくても、letrec 構文さえあれば Scheme のプログラムは書けるということになります。

2.4 高階関数（Higher-Order Functions）

Scheme では、λ閉包を数値や文字列のようなデータと同様に扱うことができます。したがって、特殊な構文を使うことなく、λ閉包を引数として受け取ったり、計算結果としてλ閉包を返す関数を定義することができます。引数として関数を受け取るような関数を「高階（higher-order）関数」と呼びます。

先ほどの平方根を求めるプログラムを思い出してください。あのプログラムは、$g(t) = t^2 - x$ で定義される関数 g に対してニュートン法を用いたものです。[*4]ニュートン法は、より一般的な連続微分可能関数の零

[*4] x は、プログラム上のパラメータ x に引数として与えられる実数を表します。

点を求めることのできるアルゴリズムなので、他の関数に対してニュートン法を適用するプログラムを書くと、sqrt と似たようなコードになるはずです。様々な関数の零点を求めなければならい状況では、同じようなコードを何度も書くのは非生産的ですが、関数を受け取る関数を使えばその問題は解決されます。

sqrt-iter のパラメータ x は、本質的にはニュートン法を適用する関数を定めるためにあるので、x の代わりに関数を受け取るパラメータ g を用意します。また、ニュートン法では導関数も必要なので、導関数を受け取るパラメータ d も追加します。(理論的には、関数を受け取りさえすれば導関数を知ることができるのですが、導関数を求めるプログラムを書くのは簡単ではありません。) また、これに合わせて、good-enough-sqrt? と improve-sqrt も変更します。

```
(define (newton-iter g d guess)
  (if (good-enough? g guess)
      guess
      (newton-iter g d (improve g d guess))))
(define (good-enough? g guess)
  (< (abs (g guess)) 0.0000001))
(define (improve g d guess)
  (- guess (/ (g guess) (d guess))))
```

この newton-iter は関数、導関数、初期値を受け取って、ニュートン法による零点の計算結果を返します。例として、以下のように sqrt2 を定義すれば、sqrt と似た計算結果を返すことが確認できます。

```
(define (sqrt-base x) (lambda (t) (- (square t) x)))
(define (sqrt-deriv x) (lambda (t) (* 2 t)))
(define (sqrt2 x) (newton-iter (sqrt-base x) (sqrt-deriv x) 1.0))
```

関数を返す関数も抽象度の高いプログラミングに役立ちます。エラトステネスの篩という素数を列挙するアルゴリズムの実装を考えてみましょう。通常のエラトステネスの篩では、2 から適当な数までの整数を列挙したリストを用意して、そこから素数のリストを抽出します。もちろん、その方針でも実装できるのですが、今回は少し趣向を変えて、篩を意味する関数を計算することにします。

```
(define (sieve m)
  (if (< m 2)
      (lambda (n) #t)
      (let ((f (sieve (- m 1))))
        (if (f m)
            (lambda (n) (and (f n) (< 0 (modulo n m))))
            f))))
```

ここで、modulo は整数同士の除算による余りを求める組込み関数です。(商を求める組込み関数は quotient です。) この sieve の返す値は、引数以下の素数で割り切れないかどうかを判定する述語になります。例えば、((sieve 4) 7) は #t、((sieve 4) 9) は #f に評価されます。末尾再帰によって定義することも可能で、その場合、累積器 (パラメータ f) が関数を累積することになります。

```
(define (sieve-iter m i f)
  (cond ((< i 2) (sieve-iter m 2 f))
        ((< m i) f)
        ((f i)
         (sieve-iter m (+ i 1)
                     (lambda (n)
                       (and (f n) (< 0 (modulo n i))))))
        (else (sieve-iter m (+ i 1) f))))
(define (sieve2 m)
  (sieve-iter m 2 (lambda (n) #t)))
```

末尾再帰版を見れば、これが通常のエラトステネスの篩と同等の計算をしていることがよく分かると思います。ただし、sieve や sieve2 は素数の判定には便利ですが、素数を列挙するのには向いていません。(sieve2 100) を計算するときには、実質的に内部で 100 以下の素数を列挙してしまっているので、それを取り出すためにはプログラム自体を少し変更する必要があります。ちなみに、素数を判定する述語 prime? は以下のように定義することができます。(floor は実数を切り捨てて整数にする組込み関数です。)

```
(define (prime? n)
  ((sieve2 (floor (sqrt n))) n))
```

