RacketTextbook-JP

~1から100までのRacket~

Kudzuyu

目次

序文

概要

　この文書は、現在日本語の情報が少ないプログラミング言語Racketについて、プログラミング初心者の方でもわかりやすいように解説し、Racketのプログラミングがある程度できるようになることを目指しています。これからプログラミングを始める人のために経験者の方には少し表現がくどくなっているかもしれませんが、ご了承ください。最新版は、Githubのリポジトリからダウンロードすることができます。

この文書の対象者

　以下のような人には、この文書が役に立つかもしれません。

- 関数型言語に興味がある人
- Racket以外のLisp系言語をやっていて、Racketに興味がある人

　全くのプログラミング初心者という人も理解できるように心がけていますが、他の言語との比較の部分はわかりにくいかもしれません。(最初の言語にRacketを選ぶ人は少ないと思いますが)

英語について

　この文章の中では中学生程度の英単語を使います。しかし、Racketでプログラミングをする中で、エラーメッセージなどは英語で出力されます。巻末にエラー一覧を載せていますが、英語は読めたほうがいいでしょう。

はじめに-RacketとLisp

　1950年代にジョン・マッカーシーにより、Lispというプログラミング言語が作られました。Lispは当時の主流であったプログラミング言語(FortranやCobolなど)とは構造がかけはなれていたためあまり流行ることもなく、また当時のコンピュータの処理性能にたいしては重たい言語であったためさほど普及しませんでしたが、他の言語にはない様々な特徴があったため、人工知能などの研究用として残りつづけます。

　Lispはその後、2つの大きな方言(元のプログラミング言語にたいし基本構造はかわらないが、細かな違いや改良がみられるプログラミング言語)を産みました。Common LispとSchemeです。それぞれに目指しているものが違い、Common Lispは多機能で様々なプログラムができるように、Schemeは言語としてのシンプルさを追求していました。 RacketはSchemeから派生したLisp方言の1つです。かつてはPLT Schemeと呼ばれていました。RacketはSchemeとの互換性もありながらかなり多機能な言語となっており、Common LispとSchemeの中間的存在と言えるかもしれません。

Racketの特徴

- RacketはSchemeから派生したため、Schemeとよく似ています。そのため、Schemeを強化するために作られたライブラリ集srfiを組み込んで使うことができます。（現在Racketで使えるsrfiは、Racketのsrfiのページを参照してください。）
- Racketにはデフォルトで GUIの開発に必要な関数郡が揃っています。これにより、これまでのCommon LispやSchemeでは環境をそろえることすら難しかったGUIについて、手軽に触れることができるようになっています。
- Schemeは、必要最小限の言語であることを目指しているため、実用にはあまり向かない言語と言われていますが、Racketは前述のGUIを始め多数のライブラリを揃えており、本格的なソフトウェア開発にも適しています。

Racketプログラミングのための準備

Racketは公式がプログラミング用のソフト(IDE)を配布しているので、とりあえずはこれを使うことをお勧めします。自分のお気に入りのエディタがあるひとはそれを使ってもいいとおもいますが、マイナーな言語ですのでエディタによってはシンタックスハイライト(ソースコードの色分けや、文法ミスの検出などの機能のこと。ソースコードを見やすくするためのもので、なくても問題はありませんがあったほうが捗ります)がないこともあります。

インストール

- **Windows**
  Racket公式サイトにアクセスし、画面上部の「Download」をクリック。Distributionが「Racket」、PlatformがWindowsであることを確認して、ダウンロード。exe形式なので、そのままインストールしてください。一応ミラーサイトもあります。
- **Mac**
  Windowsと同じようにRacketの公式サイトからダウンロードしてください。他の方法もあるかもしれませんが、筆者はMacを使ったことがないのでわかりません。公式サイトからで問題はないと思います。
- **Linux(Debian,Ubuntu系)**
  Windows、Macと同じようにRacketの公式サイトからダウンロードする方法と、aptリポジトリを使う方法があります。公式リポジトリは ppa:plt/racketです(アップデートを自動でやってくれるため、こちらのほうがおすすめです。)
- **Linux(Archlinux系)**
  pacmanからインストールできます。AURには、githubのコードを用いた開発中の物も上がっています。
- **Minimal RacketとRacketの違いについて**
  Racketをダウンロードするときに、Minimal Racketというものもあるとおもいます。これは、Racketとしての必要最低限の機能だけを搭載し、あとから機能を追加できるようにしたものです。この文書は普通のRacketを導入することを前提で書いていますので、特に理由がなければRacketをダウンロードすることをおすすめします。

起動および日本語化

インストールすると、ソフト一覧に、DrRacketが追加されると思います。これがRacketのIDEです。この記事では、これを使ってプログラミングすることを想定して書かれています。他のエディタを使っている人は、適宜読みかえてください。

DrRacketを起動すると英語版として立ちあがるかもしれません。このままでも問題はありませんが、日本語がいいという方は、メニュータブの右端「Help」をクリックしてください。さまざまな言語でかかれたメニューの中に、「DrRacketを日本語で使う」という項目があると思います。クリックすると、DrRacketを再起動するか聞かれるので、承認して再起動させましょう。これを行なうことで、それ以降はDrRacketを日本語で使えるようになります。

ライセンスと商用利用について

Racketは、LGPL(Lesser General Public License)により配布されています。このライセンスは商用利用に関して問題が起きることがあります。(Googleなどで、「LGPL 商用」などで調べると、様々な意見が混在していることがわかると思います。)Racketのサイトでは、商用開発に関して制限するつもりはないと書かれていますが、まだ見送ったほうがいいでしょう。もしどうしても商用プログラムを開発したいという方は、同じScheme系の処理系のGaucheや、有名なLisp系方言の一つであるCommon Lispなどで開発することをすすめます。

