プログラミング言語をつくろう！

序章

　プログラマーは魔法使いである。

　０と１という糸を紡ぎ、どんな布をも織り成すことができる。鉛を金に変え、機械に命を吹き込むことができる。あまた多く存在するプログラマーの中の頂点、いわゆる”ハッカー”と呼ばれる人間達にとってみれば、石をパンに、水を葡萄酒に変えることだって造作の無いことだろう。そしてそのプログラマーにとって魔法の杖と等しい物がプログラミング言語だ。世の中にはありとあらゆる魔法の杖が存在し、そのどれもが独自の性質を持っている。多くのプログラマーに支持されているものもあれば、少数の熱狂的な”信者”によって支持されているものもある。高度で力強い魔法を生み出せるものもあれば、優雅でエレガントな魔法を生み出せるものもある。しかし、その両方を満たすプログラミング言語は、残念なことに、この世には存在しない。なぜならそれぞれが他との優位性をもつのと同様、他との劣位性をもつからだ。では我々プログラマーはそれら欠点を甘んじて受け入れなければならないのか？　否、方法はある。

自分で欠点のない言語を作ればいいのだ。

第1章 Lispを学ぶ

　「言語を作る」

　とても興味をそそられる文面だ。実際、それは楽しい。しかしゼロから作るには大きな壁が立ちはだかる。

　「言語を作る」という行為に含まれる意味は二つあり、ひとつは自分で文法を定義すること、もう一つはその処理系を作成することだ。そして、前者をゼロからこなすことは初心者にはひどく難易度が高い。文法やオブジェクトシステムの設計、言語の全ての箇所においてわずかな不備も許されないからだ。よって文法や言語設計は、少なくとも基本部分に関しては、他の言語から借りる必要がある。しかしそうなると言語作成の醍醐味とも言える「言語の設計」を行う楽しみが無くなってしまうことも事実。理想を言えば、文法は他から拝借しつつ、自分の手でそれらの文法に改造を加えたい。できれば処理系の作成

も楽な文法がいい。このようなわがままな願いを叶えてくれる言語などそう簡単には見つかりそうもない。だが実は、この世には存在する。これらの拡張性と機動性を備えた言語が。それがLispだ。

Lispは1956年に発明されたとても古い言語だ。その文法は類を見ないほど奇怪で、見た目にはカッコを多用する文法構造になっている。しかしそれは見た目の奇怪さとは裏腹に非常に強力な言語でもある。Lispのもつ「強さ」は全てその文法の単純さから生み出される。それは「Lispは文法がない」とまで言われるほどのシンプルさなのだ。

文法が簡単であることは、すなわちその処理系の作成が簡単であることと直結する。そして、Lispの文法はただ単純なだけではなく、拡張性に富んでいる。ユーザーはLispという言語の中で自己流の文法を定義し、その文法でプログラムを記述することができる。いわゆる「言語内言語」の作成が非常に簡単なのだ。このメタな性質は、まさに「自分で文法を作る」という目的に合致してはいないだろうか？

これらシンプルさと拡張性の高さが、Lispを採用する明確な理由である。ではLispの具体的な解説に移ろう。実はLispとは特定の言語を指すわけではない。Lispという単語が指し示すのは「語族」だ。今回はLisp言語族のうちの「Scheme」という言語を文法として採用する。SchemeはLisp語族の中でも特に文法が簡潔で、それはすなわち強力であることを意味する。もし興味があればCommon Lispなども自学していただけると嬉しい。

では、Schemeの文法についての基本的な解説を行おう。

Lispとは全てがS式と呼ばれるモノで表現される言語だ。たとえば1+2という式を計算するには

```
(+ 1 2)
```

と表現する。これは逆ポーランド記法と呼ばれる数式表現と等しい。そしてLispでの式表現はすべてこの逆ポーランドで行われる。つまり、1+3×(10+13)を計算するには

```
(+ 1 (* 3 (+ 10 13)))
```

とする。また同じ方法で m = 1 のような数学的表現も可能だ。

```
(define m 1)
```

Lispはラムダ計算という関数を抽象化した計算体系をモデルにしており、関数の定義と実行は数値の定義や実行と同じように行われる。ラムダ計算においてパラメータとしてxをとり、x*xを返す関数はλx. x*xと表される。そしてLispではこうだ。

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

また、f(x) = x*xはf = λx. x*xとして扱うことで

```
(define f (lambda (x) (* x x)))
```

と書ける。このようにLispは非常に数学的だ。そして計算機科学的だ。Lispの構造は非常に単純であり、処理系の実装の手始めには最適なのだ。

以上が簡単なLispの文法解説であるが、詳しい文法はこのレポート内で必要に応じて説明してゆく。まずはカッコを多用する特殊な文法であることを理解してほしい。

第2章 構文解析について学ぶ

プログラムとは、プログラマーが書くソースコードである。処理系はそのソースコードを実行する。ソースコードとは、テキストにすぎない。つまり処理系は、テキストを読み込み、それをプログラムとして”解釈”できなければならない。

通常、ソースコードはプログラム内で扱いやすいよう一度抽象構文木(AST: Abstract Syntax Tree)というデータ構造に変換される。その生成されたデータ構造を処理することでプログラムは初めて動作するようになるのだ。

プログラムのソースコードのような、我々の話す自然言語に近い、いわゆる「文脈自由文法」のテキストから抽象構文木を作成するには概して二つのステージを踏まなければならない。一つ目が字句解析、もう一つが構文解析だ。

字句解析(Lexical Analysis)とはいわばソースを「単語」に分けていく処理のことで、スキャナとトークナイザという2つの部分で成り立っている。スキャナが単語の区切りを判別して、分けた単語をトークナイザに渡す。トークナイザはその単語の「品詞」を判別し、トークンと呼ばれるプログラム言語内での最小単位を返すという仕組みだ。