■演習問題　2 つの 1 引数関数を受け取ってそれらの合成関数を返す関数を定義せよ。

2.5　カリー化（Currying）

唐突ですが、以下で定義される関数 curry を考えます。

```
(define (curry f) (lambda (x) (lambda (y) (f x y))))
```

curry は、2 引数関数を受け取って 1 引数関数を返す関数です。(curry f) は、ほとんど f と同じような働きをします。例えば、以下のように + に適用したものは、(+ 1 2) と同じ計算結果を得ることができます。

```
(((curry +) 1) 2)
```

この curry の行う操作を、カリー化（Currying）と呼びます。3 以上の引数を持つ関数に対しても、同様にカリー化は定義されます。カリー化を使うと、一部の引数にだけ関数を適用することが可能になります。末尾再帰版の階乗関数 fact2 と同じ働きの関数 fact3 は、curry を使うことで次のように書くことができます。

```
(define fact3 ((curry fact-iter) 1))
```

他にも、カリー化を使うことで、上の演習問題で定義した関数の合成を多引数関数に適用することが可能になります。関数の curry を使うかどうかにかかわらず、カリー化自体は関数型プログラミングでよく使われる技術なので、覚えておくとよいでしょう。

3　ペアとリスト（Pairs and Lists）

この章では、リストに関連する組込み関数を紹介し、リスト上の関数を定義する手法について述べます。

3.1　式とリスト（Expressions as Data）

実は、まだ紹介していない重要な特殊フォームに quote 式があります。quote 式を評価すると、結果が式になります。とりあえず、次の例を処理系で実行してみてください。

```
(quote (+ 1 2))
```

(+ 1 2) と表示されたと思います。これは、quote 式の評価結果が (+ 1 2) という式になっていることを意味しています。quote の引数に当たる部分は、通常の式のようには評価されないので、変数が書いてあってもそのまま残ります。

```
(quote x)
```

quote を使うと、評価結果を変数にすることもできます。評価結果としての変数は、本来の変数としての意味を失っているので、「シンボル」と呼ばれます。このことから派生して、通常の構文上の変数もシンボルと呼ぶことがあります。Scheme のデータとしての文字列とシンボルを混同しないように注意してください。定数データとしての文字列は、Scheme のコード内の表現や処理系の表示では「"」で囲まれています。

quote の後ろには、どんな式を書いてもかまいません。実行するとエラーになるようなものも許されます。

```
(quote (1 2 3 5 7 11))
```

評価結果としての式は、リストデータと区別が付きません。これは処理系の表示の問題ではなく、Scheme の仕様として区別が付きません。実際、list? を使った以下の式の評価結果は #t です。

```
(list? (quote (1 2 3 5 7 11)))
```

また、前に説明した list を利用して作るリストと quote によるリストは同じものになります。これは、equal? を使った以下の式を評価することで確かめられます。

```
(equal? (quote (1 2 3 5 7 11)) (list 1 2 3 5 7 11))
```

quote には略記法があって、先の例は次のように書くことができます。

```
'(1 2 3 5 7 11)
```

「'」を対で使わないプログラミング言語は少ないので、注意してください。コードの読みやすさに配慮して、今後は略記法の方を優先して使うことにします。[*5]

[*5] define や let* のところで使った略記という言葉は、「式の意味が同じ」ということだったのですが、quote に関する略記はより強い性質を持っていて、「構文解析の結果が同じ」ということを意味しています。

3.2 ペア（Pairs）

Scheme は、リストの他にペア（pair）というデータを扱うことができます。ペアを返す関数は cons です。

```
(cons 1 2)
```

上の式を評価すると、(1 . 2) と表示されたはずです。これは、1 と . と 2 が並んだリストを意味しているのではなく、1 と 2 からなるペアを意味しています。ペアは「.」を用いて表されます。この記号を「ドット」と呼ぶことにします。ドットは Scheme のシンボルとして利用することも許されていません。丸括弧と同じような扱いだと考えてください。

逆に、ペアを受け取る関数は car と cdr です。car はペアの第 1 要素を、cdr は第 2 要素を返します。

```
(car (cons 1 2))
(cdr (cons 1 2))
```