Gaucheでは、Racket特有の関数以外は共通に使うことができますし、Common Lispは機能の点において、Racketにおとりません。(文法がすこし違っていますが、なれれば気にならないでしょう。)

ドキュメントについて

　Racketは、公式サイトにてドキュメント(辞書のようなもの)や初心者向けの文書が公開されていますが、全て英語なので日本人の学習には非効率です。ただ、新機能のアナウンスや新機能の解説などはそちらの方が早いので、英語が読める人はのぞいてみるといいでしょう。

まとめ

- Lispの一方言、Racket
- ダウンロードは公式サイトから
- 開発環境もついてくるので手軽に始められます。
- 商用利用は難しそうですが、商用利用可能でよく似た言語があります。そちらを利用してみるのもいいでしょう。

第一部 プログラミングRacket 基礎編

ここではRacketの基本的な文法について解説します。Racketの派生元であるSchemeの解説としても使えると思います。 文章中にでてくる練習問題は、章末に解答を載せてあります。全てやりきれば、Racketで書かれたソースコードはほぼとまどうことなく読めるようになり、自分自身でもプログラミングができるようになるでしょう。

この部では、以下のような事を学びます。

- RacketでHello World!の出力
- リストの操作や合成
- 算術演算子と条件演算子
- 関数定義
- 条件分岐
- 繰り返しと再帰定義
- ラムダ式

Racket基礎その1 Hello World!とリスト

まずはお決まり、Hello World!

DrRacketを起動させると、#lang racket と書かれた部分とその下の「>」というの部分が目につくと思います。「#lang racket」と書かれている部分はエディタと呼ばれ、プログラムを実際に書いていく所です。

「>」という部分は、Lisp系の言語の開発環境に付属する対話環境とよばれるものです。他のプログラミング言語とはすこし違いますが、使っていくうちになれます。

それでは、Racketのプログラミングを始めていきましょう。まずは、色々な言語で最初にやる画面に「Hello World!」と表示するプログラムを作ってみましょう。以下のコードを書きこんでください。全ての文字は、半角を使用してください。(#lang racketの部分は、もとから書かれていますので、これ以降は基本省略します)

```
#lang racket

(display "Hello World!")
```

書き込んだら実行してみましょう。ソフト上部の「実行」と書かれた緑色の三角形のボタンを押してください。(保存しなくても実行できます。) 実行すると、画面下部の対話環境に、以下の様に表示されると思います。エラーメッセージがでた人は、英語のつづりが間違っていないかや、カッコが半角であるか、全角空白が入っていないかどうかを確認してください。

```
Hello World!
>
```

では、このプログラムについてみていきましょう。 Racketの世界では、全てのプログラムおよびデータは()-丸カッコでくくられます。これをリストといいます。Racketの世界では、プログラムとプログラムで扱うデータを同じようにリストで表現します。このうち、プログラムとしてのリストは、先頭の要素および残りの要素に大別されます。このプログラムでいうと、「display」が先頭の部分にあたり、「"Hello World!"」が残りの部分にあたります。

プログラムとしてのリストの先頭と残りには、それぞれ役割があります。先頭の要素が関数(プログラムの名前)であり、大まかな動作を決定します。残りの要素は「引数(ひきすう)」と呼ばれ、そのプログラムの動作を細かく決定するものです。 この場合「display」は「画面に引数にある文字を出力せよ」という関数になります。従って、引数である「Hello World」が出力されます。 "Hello World"の""の中を自分の好きな文字列に変えて、実行してみてください。自分の入れた文字列が表示されたはずです。displayという命令が変わっていないので画面に文字列を出力するということは同じですが、引数が違うため動作が違うのです。

なお、"Hello World" を囲んでいる「"」は、中に含まれた文字をそのままdisplayに送るための印のようなものです。今はあまり気にしないでください。

Racketの構造

Racketでは、リストを次のように括弧で括って書きます。次のリストは、a、b、cの3つの要素をもつリストです。

```
(a b c)
```

リストは多層構造にもできます。次のリストは、1年の各月とその月の日数を表わしたリストです。

```
((1 31)(2 28)(3 31)(4 30)(5 31)(6 30)(7 31)(8 31)(9 30)(10 31)(11 30)(12 31))
```

数字と単語を組みあわせることもできます。次のリストは、出席番号と名前のリストです。

```
((1 John)(2 Jack)(3 Chris)(4 Thomas) ...)
```

これ以外にもリストで表わせる情報はたくさんあります。みなさんも2、3個作ってみてください。

評価

ここまで見てきたリストは、いづれも「データとしてのリスト」でした。これだけでは、(display "Hello World")と見分けがつきません。Racketはどのようにしてデータか、プログラムかを区別しているのでしょうか。

プログラムとしてのリストは、最初が関数、残りの部分が引数であるという話をしました。 リストの先頭要素が関数であった場合に、その関数を実行することを評価といいます。さきほどの

```
(display "Hello World!")
```

のプログラムは、関数「display」に引数「"Hello World!"」を評価させていたということです。

では、このdisplayに、リストを与えてみましょう。 (これを実行するとエラーがでます)

```
(display (1 2))
```

このエラーは、リストを引数に与えたからでたわけではありません。引数として与えられたリストの先頭のアトムが、関数でなかったからおきたものです。

プログラムを実行すると、Racketはリストの先頭の要素を見にいきます。displayはRacketで関数として登録されているので、その定義にしたがってリストの残りの要素を画面に出力しようとします。すると、残りの要素はリストということになりました。このときRacketは、そのリストを評価しにいきます。このプログラムでは、(1 2)の部分です。しかし先頭の要素が「1」で、関数ではないためRacketは評価ができず、エラーを返したのです。 このような場合にはどうすればいいでしょうか。次のプログラムを見てください。

```
(display '(1 2))
```