一方構文解析(Syntactic Analysis)は字句解析によって得られたトークンの羅列を、「意味付け」してゆく。自然言語で言うところの「とある形容詞がどの名詞に掛かっているか」ということを判別する作業だ。トークンとトークンの結びつきを解釈し、それらをつなぎ合わせる。その関係性の集合体が木であり、抽象構文木なのだ。

実は、よく知られている中でも構文解析を行う構文解析器には様々な種類があり、ボトムアップ解析かトップダウン解析か、左端導出か右端導出か、など多岐に渡る分類がなされている。そのなかでも今回は「予言的パーサー」という種類の構文解析器を使用することにする。これはトップダウン解析の一種、LL法の実装方法のひとつである再帰下降パーサーの、さらにバックトラックを行わないバージョンだ。この構文解析器を選んだのにはLisp自身の特異な性質が関係している。

そもそもLispは、それ自身が生の抽象構文木であると言われている。Lispのソースコードがデータ構造としての性質も持ち合わせているということだ。この「データとプログラムを分けない」という特徴は他の言語にはまずないもので、これがLispに奇妙かつ強力な力を与えている。

そして、Lisp自身のもつ構造は左端導出およびトップダウン解析器と相性がいい。 しかもバックトラックも不要だ。 それら文法の単純さこそが、実装が簡単だが肥大化しやすい再帰下降構文解析を使うことができる所以である。そして筆者は実際に、LL(1)（＝LL法でかつ先読みが１以下）で全ての実装を終えることができた。これほど簡単に構文解析が行えかつこれほど強力な言語はそうは存在しないだろう。

第3章 字句解析器を実装する　〜トークン編〜

では実際に構文解析器を実装していこう。今回、処理系は全てPythonを使って実装することにする。これは関数をファーストクラスとして扱えることが大きい。また、筆者の個人的な好みでもある。

第2章で解説したように構文解析は「字句解析」「構文解析」という二つのフェーズに分けることができる。これら二つの最も大きな違いは「意味を解釈するか否か」という点である。字句解析は与えられた「形式的な」文法に、データが準じているかを調べ、それを字句にわける。何はともあれ実際のソースコードを見てみよう。

```
class Token(object):
   DOT = 1
   QUOTE = 2
   QUASIQUOTE = 3
   UNQUOTE = 4
   UNQUOTE_SPLICING = 5
   LPAREN = 6
   RPAREN = 7
   EOF = 8
   STRING = 9
   INTEGER = 10
   RATIONAL = 11
   REAL = 12
   COMPLEX = 13
   SYMBOL = 14
   BOOLEAN = 15
   TERMINAL = whitespace+"(),'"

   def __init__(self, kind, value=None):
      self.kind  = kind
      self.value = value
```

簡単な解説（本文と並列して読み進めて欲しい）：
トークンは全てこのクラスのインスタンスとして表される。16個の定数はトークンの種類を現していて、例えばLPARENという種類のトークンは
Token(Token.LPAREN)
と表す。後述するToken.SYMBOLなどは値が不定で、インスタンス内にデータを保存しておく必要があるため以下のようにする。
Token(Token.SYMBOL, “a-symbol”)

Tokenクラスは Token一つ一つを表す。Tokenには一般的に品詞（型）が存在し、それを構文解析器がそれを見て、文に意味付けを行ってゆく。ここではself.kindが型、self.valueが実際のトークンのデータだ。LexerクラスはSchemeのソースコードを受け取り、それを前から順にscanしてゆく。

第1章でも少し触れたが、Lispは全てがS式である。全てのS式はリストとアトムのどちらかに分類される。例えばLisp内で

(3 “String” -3.14 a-symbol)

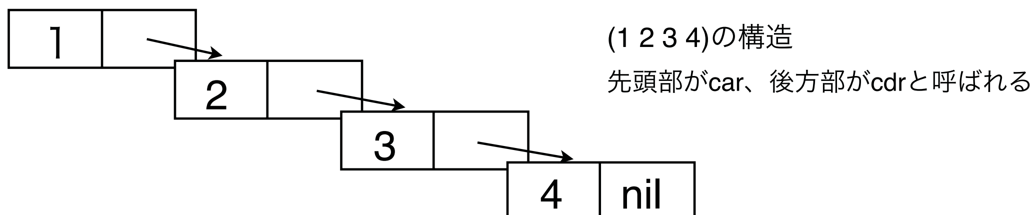
と記述すればそれは「3」と「”String”」と「-3.14」と「a-symbol」から構成される「リスト」として解釈される。リストは空白区切りであり、空白にはいわゆる「“ “(スペース)」だけでなく、改行を使うこともできる。つまり

```
(3       “String”
      -3.14
  a-symbol)
```

のように書いても宜しい。このリストは上述のリストと全く同じである。そして、「3」「”String”」「-3.14」「a-symbol」のそれぞれがアトムである。アトムはLispの中における「リストではないもの」の総称だ。アトムには整数、文字列、実数、シンボルなどが含まれる。整数に解釈できるものは整数に解釈するし、“(ダブルクオーテーション)でかこまれた文字は文字列として扱われる。実数も然り。そしてそれらに解釈できないもの、それがシンボルだ。 上でいえばa-symbolというアトムは数字にも解釈できないし、ダブルクオーテーションで囲まれていないから文字列でもない。だからa-symbolというアトムはシンボルだ。ここで読者の皆さんはシンボルが一体なんなのかという疑問にぶちあたるだろうが、ここでは敢えて説明を省く。字句解析において必要なのは「単語の意味」ではなく「単語がどのようなルールで文章を構成しているか」だからだ。

