シンタックスが無ければ作ればいいじゃない

λ組　zick

平成 20 年 11 月 25 日

1　はじめに

1.1　前置記法はお嫌い？

「Lisp？ “1+1” をわざわざ “(+ 1 1)” とか書かないといけないんだろ？ さらには “x=2” は “(setq x 2)” だっけ？ 面倒でやってられないよ。」Lisp をちょこっとだけかじった人からこんな言葉を聞くことがあります。Lisp を書きなれてる人はこれに対して「だからそれが分かりやすいんじゃないか。」と反論しますが、大抵の場合は納得してもらえず、無駄な言い争いが始まってしまいます。

しかし、Lisp 方言の代表格である Common Lisp では “1+1” のような中置記法が絶対に使えないのかといえばそんなことはありません。元々書けないのであれば、書けるようにするまでです。

「シンタックスが無ければ作ればいいじゃない」

本記事ではシンタックスを新たに作れるという Common Lisp の脅威の力を紹介します。

1.2　目標

Common Lisp のリーダマクロ (reader macro) を用い、Common Lisp のプログラムに中置表記の式を書くことができるようにすることを、ここでの目標とします。

具体的には

```
(let (x y)
  #[ x = (y=5*(4+3)) - 2 ]
  (format t "~&x=~A~%y=~A~%" x y))
```

こんなプログラムを動かしたときに、

```
x=33
y=35
```

と出力されるようにすることとします。これなら前置記法が嫌いで中置記法しか愛せない方々からも文句は言われないでしょう。多分。

2 リーダマクロ

2.1 マクロ文字

Common Lisp はプログラムを読み込む際に、マクロ文字 (macro character) があれば特別な動作をしながら構文木 (syntax tree) を作ります。例えば「'(a b c)」という文字列が読み込まれると「(quote (a b c))」のような構文木が出来上がります (「'(a b c)」の括弧はただの文字に過ぎず、「(quote (a b c))」の括弧はリストを表す S 式の表現であることに注意してください)。これは、「'」が「(quote 次に来るもの)」という形の構文木を作るようなマクロ文字であり、「(」が「)」に来るまで式を読み込み、それをリストにするマクロ文字であるためです。

マクロ文字はプログラマが新たに設定することができ、その動作を Common Lisp のプログラムを書くことにより指定できます。例えば、文字「!」に「'」と同様の動作をさせるようにしてみましょう。

```
(defun !-reader (stream char)
  (declare (ignore char))
  `(quote ,(read stream t nil t)))
(set-macro-character #\! #'!-reader)
```

これで、今後「'(a b c)」と書く代わりに「!(a b c)」と書くことが出来るようになりました[1]。

2.2 リードテーブル

しかし、一度マクロ文字を設定してしまうと、今後ずっとその影響が残ります。マクロ文字の影響する範囲を限定するためにはリードテーブル (readtable) を一旦コピーしてからそこにマクロ文字を設定することになります。

マクロ文字の情報はリードテーブルというところに保存されます。現在のリードテーブルはスペシャル変数*readtable*に入っています。*readtable*はスペシャル変数なので let で新たに値を設定して、その let の中で set-macro-character を使うとその有効範囲は let の中だけになります。また、現在のリー

[1] 実際には既にマクロ文字に割り当てられているリーダマクロ関数を get-macro-character 関数を使って取得できるので、「(set-macro-character #\! (get-macro-character #\'))」とだけ書けば同じことが出来ます。

ドテーブルをコピーするには関数 copy-readtable を使えばいいので、先ほどのマクロ文字「!」の有効範囲を限定するには次のようにします。

```
(defun !-reader (stream char)
  (declare (ignore char))
  `(quote ,(read stream nil nil t)))

(let ((*readtable* (copy-readtable)))
  (set-macro-character #\! #'!-reader)
  (defvar *my-list* !(a b c)))
```

こうすると、*my-list*の値は (a b c) になりますが、let を抜けた後は「!」の効果はなくなるようになります。

3 演算子順位構文解析

3.1 アルゴリズム

ここでは中置表記の式を読み取るために演算子順位構文解析 (operator precedence parsing) を行います。演算子順位構文解析はその名の通り、演算子に順位を付け、それに基づいて構文解析を行う手法です。

