*プログラミング言語処理*

**第６回（平成１５年度１０月２１日）**

**スタックマシン**
**（コンパイラの準備）**

**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。

---



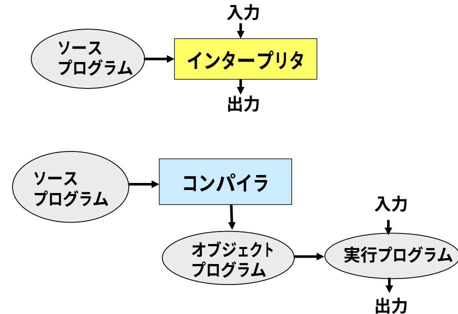
## ソース、オブジェクト、実行プログラム

- 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

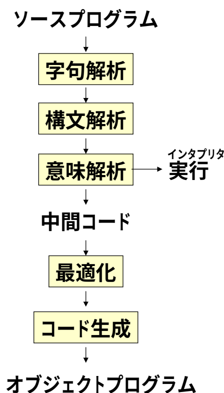



---



## 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。

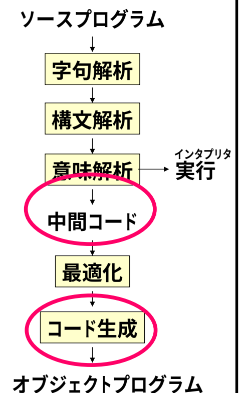



---



## 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



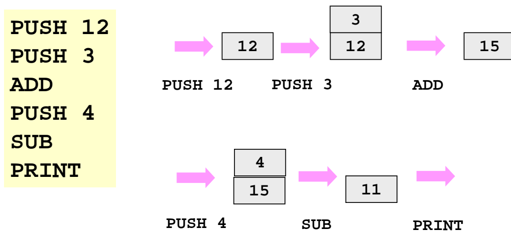
## コンパイラとは

- 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



## なぜスタックマシンか

- ◆ インタープリタでつくったtiny Cについて、コンパイラを作っていくことにする。
- ◆ 最終的には、マシンコードを直接出力するコンパイラを作るが、コード生成の考え方を簡単にするために、スタックマシンをターゲットにする。
  - スタックマシンではレジスタを扱わなくても良いため簡単になる。
  - 初回では単純な数式のコンパイルを考えたが、言語を実行するためにはインタプリタでやったように関数呼び出しやローカル変数をどのように作るかを考えなくてはならない。



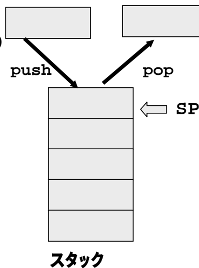
## スタックマシンのプログラム

- ◆ ここで考えるスタックマシンの「インタプリタ」のプログラムは、以下のプログラムである。
  - st_code.h : スタックマシンのコードの定義
  - st_machine.c : スタックマシンのインタプリタ
  - st_code.c : スタックマシン関連の関数



## スタックマシンとは

- ◆ スタック上で演算を行うように設計された（仮想）計算機アーキテクチャ
  - スタック（FILO: Fist In Last Out）
  - レジスタを扱わなくてもいいので、コンパイラが簡単になる。
  - 仮想計算機として、広く使われている。
    - Java VMなど
    - 実際のマシンも（昔）あった。
  - レジスタSP（スタックポインタ）がスタックの先頭を示す






## スタックマシンの命令

◆ tiny Cのターゲットとして考えるマシンの命令は、以下の２０個の命令である。

| | |
|---|---|
| POP | stackから、1つpopする。 |
| PUSHI n | 整数nをpushする。 |
| ADD | stackの上2つをpopして足し算し、結果をpushする。 |
| SUB | stackの上2つをpopして引き算し、結果をpushする。 |
| MUL | stackの上2つをpopして引き算し、結果をpushする。 |
| GT | stackの上2つをpopして比較し、>なら1、それ以外は0をpushする。 |
| LT | stackの上2つをpopして比較し、<なら1、それ以外は0をpushする。 |
| BEQ0 L | stackからpopして、0だったら,ラベルLに分岐する。 |

