プログラミング言語処理

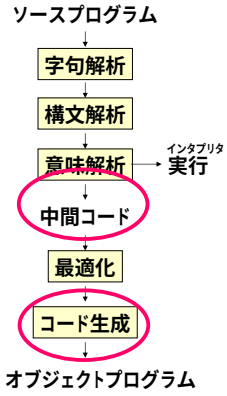
第７回（平成１５年度１０月２８日）

スタックマシンへのコンパイラ

筑波大学　佐藤三久

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。



なぜスタックマシンか

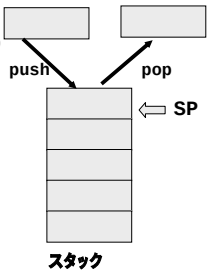
- インタープリタでつくったtiny Cについて、コンパイラを作っていくことにする。
- 最終的には、マシンコードを直接出力するコンパイラを作るが、コード生成の考え方を簡単にするために、スタックマシンをターゲットにする。
  - スタックマシンではレジスタを扱わなくても良いため簡単になる。
  - 初回では単純な数式のコンパイルを考えたが、言語を実行するためにはインタプリタでやったように関数呼び出しやローカル変数をどのように作るかを考えなくてはならない。

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)

スタックマシンへのコンパイラ

- 前回は、コンパイル対象となるスタックマシンについて説明した。今回は、スタックマシンへのtiny Cコンパイラについて解説する。プログラムは、以下のものである。
  - st_compile.h : スタックマシンのコンパイラのheader
  - st_code.h : スタックマシンのコードの定義
  - compiler_main.c : コンパイラのmain
  - st_compile.c : スタックマシンのコンパイラの関数、文の処理
  - st_compile_expr.c : スタックマシンのコンパイラの式の処理
  - st_code_gen.c : スタックマシンのコード生成
  - st_code.c : スタックマシン関連の関数
- 構文解析や字句解析部分は、インタープリタと共通
  - DefineFunc,…などのparserから呼び出される関数が異なる。

コンパイラのmainプログラム

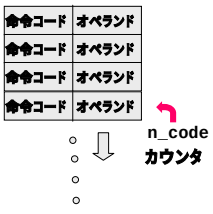
- コンパイラでは、最初に構文解析を呼び出し、構文解析ルーチンのの中で、入力された外部定義ごとにdefineFunctionやdeclareVariableが呼び出される。この関数がASTを入力してコンパイルを行う。
- したがって、コンパイラのmain プログラムは単に、yyparseを呼び出すのみである。

```
main()
{
    yyparse();
    return 0;
}
```


コードの生成ルーチン

- コンパイラでは、通常、一つ一つの関数ごとにコンパイルしていく。
- コンパイラでは、このコードをメモリに格納しておき、関数のコンパイルが終るごとに出力する
- スタックマシンの説明で述べたとおり、命令は命令コードとオペランドからなる。
- コードを格納する領域の定義は右のようになる。

```
struct _code {
    int opcode;
    int operand;
} Codes[MAX_CODE];
```



コードの生成ルーチン

```
void initGenCode()
{
    n_code = 0;
}
void genCode(int opcode)
{
    Codes[n_code++].opcode = opcode;
}
void genCodeI(int opcode, int i)
{
    Codes[n_code].opcode = opcode;
    Codes[n_code++].operand = i;
}
void genCodeS(int opcode,char *s)
{
    Codes[n_code].opcode = opcode;
    Codes[n_code++].operand = (int)s;
}
```

initGenCodeはそれぞれの関数のコンパイルの前に呼び出し、コード領域をクリア

opcodeのみの命令用

オペランドありの命令用

文字列のオペランドを持つ命令用(PRINTLN)、強制的にcastしている。

プログラミング言語処理

コードの生成ルーチン

- 関数がコンパイルが終ったら、genFuncCodeでコードを出力する。

```
void genFuncCode(char *entry_name, int n_local);
```

- なぜ、コードをためておくのか？
  - コンパイルしてみないとローカル変数が何個あるか(n_local)がわからない
  - あとで見直して、最適化できる
- これについては、関数のコンパイルで説明する。

