プログラミング入門
第１回「導入、プログラムの基礎」

二宮 崇（ ninomiya@cs.ehime-u.ac.jp ）

教科書

Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996

はじめに

この講義は関数型プログラミング言語について講義します。一般にはプログラミング言語は、手続き型、関数型、論理型、オブジェクト指向型に分類されますが、その中でも関数型プログラミング言語について勉強します。

ここでは関数型プログラミング言語について講義をしますが、関数型プログラミング言語の使い方を習得することを目標としているのではなく、関数型プログラミングの考え方や作法を習得することを目標としています。例えば、C であったり、Java であったり、Python であったり、PHP であったり、手続き型といわれるプログラミング言語はたくさんありますが、同じような言語を同じように勉強することにはそれほど意味はありません。プログラミングとは、コンピュータに計算させる手続きを記述することですが、その方法や作法は一つではなく、関数型、論理型、オブジェクト指向型と呼ばれるプログラミングパラダイムがいくつか存在し、同じ計算結果や計算手続きであっても、まったく異なる考え方でプログラムを記述することができます。例えば、手続き型言語だと 1 から 100 までの和は次のように計算できます。

```
sum ← 0; i ← 1
while i ≤ 100 do
    sum ← sum + i
    i ← i + 1
```

矢印は代入操作を意味していて、最初 sum という変数に 0 を代入し、続いて i という変数に 1 を代入することを行います。次に、i が 100 以下である間は while 以下の処理を繰り返します。while 以下ではまず、sum に sum の値と i を足した値を代入することを行い、続いて、i に i+1 を代入します(i を一つ足す)。こうすると、i は 1 から始まって、100 まで増えていき、sum にそれぞれを足していくことになるため、$1+2+\cdots+100$ を計算することになります。

次に、関数型言語では次のように計算できます。まず、次のようにsumという関数の定義を与えます。

$$sum(i) = \begin{cases} 0 & if\ i = 0 \\ i + sum(i-1) & otherwise \end{cases}$$

ここで、$sum(100)$を計算すればよいことになります。つまり、$sum(100) = 100 + sum(99)$で計算でき、$sum(99) = 99 + sum(98)$と計算できるので、これを繰り返すと、$sum(100) = 100 + 99 + 98 + \cdots + 1 + sum(0)$となりますが、最後の$sum(0)$は0となるため、$sum(100) = 100 + 99 + 98 + \cdots + 1 + 0$となるわけです。

この簡単な例をみただけでもだいぶ計算の仕方が違うということがわかるのではないでしょうか。手続き型においては、変数は書き換え可能な箱の役割を果たしていて、この箱の値を書き換えることによって計算が進んでいくことがわかります。一方、関数型では関数の定義として計算方法が記述されているため、変数の代入操作を必要としていません。

代入操作が許されない関数型プログラミング言語は、純粋関数型プログラミング言語と呼ばれ、その計算モデルは「計算は関数の適用である」としています。つまり、手続き型と比べより数学的な計算モデルとなっていて、状態という概念がなく、「変数の宣言はできるものの、変数の書き換えは許されない」という制限や、「関数は、同じ引数に対しては、同じ結果を必ず返す」といった性質があります。これは、実装しようとしているプログラムを数学的にとらえ、合理的にプログラムを実装できるようになる、ということでもあります。結果として、バグ(プログラムの間違い)が生まれにくいプログラムを効率的に実装できるようになります。また、手続き型のように代入のタイミングによって計算結果が変わるということがおきないため、並列プログラミングとも相性が良いと考えられています。プログラミングパラダイムは関数型以外にもいろいろとありますが、それぞれの異なる考え方、モデル、作法、長所、短所があり、状況や自分の好

