プログラミング言語処理

第１回（平成１５年度９月２日）

言語処理系とは

筑波大学　佐藤三久

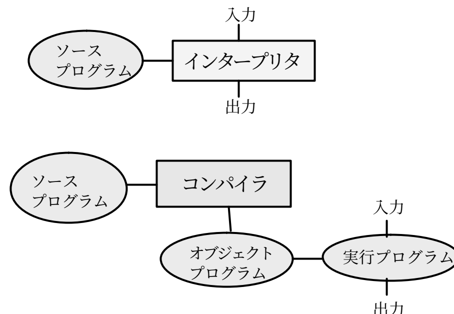
言語処理系とは

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

ソース、オブジェクト、実行プログラム

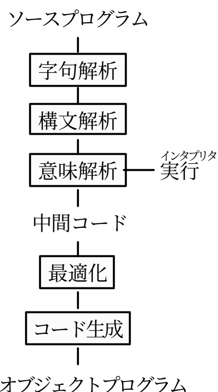
- ソースプログラム：元のプログラム
- オブジェクトプログラム：翻訳の結果と得られるプログラム
- 実行プログラム：機械語で直接、計算機上で実行できるプログラム
  - オブジェクトプログラムがアセンブリプログラムの場合には、アセンブラにより機械語に翻訳されて、実行プログラムを得る。
  - 他の言語の場合には、オブジェクトプログラムの言語のコンパイラでコンパイルすることにより、実行プログラムが得られる。
  - 仮想マシンコードの場合には、オブジェクトコードはその仮想マシンにより、インタプリトされて実行される。

言語処理系の流れ



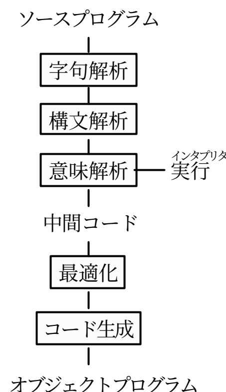
言語処理系の基本構成

- 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。
- 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。
- 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。



言語処理系の基本構成

- 意味解析(semantics analysis): 構文木の意味を解析する。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。
- 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。
- 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。

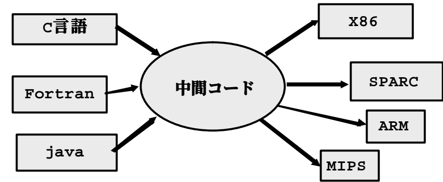


コンパイラとインタプリターの違い

- 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。
  - 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
  - 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

中間コードの役割

- 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。

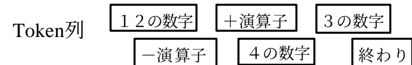
式の評価：字句解析

- 「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。

入力された文字列　’1’,’2’,’+’,’3’,’-’,’4’

⇩ 字句解析

Token列



Tokenの定義

- 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);
...
```

Tokenの種類

Tokenの種類とその値を
この２つの変数にいれて返す

１２の数字　currentToken=NUM
tokenVal=12

＋演算子　currentToken=PLUS_OP
tokenVal=?

字句解析プログラム

- getToken.c

```
#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);
}
```

字句解析だけを使うインタプリタ

- **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。**
  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。
- **電卓のアルゴリズム！**
  - 何がわるいか、欠点を考えてみよう。

字句解析だけを使うインタプリタ

- **cal.c**

```
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);
    }
  }
}
```


構文規則とBNF

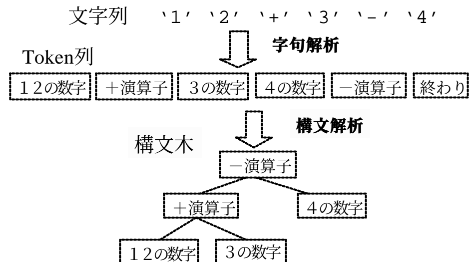
- **この「式」というプログラミング言語の構文とはどのようなものであろうか？**
- **構文規則**

  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)**
- **構文木 (AST: Abstract Syntax Tree)：上の構文を反映するデータ構造**

構文木

- **構文木 (AST: Abstract Syntax Tree)：構文規則を反映するデータ構造**



構文木のデータ構造

- **exprParser.h**

```
...
typedef struct _AST {
    int op;
    int val;
    struct _AST *left;
    struct _AST *right;
} AST;

AST *readExpr(void);
```

- どのような種類のノードなのかをいれる。NUM,PLUS_OPなど
- Op=NUMの場合にその値をいれておく
- 演算子（PLUS_OPなど）の場合の右の式と左の式
- 式を読み込み構文木を返す関数

メモリ節約のために、Unionにしてもよい。

式を読み込み構文木を作る関数

- **readExpr.h**

先読みをしていることに注意 getTokenを呼んでから readNumを呼び出す