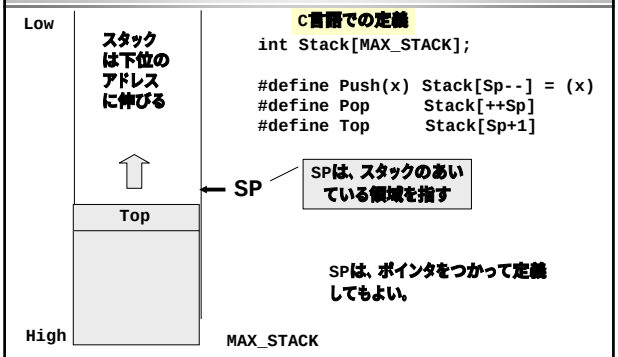
プログラミング言語処理

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。

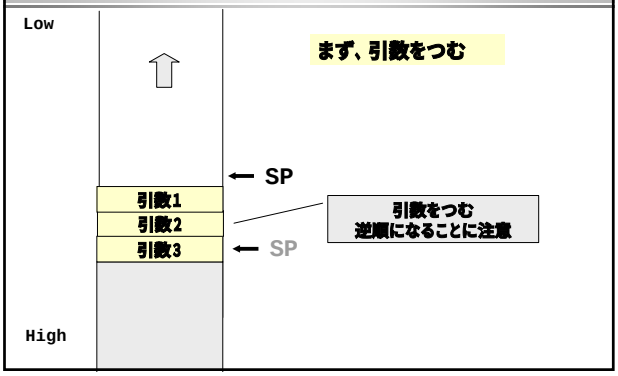
プログラミング言語処理

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

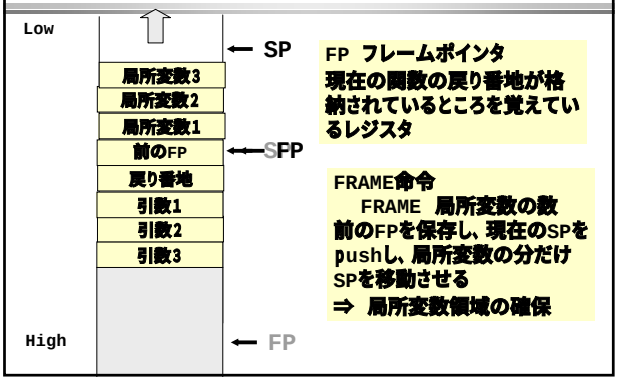


プログラミング言語処理

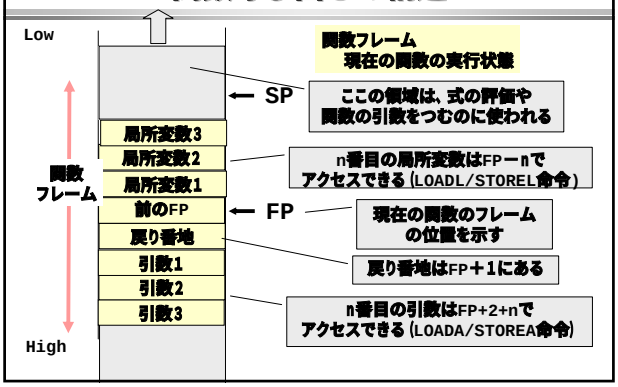
関数呼び出しの構造

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*関数のリンク規則* *(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教えておかなくてはならない。

プログラミング言語処理

コンパイラのための環境

- 関数のコンパイルするためには引数や局所変数の位置を決めなくてはならない。
  - 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。
- この変数がどこに割り当てられているかを覚えておくために、インタプリタで使った環境Envと同じようなデータ構造をつかう。
  - コンパイラでは、Envでコンパイルしているときにどの変数がスタック上のどこに割り当てられているかを覚えておく。
  - パラメータについては、パラメータの何番目かについて、Envに登録しておく。

```
#define VAR_ARG 0
#define VAR_LOCAL 1

typedef struct env {
    Symbol *var;
    int var_kind;
    int pos;
} Environment;
```