みにあわせて最も良いスタイルを選択できるようになると一段上の技術者になったといえるでしょう。

実際に用いられているプログラミング言語の多くは、手続き型をベースにしていて、拡張されることによって、上記の手続き型、関数型、オブジェクト指向型の混合となっています。例えば、C 言語は手続き型をベースとしていますが、関数型として用いることができますし、C++はオブジェクト指向型ともなっています。これから学ぶ Scheme という言語は Lisp という関数型プログラミング言語の一種（方言）ですが、Lisp はオブジェクト指向型にも拡張されていて、CLOS (Common Lisp Object System)と呼ばれる言語仕様になっています。様々な拡張が言語仕様として実現されていますが、それらの仕様はベースとなる言語に引きずられ、例えば、C や Java などの手続き型プログラミング言語では関数型プログラミング言語の本質を理解することが難しくなっています。関数型プログラミング言語で記述されるプログラムの多くは、例えば、手続き型のプログラミング言語で記述することができますが、関数型の長所を生かした記述をするには、一度関数型プログラミング言語の作法を理解しておく必要があります。これはオブジェクト指向型についてもいえることで、多くのオブジェクト指向ではないプログラミング言語でも、手続き型プログラミング言語を用いて、オブジェクト指向の作法にそったプログラムを記述することができます。重要なのは、実際に用いるプログラミング言語(C や Java や Python や Lisp など)ではなく、プログラミング作法を学ぶということです。多くの学生は手続き型プログラミング言語を最初に学びます。手続き型言語はコンピュータの動作に基づいているため、コンピュータの仕組みを理解すると同時に学ぶことができ、計算効率の点からも非常に有用な言語であると言えます。しかし、この手続き型には様々な長所があると同時に様々な短所があります。関数型にも長所と短所がありますが、関数型の長所を知った上で、手続き型プログラミング言語を使うようになれば、「より良い」プログラムを書くことができるようになります。「より良い」ということはどういうことかは、この講義で関数型プログラミング言語を学ぶにつれてわかってくるようになっていますが、より直接的なところでは、副作用のない計算の仕方、再帰呼び出し、高階関数、ラムダ式の用い方、ガーベージコレクションがついたデータ構造の実データ、ポインタの扱い方などがありますが、ここでは講義をとおして、関数型プログラミング言語をベースにしたデータの抽象化、プロセスの抽象化、関数型プログラミングの作法を理解することを目標とします。この講義では Scheme という関数型プログラミング言語で、関数型の作法を強制的に学ぶことになりますが、一度この作法を身につければ、他の手続き型プログラミング言語、例えば、C++や python でも同様の関数型

プログラムを記述することができるので、状況に応じて、より良いプログラミングスタイルを選択することができるようになります。

関数型プログラミング言語というとLispが有名で、教育用にはSchemeという言語がよく用いられています。SchemeはLispの一種ですが、Lispよりも高階関数の記述がエレガントで、計算の本質を理解するためにはより良い言語です。Scheme自体を使って今後プログラムを書いていくことは少ないかもしれませんが、関数型の本質を理解するためにはSchemeを使って勉強するのが最も良いと思います。一度関数型のスタイルを身につければ、他のC++やPythonでも同じスタイルのプログラムを記述できるようになります。

この講義は、次の本をつかって行います。

計算機プログラムの構造と解釈 第二版, ジェラルド・ジェイ サスマン, ジュリーサスマン, ハロルド エイブルソン (著), 和田英一 (訳), ピアソン・エデュケーション, 2000年

この本はMITで計算機科学を教えるために作成された教科書の邦訳です。非常に優れた教科書として有名で、実際に東京大学、京都大学、東京工業大学など日本でも著名な大学の講義や演習で用いられています。できれば原著で読むことをお薦めします(そのほうが内容がよくわかると思います)。原著は次の本で、こちらを教科書として用いてもかまいません。

Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996

原著の教科書は次のサイトからHTML形式で読むことができます。

http://mitpress.mit.edu/sicp/

また、本講義の資料、レポート問題等は次のページおよびmoodleのサイトからダウンロードできます。

http://aiweb.cs.ehime-u.ac.jp/~ninomiya/itp/

SCHEME の設定

この講義では Racket と呼ばれる Scheme の実装を用います。Racket はかつて PLT Scheme と呼ばれていたプログラミング言語で、Scheme の勉強用によく用いられています。また、様々な Scheme の変形(Variant)もサポートしています。