このプログラムを実行すると、エラーが出ずに、(1 2)と出力されます。実は、リストの前に'(クオート)を付けるとRacketはそのリストを単なるデータとみなし、評価をしないのです。データとしてのリストと、関数としてのリストはこのようにして区別されるのです。 皆さんも様々なリストを入れてみてください。なお、Hello worldの時のように"(1 2)"としても出力することができますが、リストに対しては、'を使ったほうが便利なことが多いのでこちらを使っていきましょう。

コメント

Racketをこれからプログラミングしていく中で、プログラムの内容を日本語でメモしておきたいと思うかもしれません。プログラムが小さいうちならいいのですが、数百行から数千行といった巨大なものになると、一部のプログラムを見ただけでは理解できないものがでてくるでしょう。プログラミング言語にはコメントと呼ばれる機能があり、コンピュータが無視する文章をプログラム中に埋めこむことができます。

プログラムの最初にセミコロン(;)を書くと、その行の残りの文字はRacketに認識されなくなります。

```
#lang rakcet

(display "Hello World")
; 日本語訳「こんにちは、世界」
```

また、行の途中にセミコロンを置くと、その行のセミコロンより後をコメントにすることができます。

```
#lang racket

(display "Hello World") ;日本語訳「こんにちは、世界」
```

コメントは多すぎるとプログラムが逆に見づらくなってしまいます。実際には上のような短いプログラムでコメントは必要ないでしょう。

問題

- (Racket Ball Tennis) というプログラムがあったとします。(架空の物です)。この時、関数と引数はどれでしょう。
- 次のプログラムからコメントを消してください。動作が削除前と削除後でかわらない事を確認してください。。

```
(display "日本語も出力できます。") ; 日本語も
; 出力できます。
```

まとめ

- Racketはリストとよばれる括弧()を使った構造で、プログラムやデータを書く。
- リストは、多層構造にすることができる。
- Racketのプログラムを実行すると、リストの先頭の要素は
- リストを評価すると先頭を関数とし残りの部分を引数として実行する。
- リストを評価されたくない時は、前カッコの前に'(クオート)をつける。
- セミコロンを使って、プログラムにコメントを付けることができる。

解答

関数:Racket 引数:Ball、Tennis

```
(display "日本語も出力できます。")
```

Racket基礎その2 リストの操作とアトムについて

対話環境の使い方

さて、前回まではエディタ部分にプログラムを書いて実行してきました。しかし、一々前回のプログラムを消して書きなおすのも面倒です。これまでプログラムを実行すると下の対話環境に出力されていましたが、そこに直接プログラムを書いてみましょう。次のプログラムを書いて、エンターキーを押してください。

```
>(display '(one two three))
'(one two three)
```

エディタ部分に書いて実行したときと同じように、displayが実行されました。このように対話環境はテキストに書かれたプログラムの実行だけでなく、直接プログラムを実行することもできるのです。そして、この方式だと、直ぐに次のプログラムを書きこむことができて便利です。 今の所、対話環境だけで押し通してもRacketの習得上問題はないので、当分は手軽に対話環境にプログラムを直接打ち込む方式でいきましょう。(プログラムをテキストに書きこむ方式がいいという人は、別にかまいません。結果がかわるわけではありませんから)やがてテキストに書くプログラムが必要になったら、あらためてテキストに戻ることにしましょう。 なお、対話環境に書きこんで実行している時は、プログラムの先頭に「>」を付けてありますので、それが無いときはテキストに書きこむ方のプログラムだと考えてください。

リストの各要素、アトム

もう少しプログラミング以前の話がつづきます。 前回までで、リスト(1 2 3)の1や2などを、要素と呼んできましたが、これからはRacketおよびLispの伝統にしたがって、アトムとよぶことにします。アトムとは英語で「元素」の意味であり、リストを構成しているものという感じがしますね。アトムが組みあわされたリストは、「分子」といったところでしょうか。

```
((a b) (c (d e)))
```

上のように多層構造のリストの場合、アトムは、a、b、c、d、eの5つのことです。全てのカッコをとったのこりと考えるとわかりやすいでしょう。

リスト操作のおきまり、carとcdr

では今回の本題です。今回は、リストの中から特定のアトムを抜きだす操作を覚えましょう。次のプログラムを実行してください。

```
(car '(a b c))
```

```
(cdr '(a b c))
```

いかがでしたでしょうか。それぞれリストの先頭のアトム「a」および残りの部分「'(b c)」が出力されたと思います。クダー部のリストに'がついているのは、評価前のリストがすでに'で評価されないようにしていたからです。また、カー部であるaの方に'がついていないのは、数字に関しては評価しないという取り決めがあるからです。(先程のdisplayとは、出力時の文字の色が違うかもしれませんが、いまは気にしないでください) これらの関数はRacketの先祖であるLispができたころから存在していた関数です。Racketに限らずLisp系の言語はリストの操作にこの名前を使っている言語が多いです。 これまでリストを先頭の要素と残りの要素というふうに

分けてきましたが、これらの関数名に従い、これからはそれぞれ先頭の要素をcar(カー)、残りの要素をcdr(クダー)と呼ぶことにします。
　リストは、多層構造にすることもできるという話をしました。では、次のプログラムを見てください。

```
(car '((1 2)(3 4)))
```

```
(cdr '((dog)(cat (mouse rabbit))))
```

　これを実行すると

```
>(car '((1 2)(3 4)))
'(1 2)
>(cdr '((dog)(cat (mouse rabbit))))
'(cat (mouse rabbit))
```

　このようになりました。リストが多層構造になっている時にcarやcdrをとると、リストの中のリストはアトムと同じように扱われます。(car '((a b) c))が、aではなく'(a b)となる事に注意してください。

問題

　以下のリストについてcarとcdrを推測してください。 ヒント:carとcdrを考える時のコツは、リストの一番外側の「(」前カッコを、右に一つずらすということです。

```
(1 2 3 4 5)
(((1 2)(3 4)))
(((1))(1 2))
```