そのアルゴリズムの基本方針は「新たに読み込んだ演算子が前に読み込んだ演算子より優先順位が低ければ、前に読み込んだ演算子を還元 (reduce) する、そうでなければ新たに読み込んだ演算子をシフト (shift) する」というものです。詳細な説明は参考文献 [1] に譲るとして、ここでは次のようなアルゴリズムを利用しました。

```
# s はスタックトップを表す
# a<<s は a より s の方が優先順位が高いことを表す
# a>>s は s より a の方が優先順位が高いことを表す
# s=a は s と a の優先順位が等しいことを表す
# 文字列終端記号$と開き括弧の優先順位は最も低いとする
1．スタックに$を PUSH
2．字句を 1 つ読み込み a に代入
3．a が演算子と括弧でなければ
        a を還元
        2 に戻る
4．a<<s または (a=s かつ s が左結合) ならば
        s をスタックから POP して還元
        4 に戻る
5．a>>s ならば
        a をスタックに PUSH
6．a が閉じ括弧ならば
        対応する開き括弧を POP するまで、スタックから POP しながら還元
7．a が$ならば
```

表 1: 「3*2+1」の構文解析

字句	スタック	動作 (理由)	結果
3	[$]	reduce(演算子以外)	[3]
*	[* $]	shift(* ≫ $)	[3]
2	[* $]	reduce(演算子以外)	[2 3]
+	[$]	reduce(+ ≪ *)	[(* 3 2)]
1	[+ $]	reduce(演算子以外)	[1 (* 3 2)]
$	[]	reduce($ ≪ +)	[(+ (* 3 2) 1)]

終了
８．２に戻る

ここで言う「文字列終端記号」とはストリームの終端で「(read stream nil :eof)」を評価した際に得られる「:eof」のことです。文脈自由文法における「終端記号」とは無関係です。また、このアルゴリズムでは簡単のため文字列終端記号と括弧を演算子と同様に扱ってますが、後ほど作成するプログラムでは特別扱いしています。

還元の処理ですが、結果格納のためにもう一つスタックを用意して、演算子以外はそのまま (必要であれば何か処理を施して)PUSH する、演算子は結果格納用のスタックを好きにいじってから何かを PUSH する (場合によっては何も PUSH しない) ことにします。

このアルゴリズムを用い「3*2+1」の構文解析を行い、構文木「(+ (* 3 2) 1)」が出来上がる様子を表 1 に示します。

4 Common Lisp での実装

4.1 字句解析

構文解析を行う前に字句解析 (lexical analysis) を行う必要がありますが、ここでは字句解析を read 関数に任せることにします。read は現在のリードテーブルの規則に沿って 1 つだけ字句 (token) を読み込みます。

そうすれば、演算子に自身を表す文字[2]を返すようにマクロ文字を設定するだけで、字句解析を行うことが出来ます[3]。ただし、この方法では演算子は一文字という制限が掛かってしまいますし、二項演算子の「-」(引く) と単項演算子の「-」(マイナス) の区別が出来なくなります。本記事では簡単のためこの制限を受け入れることにします。

[2]演算子+なら文字#\+といった具合です
[3]これは set-macro-character の第 3 引数を指定しなければその文字が区切り文字に設定されるためです。

4.2 全体の構成

さて、最初に立てた目標を達成すべくソースを書き始めます。まず、ストリームから字句を読み取り構文解析を行う関数 (仮に my-parser とします) を定義し、そして、set-dispatch-macro-character 関数を用い「#[」が読み込まれたら、「]」を読み込むまで ny-parser に読み込みを任せるようにします。つまり以下のようなソースとなります[4]。

```
(defun my-parser (stream &rest rest)
  (declare (ignore rest))
  ;; 区切り文字を設定
  ;; 実際に構文解析し構文木を返す
  )
(set-dispatch-macro-character #\# #\[ #'my-parser)
```

あとは、my-parser をガリガリと書いていけば一応完成するのですが、せっかく Common Lisp を使うのですからマクロを用いて、必要なものを指定すれば勝手に関数を生成してくれるようにしましょう。

構文解析器に必要なパラメータは

- リテラル (演算子と括弧以外の字句) に対する処理
- 演算子とその優先順位、結合性、還元時の処理
- 開き括弧と閉じ括弧、還元時の処理
- 文字列終端記号

の 4 つなので、これら 4 つを指定と関数名を指定すると構文解析を行う関数を定義するマクロ def-op-parser を作ります。こうしておくことで複数の構文解析器が欲しくなったときに楽が出来ます。

ここでは、演算子以外に対する処理は特になく、演算子は「*, /, +, -, =」の五種類とし、優先順位は「=」が一番低く、「*, /」が一番高いように設定し、「=」のみを右結合としそれ以外は左結合とします。それぞれの演算子は還元時に「(* x y)」「(/ x y)」「(+ x y)」「(- x y)」「(setq x y)」という式を生成するようにします。また、括弧は「()」のみを用いて、還元時には括弧の中身をそのまま返すことにします。文字列終端記号は「]」です。以上の要求を満たす構文解析を行う関数を以下のように定義できるようにします。

```
(def-op-parser my-parser (input stack acc)
  ()
  (((#\+ 2 :left)
    (let ((x (pop acc)) (y (pop acc)))
```