その「単語のルール」というのに関連して、もうひとつ、リストについて詳しく解説しておく必要がある。字句解析とは単語に分ける作業であるから、リストがどのような単語から構成される文章なのか、それを明確に知る必要がある。

実は、Scheme（と他のほとんどのLisp）ではリストはリストではない。実はリストはペアと呼ばれる最小単位の物体が連続しているのだ。



ペアは二つのアトムが繋がった、まさに「ペア」であり、上図のようにペアが数珠つなぎになって見かけ上リストになっているのだ。Lispではこのペアを明確に記述することができる。たとえばペアの最初のセルにa、後ろのセルにbが入っているペアは

(a . b)

と記述することで作成できる。そして、リストも同じようにこの”.”を使って明示的にペアの数珠つなぎとして記述できる。

(3 . (“String” . (-3.14 . (a-symbol . nil))))

これは前述の例と全くおなじリストを表している。むしろ.を使わない最初の例の方が実は省略記法だったのだ。ここで登場するnilもLispの根幹に関わる重要な概念であるが、ここでは一旦飛ばすことにする。字句解析においてはシンボルと同じように扱えばそれで良い。

ではこれらを踏まえて、字句解析器を完成させる。ソースコードは次章に詳細な解説とともにまとめて掲載している。

最終的に完成した字句解析器では、(+ 1 2)というコードを入れると

```
Token(Token.LPALEN)
Token(Token.SYMBOL, “+”)
Token(Token.INTEGER, 1)
Token(Token.INTEGER, 2)
Token(Token.RPAREN)
Token(Token.EOF)
```

と解釈されるようになる。同じく(3 . (“String” . (-3.14 . (a-symbol . nil))))は

```
Token(Token.LPALEN)
Token(Token.INTEGER, 3)
Token(Token.DOT)
Token(Token.LPAREN)
Token(Token.STRING, “String”)
Token(Token.DOT)
Token(Token.LPAREN)
Token(Token.REAL, -3.14)
Token(Token.DOT)
Token(Token.LPAREN)
Token(Token.SYMBOL, ‘a-symbol’)
Token(Token.DOT)
Token(Token.SYMBOL, ‘nil’)
Token(Token.RPAREN)
Token(Token.RPAREN)
Token(Token.RPAREN)
Token(Token.RPAREN)
Token(Token.EOF)
```

と解釈され、これら18個のトークンに分割される。

第4章 字句解析器を実装する　～実装編～

ここでは前章で解説しきれなかった字句解析器Lexerクラスの実際の実装について駆け足で説明していきたい。

実装自体は再帰下降パーサーだが、一部バックトラックを使っているため予言的パーサではない。しかし構文解析ならまだしも字句解析においてLL(2)で実装が完了していることはなかなか興味深い。まさにLispの文法の簡単さ、機動性の高さを思い知らされる一面であった。

```
class Lexer:
    '''Do lexical analyzation for scheme source code'''

    def __init__(self, exp):
        self.exp = list(exp)

    def scan(self):
        if not self.exp:
            return Token(Token.EOF)

        ch = self.exp.pop(0)

        if ch.isspace():
            return self.scan()
        elif ch == '(':
            return Token(Token.LPAREN)
        elif ch == ')':
            return Token(Token.RPAREN)
        elif ch == '\'':
            return Token(Token.QUOTE)
        elif ch == '`':
            return Token(Token.QUASIQUOTE)
        elif ch == '.':
            return self.tokenize_dot()
        elif ch == ',':
            return self.tokenize_comma()
        elif ch == '"':
            return self.tokenize_string()
        elif ch == '#':
            return self.tokenize_sharp()
        else:
            return self.tokenize_atom(ch)
```

簡単な解説：

lexer = Lexer(“Lispのソースコード”)

のようにLexerのインスタンスを作成し、

t = lexer.scan()

とすることでトークンを一つずつ取り出してゆく。内部ではLispのソースコードを分解しリストとして保持している。そしてそれを１文字ずつ読み込み、その１文字で種類が決定できるトークンについてはそのトークンを返す。そうでなければバックトラックを行うために別のメソッドを呼び出す。

Lexerクラスの使い方は簡単な解説に記したとおりだ。Schemeのソースコードを受けとり、それをスキャナがトークンに分解する。ソースコードが全てトークンに分割され

きったかどうかはトークンのkindがToken.EOFかどうかで判定する。まだ説明していないLispの文法があるためソースには意味不明な部分もあるだろうが、

‘(1 2 3)
や
`(lambda ,(map car args) ,@body)
など「’」「`」「,」「,@」を使った記法があることを念頭に置いておいて欲しい。

```
def tokenize_dot(self):
    ch = self.exp[0]
    if ch in Token.TERMINAL:
        return Token(Token.DOT)
    else:
        return Token(Token.REAL, Real('.'+self.read_atom()))
```

scan時に最初の文字が(や)である場合はそのままその文字をTokenとして返せばいいのだが、1文字目では判断のつかない「品詞」も存在する。例えば.について言えば、ドットの次の文字が空白や(などであれば(1 . 3)のようなペアを表すドットだと判断できるが、ドット以降の文字が整数であった場合（例えば“.38192”など）は小数として解釈されなければならない。そのために1文字の先読みをして、バックトラックすることによってどちらかを判定している。これは,と,@のような場合においても同じだ。（tokenize_commaの実装もこのtokenize_dotと同じような構造となっている。）

```
def tokenize_string(self):
    s = ''
    while True:
        ch = self.exp.pop(0)
        if ch == '"':
            break
        elif ch == '\\':
            ch = self.exp.pop(0)
        s += ch
    return Token(Token.STRING, s)
```