```
AST *readExpr()
{
    int t;
    AST *e,*ee;
    e = readNum();
    while(currentToken == PLUS_OP ||
      currentToken == MINUS_OP){
       ee = (AST *)malloc(sizeof(AST));
       ee->op = currentToken;
       getToken();
       ee->left = e;
       ee->right = readNum();
       e = ee;
    }
    return e;
}
```

- 新しいASTを生成
- 前に読んだ式を左にする
- 数字を読む
- 新しいASTを現在のASTにセット

式を読み込み構文木を作る関数

◆ readExpr.h

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

新しいASTを生成

NUMのASTを作る

数字でなければ、エラー

インタプリタ：構文木の解釈

◆ 式のインタプリタ：構文木を解釈して実行する
（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

式のインタプリタのプログラム

◆ evalExpr.c

式を実行して、その値を返す関数

```
int evalExpr(AST *e)
{
    switch(e->op){
    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);
    }
}
```

OpがNUMであれば、その値を返す

演算子であれば、右のASTを評価した値と左を評価した値を演算して返す

式のインタプリタ（main)

◆ Interpreter.c

```
#include <stdio.h>
#include <ctype.h>
#include "exprParser.h"
int main()
{
    AST *e;
    getToken();
    e = readExpr();
    if(currentToken != EOL){
      printf("error: EOL expected¥n");
      exit(1);
    }
    printf("= %d¥n",evalExpr(e));
    exit(0);
}
```

先読みをすることをわすれずに！

式の読み込み

式の実行！！！
結果のプリントアウト

インタプリターの実行

◆ インタプリターのコンパイル

```
% cc -o interpreter  interpreter.c getToken.c
  readExpr.c evalExpr.c
```

◆ インタプリターの実行（標準入力から）
– input_fileに式を書いておく。
– 標準入力から、入力

```
% ./interpreter  < input_file
```

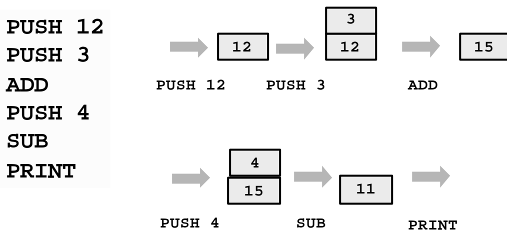
◆ 前のプログラムとの違いは式の意味を表す構文木が内部に生成されていること
◆ 構文木の意味を解釈するのがインタプリター

コンパイラとは

◆ 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
◆ 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、スタックマシンのコードを仮定することにする。
– PUSH n ： 数字nをスタックにpushする
– 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
– SUB : スタックの上２つの値をpopし、減算を行い、pushする
– PRINT: スタックの値をpopし、出力する

コンパイラによるコードの例

◆ 12+3-4 のスタックマシンへのコンパイル



コード生成の準備

◆ 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;
```

- スタックマシンのコードの定義
- コードのための構造体
- コードを格納するための領域
- コードの数

式のコンパイルの手順

◆ 式をスタックマシンのコードの列に変換し、それを格納する
- （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

式のコンパイルのプログラム

◆ compileExpr.c

```
void compileExpr(AST *e)
{
    switch(e->op){
    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;
}
```

- 構造はインタプリタによく似ている
- 実行する代わりにコードを生成
- NUMであれば、PUSHのコードを生成
- 左の式と右の式のコードを生成
- 演算に対するコードを生成
- 次のコードへ

コードの出力

◆ codeGen.h　スタックマシンのコードをC言語で出力

本当はアセンブラを生成

```
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;
        }
    }
```


式のコンパイラ（全体）

◆ compiler.c

```
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);
}
```

- readExprを呼ぶ前にTokenの先読みを忘れないように
- 式の読み込み
- コードのカウンターの初期化
- 式をコンパイル
- 最後に結果をプリントするコードを加える
- コードをC言語にして出力

コンパイラ実行の手順

- コンパイラのコンパイル

```
% cc –o compiler compiler.c getToken.c readExpr.c
  comileExpr.c codeGen.c
```

- コンパイラの実行
  - ソースをinput_fileに書いておく。
  - コンパイラで、コンパイルして、Cのプログラムを生成
  - Cのプログラムをコンパイル

```
% ./compiler < input_file > output.c
% cc output.c
```

- 実行プログラムの実行

```
% a.out
```


おわりに

- 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。

課題 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。<u>できたプログラムを提出すること。</u>