変数名

引数VAR_ARGか局所変数VAR_LOCALか

関数フレーム上の位置

プログラミング言語処理

関数のコンパイル

- yyparseの中で、関数定義が入力されるとdefineFunctionが呼び出される。

```
void defineFunction(Symbol *fsym,AST *params,AST *body)
{
    int param_pos;
    initGenCode();
    envp = 0;
    param_pos = 0;
    local_var_pos = 0;
    for( ;params != NULL; params = getNext(params)){
       Env[envp].var = getSymbol(getFirst(params));
       Env[envp].var_kind = VAR_ARG;
       Env[envp].pos = param_pos++;
       envp++;
    }
    compileStatement(body);
    genFuncCode(fsym->name,local_var_pos);
    envp = 0;  /* reset */
}
```

コード生成の初期化

パラメータのカウンタを0に

ローカル変数のカウンタを0に

パラメータをひとつつづ取り出して、環境にセットしていく

関数本体の文のコンパイル

これがおわるとCodeには出力されたコードははいっている

コードの出力

local_var_posは局所変数の数になっている

プログラミング言語処理

関数のコンパイル

- genFuncCode
  - コードをためておく理由は、全体をコンパイルしてみないと局所変数の数がわからないため

```
void genFuncCode(char *entry_name, int n_local)
{
    int i;
    printf("ENTRY %s¥n",entry_name);
    printf("FRAME %d¥n",n_local);
    for(i = 0; i < n_code; i++){
        printf("%s ",st_code_name(Codes[i].opcode));
        switch(Codes[i].opcode){
        case PUSHI:
        case LOADA:
        /* 省略 */
    }
    printf("RET¥n");
}
```

ENTRYの出力

FRMAEの出力

あとは、全部のコードを順番に出力

最後に、RETを出力

プログラミング言語処理

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。

プログラミング言語処理

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

変数の値をスタックにつむコードを生成する関数

変数への代入のコードを生成する関数

プログラミング言語処理

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);
```


変数のロード

- 変数はパラメータや局所変数があるについては、上に述べたようにEnv に記録されている。
  - Envを探し、それが引数であれば、 LOADAを生成する。
  - 局所変数であれば、LOADLを出力することになる。

```
void compileLoadVar(Symbol *var)
{
    int i;
    for(i = envp-1; i >= 0; i--){
       if(Env[i].var == var){
            switch(Env[i].var_kind){
            case VAR_ARG:
               genCodeI(LOADA,Env[i].pos);
               return;
            case VAR_LOCAL:
               genCodeI(LOADL,Env[i].pos);
               return;
            }}}
    error("undefined variable¥n");
}
```

新しいものから探す

変数が、見つかったらvar_kindをみる

パラメータ変数には、LOADAを生成

局所変数には、LOADAを生成

みつからなかったらエラー

変数のストア

- 右辺の式をコンパイルし、右辺式の結果をスタック上に計算するコードを生成し、つぎに、STOREAまたはSTORELを生成する。

```
void compileStoreVar(Symbol *var,AST *v)
{
    int i;
    compileExpr(v);
    for(i = envp-1; i >= 0; i--){
       if(Env[i].var == var){
            switch(Env[i].var_kind){
            case VAR_ARG:
               genCodeI(STOREA,Env[i].pos);
               return;
            case VAR_LOCAL:
               genCodeI(STOREL,Env[i].pos);
               return;
            }
       }
```

右辺式をコンパイル

STOREA, STORELを出す以外は、compileLoadVarと同じ

文のコンパイル

```
void compileStatement(AST *p)
{
    if(p == NULL) return;
    switch(p->op){
    case BLOCK_STATEMENT:
       compileBlock(p->left,p->right);
       break;
    case RETURN_STATEMENT:
       /*
        * 省略
        */
    default:
       compileExpr(p);
       genCode(POP);
    }
}
```

それぞれの文の処理の関数を呼び出す

式が文になる場合

式をコンパイル

ただし、式をコンパイルするとその値はスタックにつまれているので、POPして戻しておく

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、L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
```