大学でのインストール

windows および linux 上にインストールされています。windows ではスタート画面の一覧から DrRacket を選択して実行すれば ok です。linux では、コマンドラインから

drracket

と実行すれば ok です。

自宅のマシンでのインストール

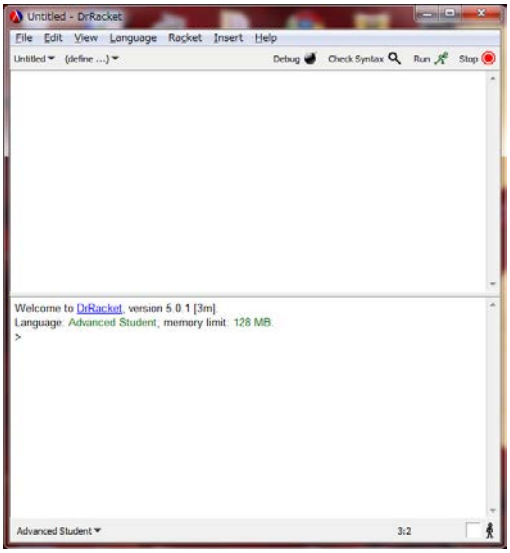
まず、次のホームページをみてみましょう。ここが Racket のホームページです。

http://racket-lang.org/

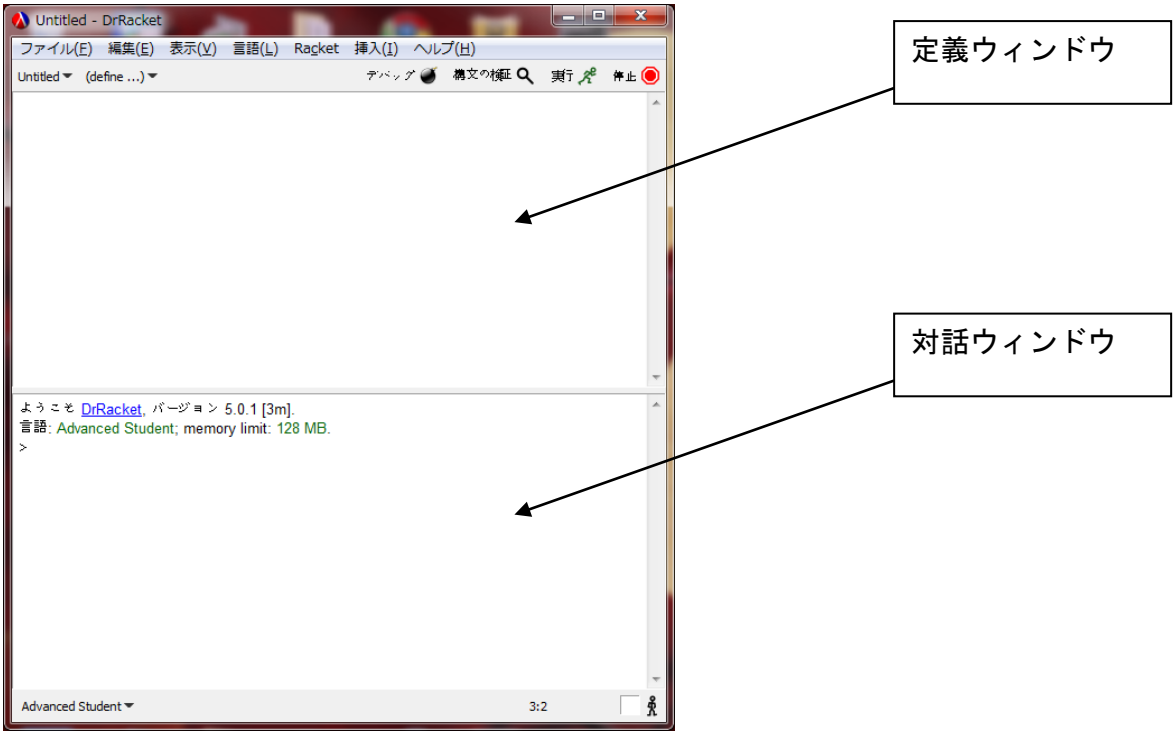
この中から、Download を選択します。次に Windows x86 を選択して(もしくは今インストールしようとしているマシンの OS を選択)、Download ボタンを押します。次に Main download (USA, Massachusetts, Northeastern University) を選択します(他のところを選んでもかまいません)。このダウンロードした exe ファイルを実行して、インストールします。

