RacketTextbook-JP

Kudzuyu

2016 年 5 月 7 日

目次

0 序文と下準備

0.1 この文書について

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。

0.2 はじめに-Racket と Lisp

1950 年代に、ジョン・マッカーシーという人により、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機能があったため、人工知能などの研究用として残りつづけました。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。

0.2.1 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になっていますが、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。

0.3 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。

0.3.1 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。リポジトリは ppa:plt/racket です (公式リポジトリで、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。)
- **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 というものが選択肢にあるとおもいます。これは、Racket としての必要最低限の機能だけを搭載し、あとからパッケージという形で Racket に機能を追加できるようにしたパッケージです。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。

0.3.2 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。

0.3.3　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。これは、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。(どうして Racket が LGPL で配布されているかというと、使っているライブラリの中に、LGPL で配布されているものがあるからです。) 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。)

0.3.4　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のがいいでしょう。

0.3.5　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。
- 商用利用はおすすめしません。

1　プログラミング Racket 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。

1.1 Racket 基礎その 1 Hello World!とリストに関する各種名称

1.1.1 まずはお決まり、Hello World!

DrRacket を起動させると、#lang racket と書かれたエディタ部分、その下の、シェルの部分が目につくと思います。これは、Lisp 系の言語の開発環境にはかならずといっていいほど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)

```
1 #lang racket
2
3 (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、カッコが半角であるかや、全角空白が入っていないかどうかを確認してください。

```
1 Hello World!
2 >
```

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。先頭の要素がプログラムの名前であり、残りの要素が「引数 (ひきすう)」と呼ばれる、そのプログラムの動作を決定するものです。このプログラムの場合、「display」というのが画面に文字を入力せよという命令 (関数といいます) であり、「“Hello World!”」が display に対する引数となっています。 “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

ため微妙に動作が違うのです。

1.1.2 Racket の構造

先ほど、関数とデータが、同じリストという構造でできているという話がありました。今度は、Racket のデータについても見ていきましょう。Racket では、リストを次のように書きます。次のリストは、a、b、c の 3 つの要素をもつリストです。

```
1 (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を表わしたリストです。

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

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

```
1 (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。

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

これを実行すると、エラーがでます。

```
1 . . application: not a procedure;
2  expected a procedure that can be applied to arguments
3   given: 1
4   arguments.:
5 >
```

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。

```
1 (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。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、リストに対してはこちらを使っていきましょう。

1.1.3 まとめ

- Racket はリストとよばれる () 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 car 部を関数とし cdr 部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。

1.2 Racket 基礎その 2 リストの操作とアトムについて

1.2.1 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。

```
1 >(display '(one two three))
2 '(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。

1.2.2　リストの各要素は、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。

```
1 ((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。

1.2.3　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。

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

```
1 (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いにしえの関数です。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名前が多いです。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。

```
1 (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それでは練習問題です。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。**練習 1**

```
(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、気にとめておい

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。

1.2.4 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。

```
1 >(cdr '(1 2 3))
2 (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。

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

```
1 >(car (cdr '(1 2 3)))
2 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 のプログラムなので、中の cdr を先に評価して評価した値を car に渡すのです。このようにして、リストの 2 番目や 3 番目をとりだすことができました。ところで、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。

```
1 >(cadr '(chicken pork beef))
2 'pork
3 >(caddr '(tuna salmon swordfish))
4 '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。

```
1 (car (cdr '(1 2 3)))
2 (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とおもいます。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も、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。

1.2.5 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。

1.2.6 練習問題の解答

```
1 >(car '(1 2 3 4 5))
2 1
3 >(cdr '(1 2 3 4 5))
4 (2 3 4 5)
5 >(car '((1 2)(3 4)))
6 ((1 2))
7 >(cdr '((1 2)(3 4)))
8 ((3 4))
```

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

1.3 Racket 基礎その 3 リストを作る

1.3.1 car と cdr の逆関数、cons

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。

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

1.4 Racket 基礎その 4 算術計算と条件演算子

1.4.1 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。今回は少し趣向がかわりまして、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。これに関しては覚えてください。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。なお、足し算とかけ算に関しては、引数をとらない関数も実行することができます。

```
>(+)
0
```

```
>(*)
1
```

これは数学的な考えにもとづいていて、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
```

これは計算が楽になりそうですね。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ます。一方、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。

1.4.2 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
>(atom? '(a b))
#f
>(atom? 'a)
#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、atom?は引数がアトム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。勿論、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ます。「<」と「>」に関しては、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
```

1.5 関数定義

1.5.1 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。

```
1 (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。

```
1 (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。

```
1 >(square 15)
2 225
```

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、基本関数だけで作ることもできます。

```
1 >(square (car '(11 12 13)))
2 121
```

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。では define はどのような動きをするのでしょうか。S 式同士を接続するというのが、define の動き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。ところで、シェルで define したときに、これまでと違うことがあることに気付かれたでしょうか。これまでの関数は、即時結果を返すものでしたが、define はなにもかえってきていません。実は define は実行されても返り値を返さないのです。### 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。

```
1 >(define x 20)
2
3 >(+ x 10)
4 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を、簡略化して使いまわすことができます。

```
1 (define pi 3.14)
```

```
2 (define (circle-area r)
3   (* 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、同じ定数を使う別のプログラム (上のプログラムでいえば、pi を定義したことによって、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になることが多いようです。

1.5.2 define の注意点

今後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。

```
1 >(* 2 3)
2 6
3 >(define * +)
4
5 >(* 2 3)
6 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。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を確認するには、Racket のドキュメントで検索してください。## 条件分岐は多種多様この回は、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プログラムを作っていきましょう。

1.5.3 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。もしも複数の動作をしたい時は、次のようにします。

(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。”))))

1.5.4 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)
    ...
)
```

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")
```

```
5         ((< 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。

```
1 >(+-0 4)
2 plus
3 >(+-0 -1)
4 minus
5 >(+-0 0)
6 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。さて、if でも cond でも気をつけなければいけ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。

```
1 (define (test-task x)
2   (cond
3     ((< x 80) 'print)
4     ((< x 50) 're-test)
5     (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。

```
1 >(test-task 80)
2 'pass
3 >(test-task 60)
4 'print
5 >(test-task 30)
6 '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) が評価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。

```
1 (define (test-task x)
2   (cond
3     ((< x 50) 're-test)
4     ((< x 80) 'print)
5     (else 'pass)))
```

1.6 繰り返しと再起定義

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、「for」という関数を使います。for の形は次のようになります。

```
```

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

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

- リストの長さが 2 のとき

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

- リストの長さが 3 のとき

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

法則が見えてきました。これを進めていくと下のようになります。

- リストの長さが n のとき

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

```
1 (define (last ls)
2   (if (null? (cdr ls))
3     (car ls)
```

```
4     (last (cdr ls))))
```

再帰定義は、多くの事をやっているわけではありませんが、初心者には理解が難しいかもしれません。わからなければここは読みとばしてもけっこうです。

2　プログラミング Racket ソフトウェア開発編