## スタックマシンの命令

| | |
|---|---|
| BEQ0 L | stackからpopして、0だったら,ラベルLに分岐する。 |
| LOADA n | n番目の引数をpushする。 |
| LOADL n | n番目の局所変数をpushする。 |
| STOREA n | stackのtopの値をn番目の引数に格納する。 |
| STOREL n | stackのtopの値をn番目の局所に格納する。 |
| JUMP L | ラベルLにジャンプする。 |
| CALL e | 関数エントリeを関数呼び出しをする。 |
| RET | stackのtopの値を返り値として、関数呼び出しから帰る。 |
| POPR n | n個の値をpopして、関数から帰った値をpushする。 |
| FRAME n | n個の局所変数領域を確保する。 |



## スタックマシンの命令

- tiny Cのターゲットとして考えるマシンの命令は、以下の２０個の命令である。

| | |
|---|---|
| CALL e | 関数エントリeを関数呼び出しをする。 |
| RET | stackのtopの値を返り値として、関数呼び出しから帰る。 |
| POPR n | n個の値をpopして、関数から帰った値をpushする。 |
| FRAME n | n個の局所変数領域を確保する。 |
| PRINTLN s | sのformatで、printlnを実行する。 |
| ENTRY e | 関数の入口を示す。(擬似命令) |
| LABEL L | ラベルLを示す。(擬似命令) |



## スタックマシンでの演算

- POPや、PUSHI, 演算ADD,SUBなどは、最初の講義で解説した通り、スタックに値をセットしたり、演算したりする命令である。
- 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。
- その手順は、
  - 式が数字であれば、その数字をpushするコードを出す。
  - 式は変数であれば、その値をpushするコードをだす。
  - 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。



## 式のコンパイル

- st_compile_expr.c

```
void  compileExpr(AST *p)
{
    if(p == NULL) return;
    switch(p->op){
    case NUM:
      genCodeI(PUSHI,p->val);
      return;
    case SYM:
      compileLoadVar(getSymbol(p));
      return;
    case EQ_OP:
      compileStoreVar(getSymbol(p->left),p->right);
      return;
    case PLUS_OP:
      compileExpr(p->left);
      compileExpr(p->right);
```

定数の場合には定数をスタックにつむコードを生成する



## 式のコンパイル

- st_compile_expr.c

```
    case PLUS_OP:
      compileExpr(p->left);
      compileExpr(p->right);
      genCode(ADD);
      return;
    case MINUS_OP:
      compileExpr(p->left);
      compileExpr(p->right);
      genCode(SUB);
      return;
    case MUL_OP:
      compileExpr(p->left);
      compileExpr(p->right);
      genCode(MUL);
      return;
    case LT_OP:
```

左の式をコンパイルして、実行すると左の式がスタックに残るコードを生成

同じく右も。。。

スタック上の2つの値を加算する命令を生成

2項演算に関してはおなじようなコードを生成する



## 式のコンパイル

```
  case LT_OP:
      compileExpr(p->left);
      compileExpr(p->right);
      genCode(LT);
      return;
    case GT_OP:
      compileExpr(p->left);
      compileExpr(p->right);
      genCode(GT);
      return;

    case CALL_OP:
      compileCallFunc(getSymbol(p->left),p->right);
      return;

    case PRINTLN_OP:
      printFunc(p->left);
```

*プログラミング言語処理*

## 制御文のコード

- JUMP命令は、LABEL文で示されたところに制御を移す命令である。
- このスタックマシンは分岐命令は、BEQ0命令しかない。この命令は、スタック上の値を popして、これが0だったら、分岐する命令である。
- これを組みあわせてIF文をコンパイルする。

```
...条件文のコード...
BEQ0 L0    /* もし、条件文が実行されて、結果が0だったら，L0に分岐*/
...thenの部分のコード...
JUMP L1
LABEL L0
... elseの部分のコード...
LABEL L1
```

