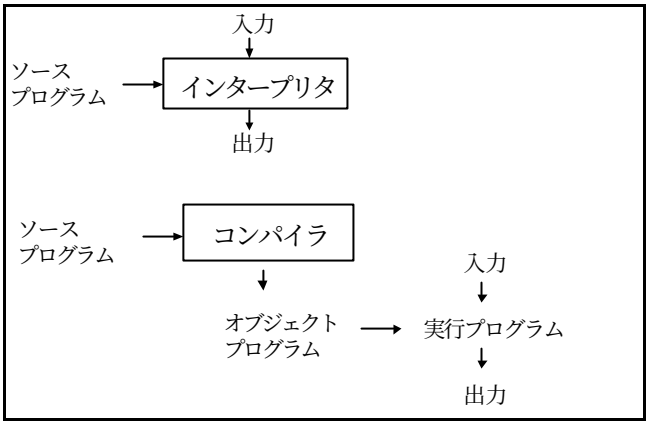
[1回目　２００２・９・３]

言語処理系とは

言語処理系とは、プログラミング言語で記述されたプログラムを計算機上で実行するためのソフトウエアである。そのための構成として、大別して２つの構成方法がある。

１、インタープリター（interpreter,翻訳系）：言語を意味を解析しながら、その意味する動作を実行する。

２、コンパイラ（compiler,通訳系）：言語を他の言語に変換し、その言語のプログラムを計算機上で実行させるもの。狭い意味でコンパイラは、言語を機械語に変換し、実行するものであるが、他の言語、あるいは仮想機械コードに変換するものもコンパイラと呼ぶ。他の言語に変換するときには、特に translator と呼ぶ場合もある。

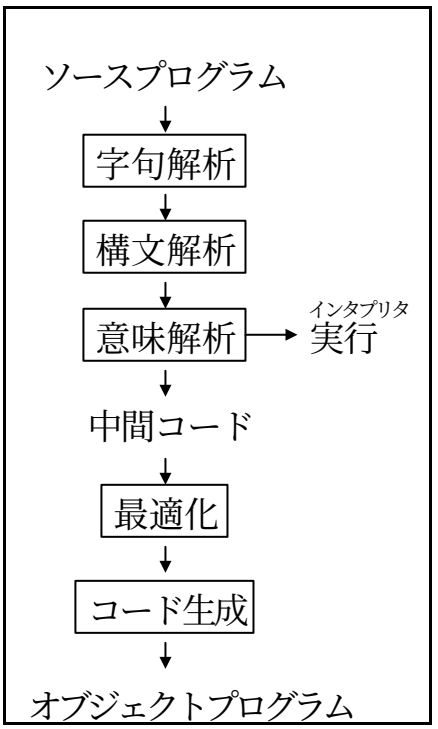


元のプログラムをソースプログラム、翻訳の結果と得られるプログラムをオブジェクトプログラムと呼ぶ。機械語で直接、計算機上で実行できるプログラムを実行プログラムと呼ぶ。オブジェクトプログラムがアセンブリプログラムの場合には、アセンブラにより機械語に翻訳されて、実行プログラムを得る。他の言語の場合には、オブジェクトプログラムの言語のコンパイラでコンパイルすることにより、実行プログラムが得られる。仮想マシンコードの場合には、オブジェクトコードはその仮想マシンにより、インタプリトされて実行される。

言語処理系の基本構成

コンパイラにしてもインタプリターにしても、その構成は多くの共通部分を持つ。すなわち、ソースプログラムの言語の意味を解釈する部分は共通である。インタプリターは、解釈した意味の動作をその場で実行するのに対し、コンパイラではその意味の動作を行うコードを出力する。

言語処理系は、大きく分けて、次のような部分からなる。



１、字句解析(lexical analysis): 文字列を言語の要素（トークン、token）の列に分解する。

２、構文解析(syntax analysis): token 列を意味を反映した構造に変換。この構造は、しばしば、木構造で表現されるので、抽象構文木（abstract syntax tree）と呼ばれる。ここまでの言語を認識する部分を言語の parser と呼ぶ。

３、意味解析(semantics analysis): 構文木の意味を解析する。インタプリターでは、ここで意味を解析し、それに対応した動作を行う。コンパイラでは、この段階で内部的なコード、中間コードに変換する。

