[3 回目　２００２・９・１７]

数式の構文解析：top-down parser の作り方

前回、簡単な数式の処理系を解説した。構文は、以下のような **BNF(Backus nur form, Backus normal form)**,による構文規則で記述される。

足し算の式　:= 式　＋の演算子　式
引き算の式　:= 式　－の演算子　式
式　:= 数字　| 足し算の式 ｜　引き算の式

ここで、構文の最終的な要素に現れるものを**終端記号(terminal symbol)**、それ以外のほかの構文規則によって定義される記号を**非終端記号(non-terminal symbol)**と呼ぶ。ここでは、非終端記号を<>で囲んで表すことにする。

1. 構文規則は，非終端記号<T>に対して、<T> := e (e は構文規則)で、表現される。これは、非終端記号<T>は、構文規則 e によって置き換えられることを意味する。
2. e は空でもよい。
3. e は、非終端記号、終端記号、もしくは e1 | e2、e1 e2、e1* のいずれか。e1 | e2 は、e1 もしくは e2 であることを意味し、e1 e2　は e1 の次に e2 が現れることを意味し、e1*は、e1 の 0 個以上の繰り返しを意味する。

(...)は、構文規則のまとまりを示す。e1 | e2 | e3 は、((e1 | e2) | e3)を、e1 e2 e3　は　((e1 e2) e3）と同じである。正規表現と似ていることを注意しよう。

ここでは、構文規則の左辺が、一つの終端記号<T>だけという文法を考える。これはすなわち、どのような場合でも<T> := e の規則を使って、右辺に置き換えることができることを意味し、このような文法を**文脈自由文法**とよぶ。この制限を取り払って、例えば、

```
e1 <T> e2 := ...
```

というような、e1 e2 の間に構文要素<T>が現れたときだけ、置き換えることができるというような文法が考えられるが、このような文法を**文脈依存文法**と呼ぶ。

構文規則に対し、構文解析を作る方法を紹介しよう。例えば、

```
<A> := a <B> c
```

という構文規則があったとすれば、<A>のための構文解析関数 readA は以下のように作ることができる。

```
readA() {
   a を読み込む；
   readB(); /* B を読み込むための関数を呼ぶ*/
   b を読み込む；
}
```

これは、これから読み込むものの形を先に仮定してしまってから、それに合致するかを調べていく構文解析法である。このような構文解析法を**再帰的下向き構文解析(recursive decent parsing)**あるいは**top-down parser** と呼ぶ。前回で解説した数式の構文解析もこの方法によるものである。

さて、前回の数式を再度考え、括弧や乗算をいれて考えてみることにする。構文規則を再度定義してみると、

```
<expression> := <expression> <expr_op> <term>
<term> := <term> <term_op> <factor>
<factor> := number | '(' <expression> ')'
<expr_op> := '+' | '-'
<term_op> := '*'
```

ここの定義で、加減算の優先順位を考慮して、生成される構文木を反映して構文規則が作られていることに注目。しかし、これを上の方法で書くと

```
readExpr() {
    readExpr();
    readExprOp();
    readTerm();
}
```

となってしまって、readExpr が再帰的に呼ばれて，うまく行かない。この問題を、**top-down parser の　左再帰性の問題**と呼ばれている。すなわち、最終的に

```
<T> := <T> e
```
となる、文法規則ではうまくいかないのである。このために、前回のプログラムでは、
```
readExpr() {
   readTerm();
   while(readExprOp() is OK)
       readTerm();
}
```
とした。これは、
```
<expresssion> := <term> <expression1>
<expression1> := <expr_op> <term> <expression1> | e
```
と書き換えたのと同等である。e は空を示す。

このほかに、
```
<T> := b ( c| e )
```
はうまく行かない。(c | e)は、c を読むか、それともなにもしないかという意味であるが、この場合は、<T>の次に何がくるかによって、c を読むかどうかが決まるので、top-down parser では、処理ができないことになってしまう。

Lisp とは

Lisp 言語の歴史は Fortran の次に古い言語で、1959 年に MIT の MacCarthy によりＡＩのための言語として、考案された。その特徴は

1. Ｓ式による簡単なシンタックス。
2. ラムダ論理による論理的な意味論
3. ＡＩのための記号処理ができる。リスト処理ができる。
4. インタープリタによる interative な処理
5. リストとしてプログラム自信をデータと扱えるメタな言語機能

特に 1980 年代には AI 言語として注目され、様々な言語に影響を与えた。この後、様々な処理系が開発されたが、現在標準的に使われているのは、common Lisp である。その他の例としては、有名な editor である emacs はその機能を emacs lisp で拡張できるようになっている。

Lisp インタープリタの製作

インタープリタの代表的な例題として、Lisp(リスプ)言語の簡単なインタープリタを作って、その仕組みを解説する。Lisp の場合は内部データ構造と外部表現が一致しているために、理解しやすい。

簡単版 Lisp の仕様

Lisp のプログラムは、S 式（Symbolic Expression）と呼ばれる構文で記述する。Ｓ式とは、以下のものである。

(関数名　式　式　式　...)