文字列は単純である。scan時に始めの文字が”であれば以降文字列が続くことが分かり、次に現れる”が終端であると簡単に判定できる。ここではエスケープ文字にも対応している。

```
def tokenize_sharp(self):
    ch = self.exp.pop(0)
    if ch == 't':
        return Token(Token.BOOLEAN, t)
    elif ch == 'f':
        return Token(Token.BOOLEAN, f)
```

実はアトムには数、文字列、シンボル以外にもう一つの種類がある。それが真偽値だ。#tと#fで表される。これらも先頭文字の#だけではどちらのトークンか判別できないため、バックトラックしている。

```
def tokenize_atom(self, ch):
    atom = ch+self.read_atom()
    try:
        return Token(Token.INTEGER, Integer(atom))
    except ValueError:
        try:
            return Token(Token.REAL, Real(atom))
        except ValueError:
            try:
                return Token(Token.RATIONAL, Rational(atom))
            except ValueError:
                try:
                    return Token(Token.COMPLEX, Complex(atom))
                except:
                    return Token(Token.SYMBOL, Symbol(atom))
```

アトムの種類の判定にはそれぞれPythonの機能をつかっている。つまりPythonでint(n)がエラーになればnは整数ではない、という判定の仕方である。IntegerやRealといったクラスはintやfloatを継承しており、それぞれSchemeに合わせた拡張を施してある。一部はPython由来の判定法ではなく、自前で正規表現による判定もしている。ただしそれらは単純なハックにも関わらず膨大な行数を消費しているので、ここでは明示的な解説はしない。無論、これらはintやfloatといった組み込み型でも代用可能である。

第5章 構文解析器を実装する

字句解析が完了したので、次に構文解析器を実装する。これは第2章で宣言したとおり予言的パーサーとして実装していく。ではさっそくソースコードを見ていこう。今回はLexerのように細切れにするのではなく、1度に全部だ。

```
class Parser:
    def __init__(self, data):
        self.lexer = Lexer(data)
        self.token = None
        self.move()

    def move(self):
        self.token = self.lexer.scan()

    def get_sexp(self):
        if self.token.kind in (Token.RPAREN, Token.DOT):
            raise Exception("syntax error")
        elif self.token.kind == Token.EOF:
            return eof
        elif self.token.kind == Token.LPAREN:
            self.move()
            return self.get_list()
        elif self.token.kind == Token.QUOTE:
            self.move()
            return self.expand_read_macro('quote')
        elif self.token.kind == Token.QUASIQUOTE:
            self.move()
            return self.expand_read_macro('quasiquote')
        elif self.token.kind == Token.UNQUOTE:
            self.move()
            return self.expand_read_macro('unquote')
        elif self.token.kind == Token.UNQUOTE_SPLICING:
            self.move()
            return self.expand_read_macro('unquote-splicing')
        else:
            atom = self.token
            self.move()
            return atom.value

    def get_list(self):
        if self.token.kind == Token.RPAREN:
            self.move()
            return nil

        cell = Cell()
        cell.car = self.get_sexp()

        if self.token.kind == Token.DOT:
            self.move()
            cell.cdr = self.get_sexp()
            self.move()  # RPAREN
        else:
            cell.cdr = self.get_list()

        return cell

    def expand_read_macro(self, macro):
        cell = Cell()
        cell.car = Symbol(macro)
        cell.cdr = Cell(self.get_sexp(), nil)
        return cell
```

明らかに、完成された言語のパーサーとは思えないほど、美しく、コンパクトだ。これがLispのLispたる所以である。第2章で話したようにLispはそれ自身が生の抽象構文木で

あるため、字句解析が完了しトークンの判別も完了した時点になっては、最早やることがないのだ。

では技術的に解説していこう。実装は典型的な予言的パーサだ。内部にLexerを保持し、先読みしたトークンをself.tokenに保存している。Lispは説明したように全てがS式で構成されるから、Parser.get_sexp()を繰り返せばよい。eofが返るまでに全てのS式が析出する。

S式の先頭が”(“である場合はそのS式がリストであると判定できる。リストはペアの再帰的な構造なので、それに合わせて実装も再帰的なものにしている。こうすることでバグもなく、美しいコードになる。唯一のコーナーケースはリストの中身がないときだ。つまり

()

のような状態である。これは明らかにペアではない。実はこれこそがnilだ。nilは空のリストである。しかしペアではない。だがnilはアトムでもある。Lisp上、最もユニークと評されるものである。ただし他のLisp語族とは違い、Schemeではリストの終端を表す以外にそれほど大きな役割を担っているわけではない。

では見慣れない‘quote’や‘quasiquote‘といった部分に焦点を当てよう。これらは見て分かるとおり、字句解析で「‘」「`」「,」「,@」を変換したトークンである。これらはLispではリードマクロと呼ばれる、いわゆる糖衣構文、省略表記のことだ。といっても名前はどうでもいい。構文解析で大事なのは「単語の意味」だ。

これら「‘」「`」「,」「,@」はS式の前に付帯し、S式を変換する。たとえば

‘(1 2 3)

というS式は

(quote (1 2 3))

というS式に変換されるべきである。同様に`や,、,@はそれぞれ「quasiquote」「unquote」「unquote-splicing」というシンボルで覆われるべきである。

では最後に与えられたソースコードから全てのS式を取り出すユーティリティ関数を作成して章を終わりとしよう。オブジェクト指向の観点から言っても明確な理由のないクラスメソッドは個人的に嫌いなので、関数として定義する。ジェネレーターとして実装することでイテレータとしても使えるようにしてみた。

```
def get_sexps(exp):
    parser = Parser(exp)
    while True:
        sexp = parser.get_sexp()
        if sexp is eof:
            break
        else:
            yield sexp
```