以下のように書くとエラーになることに注意してください。

```
(car (1 . 2))
```

ペアは定数データのような単独で評価可能な式ではないので、コード中にペアデータを直接記述することはできません。データのペアが欲しい場合は cons で作る、というのが実用的な解決方法です。

長さ 2 のリストがあれば、理論上ペアは必要ないと言えますが、Scheme では寧ろ逆の状況で、リストがペアによって実現されています。Scheme では、空リスト以外のリストは、「入れ子になったペアで右側をたどった最後の要素が空リストであるもの」として定義されます。空リストだけは例外で、「()」として表されます。空リストも、直接 Scheme の式として評価するとエラーになるので、やはり注意してください。以下のような式を評価すれば、最後の評価結果がリストになっていることが確認できると思います。

```
(define empty-list (list))
(cons 1 (cons 2 (cons 3 empty-list)))
```

ところで、この表示を見て、おかしなことに気が付いたかも知れません。上で見た cons 式の評価結果に倣えば、今回は (1 . (2 . (3 . ()))) となるはずです。それがなぜリストとして表示されているのでしょうか？ 上で述べたリストの定義によれば、同一のリストデータに複数の表現があるということになります。処理系は、リストとして表現可能なペアデータは常にリストとして表示します。

また、右側が入れ子になったペアはリストでない場合にも、簡単な記法があります。(1 2 3 . 4) のようにドットの左側に複数書いた場合のペアのような表記は、ペアが入れ子になった (1 . (2 . (3 . 4))) と同じデータを意味します。リスト表記が可能な場合を除いて、処理系はこの記法が使える場合はできるだけこの記法を使って表示します。

ペアをドットで表記するのは、処理系だけではありません。プログラム上も同じ表記を使うことができます。Scheme ではリストデータと式には区別がなかったので、式もやはり特殊なペアであると言うことができます。つまり、以下の式のようなものはどれも等価ということになります。どの表記を用いても処理系はこれらを同じものとして扱います。

```
(+ 1 2)
(+ . (1 2))
(+ . (1 . (2)))
(+ . (1 . (2 . ())))
(+ 1 . (2 . ()))
(+ . (1 2 . ()))
```

また、これまで式として許されるのはリストだけでしたが、一般のペアも式として認めることにします。そうしたところで、ほとんどは評価するとエラーになるので、実質的には、quote の後ろにペアを書けるようになるだけです。ちなみに、上に挙げた式に quote を付けて実行してみれば、処理系が全ての式を同じに扱っているということが分かります。

ちなみに、car と cdr はペアの要素を取り出す関数でしたが、リストもペアであることを考慮すると、car はリストの先頭要素を返す関数、cdr はリストの先頭要素を取り除いた残りを返す関数と解釈することができます。cons もリストを延長する関数とみなすことができます。

Scheme には、データ構造は基本的にペアしか存在しません。他の複雑なデータ構造を使いたい場合も、ペアとリストを用いて表現するので、Scheme におけるプログラミングではリスト上の関数を書くことが非常に重要になります。

3.3 関数適用（Application）

リスト上の関数を定義する前に、関数適用の評価方法について少し補足があります。以下の例を思い出してください。

```
(define (square x) (* x x))
(square 3)
```

以前の説明では、(square 3) を評価すると、パラメータ x の部分が 3 に置き換わり、(* 3 3) が計算されるとなっていました。しかし、実際には置換されるわけではなく、変数 x が値 3 として定義された状態で、(* x x) が評価されます。つまり、以下の式が評価されるのに近いということになります。

```
(let ((x 3))
  (* x x))
```

数値を扱っている分には、特にこの辺りの違いを気にする必要はないのですが、リストやペアを受け渡しするようになると、この違いは重要になります。次のようなプログラムを考えることにします。

```
(define (my-car x) (car x))
(my-car (list 1 2))
```

この関数適用を評価すると、まず (list 1 2) が評価されます。より正確には、my-car と list という変数式も評価されますが、関数になるだけなので気にしないことにします。すると、その評価結果 (1 2) を用いて、関数の本体部分の評価が始まります。このとき、パラメータを値で本当に置き換えてしまうと、以下の式が評価されることになります。

```
(car (1 2))
```

