プログラミング言語処理

第８回（平成１５年度１１月１４日）

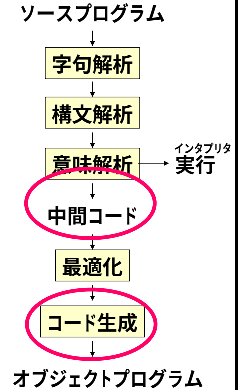
レジスタマシンへのコンパイラ

筑波大学　佐藤三久

プログラミング言語処理

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。



プログラミング言語処理

レジスタマシンへのコンパイラ

- 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。
  - 演算はスタック上で行われる。
- レジスタマシンでは、演算はレジスタ上で行わなくてはならない
  - レジスタの数は限られている（x86では、６個）
  - 変数をレジスタに割り当てる（レジスタ割り当て）
  - 足りない場合にはスタック上に退避しなくてはならない。

プログラミング言語処理

レジスタマシンへのコンパイラ

- 今回は、実際のマシン、x86(Pentium)へコンパイルすることにする。
- スタックマシンではコンパイラが作り安いマシンであるが、実際のマシンではレジスタがあり、これらを使ったコードを生成しなくてはならない。
- 説明するプログラムは以下のものである。
  - reg_compile.h : レジスタマシンへのコンパイラのheader
  - reg_code.h : レジスタマシン用の中間コードの定義
  - compiler_main.c : コンパイラのmain
  - reg_compile.c : レジスタマシンへのコンパイラの関数、文の処理
  - reg_compile_expr.c : レジスタマシンへのコンパイラの式の処理
  - x86_code_gen.c : x86用のコード生成

プログラミング言語処理

IA32命令セット：x86(Pentium)プロセッサ

- 演習室に導入されているプロセッサであるx86のIA32命令セットについては機械語序論において詳しく解説した。
- ここではコンパイラを作成するのに必要な簡単な命令について説明する。
  - なお、命令の記述形式にはAT&T形式とIntel形式があり、Linuxのアセンブラでは AT&Tを使っているので、これで説明する。この AT&T形式では、書き換えられるdestinationを後に書く。

プログラミング言語処理

IA32命令セット

- レジスタの構成
  - プロセッサには、
    - 整数レジスタが %eax,%ebx,%ecx,%edx,%edi,%esiの６個、
    - 浮動小数点レジスタが８個
      - tiny cでは、整数レジスタのみをつかう。
    - プログラムカウンタ %pc,
    - スタックポインタ%esp
    - フレームポインタ（ベースポインタ）%ebp
- オペランドの記述
  - レジスタの時には%をつけて、%レジスタ名と記述する。
  - メモリ参照は、offset(%レジスタ)と記述する。
  - 即値は$nと$をつけて記述する。

IA32命令セット

◆ 命令セット

ロード命令	movl offset(reg),dst	reg+offsetのメモリの 1word(32bit) の内容を、dstのレジスタに格納する。
ストア命令	movl src, offset(reg)	reg+offsetへ、srcの1wordの内容 を格納する。
即値ロード命令	movl $int, dst	intの数値を、dstにセットする。
レジスタ間移動命令	movl dst,src	srcのレジスタの内容をdstにコピーする。

演算命令	addl src1,dst2
	subl src1,dst2
	mull src1,dst2
		src1,dst2のレジスタの内容を加算し、dst2にセットする。減算命 令sublは、addlと同じである。乗算命令mullであるが、dst2にはeaxしか使え ない。32ビットの乗算ではedxに上位32ビット、eaxには下位32ビットがセット される。