[4] my-parser は 1 つ以上の引数を受け取り、第 2 引数以降を無視するように定義していますが、これは、set-macro-character と set-dispatch-macro-character の両方に使えるようにするためです。詳しくは参考文献 [2] の両関数の項目をお読み下さい。

```
    (push `(+ ,y ,x) acc)))
 ((#\- 2 :left)
  (let ((x (pop acc)) (y (pop acc)))
    (push `(- ,y ,x) acc)))
 ((#\= 1 :right)
  (let ((x (pop acc)) (y (pop acc)))
    (push `(setq ,y ,x) acc)))
 ((#\* 3 :left)
  (let ((x (pop acc)) (y (pop acc)))
    (push `(* ,y ,x) acc)))
 ((#\/ 3 :left)
  (let ((x (pop acc)) (y (pop acc)))
    (push `(/ ,y ,x) acc))))
(((  #\( #\) ) ))
#\])
```

それでは、def-op-parser を作っていきましょう。

4.3 def-op-parser の引数

上記 def-op-parser の使用例を振り返ってください。第 1 引数は定義する関数の名前です。第 2 引数はというと、演算子順位構文解析の節で出てきた「読み込んだ字句」「スタック」「結果用のスタック」の 3 つを表す変数です。これらは次に説明するリテラル、演算子、括弧の指定において使用します。

さて、第 3 引数ですが、リテラルに対する処理の指定です。これは次の様な文法で指定することにします。

```
((test-form form1 form2 ...) ...)
```

ここで、「...」は「0 回以上の繰り返し」を表します。test-form を評価し、それが真であれば form1 以降の式が左から順に評価されます。これは cond の節とほぼ同じであり、次のように使います。

```
(((stringp input) (push (intern input) acc))
 ((consp input) (error "bad expression"))
 ((symbolp input) (push input acc)))
```

全ての test-form が偽になった場合、その記号は acc に push されます (つまり上の例の ((symbolp input) (push input acc)) は本当は不要です)。

第 4 引数は演算子に対する処理の指定です。これは次のような文法で指定します。

```
(((op-char precedence association) form1 form2 ...) ...)
```

op-char には演算子を表す文字を指定し、precedence には優先度を表す 1 以上の整数 (大きい方が優先度が高い) を指定、association には:left か:right を

指定します。form1, form にはこの演算子が還元するときに行う処理を指定します。

第 5 引数は括弧に対する処理の指定です。これは次のような文法で指定します。

```
(((open-char close-char) form ...) ...)
```

open-char は開き括弧を表す文字、close-char は閉じ括弧を表す文字を指定し、form には還元時の処理を書きます。form は省略可能です (そうすると、括弧の中身が結果用スタックのトップに残ることになります)。

第 6 引数は文字列終端記号を表す文字を指定します。

第 3 引数から第 5 引数まではなんだか cond の文法と似ていますね。これは後で cond の中に突っ込みやすくするための工夫です。それはさておき、それらの文法から特定のものを取り出しやすいように以下のマクロを定義します。

```
(defmacro op-char (spec)
  `(first (car ,spec)))
(defmacro op-precedence (spec)
  `(second (car ,spec)))
(defmacro op-association (spec)
  `(third (car ,spec)))
(defmacro op-body (spec)
  `(cdr ,spec))
(defmacro lt-test (spec)
  `(car ,spec))
(defmacro lt-body (spec)
  `(cdr ,spec))
(defmacro pr-open (spec)
  `(first (car ,spec)))
(defmacro pr-close (spec)
  `(second (car ,spec)))
(defmacro pr-body (spec)
  `(cdr ,spec))
```

4.4 区切り文字の生成

構文解析を始める前に、全ての演算子、括弧及び文字列終端記号を区切り文字に設定します。まず、「指定された文字を “その文字自身を返すマクロ文字” に設定する式を返す」関数 set-self-macro-character を定義し、演算子、括弧は map 系の関数を用い全ての記号に適用し、文字列終端記号は直接適用することにします。

```
(defun set-self-macro-character (char)
  `(set-macro-character ,char
                        #'(lambda (s c)
                            (declare (ignore s c))
```

```
                              ,char)))

(defun set-op-macro-character (operators)
  (mapcar #'(lambda (op)
              (set-self-macro-character (op-char op)))
          operators))

(defun set-pr-macro-character (parentheses)
  (mapcan #'(lambda (pr)
              `(,(set-self-macro-character
                   (pr-open pr))
                ,(set-self-macro-character
                   (pr-close pr))))
          parentheses))
```