第6章 Lispの意味論

そろそろLispの意味論を語るべき時だろう。これまでの5章を使い、我々はLispの形式的な構造を理解することができた。表面的なルールを学ぶことができた。これからはそれらの「意味」について学ぶ時だ。構文解析によって得られた抽象構文木をどのように実行していくのか。それさえ分かってしまえばあとはほとんど機械的な作業だけが待っている。なぜならコンピュータにとってもっとも扱いにくい「生のソースコード」を扱いやすい「木構造」に変換することができたからだ。これらを踏まえ、Lispのセマンティクスを大まかに解説する。

最初の例で、1+2を計算する式として

(+ 1 2)

というS式を提示した。これについてじっくり考察してみよう。あなたは今これがリストであることを知っている。そしてリストの中のアトムが順に「シンボル」「整数」「整数」であるということを知っている。ではシンボルとは一体何者なのだろうか？種明かしをすれば、シンボルとは数学で言う変数である。例えば数学で

a = 1

f(x) = x^2

という表現は、aに「1」が、fに「xを引数に取りx^2を返す関数」が意味付けられている（束縛されている）ことを表す。この概念を使うことで、+というシンボルには『引数を2つとり、足しあわせた数を返す関数』が束縛されている、と考えることができる。そして(+ 1 2)というリストこそ、まさに、関数適用なのである。Lisp内において全てのリストは関数適用である。つまり

(f x (g 1 2 3))

というリストは、数学的に記述すれば、

f(x, g(1, 2, 3))

という値である。

もしかすると読者の皆さんの中には以上の説明を読んでひとつの疑問を浮かべた人もいるだろう。そう、こういう疑問だ：

「『(3 “String” -3.14 a-symbol)』という表現は正しいのか。

つまり、リストの先頭がシンボルでなければ、どうなるのか。」

さよう。これは文法的には正しくても意味として誤りだ。3は整数であって関数ではない。リストの先頭は常に関数でなければならない。そしてリストの先頭以外の要素を引数として適用可能でなければならない。そうでなければエラーとなる。これはつまり、リストの先頭がシンボルであっても起こりうるエラーだ。それは例えば、

```
(f 1 2 3)
```

の時。fはもちろんシンボルだが、

・fに「関数でない何か」が束縛されている場合。

・fに関数が束縛されていたとしても、fが2引数関数であった場合。

・fがたとえ3引数関数だったとしても、引数として整数以外の物（文字列など）を期待していた場合など。

我々は評価器を作成するにあたってこれらのエラーの対策もしっかり講じておかなければならない。なぜならエラーは脆弱性、コンピュータの暴走、データの欠損など全ての負の事実の根源だからだ。例えばもしエラーによって束縛される値が変化することがあれば、悪意のあるハッカーがそのエラーを意図的に引き起こして任意の値を束縛させ、彼らの思うがままにコンピュータを操作することが可能になってしまうかもしれない。もしくは一つのエラーがコンピュータを暴走させてしまうかもしれない。だから、我々はエラーに起因する予期せぬ動作について少し敏感になる必要がある。今後実際のソースコードを見てもらえれば分かるが、今回の関数の先頭がシンボルでないエラーについては、

```
raise Exception(“Atom not callable”)
```

と、エラーを発生させて対処した。このように万全のエラー対策を施した上で、処理系は絶対の信頼を得なければならないのだ。

では評価器の実装について話を戻そう。リストとシンボルの真の意味に続いて、次は「評価」について話さなければならない。評価とは、すなわち実行である。Lispでは引数は実行される。すこし込み入った話をするため、先にいくつか覚えて覚えておいて欲しいことがある。一つは束縛だ。

```
(define a 1)
```

のようにdefineというシンボルを使うことで、aに1を束縛できる。

二つ目に関数の作成だ。

```
(lambda (x y) (+ x y))
```

これは引数を二つとり、それらを足し合わせた結果を返す関数だ。しかし、この関数に名前はない。名前を付ける必要があれば、この関数を束縛する。

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

これで

```
(f 1 2)
```

は3になるはずだ。これらの構文は今までとは違う少し例外的な要素をふくんでいるので、何か疑問を抱くだろうが今はそれを横にそっと置いておいて欲しい。

ではここで評価について解説していく。たとえば

```
(define a 1)
(define b 2)
(+ a b)
```

というソースがあったとしよう。これは3になる。すなわち、+という関数に渡されるのはaやbといったシンボルではなく、aに束縛されている値、bに束縛されている値である。つまりaとbは評価されている。天下り的に解説すれば以下のようになる。

・数や文字列は評価するとそれ自身が返ってくる。

・シンボルは評価すると束縛されている中身が返ってくる。

・リストは評価すると、関数を実行する。

例えば先ほど定義した2引数を足し合わせる関数fについて考えよう。

```
(define f (lambda (x y) (+ x y)))
(define a 1)
```

このとき、

```
(f a (+ 1 3))
```

について考える。これはリストなので関数を実行しようとする。そのために、まずリストの中身をそれぞれきっかり一回評価する。fはシンボルなのでその中身が返る。

```
((lambda (x y) (+ x y)) a (+ 1 3))
```

aもシンボルなので中身に評価される。

```
((lambda (x y) (+ x y)) 1 (+ 1 3))
```

(+ 1 3)はリストなので、実行する。が、そのために+、1、3を評価しなければならない。+を評価すると二つの引数の和を返す関数が返る。そして、1と3は数字なのでそのまま1と3に評価される。つまり、二つの引数の和を返す関数に1と3を適用している。つまり答えは4だ。

((lambda (x y) (+ x y)) 1 4)