４、最適化(code optimization): 中間コードを変形して、効率のよいプログラムに変換する。

５、コード生成(code generation): 内部コードをオブジェクトプログラムの言語に変換し、出力する。例えば、ここで、中間コードよりターゲットの計算機のアセンブリ言語に変換する。

コンパイラの性能とは、如何に効率のよいオブジェクトコードを出力できるかであり、最適化でどのような変換ができるかによる。インタープリタでは、プログラムを実行するたびに、字句解析、構文解析を行うために、実行速度はコンパイラの方が高速である。もちろん、機械語に翻訳するコンパイラの場合には直接機械語で実行されるために高速であるが、コンパイラでは中間コードでやるべき操作の全体を解析することができるため、高速化が可能である。

また、中間言語として、都合のよい中間コードを用いると、いろいろな言語から中間言語への変換プログラムを作ることで、それぞれの言語に対応したコンパイラを作ることができる。

例題：式の評価

さて、例として最も簡単な数式の評価について、インタプリターとコンパイラを作ってみることにする。目的は,

12 + 3 - 4

の式の入力に対し、この式を計算し、

11

と出力するプログラムを作ることである。これは、式という「プログラミング言語」を処理する言語処理系である。「式」という言語では、token として、数字と"+"や"-"といった演算子がある。

まずは、字句解析ではこれらのトークンを認識する。例えば、上の例では、

１２の数字、＋の演算子、３の数字、－の演算子、４の数字、終わり

という列に変換する。このプログラムが geToken.c である。

これをいわゆる構文解析しなくても、直接実行する（計算してしまう）インタプリターは簡単にできる。その動作は以下のような動作である。

1、現在の結果を変数 result に覚えておく。また、直前の演算子を変数 op に覚えておく。

2、関数 getToken を呼んで、数字であれば、現在の結果と今の数字の値との計算を行う。但し、最初の数字（まだ、op がない）の場合には、現在の結果に入力された数字を格納する。

3、終わりがきたら、現在の数字を出力する。

これが、いわゆる電卓のアルゴリズムである。（この電卓の欠点を考えてみよ！）

ＢＮＦと構文木

では、この「式」というプログラミング言語の構文とはどのようなものであろうか。例えば、次のような規則が構文である。

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

このような記述を、BNF (Backus Naur Form または Buckus Normal Form)という。

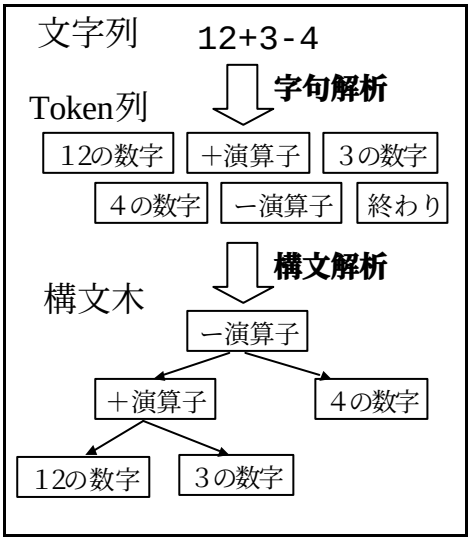
このような構造を反映するデータ構造を作るのが、構文解析である。図に示す。この構文木を作るプログラムが、readExpr.c である。