4.5 シフトと還元の処理

さて、次にスタックトップと読み込んだ字句を比較して、シフトか還元を行う処理を作ります。演算子順位構文解析の肝となる部分であるだけに、少々長いコードとなっていますが、演算子順位構文解析のアルゴリズムとあわせてみたらそれほど難しいものではありません。

```
(defun make-action-table (op operators parentheses stack)
  (let ((i-prec (op-precedence op))
        (i-char (op-char op)))
    `(ecase (car ,stack)
       ,@(mapcar
          #'(lambda (op2)
              (let ((s-prec (op-precedence op2))
                    (s-assoc (op-association op2))
                    (s-char (op-char op2)))
                `((,s-char)
                  ,@(if (or (> i-prec s-prec)
                            (and (= i-prec s-prec)
                                 (eq s-assoc :right)))
                        `((push ,i-char ,stack)
                          :shift)
                      `((pop ,stack)
                        ,@(op-body op2)
                        :reduce)))))
          operators)
       (,(mapcar #'(lambda (pr) (pr-open pr))
                 parentheses)
        (push ,i-char ,stack))
       ((:bottom)
        (push ,i-char ,stack)
        :shift))))
```

この関数は第 1 引数に指定された演算子を読み込んだ際に、スタックトップにあわせた動作を行うコードを生成します。スタックには演算子か開き括弧しか PUSH されないので、ecase を使って全ての場合に対する場合分けを行います。また、演算子順位構文解析のアルゴリズムの説明では最初にスタックに文字列終端記号を PUSH していましたが、:bottom というシンボルを PUSH するように若干アルゴリズムを変更し、それに対する処理も書いています。また、シフトが行われた際には:shift、還元が行われた際には:reduce が評価されます。これら 2 つの値は後ほど使用します。

4.6 リテラルの指定を cond の節に展開

ここからしばらくは、読み込んだ字句に対する場合分けのコード生成を考えて生きます。この場合分けは cond で行うことにします。

リテラルの指定はそのまま cond の節として利用できるのですが、一応、関数を一段挟むことにしましょう。

```
(defun make-literal-cond-clause (literals)
  (mapcar #'(lambda (l)
              `(,(lt-test l)
                ,@(lt-body l)))
          literals))
```

色々してますが、現時点の仕様では literals と同じ形のリストが返されます。

4.7 演算子の指定を cond の節に展開

次に、演算子が入力された場合の処理を作ります。入力が演算子と同じ文字であれば、シフトまたは還元を行いますが、シフトと還元を行うコードの生成は既に作ってあります。還元は可能な限り連続して行うので次の様なソースになります。

```
(defun make-operator-cond-clause
       (operators parentheses input stack)
  (mapcar #'(lambda (o)
              `((eql ,(op-char o) ,input)
                (do ()
                    ((not (eq ,(make-action-table
                                o operators
                                parentheses stack)
                              :reduce))))))
          operators))
```

4.8 括弧の指定を cond の節に展開

同様に、括弧が入力された場合の処理を作ります。開き括弧が入力されればシフト、閉じ括弧が入力されれば対応する開き括弧を見つけるまで、スタックを POP しながら還元を繰り返しますが、この処理はすぐ後で定義する reduce-to 関数に任せ、先に cond の節に展開するコードを書いてしまいます。

```
(defun make-parenthesis-cond-clause
       (parentheses operators input stack)
  (mapcan #'(lambda (p)
              `(((eql ,(pr-open p) ,input)
                 (push ,input ,stack))
                ((eql ,(pr-close p) ,input)
                 ,(reduce-to (pr-open p) operators
                             parentheses stack)
                 ,@(pr-body p))))
          parentheses))
```

4.9 特定のものを見つけるまで還元

さて、先ほど使用した関数 reduce-to を作ります。この関数はスタックから特定のものを POP するまで還元を繰り返します。還元を行うために優先順位 0 (def-op-parser で指定できるのは 1 以上なのでもっとも優先順位が低い) の演算子に対する処理を生成し、利用しています。

```
(defun reduce-to (to operators parentheses stack)
  `(do ()
       ((or (eq (car ,stack) ,to) (null ,stack))
        (when (null ,stack)
          (error "Syntax error! (by op-parser)"))
        (pop ,stack))
     ,(make-action-table '((#\Null 0 :left) nil)
                         operators parentheses stack)))
```

