[第４回目２００２・９・２４]

Lisp インタープリタの拡張

前回、簡単な Lisp インタープリタを代表的な例題として作った。今回は、これを本格的なプログラムが書けるように拡張してみよう。前回の Lisp では、以下の機能をつくった。

１．組み込みの四則演算子による整数の計算
２．変数が使える。変数の代入と参照。
３．関数定義ができる。
４．If 文により、条件分岐ができる（課題３）

さて、プログラミング言語として、これ以外に必要な機能は何であろうか。Cや Java など、近代的なプログラミング言語にはいろいろな機能がある。

- while 文や for 文、switch 文などの豊富な制御構文
- 局所変数とブロック（複文）
- 関数の途中から関数の値を返す return 文
- 豊富なデータ型と構造体
- 配列や文字列
- ポインター

このほかにもいろいろあるが、ここで、考えてみる機能は以下の通りである。

１．while 文
２．局所変数とブロック文
３．関数の値を返す return 文
４．文字列と配列
５．print 文

while 文：制御文

while 文に関しては、以下のような定義にしてみることにする。

(while 条件式　式)

これについては、インタプリターevalObject を使って、条件式を実行し、その値が真（すなわち０でない）になるまで、式を実行すればよい。これを追加する手順は、まず、while のシンボルを定義し、たとえば、whileSym としておく。evalObject 内ではじめが whileSym であったら、while 文を実行する whileFunc を残りのリストを引数にしてよびだす。

```
whileFunc(getNext(p));
```

条件式に当たる１番目の引数を実行し、これが真（すなわち０でない）の間、２番目の引数を実行する。このほかにも、for 文などの制御構造も、シンタックスを決め、その意味にしたがってどの部分を実行するかを制御すれば実装することができる。

```
int whileFunc(Object *args)
{
  while(evalObject(getNth(arg,0) != 0)
      evalObject(getNth(arg,1));
  return 0;
}
```

ブロック文と局所変数

局所変数のあるブロック文（式）については以下の定義としよう。

(block (局所変数１ 局所変数２ ...) 式１ 式２ 式３)

この文では、局所変数をつくり、ならぶ式を順番に実行することにする。Lisp では式は必ず値を返さなくてならないので、便宜的に最後の式の値を返すことにしよう。

式 1,2,3...を実行している間に局所変数が現れた場合には、宣言された局所変数を参照しなくてはならない。しかし、このブロック文が終わった時には、元の値に戻さなくてはならない。つまり、有効範囲（スコープ）を持つ。なお、ローカル変数として宣言されていない変数は、シンボルテーブルの中の値が参照される。これはいわば、大域変数といえる。

局所変数に対する処理は基本的には関数のパラメータ変数に対する式と同じである。block 文に現れた局所変数について、現在の環境に登録しておく。パラメータの場合には引数と結合しておいたが、局所変数に関しては何の値でもかまわない（つまり、未定義）。具体的には、

１．局所変数のある部分のリストを取り出す。
２．局所変数のリストから、局所変数のシンボルを取り出す。

```
int blockFunc(Object *args)
{
    Object *local_vars;
    int v;
    int envp_save;

    envp_save = envp;
    local_vars = getFirst(args);
    while(local_vars != NULL){
            Env[envp++].var = getSymbol(getFirst(local_vars));
            local_vars = getNext(local_vars);
    }
    args = getNext(args);
    while(args != NULL){
            v = evalObject(getFirst(args));
            args = getNext(args);
    }
    envp = envp_save;
    return v;
}
```

３．環境に登録する。Env[envp++] = 局所変数のシンボル
４．これを、局所変数の全部に繰り返す。
このあとで、式 1,2,...を実行する。この間で、変数が参照される場合には、関数 getValue/setValue で、値を参照するために Env にある変数の値が参照されることになる。
式が全部実行しおわったら、envp は元に戻しておく。これによって、局所変数は取り消されることになる。evalObject から呼び出す block 文に対応する部分について示す。

return 文

関数の途中から return できると便利な場合がある。たとえば、C言語では

```
foo(){
   if(...) return 100;
  ...
}
```

のように、途中で return 文を実行すると途中から、関数を終了し値を返すことができる。この機能を作ってみることにする。retrun 文は以下のように書くことする。

```
(return 式)
```

式の値を実行中の関数の値として返す。

インタープリターでは関数本体を実行するときには、evalObject で再帰的に呼び出しながら、実行している。関数の呼び出しの部分を思い出してみよう。関数 callFunc で、まず、引数とパラメータ変数を結合し、本体の式を evalObject で実行する。その中でさらに、evalObject が呼び出されて実行が進んでいく。その途中で、return 文が実行されたときには、最初の callFunc のところに戻ってこなくてはならない。