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部分の文をコンパイルする。
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部の文をコンパイル。
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。

IF文のコンパイル

```
void compileIf(AST *cond, AST *then_part, AST *else_part)
{
    int l1,l2;

    compileExpr(cond);
    l1 = label_counter++;
    genCodeI(BEQ0,l1);
    compileStatement(then_part);
    l2 = label_counter++;
    if(else_part != NULL){
      genCodeI(JUMP,l2);
      genCodeI(LABEL,l1);
      compileStatement(else_part);
      genCodeI(LABEL,l2);
    } else {
      genCodeI(LABEL,l1);
    }
}
```

条件式のコンパイル
BEQ0の生成
then部の文のコンパイル
else部がある場合
else部のコンパイル

関数呼び出しのコンパイル

- 関数呼び出しのコンパイル（compileFuncCall）は、引数をスタックに積んで、CALL命令を出す。
  - 引数をスタックに積むのは、式の実行が終わるとスタック上につまれるはずなので、単に引数をコンパイルすればよい。
  - その後に、 CALL命令を生成し、
  - 引数をスタックからpopして、結果をpushする命令POPR命令を生成しておく。

関数呼び出しのコンパイル

```
void compileCallFunc(Symbol *f,AST *args)
{
    int nargs;
    nargs = compileArgs(args);
    genCodeS(CALL,f->name);
    genCodeI(POPR,nargs);
}

int compileArgs(AST *args)
{
    int nargs;
    if(args != NULL){
      nargs = compileArgs(getNext(args));
      compileExpr(getFirst(args));
      return nargs+1;
    } else return 0;
}
```

引数のコンパイル
CALL命令の生成
POPR命令の生成
関数の引数を逆順に評価している
同時に引数の数を数えていることに注意

局所変数のコンパイル

- Block文の処理、局所変数をコンパイルする

```
void compileBlock(AST *local_vars,AST *statements)
{
    int v;
    int envp_save;

    envp_save = envp;
    for( ; local_vars != NULL;local_vars = getNext(local_vars)){
       Env[envp].var = getSymbol(getFirst(local_vars));
       Env[envp].var_kind = VAR_LOCAL;
       Env[envp].pos = local_var_pos++;
       envp++;
    }
    for( ; statements != NULL; statements = getNext(statements))
       compileStatement(getFirst(statements));
    envp = envp_save;
}
```

局所変数を一つつづ取り出し、環境にセットしていく VAR_LOCALにする
local_var_posで数えている
文を順番にコンパイル
環境をもとにもどしておく

return文のコンパイル

- RET命令は、スタックのtopの値を返す
- return文のコンパイルはスタック上に式の値を残し、RET命令を生成すればよい。

```
void compileReturn(AST *expr)
{
    compileExpr(expr);
    genCode(RET);
}
```

なお、式exprがNULLの場合は結果はなにも残らないので、
RETが実行されるとその時点でのtopの値が返されるが、値は不定である。

While文、For文

- 考えてみてください。

プログラミング言語処理

変数と配列の宣言

- 変数と配列宣言についても、省略してある。
- インタプリタと同様に、変数と配列宣言が入力されると、declareVariableと declareArrayがyyparseから呼び出される。
- 前回説明したスタックマシン st_machineには、大域変数を扱う機能がない。これを扱うためにはどのようなコードが必要なのかについて、考えてみよ。

プログラミング言語処理

コンパイラとスタックマシンの実行

- さて、説明したコードをコンパイルして tiny-cc-stを作る。 tiny-cc-stは、標準入力から呼んで、コンパイルの結果のコードを標準出力に出力するようになっている。
  - 例えば、プログラムfoo.cをコンパイルして、コードfoo.iを作るには、

```
% tiny_cc_st < foo.c > foo.i
```

  - st_machineもコードは標準入力から読むようになっているので、

```
% st_machine < foo.i
```

  - 連続して動かす場合には

```
% tiny_cc < foo.c | st_machine
```

プログラミング言語処理

次回

- レジスタマシンへのコンパイラについて説明する
  - 機械語序論のx86コードを参考にすること