比較演算命令	cmpl src2,src1	src1からsrc2を減算し、condition code をセット

IA32命令セット

◆ 命令セット

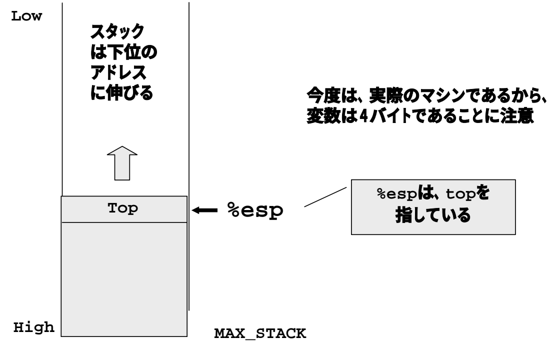
比較演算命令	cmpl src2,src1	src1からsrc2を減算し、condition code をセットする。src2はレジスタでなくてはならない。
条件分岐命令	je label jl label jg label	jeは、上の比較演算命令のcondition codeをみて、等 しい場合にlabelに分岐する。このほかにsrc1よりもsrc2が小さい場合に分岐 するjl、大きい場合に分岐するjg命令がある。
分岐命令	jmp label	labelに分岐する。
push命令	pushl src	srcをスタックにpushする。なお、spはスタックの先頭の要素をさしている。
関数呼び出し命令	call label	戻り番地のアドレス（次の命令のアドレス）をpushし、labelに分岐する。
leave命令	leave	これは、ebpさしているアドレスにespをセットし、pop した内容をebpにセットする。
リターン命令	ret	スタックからpopしたアドレスに分岐する。

関数の呼び出し規則

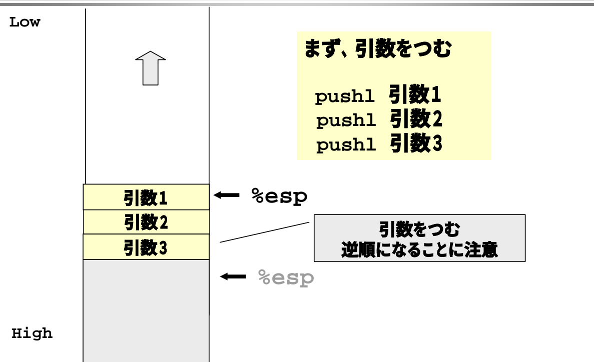
- 実際のマシンでは呼び出し規則は決められており、命令を組み合わせて行わなくてはならない
- 呼び出し側では、スタック上に引数をpushし、call命令を用いる。
  - ラベルfooにjumpした時には、スタック上に戻り番地がpushされる。
- 関 数呼び出しが終わって、戻ってきたときには、スタックポインタを元に戻して おかなくてはならない。
  - 従って、pushした引数個数分だけ、%espを加算して戻 す。なお、関数のもどり値は、eaxに入れることになっている。

```
pushl 引数2
pushl 引数1
call foo
addl 引数の個数*4, %esp
```