関数名は、ユーザが定義した関数でも、処理系であらかじめ定義されているものでもよい。Lisp では、この２つには区別はない。実行は引数の式を評価し、その値を引数として、指定された関数を呼び出す。式は、変数のシンボル、数字、またはＳ式である。式は、必ず値を持つものとする。

例えば、(+ (- x 2) 3)では、(- x 2)を計算し、その結果と 3 を引数として、関数＋を呼び出す。但し、これに従わない例外が若干ある。例えば、条件分岐を以下のような式で与えることにする。

(if　条件式　then-式　else-式)

この場合は、まず条件式を評価してから、その結果にしたがって、then または else の部の式を評価する。この if は通常の関数とは違う実行順序が必要である。また、変数の代入を次のような式で行うことにする。

(=　変数　式)

この式では、変数のほうは評価せずに、式を評価した値を代入する。（評価してしまうと、変数の値になってしまう）

本来の Lisp では、数値のほかにリストや記号、文字列などのデータが扱えるが、ここで作る Lisp では、以下のような制限を加えたものとする。

１．プログラムで扱うデータは整数のみ
２．条件分岐は、上に挙げた if の式とし、条件は 0 は false、それ以外の値は true として扱う。
３．変数の代入は上に挙げた=の式で行う。
４．システムで用意する関数は、四則演算と＞、＜などの大小比較のみとする。
５．リストは扱わないので、CG(ガーベージコレクション)はなし。

関数定義は以下の関数で行うことにする。

(define 関数名　(パラメータ１ パラメータ２ ....)　式)

例えば、

```
(define foo (x y) (+ (* x x) (* y y))
```

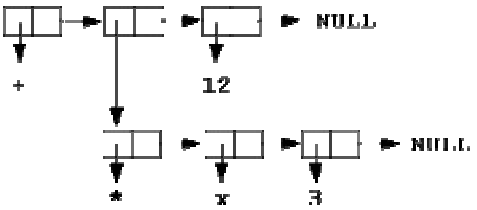
は、x と y を引数として、2 乗の和を返す関数 foo を定義する。

```
(foo 2 3)
```

では、まず、パラメータ x に2、パラメータ y に 3 をセットし、(+ (* x x) (* y y))を実行する。(* x x)は、４、(* y y)は９になって、この関数の値は 13 となる。この関数の実行後は、x と y のセットした値は元に戻ることに注意。

```
typedef struct object {
    int kind;    /* NUM, SYM, LIST */
    int val;     /* 数値の時, 値  */
    Symbol *sym; /*  シンボルの時、*/
     struct object *left,*right;  /*  リストの時*/
} Object;
```

(+ (* x 3) 12) の木構造



S 式のデータ構造と読み込み

まず、S 式は以下のデータ構造をつかって表現することにしよう。図は、リストのみの構造を示したものである。リストの場合は、kind が LIST になり、右と左にオブジェクトへのポインターを入れる。数字 12 やシンボル x は、NUM や SYM の kind を持つ Object になる。その場合の数字や要素は、val, sym のフィールドに格納する。リストの左は同じレベルの要素をリンクするのに用いて、リストの右はリストに入っている要素へのポインタが入る。右に例を示す。リストの終わりは NULL を格納しておく。

```
typedef struct symbol {
    char *name;    /*  名前の文字列   */
    int val;       /*  変数の場合の値   */
    Object *func;  /*  関数の場合の定義   */
} Symbol;
extern Symbol SymbolTable[];
```

シンボルテーブル

シンボルは、同じ名前のシンボルを１つのデータで管理するもので、以下のようなデータ構造である。これをつかって、表 Symbol SymbolTable[]で管理する。S 式のデータ構造の変数などのシンボルはこのデータ構造を通じて、同じ名前は同じデータ構造で管理される。関数 objectRead は、入力から S 式を入力し、データ構造に変換する。文字解析は lex でも記述できるが、lex.c に定義してある。ここに、これらのデータ構造を操作する関数(object.c)について説明しておく。

- Object *makeNum(int val): val の値を持つ NUM の Object を作って返す。
- Object *addList(Ojbect *left,Object *right): リスト LIST のオブジェクトを作って返す。
- Object *getNth(Object *p,int nth): リスト p の nth 番目のオブジェクトを返す。
- Object *getFirst(Object *p): リストの最初のオブジェクトを返す getNth(p,0)と同じ。
- Object *getNext(Object *p): リストの先頭の要素を除いたリストを返す。
- Symbol *lookupSymbol(char *name): シンボルテーブルから、同じ名前を持つシンボルを返す。なければ、シンボルテーブルに登録。
- Object *makeSymbol(char *name): 名前 name を持つシンボルのオブジェクトを返す。
- Symbol *getSymbol(Object *p): シンボルのオブジェクトから、シンボルを返す。

インタープリタ：変数の扱い

まずは、簡単なインタープリタを作ってみることにする。変数を考えなければ、大体は式の評価でつくったインタープリタと同じである。システムで定義してある四則演算に関しては、あらかじめ+,-,*,/などの記号に対してはシンボル定義をしておく(関数 initEval)。