これで引数を全てきっちり一回評価することができた。よってリスト自身の評価、すなわち関数適用を行い、答えはめでたく5となる。

これで関数を実行する手続きである「評価」を理解することができた。では最後に「関数」自身について解説してこの章を終えることとしよう。実はLispには関数というものは存在しない。存在するのは手続きだけだ。（言葉のあやだが。）手続きとは引数を適用できるものだ。つまり、今まで「関数」として解説してきた物のことだ。ではなぜ関数と言わないのだろうか。それはLispにおける手続きには大きく分けて4種類存在するからだ。それらは「特殊形式（構文）」「原始関数」「クロージャ」「マクロ」と呼ばれる。これらはLispを書いている最中においては特に区別する必要はない。が、処理系の実装においてはこの差は非常に重要だ。

まず「原始関数」とは文字通りプリミティブな関数のことだ。それらには+や-といった組み込み型に対する基本処理などが含まれる。整数や文字列といった組み込み型がLisp内で定義された値でない以上、これらに対する処理は処理系自身が提供しなければならない。内部的には、例えば+というシンボルにはPrimitiveクラスのインスタンスが束縛されており、そのインスタンスはpythonの関数自身を内包している。

クロージャとはLisp内でlambdaを使って定義された関数のことだ。これについては後述する。

特殊形式については具体例を出して解説してゆこう。特殊形式の中にifという構文がある。これは

(if #t 1 2)

などのように使用し、一つ目の引数が#tなら二つ目の引数を、#tでなければ三つ目の引数を返す手続きだ。一見、至極簡単な手続きだ。だが一つ落とし穴がある。今までの例通りであれば、第一引数の値にかかわらず、引数は評価されてしまうのだ。

一つ例を挙げる。Schemeではdisplayという原始関数があり、これはpythonでいうprint文に相当する。つまり

(display “Hello, World”)

とすることで、引数として与えられた文字列を標準出力に出力する。これを使ってif文を記述してみよう。

(if #t (display “TRUE”) (display “FALSE”))

一見、これは(display “TRUE”)だけが実行されるように見える。しかしここで、評価の話を思い出して欲しい。リストを評価するとき、まずリストの全ての中身を評価する。つ

まり、(display “TRUE”)も(display “FALSE”)も第2引数に関わらず評価（実行）される。これを防ぐための手続きが特殊形式と呼ばれるものだ。そして最後に残ったマクロは、Lisp内で定義する特殊形式のことだ。つまり以下のような表になる。

	引数を先に評価する	引数を評価しない
処理系側で定義する	プリミティブ	特殊形式
Lisp内で定義する	クロージャ	マクロ

では実際にLispの評価器を作成する。

第7章 評価器evalの作成　〜レキシカルスコープ編〜

前述したようにLispはS式の評価の連鎖によって「実行」される。S式を評価する再帰的な関数を一つ定義してしまえば処理系の作成はほとんど完了したことになる。その関数こそevalとよばれる関数だ。しかしevalを実装する前にひとつこなして置かなければならないことがある。変数テーブルの作成だ。例えばa-symbolというシンボルに値を束縛したとき、「”a-symbol”にXという値を束縛した。」という情報を保存しておく、言わばデータベースが必要になる。このデータベースはシンボル自身の評価にも必要で、高速かつ堅牢でなければならない。またSchemeにおける変数テーブルは若干変則的なのでそのことについても留意しておく必要がある。

Schemeが他のLisp語族と大きく違う点のひとつとしてレキシカルスコープというのが挙げられる。これは変数テーブルの実装方法の一つのことだ。例を挙げよう。

Schemeにはletという構文がある。変数を定義するための構文だ。

```
(define a 1)
(define returna (lambda () a))
(returna)
```

このとき最後の(returna)が1となることは承知であろうと思う。では続けて以下を実行してみる。

```
(define f (lambda (a) (returna)))
(f 2)
```

このとき(f 2)は1となる。しかしよく考えて欲しい。(f 2)を評価する際には以下のようになっているはずだ。

(f 2)
((lambda (a) (returna)) 2)
そしてここでaに2が束縛される。そして

((returna))
((lambda () a))
(a)
となり、aに束縛されている値が返される。ここで1が返されるということはつまり、returnaに束縛した際のaが保存されているということだ。関数を定義して局所束縛をする際はスコープが変わり、そのスコープを関数ごとに保存しておかなければならないということを意味するのだ。

もしもこの解説が理解できなくても慌てないでほしい。第8章以降の具体的な実装からその意図を改めて汲みとっていただきたい。

ではこのネストしたスコープを実現するためのテーブルを実装する。シンボルという一意の値にそれぞれ対応する値を結び付けたいのでハッシュテーブルが理想的だ。そしてそれはPythonの組み込み型でもある。

```
class Env(dict):
    def __init__(self, outer=None):
        self.outer = outer

    def find(self, var):
        if var in self:
            return self[var]
        elif self.outer is not None:
            return self.outer.find(var)
        else:
            raise Exception("Not found a symbol: "+str(var))
```

Envの本質はシンボルをキーとしてそのシンボルが束縛しているS式を返すハッシュテーブルである。シンボルの検索はそのシンボルを評価した地点から外側のスコープへと順番に行われていく。とあるEnvのインスタンスは、自身のひとつ外のEnvを保持しており、そのEnv自身も自身のひとつ外のEnvを保持している。そして、シンボルの検索もそれにしたがって行ってゆく。変数aをEnvのとあるインスタンス、envから検索するには、

env.find(a)
とするだけでよい。env自身にaがなければ、その外側のenvから検索する。もしaが最外のEnvでも見つからなければ、‘Symbol not found’というエラーが返る。また、dictクラスを継承している利点として、シンボルへの束縛が

env[symbol] = sexp

とするだけで済むというのも嬉しいポイントだ。

最後にシンボルの束縛に関連してもう一つ便利な関数を定義しておこう。リストの正体が数珠つなぎになったペアであることはすでに解説した通りだ。リストは内部に「.」を書くことで明示的にペアの連続として記述することができる。