この関数は括弧の対応を取る以外にも、最後に文字列終端記号を読み込んだ際に使用します。

4.10 いよいよ完成

必要なパーツは全て出来上がりました。あとはこれらを組み合わせるだけです。そう、ついにマクロ def-op-parser を書くときが来たのです。

```
(defmacro def-op-parser (name (input stack acc)
                         literals operators parentheses
                         &optional (delimiter :eof))
  (let ((stream (gensym))
```

```
    (rest (gensym)))
`(defun ,name (,stream &rest ,rest)
  (declare (ignore ,rest))
  (let ((*readtable* (copy-readtable)))
    ,@(set-op-macro-character operators)
    ,@(set-pr-macro-character parentheses)
    ,@(when (characterp delimiter)
        (list (set-self-macro-character delimiter)))
    (do ((,input (read ,stream nil :eof nil)
                 (read ,stream nil :eof nil))
         (,stack (list :bottom))
         ,acc)
        ((or (eq ,input :eof)
             (eql ,input ,delimiter))
         ,(reduce-to
            :bottom operators parentheses stack)
         (car ,acc))
      (cond ,@(make-literal-cond-clause literals)
            ,@(make-operator-cond-clause
                operators parentheses input stack)
            ,@(make-parenthesis-cond-clause
                parentheses operators input stack)
            (t (push ,input ,acc))))))))
```

まず、区切り文字を設定して、後は文字列終端記号を読み込むまでループ。ループの内容はというと、cond を使ってリテラル、演算子、括弧に対する場合分けを行い、どれにも当てはまらなければシフト。以上の処理を行う関数を定義。

最も長いマクロではありますが、大したことはやってません。何はともあれこれで完成です。def-op-parser に関するプログラムを全てロードし、以前出てきた def-op-parser の使用例をコンパイルしてロードすると、目的を満たす構文解析を行う関数 my-parser が出来上がるので、これまた以前出てきたとおりに set-dispatch-macro-character で登録してやると、全ての準備が整います。

4.11 いざ実行！

高まる気持ちを抑えつつ、ゆっくりと目標であった式を入力。そして、Enter を押し式を評価。

```
CL-USER> (let (x y)
           #[ x = (y=5*(4+3)) - 2 ]
           (format t "~&x=~A~%y=~A~%" x y))
x=33
y=35
NIL
```

見事動作しました[5]。もう「Lispが中置記法が使えなくて不便な言語」なんて言わせません!!!

5 終わりに

5.1 効率

「へー、Lispってこんなことも出来るんだ。けど遅いんじゃない？」苦し紛れにこんな反応を示す方もいるでしょう。確かに私が書いたコードの効率がいいとはとても言いがたいです。

しかし、中置記法が本来のLispの式に変換される処理は「リード時」に行われます。具体的には、インタプリタにソースが読み込まれるときや、ソースをコンパイルするために読み込むときです。インタプリタで実行する際には若干のロスが生まれてしまいますが、コンパイルしてしまえば、コンパイル時間が若干遅くなるだけで、実行時には何一つ余計な時間は掛かりません。

言語自体のシンタックスを追加するという高度な抽象化を行いながら、実行時に余計な時間が掛からないというのは、Common Lispならではでしょう。

5.2 今後の展望

今回作ったのは「Common Lispのコードに別のシンタックスを埋め込む」といったものでしたが、私には「全く別の言語をCommon Lispのコードとしてリードする」という野望があります。リードさえ出来てしまえばコンパイルはCommon Lispの処理系ががんばってくれるので、実質コンパイラが出来てしまうことになります。

実行時にCommon Lispの処理系が必要なことが気になるのであれば、ECL[6]のような処理系を使えばCのソースを吐かせたり、Cからコンパイル済みのファイルを読み込ますことが出来てしまうので、そういった手段を使うのもいいかもしれません。

だらだらと長い記事になってしまいましたが、最後までお読み頂きありがとうございました。本記事により少しでもCommon Lispの面白さが伝われば何よりです。

```
(defvar lisp '#1=(#1# is SUGOKU powerful!))
```

[5]実際にはREPLを使って関数やマクロをいくつか作るごとに試していたため、ほぼ確実に動くのが分かっていたんですけどね (笑)

[6]http://ecls.sourceforge.net/

参考文献

[1] 中田育男：
コンパイラの構成と最適化
朝倉書店

[2] Kent Pitman, X3J13 Project Editor :
Common Lisp HyperSpec
(http://www.lispworks.com/documentation/HyperSpec/Front/index.htm)