　いかがでしたか。章末に答えをのせていますが、実際に打ちこんでしらべた方がはやいかもしれません。もし巻末の答えが実際の挙動と違っていた場合は、githubのissuesにお願いします。 ではもうひとつ、

```
(a)
```

　これのcarとcdrをとってみましょう。carは予測がつくと思いますが、cdrはどうなるでしょう。

```
>(car '(a))
a
>(cdr '(a))
'()
```

　中にアトムがない、ただのカッコがでてきました。これは空リストとよばれるもので、要素を1つも持たないリスト(要素0個のリスト)です。では、このリストのcarとcdrを取るとどうなるでしょう。

```
>(car '())

>(cdr '())
```

どちらもエラーがでました。Racketでは空リストのcarやcdrをとってはいけないことになっています。これは後々条件分岐の所で重要になってきますので、覚えておいてください。(他のLisp系言語をやったことのあるひとは、空リストのcdrがエラーになることや、nilが出ないことなどに違和感をいだくかもしれませんが、わりきってください) Racketの定義では、carとcdrは、1つ以上のアトムを含むリストを引数にとることとなっています。

carとcdrの複合技

ここまで、carとcdrでリストを評価するとどこのアトムやリストがでてくるのかをみてきました。では、リストの2番目や3番目の要素が必要になったときは、どうすればいいでしょう。 ここで、cdrに注目してみましょう。

```
>(cdr '(1 2 3))
(2 3)
```

上のプログラムで、carをとったあとのリストを見てみると、これもリストであり、carやcdrをつかうことができる構造になっています。これで、リストの2番目のアトムを取りだすめどがたちました。

```
>(cdr '(1 2 3))
'(1 2)
>(car '(1 2))
1
```

このようになりました。今は一回づつやりましたが、これは組みあわせて使うことができます。

```
>(car (cdr '(1 2 3)))
1
```

このプログラムを実行すると、Racketは始めにリストの先頭要素をみて、carなので残りのリストを評価しようとします。しかし、評価すべきリストもcdrというプログラムなので、中のcdrを先に評価して、帰ってきた値を更にcarに渡すのです。 これでリストの2番目や3番目をとりだすことができました。Racketにはリストの2番目や3番目を一発で返す関数も用意されています。cadrや、caddrなどです。

```
>(cadr '(chicken pork beef))
'pork
>(caddr '(tuna salmon swordfish))
'swordfish
```

carやcdrの組みあわせ技なので、命名規則もそれにのっとっています。c{}rの{}のあいだには、carを表すaか、cdrを表わすdかが、その関数を使う順番に **後ろから** はいっています。つまり、caddrならcdr→cdr→carと実行したのとおなじということです。一見後ろからいれるのは不思議に思えるかもしれませんが、先程の入れ子プログラムをみるとわかりやすいかと思います。

```
(car (cdr '(1 2 3)))
(cadr '(1 2 3))
```

この2つのプログラムは同じはたらきをします。cadrのaとdの並び方が入れ子にしたプログラムのcarとcdrの位置関係と同じことに気づかれたでしょうか。わからなくなったときは入れ子のプログラムを思いだすと理解しやすいでしょう。

では、次のようなリストを考えてみましょう。 食品の名前とその値段のリストです。

```
((Egg 100) (Milk 200) (Tomato 150))
```

このリストにおいて、例えばMilkの値段を取りだしたいときは、carとcdrをどう使えばいいでしょう。

困った時は、とりあえずcarやcdrを使ってみましょう。

```
>(car '((Egg 100) (Milk 200) (Tomato 150)))
'(Egg 100)
>(cdr '(Egg 100) (Milk 200) (Tomato 150))
'((Milk 200) (Tomato 150))
```

なおcarとcdrの複合関数はいくらでも使えるというわけではなく、caddddrくらいまでとなっています。これ以降のアトムを取りだしたいときは、(cadddr (cddddr ...))という風にするのが得策でしょう。あえてリストの20番目を一発でとる関数などがほしい人は、ここから何章か進んだ関数定義の章や再帰定義の章などを学習することで自分で作ることができるようになります。

まとめ

- リストを構成する各要素のことをアトムという。
- リストの先頭のアトムはcar、先頭を除く残りのリストはcdrで取り出すことができる。
- carとcdrはリストを入れ子にすることで2番目以降のアトムを取りだすことができる。
- 入れ子にする以外にも、リストの5番目程度まではcadrやcaddrなどの関数が用意されている。
- それ以上の場合は、入れ子にするか後々関数定義や再帰定義などを学ぶことで自分で作ることができる。

練習問題の解答

```
>(car '(1 2 3 4 5))
1
>(cdr '(1 2 3 4 5))
'(2 3 4 5)
>(car '(((1 2)(3 4))))
'((1 2))
>(cdr '(((1 2)(3 4))))
```

```
((3 4))
>(car '(((1))(1 2)))
((1))

>(cdr '(((1))(1 2)))
(1 2)
```

Racket基礎その3 リストを作る

consとlist

　前回は、リストの中からアトムやリストを抜き出す操作をやってきました。今回は、リストを作りだす関数についてです。 リストを作りだすには、consという関数を使います。

```
>(cons 'a '(b))
(a b)
```

　3つ以上のリストも作れます。

```
>(cons 'a (cons 'b (cons 'c '(d))))
(a b c d)
```

　また、carとcdrで分解したリストは、consで合成しなおすことができます。

```
>(car '(a b c))
'a
>(cdr '(a b c))
'(b c)
>(cons 'a '(b c))
'(a b c)
>(cons (car '(a b c)) (cdr '(a b c)))
'(a b c)
```

　とはいえ、このようにしてリストを作るのは非効率な気がしますね。(cons 'a 'b 'c 'd) -> (a b c d)とやってほしい気がします。Racketにはlistという、もっと簡単にリストを作りだす関数も用意されています

```
>(list 'a 'b 'c 'd)
'(a b c d)
>(list '(a) '(b) 'c 'd)
'((a) (b) c d)
```