この式は処理系で直接実行すればエラーになります。つまり、実際に行われている計算はこれではないということです。では、一体何が行われているのでしょうか？ 先ほど述べたように、実際には、(car x) がそのまま評価されています。ただし、トップレベルで (car x) を評価してしまうと、変数 x が未定義だったり、異なる値を指していたりするので、変数 x の値が (1 2) であるという条件が課せられています。上のように let を用いて、似た状況を表現するなら以下のようになります。

```
(let ((x (list 1 2)))
  (car x))
```

(1 2) ではなく (list 1 2) となっているのは、let による局所変数の宣言においても、変数が定義される前にその値となるべき部分の式が評価されてしまうからです。

このことを踏まえた上で、以降でリスト上の関数を定義することにします。

■演習問題　以前定義した sieve2 を変更して、素数のリストを返す関数を定義せよ。(素数のリストのための累積器を引数に加えればよい。)

3.4　帰納法（Induction）

リストは特殊なペアのことでしたが、別の観点からリストを特徴付けることを考えます。リスト全体のなす集合は、次の条件を満たす最小の集合だと言うことができます。

1. 空リストはリストである。
2. データとリストのペアはリストである。

このような定義を帰納的（inductive）定義と呼びます。[*6]帰納的に定義された集合を定義域とする関数を書くには、集合の定義に合わせて関数を書けばよいということが知られています。例を挙げて具体的に説明します。

```
(define (my-length l)
  (if (null? l)
      0
      (+ 1 (my-length (cdr l)))))
```

この my-length はリストの長さを計算する関数です。ここで、null? はリストが空であるかを判別する述語です。したがって、この関数は引数が空リストであるかどうかで場合分けをしている、ということが分かります。引数が空リストの場合は 0 を返しています。そうでない場合には、再帰的に my-length を適用しています。この構成は、上に書いたリストの定義とよく似ています。これがリストを受け取る関数の基本的な書き方になります。

よく使う組込み述語には、null? の他に、ペアかどうかを判別する pair? があります。pair? を適用した結果が真なら、car や cdr を適用してもエラーにならないことが保証されます。

[*6] これを定義とするには、データ全体が定義されている必要があります。データ全体も帰納的に定義することができます。

リストを受け取ってリストを返す関数も同様に書くことができます。例えば、以下の my-map は、引数として与えられたリストの要素全てに、別の引数として与えられた関数を適用して得られるリストを返す関数です。(my-map square (list 1 2 3 4)) の評価結果は (1 4 9 16) になります。

```
(define (my-map f l)
  (if (null? l)
      l
      (cons (f (car l)) (my-map f (cdr l)))))
```

my-length と同じ働きをする関数 length、my-map と同じ働きをする関数 map はそれぞれ組込みで与えられています。他にも、複数のリストを結合したリストを返す append、逆順のリストを返す reverse などが組込み関数として用意されていますが、同様の機能の関数を自分で定義することも可能です。演習問題として、是非挑戦してみてください。

また、次のように定義される関数 my-fold を使えば、リスト上の関数の多くを明示的な再帰なしに書くことができます。(処理系によっては、これと同様の働きをする組込み関数が存在することがあります。)

```
(define (my-fold op init l)
  (if (null? l)
      init
      (my-fold op (op (car l) init) (cdr l))))
```

■演習問題　リストを受け取って、何らかの順序で要素を整列したようなリストを返す関数を定義せよ。

3.5　連想リスト（Association Lists）

全ての要素がペアであるリストを連想リスト（association list）と呼びます。ペアの第 1 要素をキーとみることで、連想リストをデータベースとして使うことができます。

```
(define sample-alist
  '((1 . "One") (2 . "Two") (3 . "Three") (4 . "Four")))
```

この例では、sample-alist は数字をキーとした文字列のデータベースとみなすことができます。データベースからデータを取り出すには、関数 assoc を使います。

```
(assoc 3 sample-alist)
(assoc 7 sample-alist)
```

assoc は、第 1 引数をキーとして第 2 引数にあるデータベースを探します。キーが合致するペアが見つかった場合には、そのペアが返り値となります。見つからなかった場合は、評価値が #f となります。assoc が中のデータそのものではなくペアを返すのは、データとして #f が入っている場合とデータがない場合を区別するためです。データベースにキーが合致するペアが複数ある場合には、データベースリストの前方にあるものが優先されます。

assoc はキーの合致判定に equal? を使いますが、eq? を使いたい場合は assq を使います。equal? はリストを再帰的にたどって 2 つのリストが等しいかどうかを判別しますが、eq? はメモリ上の同一物かどうかを判別します。別々の cons や list で作られたリストは、eq? では異なるものと判別されます。数値やブール値の判別に関しては、eq? も equal? も変わりません。

```
(define l1 (list 1 2))
(define l2 (list 1 2))
(define l3 l1)
(equal? l1 l2)
(eq? l1 l2)
(eq? l1 l3)
```