*プログラミング言語処理*

## IF文のコンパイルの手順

1. 条件式の部分のコンパイルする。これが実行されるスタック上には、条件式の結果が積まれているはずである。
2. ラベルL0を作って、BEQ L0を生成。
3. then部分の式をコンパイルする。
4. これが終わるとIF文を終わるため、ラベルL1を作って、ここにJUMPする命令を生成する。
5. 条件文が0だったときに実行するコードを生成する前に、LABEL L0を生成する。
6. else部の式をコンパイル。
7. then部の実行が終わったときに飛ぶ先L1をここにおいておく。

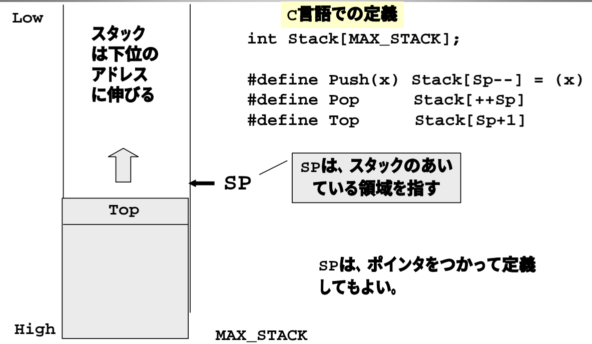
*プログラミング言語処理*

## 関数呼び出しの構造

- スタックマシンは以下の３つのレジスタを持つ。
  - SP : スタックポインタ。スタックのtop（の上）を指しているレジスタ。
  - FP : 実行中の関数の情報を保存しているところを指すレジスタ。ここからの相対で、引数や局所変数にアクセスする。
  - PC : プログラムカウンタ。現在実行している命令のアドレスを持つ。