　一番最後の例でわかるように、car/cdrとconsは、互いに逆の動作をする関係にあります。(cons 'a 'b 'c 'd)としないのはこのためです。
　あきらかにconsよりも楽になりました。これを見るかぎりは、リストを作る関数はlistだけでいい気がします。なぜconsがあるのでしょう。 実は、consでないとできないことがあるのです。car/cdrとconsを組みあわせたプログラムを見てみましょう。

```
>(cons (car '(a b c)) (cdr '(a b c)))
'(a b c)
```

listを使うと、このようになります。

```
>(list (car '(a b c)) (cdr '(a b c)))
'(a (b c))
```

問題

- 次のリストをconsを使って作ってください。 ```racket (A B (C D (E)) (F))
((A (B (C (D (E F))))))
((((((A B) C) D) E) F)) ``` いかがでしょうか。リストは他の言語では表舞台にあまりでてきませんが、Racketを初めとするLisp系言語ではこれでもかと使われます。特に、Lisp族に特有の機能を使う時にその真価が発揮されますので、しっかり覚えておきましょう。

まとめ

- carとcdrで分解したリストは、consで合成しなおすことができる。
- リストを作る関数はcons以外にもlistなどがある。
- listでは作れない構造のリストもconsなら作れる。

解答

```
>(cons 'A (cons 'B (cons (C D (cons 'E)) 'F)))
```

Racket基礎その4 算術計算と条件演算子

リストで計算

前回までは、リスト操作の基本関数であるcarとcdr及びその派生関数について解説してきました。今回は少しかわって、計算を行なう関数と条件演算子とよばれるものについてです。次回以降の自分で関数定義をしていく章において重要となってくるので、がんばってください。

まずは、リストを使って計算をしましょう。Racketには、displayやcarなどのように、数学的な計算を行う関数も用意されています。四則演算を行う関数のことを特別に、算術演算子といいます。

```
(+ 2 4)
(- 12 4)
(* 5 9)
(/ 35 7)
```

これらは上から順に、足し算、引き算、かけ算、割り算をあらわしています。かけ算で×を使わないのは、キーボードに×の記号を表す文字がなかったからです。割り算が/なことに関しては、コンピューター上で分数をあらわす時の約束としてこれまでにも見たことがあるとおもいます。

算術演算子がこれまでのcarなどと違う点は、引数をいくらでもとれるところです。

```
>(+ 1 2 3 4 5 6 7 8 9 10)
55
>(* (+ 2 2) (- 5 4) 2 4)
32
```

元々Racketの始祖であるLispは、数学的な考え方から生まれました。このように算術演算子を一番前において、後ろに計算する数字を並べる書き方のことを前置記法といい、私達が普段使っている、「3＋4」などの書き方は、中置記法といいます。前置記法のいいところは、複数の数字にたいして同じ演算をする(まとめて足し算やかけ算をする)時に、算術演算子を1つ書けばすむことです。この考えかたが後々、Racketにでてくる算術演算子以外の、沢山の引数をとる関数などのところで役立つことになります。

条件判定

算術演算子に続いて、条件演算子について見ていきましょう。この関数群はそれ単体では大した機能をもちませんが、状況によって動作を変えるプログラムを作る時にとても役に立つものです。 条件演算子は、あたえられた引数にたいし、「真」をあらわす#tか「偽」をあらわす#fを返します。

```
>(list? '(a b))
#t
>(null? '(a b))
#f
>(null? '())
#t
```

上にあげたのは、条件演算子をつかった例です。関数名からなんとなく機能がわかると思います。list?は、与えられた引数がリストかどうかを、null?は引数が空リストかどうかを調べます。「'」をつけるのを忘れないでください。 さて、今あげた関数はいずれも「引数が~であるか?」を調べる1引数の関数でした。しかし、これだけでは2つの物を比較することができません。Racketにはそういう関数も用意されています。

```
>(= 10 10)
#t
>(= 10 9)
#f
>(> 10 9)
#t
>(< 10 9)
#f
>(>= 10 9)
#t
>(<= 10 9)
#f
```

条件演算子「=」は、与えられた2つの数字が等しいかどうかを判定し、「<」と「>」は大きいか小さいかを判定します。 最後の「>=」と「<=」は以上、以下を判定します。=の位置は「<、>」よりも後に置かなければいけません。「=<」のようにするとエラーとなります。

条件演算子も算術演算子と同じように、3個以上の値を取ることができます。

```
>(= 8 8 8 8)
#t
>(< 1 2 3 4 10)
#t
>(< 1 2 3 3 4 4)
#f
>(<= 1 2 2 3 3 4)
#t
```

普段の数学の表記になおすと、これは以下のようになります。個々の数字の間に指定された算術演算子をあてはめてみて、正しい表記になっていたら#tが、1つでも違っていたら#fが返ります。

```
(= 8 8 8 8 ) -> 8 = 8 = 8 = 8                    <-正しい
(< 1 2 3 4 10) -> 1 < 2 < 3 < 4 <10              <-正しい
(< 1 2 3 3 4 4) -> 1 < 2 < 3 < 3 < 4 < 4         <-3 < 3と、4 < 4が正しくない。
(<= 1 2 2 3 3 4) -> 1 <= 2 <= 2 <= 3 <= 3 <= 4   <-正しい
```

Racket基礎その5 関数定義

defineで関数を定義

いよいよ今回から、自分で関数を定義していきます。これによって、一応Racketのプログラムが書けるようになります。今回からは、又プログラムを上のファイルに書きこんでいきましょう。今回使う関数は、defineというものです。defineの構造はこれまでの関数と少しちがっていて、次のようになっています。

```
(define (関数名 引数...) (関数の内容))
```