共通設定

linux 上で drracket を実行、もしくはウィンドウズ上で DrRacket というプログラムを左下のスタート（やウィンドウのアイコン）から選んで実行します。次のような画面がでているでしょうか。



Help から「DrRacket を日本語で使う」を選択すると日本語の GUI になります。その場合はもう一度 DrRacket を起動し直すようにメッセージがでるのでそのようにします。次のように日本語の GUI になります。



次に、「言語」→「言語の選択」を選択します。Choose a language をクリックして、「R5RS」を選択します。

ウィンドウの上側（定義ウィンドウ）にプログラムを書いて、実行ボタンを押すとプログラムが実行されるようになっています。書いたプログラムは「ファイル」→「定義に名前をつけて保存」を選択すると、保存できるようになります。下のウィンドウ(対話ウィンドウ)で、プログラムの実行後、プログラムで定義された関数や変数を呼び出したり、簡単なプログラムを書いたり実行することができます。

最初は下のウィンドウで簡単なプログラムを書いていくことになります。

計算機プログラムの構造と解釈(第 1 章)

1.1 プログラムの要素

・基本式: 数字や文字列などの最も基本となる式

・組合せ: 基本式を組み合わせる

・抽象化: 式に名前をつける

1.1.1 式

式は基本式とその組合せから成ります。基本式は、変数、整数、実数、ブーリアン、シンボル、文字、文字列、空リストなどがあります。Scheme の式は、半角の括弧()とスペース区切りで表現され、一番左に演算子(operator)を書き、続いて被演算子(operands)を書きます。また、Scheme は演算子を被演算子の左側におく前置記法(prefix notation)を採用しています。例えば、137 + 349 は次のように記述します。

```
(+ 137 349)
```

この式の計算結果は 486 となります。他にも、次のような式を記述することができます。

```
(- 1000 334)
```

```
(* 5 99)
```

```
(/ 10 5)
```

```
(+ 2.7 10)
```

(+ 21 35 12 7)

(* 25 4 12)

*は掛け算の演算子で、/は割り算の演算子です。コンピュータのプログラムでは×の代わりに*を使い、÷の代わりに/を使います。

次にこれらの式は入れ子にする(nested)ことができます。つまり、組合せの要素を組合せとすることができます。

(+ (* 3 5) (- 10 6))

(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

普通の数学やプログラミング言語では括弧は曖昧性がなければ省略しても良いし、余分に括弧があってもかまわない、ということになっていると思います。しかし、Schemeにおいては、括弧は「その演算子と被演算子に対し計算を行います」という意味を持ち、余分な括弧があったり、逆に括弧が足りない場合にはきちんと動作しないので、注意してください。例えば、次の式はエラーになります。

(137)

((+ 137 349))

(137)というのは、137 が演算子で被演算子が与えられていない式、というふうに解釈されます。137 というのは整数値であって、演算子ではないので、エラーが返ってきます。

1.1.2 名前と環境

define をつかって変数を定義することができます。

(define size 2)

これで size という変数に 2 という値が対応づけられます。一般的には、

(define ⟨変数の名前⟩ ⟨式⟩)

と変数を定義することになります。ただし、一度変数に値を対応づけたら、(純粋)関数型言語においては、再定義(代入)することはできません。例えば、上の例では size に 2 を対応づけたら、それを別の値として再定義することはできません。

一度定義した変数をつかって他の変数を定義することもできます。

```
(define radius (* size 5))

(define pi 3.14159)

(* pi (* radius radius))
```

値と変数名の対応付けをコンピュータは覚えていないといけません。この対応付けの記憶を環境 (environment)と呼んだり、大域環境 (global environment)と呼びます。