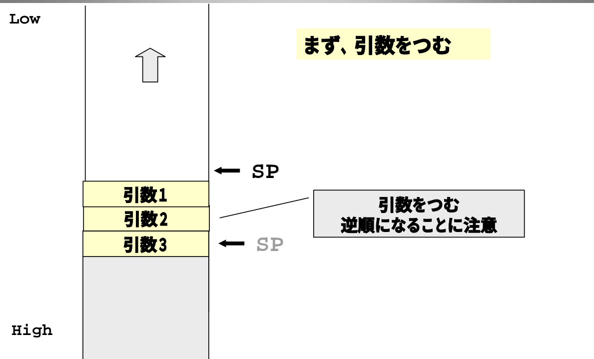
*プログラミング言語処理*

## 関数呼び出しの構造



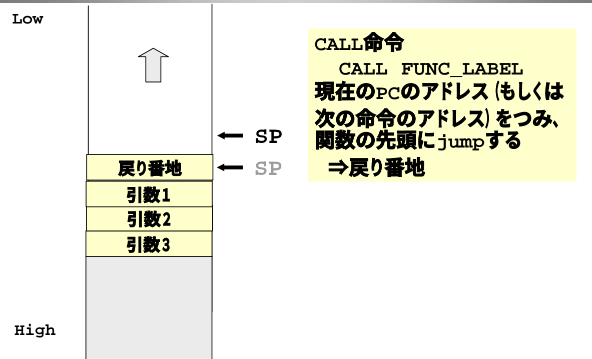

*プログラミング言語処理*

## 関数呼び出しの構造

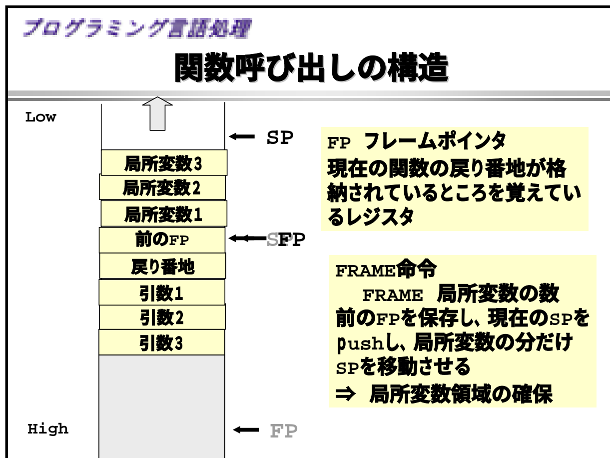



*プログラミング言語処理*

## 関数呼び出しの構造

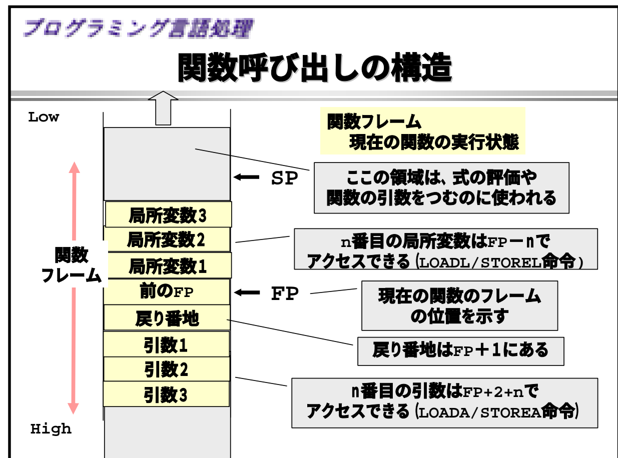


## 関数呼び出しの構造






## 関数呼び出しの構造






## 関数呼び出しの手順

- スタック上に引数を積む。
- 現在のPCの次のアドレスをスタック上に保存(push)し、関数の先頭のアドレスにjumpする。（CALL命令）
- 現在のFPをスタック上に保存し(push)し、ここを新たなFPとする。FPから、上の部分を局所変数の領域を確保し、ここを新たなスタックの先頭にする。(FRAME命令)
- 式の評価のためのstackはここから始まる。

- 引数にアクセスするためには、FPから２つ離れたところにあるので、ここからとればよい。(LOADA/STOREA命令)
- 局所変数にアクセスするためには、FPの上にあるので、ＦＰを基準にしてアクセスする。(LOADL/STOREL命令)



## 関数戻りの手順

- 関数から帰る場合には、stackに積まれている値を戻り値にする。

- 元の関数に戻るためには、FPのところにSPを戻して、まず、前のFPを戻して、次に戻りアドレスを取り出して、そこにjumpすればよい。(RET命令)
- 戻ったら、引数の部分をpopして、関数の戻り値をpushしておく。（POPR 命令）



## 関数コードと関数呼び出しの手順

- 関数の定義と関数呼び出しは以下のコードになる。

```
引数1のpush ...
引数2のpush ...
  ....
CALL foo
POPR  pushした引数の個数
  ...
```

呼び出し側(caller)

```
ENTRY foo
FRAME ローカル変数の個数
  ....
  関数本体のコード
  ....
RET
```

呼ばれる側(callee)



## 関数フレームとリンク規則

- 関数フレーム
  - 関数呼び出しごとに、戻り番地、局所変数などの情報を保持しているデータ構造

- 呼び出し側と呼ばれる側の手順を合わせておかなくてはならない。この手順を*数のリンク規則* *(linkage convention あるいはcalling sequence)* とよび、各マシンごとに定められている。

## 関数のコンパイル

- 関数のコンパイルは、以下のようになる。
  1. まず関数の名前を取り出して、ENTRY funcを生成する。
  2. パラメータ変数に番号をつける。関数が呼ばれた場合にはこの順番でスタックに積まれていることになる。これをEnvをいれておく。
  3. 関数の本体をコンパイルする。
  4. 実行されると関数の本体の値がスタックに積まれているはずなので、ここでRET命令を生成する。

- パラメータの変数や局所変数は、スタック上にその領域が確保されるが、どこに確保されるかを数えておかなくてはならない。

プログラミング言語処理

## 次回

- スタックマシンへのコンパイラについて説明する