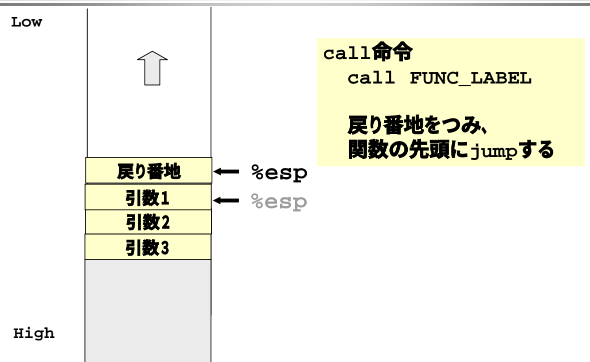

関数呼び出しの構造



関数呼び出しの構造

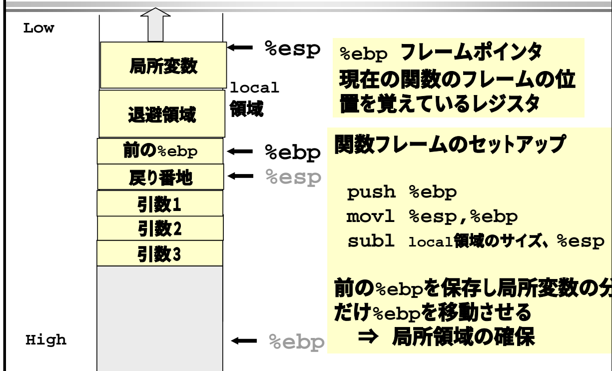


関数呼び出しの構造



プログラミング言語処理

関数呼び出しの構造



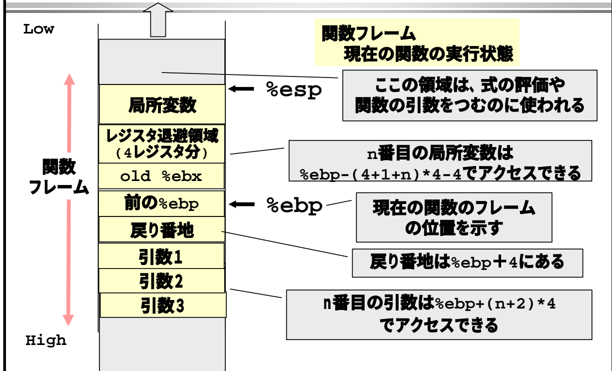
プログラミング言語処理

レジスタ、一時変数の退避

- x86では、%ebx,%ebp,%esi,%ediは、呼び出し側で保存することになっている（callee save register)
  - このコンパイラではレジスタとして %eax,%ebx,%ecx,%edx の4つのレジスタを使い %esi,%edi はつかわないので%ebxのみを退避
  - %eax,%ecx,%edxは、caller側が退避する。
- レジスタが足りなくなった場合、メモリに退避しておかなくてはならない。
  - このコンパイラでは、レジスタの待避領域は4つまでとしている
  - さらに関数呼び出しをしたときにも、レジスタをこの領域に退避しておく。
- したがって、ローカル領域のサイズは
  (4+1+n_local)*4

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

関数フレームのセットアップの手順

- 現在の%ebpをスタック上に保存し(pushl)ここを新たな%ebpとする。
- %ebpから、上の部分をレジスタ退避領域、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。
- %ebxのレジスタの保存する
- 局所変数にアクセスするためには、bpから２つ離れたところにあるので、 %ebp-(4+1+n)*4-4
- 引数にアクセスするためには、%ebpを基準にして%ebp+(n+2)*4

```
foo:    push %ebp
        movl %esp,%ebp
        subl スタック上に確保する領域, %esp
        movl %ebx,-4(%ebp)
          ... 関数の本体 ...
        movl -4(%ebp), %ebx
        leave
        ret
```

プログラミング言語処理

関数戻りの手順

- 関数から帰る場合には、戻り値を%eaxにセットする。
- 元の関数に戻るためには、
  - 退避した%ebxの復帰
  - %ebpのところに%espを戻して、まず、前の%ebpを戻す（leave命令）
  - 戻り番地を取り出して、そこにjump (RET命令)
- 戻ったら、引数を積んだ部分を%espを戻す

```
foo:    push %ebp
        movl %esp,%ebp
        subl スタック上に確保する領域, %esp
        movl %ebx,-4(%ebp)
          ... 関数の本体 ...
        movl -4(%ebp), %ebx
        leave
        ret
```

プログラミング言語処理

コンパイラの中間コード

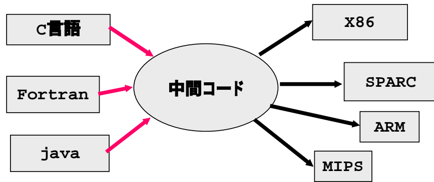
- 一般的に、コンパイラはコンパイラが作り安いように中間コードを設計し、構文解析によって得られた構文木を中間コードに変換する。
- ここで最適化などの 解析を行い、最終的にマシンコードに変換する。
- 中間コードを適当に設計する ことによって、実際のマシンから独立したものになり、いろいろなマシンに対 応できるようにもなる。

- 中間コードの生成までは、スタックマシンへのコード生成と非常によく似ている。
  - レジスタがあることを除いては！

プログラミング言語処理

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。



プログラミング言語処理

tiny Cの中間コード

- 変数rとしているのは、いわゆるプログラム上の局所変数では なく、レジスタが無限にあるとして考えた時の一時的な変数あるいは仮想的なレジスタ。
  - コード生成のフェーズにおいて、実際のレジスタが割り 当てられる。

LOADI r, n	整数nを変数rにnをセット。
LOADA r, n	n番目の引数を変数rにセットする。
LOADL r, n	n番目の局所変数を変数rにセットする。
STOREA r, n	変数rの値をn番目の引数に格納する。
STOREL r, n	変数rの値をn番目の局所に格納する。
ADD r,r1,r2	変数r1,r2を加算し、結果をrに格納する。
SUB r,r1,r2	変数r1,r2を減算し、結果をrに格納する。
MUL r,r1,r2	変数r1,r2を乗算し、結果をrに格納する。