(1 2 3 4) ≡ (1 . (2 . (3 . (4 . nil))))

実はもう一つ省略記法がある。それが以下のような場合だ。

(1 . (2 . (3 . 4)))

これは終端がnilではないのでリストではない。しかしこれの為だけにわざわざドット対をいくつも書くのは美しくない。このためにSchemeでは以下のような記法が許されている。

(1 2 3 . 4)

実はこの記法はSchemeをより強力にしている。この糖衣構文によって可変長引数の関数を定義することができるのだ。たとえば

(define f (lambda (arg1 arg2 . args) (something ...)))

という関数があったとしよう。ここで

(f 1 2 3 4 5)

とした場合を考える。これは構造的には以下のようになる。

(f . (1 . (2 . (3 . (4 . (5 . nil))))))

では関数のパラメータのリストとこの引数のリストを比較してみよう。

(arg1 . (arg2 . args))
(1 . (2 . (3 . (4 . (5 . nil)))))

そして、この場合の束縛は以下のようになる。

arg1 = 1, arg2 = 2, args = (3 4 5)

このように記述することで処理系の拡張も最小限ですみ、かつ言語としても新しい構文を用意せずに可変長引数を実現している。これらリストとリストを順に辿って束縛してゆくユーティリティ関数を定義して、envモジュールは完成だ。

```
def bind(pars, args, env):
    if isa(pars, Cell):
        env[pars.car] = args.car
        bind(pars.cdr, args.cdr, env)
    elif not pars == nil:
        env[pars] = args
    elif args != nil:
        raise Exception("Invalid number of arguments")
```

簡単な解説：
束縛される側のリストを再帰的にたどり、束縛する側のリストの先頭から順に束縛してゆく。束縛する側の数が足りなければエラーを返す。

第8章 評価器evalの作成　〜eval関数編〜

ではeval関数の実装を見てゆく。ここで関数名がevalsとなっているのはPython自身のeval関数と名前衝突を起こしてしまうからである。

```
def evals(x, env):
    if x is eof:
        return

    elif isconstant(x):
        return x

    elif issymbol(x):
        return env.find(x)

    elif islist(x):
        proc = evals(x.car, env)

        if isa(proc, Syntax):
            return proc.proc(x.cdr, env)

        elif isa(proc, Primitive):
            args = list(evalall(x.cdr, env))
            return proc.proc(*args)

        elif isa(proc, Closure):
            new_scope = Env(proc.env)
            # Evaluate parameters
            arguments = evalall(x.cdr, env)
            # Bind
            bind(proc.pars, arguments, new_scope)
            # Evaluate a closure
            return evals(proc.exp, new_scope)

        elif isa(proc, Macro):
            new_scope = Env(proc.closure.env)
            # Bind
            bind(proc.closure.pars, x.cdr, new_scope)
            # Expand
            exp = evals(proc.closure.exp, new_scope)
            # Evaluate
            return evals(exp, env)

        else:
            raise Exception('Atom not callable: {0},{1}'.format(type(x.car), x.car))
```

第6章で述べたように評価は以下の法則で行われる。

・数や文字列は評価するとそれ自身が返ってくる。

・シンボルは評価すると束縛されている中身が返ってくる。

・リストは評価すると、関数を実行する。

しかし最後の項目については後により複雑な体系について解説した。つまり

・リストは評価すると、

・クロージャなら引数を全て評価し、それを適用する。

・プリミティブなら引数を全て評価し、それを適用する。

・特殊形式かマクロなら、引数を評価せずに、適用する。

という法則だ。

ではソース内のif〜elif〜else文を一つずつ見てゆく。まずevals関数はS式及び現在のスコープを受け取る。

S式の中身がなければNoneを返す。

S式がシンボル以外のアトム（定数：数や文字列など）ならそれ自身を返す。

S式がシンボルならそこに束縛されている値をスコープから検索し、返す。スコープに存在しなければenv.findメソッドは例外を投げる。

S式がリストなら。まず先頭の手続きが4種類の手続きのうちどれかを調べる。

・もし手続きが特殊形式なら、構文クラスが保持する関数に生の状態の引数を渡す。例えばSyntaxクラス（特殊形式を表すクラス）のインスタンスのうち、if構文に対応するインスタンスは、内部に以下のような関数を保持している。proc.proc(args, env)とすれば、この関数が実行されるという仕組みだ。

```
def if_syntax(arg, env):
    if evals(arg.car, env) == t:
        return evals(arg.cdr.car, env)
    elif not arg.cdr.cdr == nil:
        return evals(arg.cdr.cdr.car, env)
```

・もし手続きがプリミティブなら、まずevalall関数を使って、引数を全て評価する。

```
def evalall(exps, env):
    if exps == nil:
        return nil
    else:
        return Cell(evals(exps.car, env), evalall(exps.cdr, env))
```

リストを渡すと中身がそれぞれ評価された後の値になった新しいリストが返される。

プリミティブはPython自身の関数であるため、引数を適用できるようにリストに変換したあと、それを実行する。たとえばenv内の’+‘というシンボルには

env[‘+’] = (lambda a, b = a+b)

という代入がなされている。Pythonの関数と、Schemeの関数の橋渡しをする役目がこのプリミティブだ。

・もし手続きがクロージャなら、同じくevalall関数を使って、引数を全て評価する。クロージャを実行するには、あたらしいスコープを作り、そのなかで中身を評価すればよい。たとえば