これまでの関数と違うところは、まず、引数をdefineの実行時に評価しないことです。例として、与えられた数を二乗する関数をみてみましょう。関数名はsquareとします。

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

これをファイルの方に書いて実行すると、テキストカーソルが下の対話環境の方に移ると思います。ここで、関数squareを実行してみましょう。

```
>(square 15)
225
```

ちゃんと答えが返ってきました。racketでは、ユーザーの作った関数はもとからある関数と同じように使うことができます。したがって関数の入れ子をユーザーの作った関数と基本関数で作ることもできます。

```
>(square (car '(11 12 13)))
121
```

defineが引数を評価しないという点について、もう一度先程のsquare関数の定義を見てみましょう。

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

このように書いた場合、もしdefineが引数を評価してしまうと、squareという関数がないのでエラーがでてしまうのです。

したがって、S式同士を接続するというのがdefineの機能の表現としていいかもしれません。例えば上のsquareの式では、(square x) と、(* x x)が接続されるのです。この後、(square 5)などという式を実行すると(* x x)という式に、 x に5を代入して、(* 5 5)が評価されることとなります。

defineで定数を定義する。

defineは値を設定することにも使えます。次の例を見てください。

```
>(define x 20)

>(+ x 10)
20
```

xに値10が設定されました。この方法をつかうと、特定の数字や文字などに名前を付けて使いまわすことができます。これを変数といいます。

```
(define pi 3.14)
(define (circle-area r)
  (* r r pi))
```

上のプログラムは最初に円周率piを設定し、次の関数で円の面積を求めています。こうしておくと、piの数値を3.1415などと変更したときに、面積のプログラムの方を変更する必要がないので、便利です。また、同じ変数を使う別のプログラム(上のプログラムではpiを使う球の体積や表面積などの関数)にも使いまわせます。実際のプログラムでは、値を定義する関数を先頭に書いておいてその下にそれを使ったプログラムを書く形式が多く見られます。

インデント

上の円の面積を求めるプログラムの中で、circle-area関数の2行目は、2文字分字下げされています。これはプログラムを見やすくする工夫で、インデントとも呼ばれます。この文書の中ではインデントは2文字分としていますが、4文字字下げの物も多くみかけます。皆さんもプログラムを作る時には、このように字下げすると見やすくなるでしょう。

インデントはスペースキーで半角空白を打ってもできますが、Tabキーを使うと、簡単です。Tabキーが半角空白2個分にならない場合は、DrRacketの設定から変えられます。

defineの注意点

後で詳しく解説しますが、Racketにおいて元からある関数とユーザーが作った関数は、同じように一ヶ所で管理されます。このことは、元からある関数(例:carや、cdr、defineなど)を名前にした新しい関数をユーザーが作った場合、そのプログラムの中では元の関数としての機能は失われてしまうということを意味しています。

```
>(* 2 3)
6
>(define * +)
>(* 2 3)
5
```

もともと「*」はかけ算の演算子ですが、「define」で「*」を「+」にしてしまったので、足し算になってしまいました。 このように、元からある関数を上書きしてしまわないように、注意しましょう。 (とはいっても、二度と変更できないわけではなく、Racketを再起動すれば元にもどります。ソースコードの中にこのようなコードが入っていると、再起動しても上書きされ続けてしまいますので注意してください) これは、Common Lisp系のLispとScheme系のLispで違うことのひとつです。(Common Lispでは、ユーザーが作った関数と元からある関数は別に管理されているので、このようなことは起きません。くわしくは「Common Lisp 名前空間」などで調べてください。) 自分が作った関数が既に定義されているかどうかは、Racketのドキュメントで検索してください。

問題

- 2次関数の値を計算する関数を作りましょう。関数の引数は4つとし、先頭の3つは2次関数の係数($2x^2 + 3x + 1$の、2や3や1)を示し、4つ目が変数(x)を表わすとします。

まとめ

- defineを使うことで、自分で関数を定義できる。

- 作った関数はcarやcdrなどと同様に使え、組み合わせることもできる。
- 元からある関数との名前が被らないように注意する。

解答

```
(define (quad-func x1 x2 x)
  (+ (* x1 x1 x) (* x2 x) x3))
```

実行結果

```
>(quad-func 4 1 2 2)
12
```

Racket基礎その6 条件分岐

この回は、Racketの条件分岐について解説します。前々回に、Racketの条件演算子について学びました。今回はこれとdefineを使って、条件によって様々な変化をする関数を作っていきましょう。

if文

最初は、単純な条件分岐である、if文を使ってみましょう。if文の構造は次のようになります。

```
(if (条件)
    (動作1)
    (動作2))
```

if文を実行すると、条件の文が評価されます。評価結果が真の場合は、動作1が、偽の場合は動作2が実行されます。 たとえば、受けとった数字が10より大きいか小さいかを判定する関数は、以下のようになります。

```
(define (ten x)
    (if (>= x 10)
        (display "10以上")
        (display "10より小さい")))
```

この関数は、引数xをとり、xが10以上が否かで、それぞれの動作をします。条件が真の時も偽の時も、片方の動作しか実行されないことに注意してください。

if文は中にif文を入れることで、2個以上の分岐も作れます。

```
(define ())
```

begin文

さて、ここまでのif文では、条件分岐した時に実行される文は1つでした。では、1つのif文に対して複数の動きをさせるにはどうすればいいでしょう。

以下の例は失敗です。複数の動作をまとめて括弧でくくるだけでは、Racketはそれをプログラムと認識してくれません。プログラムとしてのリストは、car部にリストがあってはいけないのです。

```
(define (ten1 x)
    (if (>= x 10)
    ((display "与えられた数")(display x)(display "は10以上です。"))
        ((display "与えられた数")(display x)(display "は10より小さいです。"))))
```