プログラミング言語処理

tiny Cの中間コード

GT r,r1,r2	r1とr2して比較し、>ならrに1、それ以外は0をセットする。
LT r,r1,r2	r1とr2して比較し、<ならrに1、それ以外は0をセットする。
BEQ0 r, L	rが 0だったら,ラベルLに分岐する。
JUMP L	ラベルLにジャンプする。
CALL r, n, e	引数n個で、関数エントリeを関数呼び出しをし、結果をrにセットする。
ARG r,n	rをn番目の引数とする。
RET r	変数rを返り値として、関数呼び出しから帰る。
PRINTLN r, s	sのformatで、printlnを実行する。
LABEL L	ラベルLを示す。

プログラミング言語処理

tiny Cの中間コード

- 以下の形式のコードを、*四つ組*と呼 ばれる。

  op dst,src1,src2

- このほかに、命令に近い形に表現する *RTL(Register Transfer Language)*をいう形式もある。

プログラミング言語処理

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。
  - オペランドが増えていることに注意
  - スタックマシンと比較してみよ。
- 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 operand1,
     operand2,operand3;
    char *s_operand;
} Codes[MAX_CODE];
```

命令コード	オペランド
命令コード	オペランド
命令コード	オペランド
命令コード	オペランド

n_code カウンタ

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**中間コードから、実際の命令に変換**して出力する。

```
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、「中間コードからマシンコードへの変換」で説明する。

コードの生成ルーチン

```
void initGenCode()
{
    n_code = 0;
}
void genCode1(int opcode,int operand1)
{
    Codes[n_code].operand1 = operand1;
    Codes[n_code++].opcode = opcode;
}
void genCode2(int opcode,int operand1, int operand2)
{
    Codes[n_code].operand1 = operand1;
    Codes[n_code].operand2 = operand2;
    Codes[n_code++].opcode = opcode;
}
void genCode3(int opcode,int operand1, int operand2, int operand3)
{
    Codes[n_code].operand1 = operand1;
    Codes[n_code].operand2 = operand2;
    Codes[n_code].operand3 = operand3;
    Codes[n_code++].opcode = opcode;
}
void genCodeS(int opcode,int operand1, int operand2, char *s)
{
    Codes[n_code].operand1 = operand1;
    Codes[n_code].operand2 = operand2;
    Codes[n_code].s_operand = s;
    Codes[n_code++].opcode = opcode;
}
```

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

オペランド1つの命令用

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。

式のコンパイル：中間コードへの変換

- さて、レジスタマシンへのコンパイラで大きくことなるのは、式の計算をスタックではなくて、レジスタを使っておこなわなくてはならないところである。
- 式のコンパイルは、compileExprで行う。
  - この関数では、呼び出す側で ターゲットとなる一時的な変数を作って、これを引数にして呼び出す。

    compileExpr(int target, AST *p)

  - compileExprは、引数のASTの式に対して、

    「コードを実行すると与えtargetに 結果を格納する」

    コードを生成する。
- 文として実行され、値を必要としない場合にはtargetを-1としている。一時的な変数を作るのは、大域変数tmp_counterを使って新しい変数の 番号を生成する。

式のコンパイルの手順

1. 式が数字であれば、その数字をターゲットにセットするLOADIコードを生成する。
2. 式は変数であれば、compileLoadVarを呼び出して、その値をロードする コードを生成する。
3. 式が代入であれば、まず、新しい変数を作り、それに演算結果をいれる コードを生成する。そのあとで、compileStoreVarを呼び出して、その変数の値を 変数に格納するコードを出す。
4. 式が演算であれば、左辺と右辺に対する変数を作って、それをターゲットにコンパイルし、ターゲットに演算結果をいれるコード を生成する。

式のコンパイル

- CompileExpr

```
void compileExpr(int target, AST *p)
{
    int r1,r2;
    if(p == NULL) return;
    switch(p->op){
    case NUM:
        genCode2(LOADI,target,p->val)
      return;
    case SYM:
      compileLoadVar(target,getSymbol(p));
      return;
    case EQ_OP:
      if(target != -1) error("assign has no value");
      r1 = tmp_counter++;
      compileExpr(r1,p->right);
      compileStoreVar(getSymbol(p->left),r1);
      return;
    case PLUS_OP:
```