1.1.3 式の評価

計算の手順は Scheme ではどうなっているのでしょうか？わかりやすい、ある決まった順に評価(=式の計算)が行われています。

式は次のように評価されます。

(ア)　式が基本式の場合

1. その基本式の値を返します(変数の場合は変数に対応づけされた値を返します)。

(イ)　式が組合せの場合

1. 組合せの部分式をすべて評価します。
2. 被演算子を演算子に適用して計算を行います。

これは入れ子になった式を再帰的に評価していることになります。例えば、次の式を評価するとしましょう。

```
(* (+ 2 (* 4 6)) (+ 3 5 7))
```

式全体をみると、演算子*, 被演算子(+ 2 (* 4 6)), (+ 3 5 7)の部分式からできている組合せとなっています。従って、部分式(+ 2 (* 4 6))と(+ 3 5 7)を評価することになります。(+ 2 (* 4 6))は演算子+, 被演算子 2, (* 4 6)の部分式からできていますので、同様に(* 4 6)を評価します。次に計算を行います。(* 4 6)を計算して 24、(+ 3

5 7)を計算して 15、(+ 2 24)を計算して 26、最後に(* 26 15)を計算して 390 を返します。

つまり、上記の式が与えられたとき一番内側の(* 4 6)や(+ 3 5 7)から計算し、だんだん外側を計算することになります。一番内側では、演算子、数字、文字列などの基本式だけが書かれてあって、それぞれその数、文字列が返されます。変数であった場合には環境において対応づけられた値が返されます。

RACKET での式の書き方

式を並べたものがプログラムと呼ばれることになります。プログラムを書く上で気をつけることがいくつかあります。

注意 1: 括弧の対応があっていないとエラーになります。例えば、3 ×(2 + 5)を計算したとき、

```
(* 3 (+ 2 5)
```

で終わっているとエラーとなります。正しく(* 3 (+ 2 5))と書かないといけません。括弧の対応付けを間違っていると、変数の定義が正しくされなかったり、関数の定義の範囲を間違えたりして(関数については次回習います)、プログラムがまったく正しく動かない、もしくは、読み込むことすらできないということになります。また、括弧は 1.1.1 にも書いたように、括弧の中にある演算子に被演算子を適用して計算する、という意味を持ちます。従って括弧が余分にあったり足りなかったりするときちんと計算がされません。

注意 2: スペース(空白)と括弧は特殊な記号です。例えば、変数名の途中にスペースがあると、そこで変数名がおわったと解釈されます。

```
誤)  (define max value 10)
```

そういう場合にはスペースを入れず何か別の記号でつなぐようにしましょう。-(ハイフン)でつなぐことが多いです。

```
正)  (define max-value 10)
```

また、逆に変数名や演算子、被演算子の間にはスペースが必要です。

誤） (define max-value10)

正） (define max-value 10)

誤） (+3 5)　← +3 という名前の演算子と解釈される

正） (+ 3 5)

変数名に括弧があってもだめです。

(define max(value) 10)

注意 3: 次の記号は変数名に使えません。

" # ' ( ) [ ] { } | \ , ;

しかし、逆に言えばそれ以外の記号は自由に変数名として使えます。

注意 4: 複数のスペースや改行は一つのスペースとみなされます。スペースはいくつあっても一つとみなされるので、例えば、演算子と被演算子の間はいくらあいていてもかまいませんし、改行をいれてもかまいません。

例） (+　3　5)は(+ 3 5)と同じ

例)

(+　(* 3 5)

　　(/ 3 2))

は(+ (* 3 5) (/ 3 2))と同じ。この性質を利用して見やすいプログラムを書くようにしましょう。また、DrRacket 上ではある行でタブ(Tab)を押すと括弧にあわせてその行のインデント(その行の先頭のスペース)を自動調整してくれます。

注意 5: ;を使うと;より後ろの一行はコメント文として処理されます。

; これは変数の最大値

(define max-value 10000) ; 10000 をもっと増やしてもよい

オレンジの部分がコメントとして扱われます。