複数の関数をまとめて実行するには、beginという関数を使います。

```
(define (ten1 x)
    (if (>= x 10)
        (begin (display "与えられた数")
           (display x)
```

```
            (display "は10以上です。"))
         (begin (display "与えられた数")
            (display x)
            (display "は10より小さいです。"))))
```

これの実行結果は次のようになります。最初の関数よりも見やすくなりました。

beginは、cdr部にある関数を1つずつ実行する関数です。これによって、書くというような整形された

cond文

続いて、より一般的な条件分岐であるcond文を紹介します。condはifと違い、3方向以上の分岐も可能になっています。

```
(cond
    (条件1 動作1)
    (条件2 動作2)
    (条件3 動作3)
    ...
  (条件n 動作n))
```

condは、条件を上から順に評価し、評価結果が真になったときにその条件式に対応した関数が実行されます。ifと同じように関数を定義してみましょう。

```
(define (+-0 x)
    (cond
        ((> x 0)(display "plus"))
        ((= x 0)(display "0")
        ((< x 0)(display "minus")))))
```

このプログラムを実行し、対話環境で試してみると以下のようになるでしょう。

```
>(+-0 4)
plus
>(+-0 -1)
minus
>(+-0 0)
0
```

ごらんのように、xの値によって3種類の動作をします。昔のLispは、condの方をメインにして、ifは脇役のような扱いが多かったようですが、最近の書籍等ではifの方を先に紹介する方がメジャーなようなので、このテキストもそれにならいます。

条件分岐の注意点

ifでもcondでも条件式の順番を間違えると正しく実行されないことがあります。次のプログラムはテストの点数xが50点未満、80点未満、80点以上について、それぞれの課題を表示するプログラムです。

```
(define (test-task x)
  (cond
    ((< x 80) 'print)
    ((< x 50) 're-test)
    (else  'pass)))
```

ところが、このプログラムは正常に動作しないのです。xにいくつか値をいれてみましょう。

```
>(test-task 80)
'pass
>(test-task 60)
'print
>(test-task 30)
'print
```

xに30をいれたとき、本当なら(< x 50)が評価されて're-testが返されてほしいところですが、先にある(< x 80)も正しいため'printが返されてしまいました。 このような間違いは条件式のならびを変えることで訂正できます。以下が正しいプログラムです。

```
(define (test-task x)
  (cond
    ((< x 50) 're-test)
    ((< x 80) 'print)
    (else 'pass)))
```

問題

- 割り算の余りを求めるmodという関数があります。(mod m n) -> m/nの余り この関数を使って、mがnの倍数であるときは、「mはnの〜倍です」と出力し、そうでないときは、「m/nの余りは〜です」という関数を作って下さい。

まとめ

- 条件分岐には、ifやcondを使う。
- ifは2方向、condは3方向以上の分岐に使う。
- 条件は上の物から評価されるので、順番を間違えると評価されない事がある。

解答

Racket基礎その7 繰り返し

さて、condとifを覚えたので、条件分岐するプログラムを書くことができるようになりました。このあとは、繰り返しのプログラムを覚えていきましょう。 繰り返しのプログラムを書くには、再帰というテクニックを使うか、forやdoという関数を使います。多くのLisp、Schemeの本やサイトでは再帰を先に解説することが多いのですが、理論としてわかりやすいのはforやdoなので、ここではforを先に解説することにします。forの基本形は次のようになります。forは沢山の使い方がありますが、今は1つの使い方を取りあげます(注:forはRacket独自の関数で、Schemeにはありません。Schemeではdoや再帰を使ってください) forの考え方は、次のようになります。

- 繰り返し用の変数を定義し、その変数を1つずつ変化させながら、関数を実行する。

普段私達も似たような考えの動きを行っています。名札を作る作業を考えてみましょう。

この時私達のやる仕事は * 名前と(場合によっては番号を)書く * 名前を変えて(番号を変えて)名札を書く

1つめと2つめで名札を書く動作(関数)は変わっていませんが、書く名前や番号(引数)が変わっているため、動きが変わります。これは繰り返しの原始的な実装(プログラム)であるといえるでしょう。

プログラムに戻ります。forの書き方は次のようになります。

```
(for ((変数1)(変数2) ...)
   (動作))
```

簡単な例を満てみましょう。次のプログラムは、引数xの数だけ"Hello"を出力します。

```
(define (hellos x)
    (for ((i x))
        (display "hello")))
```

```
>(hellos 4)
hellohellohellohello
```

このプログラムでは、いくつかの新しい書きかたが導入されます、1つずつ見ていきましょう。

```
(for ((i x))
```

ここでは、forの中でだけ使われる変数を設定します。(この例では、変数iの値をxにする)このように、特定の関数の中でだけ使われる変数を、局所変数とよびます。condと同じように、(())が2重になっているのは、複数の局所変数を定義することで、複雑な繰り返しを作りだせるようにするためです。 forの中では、ここで定義された値の数だけくりかえしが行なわれることになります。

```
(display "hello")))
```

ここが、繰り返される部分です。今回の例では、(i x)と、二度手間のようにして局所変数を定義しているのが不便に思われるかもしれません。(for ((i)) ...)でi回繰り返しができた方が簡単に見えます。 しかし、こうするこ

とによって、forは内部の繰り返しに毎回変わる値を入れることができるようになるのです。
以下の関数は、hellosの関数に、何回めのhelloかの表示を加えたものです。

```
(define (hellos-2 x)
  (for ((i x))
    (display "hello")
    (display i))
```

これを実行すると、

```
>(hellos-2 4)
hello1hello2hello3hello4
```

iはforの繰り返しの中で、変化していくのです。

寄り道 特殊文字

先程for文の例で作ったhellos関数は、「hello」が1行に出力されてしまいました。普段みなさんが見る文章は、改行やスペースによって見やすく整形されています。ここで、displayで改行させる方法を学びましょう。

```
>(display "hello\nhello")
hello
hello
```