これらの式の評価結果は予想通りでだったでしょうか。もし間違っていた場合には、理由をよく考えなおしてみてください。

4　評価戦略（Evaluation Strategies）

特殊フォームでは、どの部分式が先に計算されるかはフォームごとに決まっていました。lambda 式の評価の際には、部分式は評価されないことになっていますが、それについて少し見てみます。

display のような副作用がなければ、λ 閉包を作る際に lambda 式の中身を部分的に評価しても結果に違いはありません。例えば、以下で定義される 2 つの関数は、どんな引数に適用しても同じ結果を返します。

```
(define (triple x) (* 3 x))
(define (triple2 x) (* (+ 1 2) x))
```

しかしながら、Scheme ではこれら 2 つの λ 閉包は異なるデータ構造を持つことになります。この違いは副作用がある場合には、重要です。次の関数定義を処理系で評価してみます。

```
(define (zero) (display "Hello!\n") 0)
```

定義した際には特に Hello! とは表示されませんでした。次に関数適用を実行してみます。

```
(zero)
```

すると、Hello! と表示されました。つまり、変数 zero の値である λ 閉包の中に、(display "Hello!\n") という情報が入っているということになります。もし、lambda 式の評価の際に評価可能な部分式を評価しておくという規則があったとすると、zero が定義されるときに Hello! と表示され、関数適用の実行時には Hello! は表示されないはずです。

リストの形をした式で特殊フォームでないものは関数適用でした。関数適用の評価はリストの各要素に当たる式の評価から始まります。リストの要素は式なので、このルールが再帰的に適用され、複雑な関数適用は葉から根に向かって計算されていくということになります。このような評価戦略は、関数適用で関数の中身に入るより先に引数部分の評価が行われることから、「値呼び（call-by-value）」と呼ばれます。対して、引数部分の評価を後回しにして関数の中身の評価に入る戦略を「名前呼び（call-by-name）」と呼びます。名前呼びは引数の評価を後回しにしていることから、遅延評価（lazy evaluation）とも呼ばれます。

Scheme が値呼びであることを確かめるために、敢えて副作用のある以下のプログラムを実行します。

```
(let ((ignore (lambda (x) (display "second\n") 0)))
  (ignore (begin (display "first\n") 1)))
```

ここで、begin で始まる式は複数の式を順に評価する特殊フォームです。Scheme の処理系では、first、second の順に文字が表示されます。(式全体の評価結果である 0 もどこかに表示されます。) もし Scheme と同じ構文を持った名前呼びの言語があれば、そこでは、second のみが表示され、first は表示されません。

複数の引数がある関数適用の場合、引数同士の評価順序は仕様上定まっていません。また、関数適用の先頭要素も引数部分と同様のタイミングで評価されます。

```
(let ((add-and-apply (lambda (f x)
                       (lambda (y) (f (+ y x))))))
  ((add-and-apply square 1) 2))
```

このようなプログラムがあったとして、((add-and-apply square 1) 2) が評価される際には、全体の関数適用が計算される前に (add-and-apply square 1) の部分が評価されます。(この部分が計算できなければ全体も計算できないので、状況は名前呼びでも変わりません。)

上の例にあった ignore では、計算結果に引数は必要ないので、値呼びでは引数が無駄に計算されていることになります。これに対して、以下のようなプログラムを名前呼びで評価すると、(square 2) を 2 度計算することになってしまいます。

```
(let ((square (lambda (x) (* x x))))
  (square (square 2)))
```

純粋な値呼びと名前呼びでは計算の効率に関して、それぞれにメリットとデメリットがあります。Haskell のような近代的な遅延評価プログラミング言語では、一度計算した結果を覚えておいて、無駄な計算を省くように設計されています。