１．まず、オブジェクトが数値 NUM であれば、その値を返す。
２．オブジェクトがリスト(f e1 e2)であれば、まず関数部分にあるシンボルを取り出し、システムで定義されたシンボルであるかを調べ、e1 と e2 を評価し、演算を行う。
３．さて、シンボルの場合は、変数と解釈する。変数の値は、シンボルテーブルのそれぞれの val のところにいれておくとすると、オブジェクト p がシンボル(kind==SYM)だったときには、p->sym->val を返せばよい。
４．変数の代入（＝ var e1）では e1 を評価して、その値を p->sym->val にいれておく。var のほうは評価しないことに注意。

これで、インタプリターの本体は、右のように read とインタプリタの実行 evalObject を繰り返すのが Lisp のインタープリタである。

```
main(){
        initEval();
      while(p = objectRead(0)){
              v = evalObject(p);
              printf("value is %d¥n",v);
        }
}
```

関数の定義と呼び出し

さて、関数の定義は define で行う。リストの先頭が define である場合には、引数と本体の部分のリストを定義される関数のシンボルの func のところにいれておくことにする。例えば、前の foo の例では、((x y) (+ (* x x) (* y y)) を func にいれておく。なお、Lisp では式を評価したときには何らかの値を返すが、便宜的に０を返しておくことにしよう。

関数呼び出しでは、関数の実行中は x と y の値を引数の値にしておかなくてはならない。ここで、xやyの val の部分にいれておくことも考えられるが、実行が終わると元の値にもどしておかなくてはならない。この意味でこれは単なる代入と異なり、このような操作を結合(bind)するという。このためのデータ構造として、結合した変数と値のペアを記録しておくデータ構造を用意する。どの変数がどのような値と結合されているかという状態のことを環境(environment)という。

```
typedef struct env   {
    Symbol  *var;
    int  val;
} Environment;
Environment Env[MAX_ENV];
```

Env は変数と値のペアの配列で、パラメータの変数に値が結合されるごとにこの配列に記録しておく。この配列をどこまで使っているかを示すために、envp という変数を使う。変数の値を探す時には、この表を最近に結合された順に探し、この表で見つかった場合にはその値を使い、ない時にはシンボルテーブルにある値を使えばよい。関数の実行が終わったら、envp の値を元に戻せば結合はなくなる。代入で、変数の値を変える場合もこの表にある場合には、その値を変更しなくてはならない。（関数 setValue, getValue）

関数呼び出し(foo e1 e2)の手順は、

1．シンボル foo を取り出し、func のところから関数定義本体を取り出す。
2．パラメータ部分(x y)を取り出し、e1 を評価し、シンボル x と結合する。次に、e2 を評価し、シンボル y と結合する。ここでは、Env にセットするだけで、envp は変えない。
3．引数の評価が終わったら、結合したところまで、envp を移動させる。これによって、x やyの値は env にある値になる。
4．本体部分（関数定義の２つめの要素）を取り出し、評価する。その値を返す。
5．返す前に、envp の値を元にもどし、結合を解く。

この動作を行うのが、関数 callFunc である。

関数が再帰的に呼び出される場合にも、最近の結合されたものから探す（つまり、Env を逆順に探す）ことにより、最も最近に結合された値を参照できることになる。さて、以上のことを統合すると、

1．まず、オブジェクトが数値 NUM であれば、その値を返す。
2．オブジェクトがリスト(f e1 e2)であれば、まず関数部分にあるシンボルを取り出し、システムで定義されたシンボルであるかを調べ、e1 と e2 を評価し、演算を行う。
3．オブジェクト が定義されたシンボルでなければ、関数定義であるので、callFunc を使って、関数呼び出しを行う。
4．シンボルの場合は、変数と解釈する。変数の値は、getValue を使って、結合されている値を返す。
5．変数の代入（＝　var e1）では e1 を評価して、setValue を使って、結合されている値を変更する。
6．もう一つ、(if cond e1 e2) を追加してみる。この場合には、cond を評価し、cond が 0 でなければ、e1 を評価し、それ以外であれば、e2 を評価して返す。

動的束縛

このような環境の作り方はCなどの言語とはちょっと異なる。例えば、

```
(define addx (y) (+ x y))
```

という関数があった場合、(= x 10) (addx 2) では、答えが　12 になる。xは関数の外側の x が参照される。しかし、

```
(define addxy (x y) (addx y))
(= x 10) (addxy 3 2)
```

とすると、5 が返される。つまり、後者の例では、x は addxy で結合された x が参照される。つまり、どのような順番で呼び出されるかに依存してしまう。このような実現の方法を動的結合(dynamic binding)と呼ぶ（動的束縛と呼ぶこともある）。C などの言語との違いを考えてみよ。

演習課題３：

解説したプログラムは、web 上に公開してある。これを使って、階乗 fac(10)を計算しなさい。if の部分については、意図的に削除してあるので追加すること。また、必要な組み込み関数があれば、プログラムに追加すること。階乗の計算はループで簡単に記述できるが、再帰呼び出しを使って代えることができる。提出は、

- 実行した Lisp の階乗を計算するプログラムと fac(10)の値
- 追加、変更したプログラム