文字列が改行されて出力されました。改行するには、文章中に、\nという記号を入れます。\という記号はエスケープ文字と呼ばれ、後に続く文字によって、プログラムを出力するときに工夫を加えることができます。
(Windowsを利用している方は、\は￥と表示されます)

問題

-

Racket基礎その8 再帰定義

お次は、繰り返しのもう一つの書きかたである、再帰定義を見ていきましょう。再帰定義を簡単にまとめると、以下のようになります。

- 条件分岐を使い、一つに自分自身(定義している関数自身)を、少し引数を変えて呼びだすように定義する。
- 自分自身の呼びだしが終了する条件を分岐の他の部分に書いておく。

正直説明が難しいので、実際のプログラムを見ながら理解していきましょう。 先程の、helloを沢山打つだけのプログラムを、再帰定義で書きなおしてみましょう。

```
(define (hellos2 x)
  (if (1 < x)
    (display "hello"))
    (begin
      (display "hello")
      (hellos2 (- x 1))))
```

この式の動きを展開していくと、次のように実行されています。

```
(hellos2 4)
->(begin (display "hello")
         (hellos2 (hellos2 3)))

->(begin (display "hello")
         (begin (display "hello")
                (hellos2 2)))

->(begin (display "hello")
         (begin (display "hello")
                (begin (display "hello")
                       (hellos2 1))))
```

hellosに渡される引数が1つづつ減っていくのがわかります。これによってやがてifの条件が満たされ、hellosが終了することができます。自分自身の中で自分自身を呼びだすというのは、マトリョーシカにも似ていますね。

このプログラムでは、hellos2に渡される引数が1以下になった所でif文の1つめの分岐である(display "hello")が評価されます。それ以降は展開された式を1つずつ戻りながら、(display "hello")が実行されていきます。

それでは、リストの一番最後の要素をとりだす関数を考えてみましょう。リストに含まれるアトムの数を、リストの長さとよぶことにします。 初めに、手作業で要素を求めるということをやってみましょう。 このばあい、最初はcdrを打ちつづけていけばいいでしょう。ただ、それをすると最終的に空リストがのこってしまいますので、最後だけcarを使いましょう。 これは以下のようになります。

リストの長さが1のとき

- ```
  >(car '(1))
  1
  ```

- リストの長さが2のとき

```
>(car (cdr '(1 2)))
2
```

- リストの長さが3のとき

```
>(car (cdr (cdr '(1 2 3))))
3
```

法則が見えてきました。これを一般化すると下のようになります。

- リストの長さがnのとき

```
>(car (cdr (cdr ... (cdr '(1 2 3 ... n)))))
n
```

このプログラムを文章でまとめてみると、

- リストの長さが1の時はcarを実行
- リストの長さが2以上の時はcdrを実行

これを再帰定義というテクニックで実装してみましょう。プログラムは次のようになります。

```
(define (last ls)
  (if (null? (cdr ls))
        (car ls)
        (last (cdr ls))))
```

続いて、再帰定義を習うときによく使われる、階乗の関数を作ってみましょう。 階乗は $n!=n*(n-1)*(n-2)...*2*1$であらわされます。1つづつ数が小さくなっていくところなどは再帰にピッタリでしょう。

では、プログラムの構造を考えてみましょう。このプログラムでは、再帰の度に数を1つずつ変化させ、それを掛けあわせて階乗を求めていけばよさそうです。以下が階乗の関数になります。

```
(define (fact x)
  (if (=> x 0)
    1
     (* x (fact (- x 1)))))
;factとは、factrial(階乗)の省略形
```
```

いかがでしょうか。last関数とは最後の処理が少し異なりますが、一つずつリストや数を動かしていくところは同じです。これを実行すると以下のようになります。

```
>(fact 5)
120
>(fact 10)
3628800
```

再帰定義はLisp(Racket)と親和性が高いため、慣れてくると繰り返しよりも書きやすくなります。しかし、再帰定義は繰り返しに比べると関数を1つずつ展開するという点で公立が悪いため、速度が重要な場面では繰り返しや、後に出てくる再帰定義の高速化版である末尾再帰を使った方がいいでしょう。(プログラムとしての見易さは再帰定義が1版Racktにあっていると思います。試作を普通の再帰で作り、後で末尾再帰やforに書きかえるといいでしょう)

問題

- 階乗のプログラムを改造して、累乗の関数を作ってください。関数名はriseとします。実行結果は以下のようになります。

```
>(rise 2 3) ; 2の3乗
8
```

- 割り算の余りを求める関数を作りましょう。方法としては、割られる数から割る数を引きつづけていくといいと思います。関数名はmod2としましょう。(modはracketで定義されているので、重複しないようにします)実行結果は以下のようになります。

```
>(mod2 21 4)
1
```

解答

Racket基礎その9 ラムダ式と関数引数

ここまで色々な関数を定義してきました。今回はラムダ式と呼ばれる「名前のない関数」を勉強します。

第二部 プログラミングRacket ソフトウェア開発編

資料

エラー対照表

DrRacketのエラーは「[エラーを起こした関数]:[エラー内容]」という書式で書かれています。ここでは、関数名順に載せています。関数名がプログラム毎に変わる物は、関数名1などと一般化し、エラー内容のアルファベット順に並べています。

エラー文	**翻訳(意訳)**
module: duplicate definition for identifier in: 関数1	同じ名前の関数を複数回定義しています。(関数名が衝突しています)
read: expected a closing '"'	ダブルクオーテーション(")が足りません
read: expected a ')' to close '('	閉じ括弧が足りません
関数1: arity mismatch; the expected number of arguments does not match the given number expected: at least 数1 given: 数2 arguments...:	引数の数があっていません。この関数は最低[数1]個の引数を必要としますが、[数2]個の引数しかありません。
関数1: unbound identifier in module in: 関数2	関数2の中で、定義されていない関数1の関数が呼ばれています。