5　データ構造（Data Structures）

Scheme には、リストとペア以外のデータ構造がありませんが、リスト（ペア）は他の多くのデータ構造を表現するのに十分強力です。よくあるタイプのデータ型は、長さを固定したリストまたはペアとみなすことができます。C のように構造体の宣言はないので、「みなす」というのは、単に頭の中で思い描くという行為になりますが、そのデータを操作するのに特化した関数を定義することで、意図をより明確にすることができます。例えば、複素数を実部と虚部のペアで表現する場合には、以下のような関数を定義すればよいでしょう。

```
(define (complex x y) (cons x y))
(define (c-re c) (car c))
(define (c-im c) (cdr c))
(define (c-= c1 c2)
  (and (= (c-re c1) (c-re c2)) (= (c-im c1) (c-im c2))))
(define (c-+ c1 c2)
```

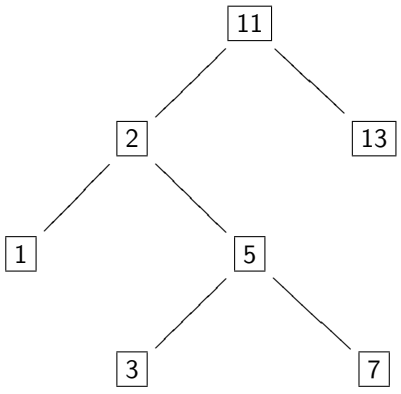
```
  (complex (+ (c-re c1) (c-re c2)) (+ (c-im c1) (c-im c2))))
(define (c-* c1 c2)
  (complex (- (* (c-re c1) (c-re c2)) (* (c-im c1) (c-im c2)))
           (+ (* (c-re c1) (c-im c2)) (* (c-im c1) (c-re c2)))))
```

また、データ構造を決める際にデータを操作する関数を定義することは、モジュール化の観点からも重要です。プログラムの作成中に、内部で用いているデータ構造を変更しなければならなくなった、という状況を考えてみましょう。プログラムの規模が大きければ大きいほど、その変更にかかる手間が多いことは容易に想像できるはずです。しかしながら、データを直接操作する関数が限られていれば、それらを変更するだけで全体の変更が完了することになります。一般に、モジュール化が十分になされていないコードは美しくないとされているので、プログラミングの際には関数を定義する手間を惜しまないように気をつけましょう。

Scheme のリストは入れ子にすることが許されているので、リストによって再帰的なデータ構造を表現することも可能です。二分木の場合、C なら以下のようにやるところですが、Scheme では長さ 3 のリストで表現することができます。

```
struct tnode {
    struct tnode *left;
    void *data;
    struct tnode *right;
}
```

今回は、リストの第 1 要素を左の部分木、リストの第 3 要素を右の部分木、リストの第 2 要素をデータだと決め、空木は空リストで表現することにします。



この木をリストで表現すると次のようになります。[*7]

```
((() 1 ()) 2 ((() 3 ()) 5 (() 7 ())) 11 (() 13 ()))
```

二分木を操作する関数は以下のようになります。ここで、cadr は `(lambda (x) (car (cdr x)))`、caddr は `(lambda (x) (car (cdr (cdr x))))` の略です。利便性のために、Scheme にはこのような関数がいくつ

[*7] 葉を空木ではなくデータにすれば、より見やすい形で表現することができますが、本稿では、空リストがリストであることとの対比から、空木も木であるとしています。

か用意されています。

```
(define (btree-empty) (list))
(define btree-null? null?)
(define btree-root cadr)
(define btree-left car)
(define btree-right caddr)
```

リストと同じく二分木は帰納的なデータ型なので、二分木上の関数は再帰的に定義することが可能です。例えば、木の深さを計算する関数 btree-depth は以下のように定義されます。リスト上の関数の定義と比較してみてください。

```
(define (btree-depth t)
  (let ((max (lambda (x1 x2) (if (< x1 x2) x2 x1))))
    (if (btree-null? t)
        0
        (+ 1 (max (btree-depth (btree-left t))
                  (btree-depth (btree-right t)))))))
```

■演習問題　二分探索木に対する検索、挿入、削除などの操作を実装せよ。(挿入や削除の場合には、二分探索木を受け取って二分探索木を返す関数を定義すればよい。)