((lambda (x y) (+ x y)) 1 2)
ならまずクロージャのための新しいスコープを作る。そしてそのスコープ内で

env[x] = 1;　env[y] = 2
としておく。わざわざ新しいスコープを作るのは、この束縛が局所束縛であり、周りの環境に影響を与えてはいけないからだ。そしてそのあたらしいスコープの中で、本体部、ここでいう(+ x y)の部分を評価する。

・もし手続きがマクロなら。マクロはユーザーがScheme内で定義する構文だと説明したが、それ以上の深追いは避けておく。ただソースを見れば分かるように、マクロは生の（＝未評価の）S式を受け取って、それを変換し、別のS式を返すクロージャを保持している。その変換が完了したあとのS式を再度評価するのがマクロだ。マクロの概念はかなり奥が深く、ここで解説するには余白があまりにも足りない。それでもここでわざわざ登場させたのはこれこそがLispの本質だからである。余裕があれば自力で調べて欲しい。

これで、evals関数は完成だ。ただしこのevals関数にこのままParserを使って作成したS式を渡しても動かない。なぜならenvにたくさんの組み込み手続きを入れて置かなければならないからだ。これらは全てを解説するには多すぎるので、最外のenvに特殊形式を登録する関数のうち、define、lambdaだけを抜粋して掲載する。あくまで参考のためであり、解説していない部分も多々あるので注意されたい。

```
def add_syntaxes(env):
    def define_syntax(arg, env):
        name = arg.car
        env[name] = evals(arg.cdr.car, env)
        return name

    def lambda_syntax(arg, env):
        pars = arg.car
        exp = arg.cdr.car
        return Closure(exp, pars, env)

    for name, cont in {'lambda': lambda_syntax, 'define': define_syntax,}.items():
        env[Symbol(name)] = Syntax(cont)
```

第9章 REPL(Read-Eval-Print Loop)の作成とその後

これでScheme本体は全て完成した。最後にインタラクティブなインタプリタを作成しよう。いわゆるREPLと呼ばれるもので、式を読み込み（Read）、それを評価し（Eval）、結果を表示する（Print）、を繰り返す（Loop）インタラクティブシェルのことだ。これに必要な関数等はすべて事前に用意してある。そしてこれがソースだ。

```
def repl():
    print '/'*5, 'NLisp Interpreter', '/'*5, '\n'
    global_env = Env()
    add_global_functions(global_env) # Load primitives and special forms

    while True:
        s = raw_input('Lisp >> ')
        for exp in analyzer.get_sexps(s):
            result = evals(exp, global_env)
            if result:
                print result
```

では、これを実行してみよう。「Lisp >> 」というプロンプトに向かって何かS式を打ち込んでみる。たとえば(+ 1 2)。もし+のプリミティブ関数を実装し、それをグローバルにちゃんと登録していれば、結果は3になる。これであなた自身の言語の完成だ。

とはいってもこのLispには文字通り何も無い。なぜなら何も作っていないから。1人前にSchemeと名乗るには数多くの標準ライブラリを実装しなければならない。それに、Schemeに必要ないくつもの根本的な機能が欠けている。たとえば末尾最適化の保証だ。いわゆるLispはその構造上再帰関数を大量に駆使したコードになりがちである。しかし再帰関数はメモリをより多く消費し、ある一定回数以上は再帰できなくなる。これを防ぐための手立てが末尾最適化とよばれる処理であり、Schemeと名乗るにはこれが必要条件だ。しかし、現時点ではそれが実装されていない。では実装は不可能なのかというと、案外そうではない。以下がevals関数の末尾最適化版だ。（変更部分のみ抜粋）

```
def evals(x, env):
    while True:
…
        elif isa(proc, Closure):
…
            x, env= proc.exp, new_scope

        elif isa(proc, Macro):
…
            x = exp
```

見て分かるとおり、全体をwhileループで囲み、return evals(...)をx = ..., env = ...へと変換している。たったこれだけで末尾最適化は完了だ。他にもSchemeと名乗るには足りない部分がいくつもある。Portの実装、ベクタ、文字型、そして継続だ。特に継続は関数呼び出しをスタックしておく必要があるため、実装が面倒だ。これらの実装はあなた自身が考察してほしい。

最終章

以上が言語作成のすべてだ。駆け足ではあったものの、文法の解説からその実装までをこれほどコンパクトに解説できたのはひとえにLispのもつ力だろう。これを機に、初心者の方にはプログラミングに興味を持ちコンピュータのもつおもしろさに触れて欲しい。またプログラミング経験のある方は、言語という低レイヤーに価値を見出してくれればと思う。

今回作成したLispはNLispと名付けた。これは筆者の名前、我が灘校、そしてこのパソコン部(NPCA)のイニシャルを取ったものだ。このNLispのすべてのソースコードは以下のURLからアクセスできる。興味が湧いたならぜひ参考にしていただきたい。

https://github.com/wasabiz/NLisp

また、いくつか参考にしたウェブサイトを記していく。後学の足しになればと思う。

http://www.unixuser.org/~euske/doc/r5rs-ja/r5rs-ja.pdf
http://practical-scheme.net/gauche/index-j.html
http://www.aoky.net/articles/peter_norvig/lispy.htm
http://www.aoky.net/articles/peter_norvig/lispy2.htm

謝辞

ここまでお読みいただき誠にありがとうございました。本日は灘校文化祭へ、そしてパソコン部へ来場いただき大変喜ばしい限りです。本文を執筆するにあたって参考にしていただいたウェブサイト様、文献の筆者様、ならびに僕にコンピュータの楽しさと喜びを教えてくれたパソコン部部員と全ての人々に感謝いたします。

by Wasabiz