この構文木を解釈して実行する、すなわちインタプリターをつくってみることにする。その動作は、

1、式が数字であれば、その数字を返す。

2、式が演算子を持つ演算式であれば、左辺と右辺を解釈実行した結果を、演算子の演算を行い、その値を返す。

このプログラムが evalExpr.c である。このプログラムでは、ExprParser.h で定義されている Expr というデータ構造を使って、構文木を作っている。こ




```
/* exprParser.h */
#define EOL 0
#define NUM 1
#define  PLUS_OP   2
#define MINUS_OP 3

extern int tokenVal;
extern int currentToken;

void getToken(void);

typedef struct _expr {
      int kind;
      int val;
      struct _expr *left;
      struct _expr *right;
} Expr;

Expr *readExpr(void);
void printExpr(Expr *e);
```

```
#include <stdio.h>
#include <ctype.h>
#include "exprParser.h"

int tokenVal,currentToken;

void getToken()
{
      int c,n;
again:
      c = getc(stdin);
       switch(c){
      case '+':
                  currentToken = PLUS_OP;
                  return;
       case '-':
                  currentToken = MINUS_OP;
                  return;
       case '¥n':
                  currentToken = EOL;
                  return;
       }
      if(isspace(c)) goto again;
       if(isdigit(c)){
                  n = 0;
                  do {
                        n = n*10 + c - '0';
                         c = getc(stdin);
                  } while(isdigit(c));
                  ungetc(c,stdin);
                  tokenVal = n;
                  currentToken = NUM;
                  return;
       }
      fprintf(stderr,"bad char '%c¥n",c);
       exit(1);
}
```

```
/* cal.c */
#include <stdio.h>
#include <ctype.h>
#include "exprParser.h"

main()
{
      int t;
      int op;
      int result;

     op = NUM;
     result = 0;
      while(1){
                  getToken()
                  switch(currentToken){
                  case NUM:
                        switch(op){
                        case  NUM:
                                    result = tokenVal;
                                    break;
                         case  PLUS_OP:
                                    result = result + tokenVal;
                                    break;
                         case  MINUS_OP:
                                    result = result - tokenVal;
                                    break;
                         }
                         break;
                  case PLUS_OP:
                  case MINUS_OP:
                        op = t;
                         break;
                  case EOL:
                        printf("result = %d¥n",result);
                         exit(0);
                  }
      }
}
```

のデータ構造は式の場合は、演算子とその左辺の式と右辺の式を持つ。数字の場合はこれらを使わずに値のみを格納する。token を読むたびに、データ構造を作っている。

解釈実行：インタプリター

evalExpr.c は、これを解釈して、式の値を計算するプログラムである。構文木の構造にしたがって、解釈する。

1. 数字の Expr つまり、kind が NUM であれば、その値を返す。
2. 演算式であれば、左辺を評価した値と右辺を評価した値を kind に格納されている演算子にしたがって、計算を行う。

これらは再帰的に呼び出しが行われていることに注意しよう。

```
/* evalExpr.c */
#include <stdio.h>
#include "exprParser.h"

int evalExpr(Expr *e)
{
      switch(e->kind){
     case NUM:
            return e->val;
     case PLUS_OP:
            return evalExpr(e->left)+evalExpr(e->right);
     case MINUS_OP:
            return evalExpr(e->left)-evalExpr(e->right);
      default:
            fprintf(stderr,"evalExpr: bad expression¥n");
            exit(1);
     }
}
```

main プログラムでは、関数 readExpr を呼び、構文木を作り、それを関数 evalExpr で解釈実行して、その結果を出力する。これが、インタプリターである。先のプログラムと大きく違うのは、式の意味を表す構文木が内部に生成されていることである。この構文木の意味を解釈するのがインタプリターである。(readExpr では１つだけ先読みが必要であるので、getToken を呼び出している)

```
/* readExpr.c */
#include <stdio.h>
#include "exprParser.h"

Expr *readNum(void);

Expr *readExpr()
{
      int t;
      Expr *e,*ee;

      e = readNum();
     while(currentToken == PLUS_OP || currentToken == MINUS_OP){
            ee = (Expr *)malloc(sizeof(Expr));
            ee->kind = currentToken;
            getToken();
            ee->left = e;
            ee->right = readNum();
            e = ee;
      }
      return e;
}

Expr *readNum()
{
      Expr *e;
     if(currentToken == NUM){
            e = (Expr *)malloc(sizeof(Expr));
            e->kind = NUM;
            e->val = tokenVal;
            getToken();
            return e;
     } else {
            fprintf(stderr,"bad expression: NUM expected¥n");
            exit(1);
      }
}
```

```
/*   interpreter.c */
#include <stdio.h>
#include <ctype.h>
#include "exprParser.h"

int main()
{
      Expr *e;

       getToken();
      e = readExpr();
     if(currentToken != EOL){
            printf("error: EOL expected¥n");
            exit(1);
      }
     printExpr(e); printf("= %d¥n",evalExpr(e));
      exit(0);
}
```

コンパイラとは

次にコンパイラをつくってみる。コンパイラとは、解釈実行する代わりに、実行すべきコード列に変換するプログラムである。実行すべきコード列は、通常、アセンブリ言語（機械語）であるが、そのほかのコードでもよい。中間コードとして、スタックマシンのコードを仮定することにする。スタックマシンは以下のコードを持つことにする。

1、PUSH n ： 数字 n をスタックに push する。
2、ADD ： スタックの上２つの値を pop し、それらを加算した結果を push する。
3、SUB : スタックの上２つの値を pop し、減算を行い、push する。
4、PRINT: スタックの値を pop し、出力する。

```
PUSH 12
PUSH 3
ADD
PUSH 4
SUB
PRINT
```