この動作を行うために setjmp/longjmp を使わなくてはならない。setjmp/longjmp は関数の現在の状態を記録しておき、呼び出された先から戻る機能である。例えば、

```
#iclude <setjmp.h>
jmp_buf env;
foo() { ...  setjmp(env); ... goo1(); ...}
goo1() { ... goo2(); ...}
goo2() {  ... longjmp(env,1);  }
```

この例では、foo の setjmp で env に状態を覚えておき、goo1, goo2 と呼び出されたときに、goo2 で longjmp をすることよって、setjmp の後に戻ってくる。setjmp では、はじめにセットしたときに 0 を、longjmp で戻ってきたときには longjmp で指定された値を返す。したがって、longjmp では第 2 引数に０以外の値を与える。

この機能を使って return 文を作ってみることにしよう。まず、戻るべき最も最近の状態 jmp_buf を覚えておくために、変数 funcReturnEnv を使う。

```
jmp_buf *funcRetrunEnv;
  ...
ret_env_save = funcReturnEnv;  /* 元の値をとっておく  */
funcReturnEnv = &ret_env;   /* 今度戻ってくるところにセット  */
if(setjmp(ret_env) != 0){ /* longjmp で戻ってきたとき */
    val = funcReturnVal;  /* return からの値をとる*/
} else {                  /* はじめにセットしたとき,本体を評価*/
    val = evalObject(getNth(func_def,1)); /*なにもなければその値  */
}
funcReturnEnv = ret_env_save;  /* 前の値に戻す*/
```

これで、return 文のほうは、

```
int callFunc(Symbol *f,Object *args)
{
      int  nargs;
      int  val;
      Object  *params,*func_def;
      Symbol  *param_var;
      jmp_buf  ret_env;
      jmp_buf  *ret_env_save;

      func_def = f->func;
      params = getNth(func_def,0);
      nargs = 0;
      while(params != NULL){
                param_var = getSymbol(getFirst(params));
                val = evalObject(getNth(args,nargs));
                bindEnv(nargs,param_var,val);
                nargs++;
                params = getNext(params);
       }
      ret_env_save = funcReturnEnv;
      funcReturnEnv = &ret_env;
      envp += nargs;
      if(setjmp(ret_env) != 0){
                val = funcReturnVal;
      } else {
                val = evalObject(getNth(func_def,1));
       }
      envp -= nargs;
      funcReturnEnv = ret_env_save;
      return val;
}

int returnFunc(Object *args)
{
      funcReturnVal = evalObject(getFirst(args));
       longjmp(*funcReturnEnv,1);
      fprintf(stderr,"longjmp failed???¥n");
       exit(1);
}
```

```
    funcReturnVal = evalObject(式)   /*  式を評価  */
  longjmp( *funcRetrunEnv,1);    /*  最近の setjmp にかえる！！！*/
```
とすることで、return の値を返すことができる。

文字列、配列

文字列や配列の機能がなければ面白くないので、付け加えてみることにする。

本来の Lisp では、実行時にデータ型をチェックしながら実行することができるが、このインタープリターでは、配列や文字列を扱う場合にはその配列や文字列のアドレスを数値として持つことにして作ってみることにした。

文字列に関しては、read されたところで、文字列を保存し、そのアドレスを数値 NUM として返すことにする。関数 println は、printf(フォーマット文字列、値)と同じ機能を持つ関数である。これを使えば、

（println "this is %d" x）

として、x の値を出力することができる。

配列も同じように、変数にアドレスを数値として持つことにする。例えば次のような仕様が考えられる。

- (array 配列名　配列サイズ)：配列を作る（宣言する）
- (getarray　配列の式　インデックス)：第 1 引数で与えられた配列のインデックス n 番目の要素を返す
- (setarray 配列の式　インデックス 値)：第 1 引数で与えられた配列のインデックス n 番目の要素に値をセットする。

なお、配列は、１次元配列のみである。

さて、以上でインタープリタは終わりである。これで、一通りのプログラムが書けるはずである。例えば、右のようなプログラムが書けるはずである。main を定義し、次に main を実行している。

次回からは、これをもとに C 風の言語を作っていくことにする。

```
(define main ()
  (block (s i)
        (= s 0)
        (= i 0)
        (while (< i 10)
              (block ()
                    (= n (+ n i))
                     (= i (+ i 1))))
    (println "sum is %d" n)))
(main)
```

演習課題４：

製作した Lisp 処理系を使って、8 クイーン問題のプログラムを書き、この問題を解きなさい。

- 8 クイーン問題とは、8×8 のチェス版に 8 個のクイーン（縦横斜めに移動できる）をおいて、お互いにぶつからない位置にクイーンを置く問題である。なお、問題は解が何個あるか求めるだけでよい（実際の位置を出力する必要はない）。
- この問題を解くために必要な機能があれば、適宜追加製作すること。

提出は、

- 実行した８クイーン問題の Lisp プログラムと８クイーン問題の解答（いくつ解があったか）
- インタプリターのプログラムとどのような機能を追加したかの説明