コンパイラは、このスタックマシンのコードを使って、式を実行するコード列を作る。例えば、図で示した例の式 12+3-4 は右のようなコードになる。stackCode.h にコードとその列を格納する領域を定義してある。

```
/.* stackCode.h */.
#define PUSH 0
#define ADD 1
#define SUB 2
#define PRINT 3

#define MAX_CODE 100

typedef struct _code {
      int opcode;
      int operand;
} Code;

extern Code Codes[MAX_CODE];
extern int nCode;
```

その手順は、

1、式が数字であれば、その数字を push するコードを出す。
2、式が演算であれば、左辺と右辺をコンパイルし、それぞれの結果をスタックにつむコードを出す。その後、演算子に対応したスタックマシンのコードを出す。
3、式のコンパイルしたら、PRINT のコードを出しておく。

この中間コードを生成するのが、compileExpr.c である。構文木を入力して、再帰的に上のアルゴリズムを実行する。コードは Codes という配列に格納しておく。

コード生成では、ここではスタックマシンのコードを C に直して出力することにしよう。C で実行させるために、main にいれておくことにする。このプログラムが、codeGen.c である。

コンパイラの main プログラムであるが、readExpr まではインタープリタと同じである。標準出力に出力されるプログラムに適当に名前をつけ（たとえば、output.c）これを C コンパイラでコンパイルして実行すればよい。（assembler のファイルの場合は as コマンドでコンパイルする。）

```
/* compileExpr.h */
#include "exprParser.h"
#include "stackCode.h"

void compileExpr(Expr *e)
{
        switch(e->kind){
      case  NUM:
                Codes[nCode].opcode = PUSH;
                Codes[nCode].operand = e->val;
                break;
       case  PLUS_OP:
                compileExpr(e->left);
                compileExpr(e->right);
                Codes[nCode].opcode = ADD;
                break;
       case  MINUS_OP:
                compileExpr(e->left);
                compileExpr(e->right);
                Codes[nCode].opcode = SUB;
                break;
        }
        ++nCode;
```

```
/*   comopiler.c */
#include <stdio.h>
#include <ctype.h>
#include "exprParser.h"
#include "stackCode.h"

int main()
{
      Expr  *e;

       getToken();
      e  =  readExpr();
      if(currentToken  !=  EOL){
                printf("error: EOL expected¥n");
                exit(1);
       }
      nCode  =  0;
       compileExpr(e);
      Codes[nCode++].opcode  =  PRINT;
       codeGen();
       exit(0);
}
```

```
/* codeGen.c */
#include "stackCode.h"

Code Codes[MAX_CODE];
int nCode;

void codeGen()
{
       int  i;
       printf("int  stack[100]; ¥nmain(){ int sp = 0; ¥n");
      for(i  =  0;  i  <  nCode;  i++){
                switch(Codes[i].opcode){
                case PUSH:
                        printf("stack[sp++]=%d;¥n",Codes[i].operand);
                        break;
                case ADD:
                        printf("sp--; stack[sp-1] += stack[sp];¥n");
                        break;
                case SUB:
                        printf("sp--; stack[sp-1] -= stack[sp];¥n");
                        break;
                case PRINT:
                        printf("printf(¥"%%d¥",stack[--sp]);¥n");
                        break;
                }
        }
        printf("}¥n");
}
```

電卓のプログラムに比べて、構文木を作るなど、ずいぶん遠周りをしたようであるが、その理由は演算の優先度や、括弧の式など、通常の数学で使われる式を正しく処理するためである。例えば、

12*3 + 3*4

の場合には、掛け算を最初にして、それらを加算しなくてはならない。この処理を反映した構文木を作ることによって、正しく処理する「言語処理系」を作ることができるようになる。

演習問題１： 掛け算、割り算の優先度を入れたインタープリターを作りなさい。token の種類に*や/に対応した演算子が増えることになる。入力として、

```
12*3 + 3*4 - 10
```

をいれて、正しく実行できることを確認しなさい。

さらに、括弧をいれた式が正しく処理できるよう拡張せよ。token の種類に括弧に対応するものが増えることになる。入力として、

12*(3+13) – 10

をいれて、正しく実行できることを確認しなさい。できたプログラムを提出すること。

次回は字句解析、lex の使い方など。