定数の場合には定数を一時変数にロードするコードを生成する

変数の値を一時変数targetにロードするコードを生成する関数

右辺の式の値をr1に計算するコードを生成

変数への代入のコードを生成する関数

式のコンパイル

- CompileExpr

```
  case PLUS_OP:
      r1 = tmp_counter++; r2 = tmp_counter++;
      compileExpr(r1,p->left);
      compileExpr(r2,p->right);
      genCode3(ADD,target,r1,r2);
      return;
    case MINUS_OP:
      r1 = tmp_counter++; r2 = tmp_counter++;
      compileExpr(r1,p->left);
      compileExpr(r2,p->right);
      genCode3(SUB,target,r1,r2);
      return;
    case MUL_OP:
      r1 = tmp_counter++; r2 = tmp_counter++;
      compileExpr(r1,p->left);
      compileExpr(r2,p->right);
      genCode3(MUL,target,r1,r2);
      return;
```

左の式と右の式を計算した値を格納する一時変数を生成する

左の式をコンパイルして、実行すると左の式がr1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

式のコンパイル

- CompileExpr

```
    case GT_OP:
      r1 = tmp_counter++; r2 = tmp_counter++;
      compileExpr(r1,p->left);
      compileExpr(r2,p->right);
      genCode3(GT,target,r1,r2);
      return;
    case CALL_OP:
      compileCallFunc(target,getSymbol(p->left),p->right);
      return;
    case PRINTLN_OP:
      if(target != -1) error("println has no value");
      printFunc(p->left);
      return;
      /* 省略 */
    default:
      error("unknown operater/statement");
    }
}
```

プログラミング言語処理

仮想レジスタの生成について

- 一時変数rは、いわゆるプログラム上の局所変数ではなく、レジスタが無限にあるとして考えた時の仮想的なレジスタ
- コンパイラが作った一時的な変数の結果は高々1回しか使わないようにコードを生成している。
  - その理由は、後で説明する実際のレジスタマシンのコードの生成を簡単にするため
- この理由から代入文自体の値は使われないように制限している。例えば、文として現れる

  x = z + 1;

  は、OKだが、

  x = (y=1)+1;

  は、だめ。

プログラミング言語処理

(スタックマシンのコンパイラと同じ)

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、インタプリータで使った環境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

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(int target, Symbol *var)
{
    int i;
    for(i = envp-1; i >= 0; i--){
        if(Env[i].var == var){
            switch(Env[i].var_kind){
            case VAR_ARG:
                genCode2(LOADA,target,Env[i].pos);
                return;
            case VAR_LOCAL:
                genCode2(LOADL,target,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

プログラミング言語処理

変数のストア

- 一時変数rを格納するコードSTOREAまたはSTOREL を生成する。

```
void compileStoreVar(Symbol *var, int r)
{
    int i;
    for(i = envp-1; i >= 0; i--){
        if(Env[i].var == var){
            switch(Env[i].var_kind){
            case VAR_ARG:
                genCode2(STOREA,r,Env[i].pos);
                return;
            case VAR_LOCAL:
                genCode2(STOREL,r,Env[i].pos);
                return;
            }
        }
    }
    error("undefined variable¥n");
```

格納する一時変数r

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

プログラミング言語処理

(スタックマシンのコンパイラと同じ)

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;
}
```

プログラミング言語処理

(スタックマシンのコンパイラと同じ)

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づつ取り出して、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

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(-1,p);
    }
}
```

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

ここだけ違う

targetを-1にして式をコンパイル

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。
- 分岐命令は、BEQ0命令しかない。この命令は、オペランドの値が 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...rに結果を残す。
BEQ0 r、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。一時変数ｒを生成してこの変数に値を計算するコードを生成する
2. ラベルL0を作って、BEQ ｒ、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;
    int r;
    r = tmp_counter++;
    compileExpr(r,cond);
    l1 = label_counter++;
    genCode2(BEQ0,r,l1);
    compileStatement(then_part);
    if(else_part != NULL){
      l2 = label_counter++;
      genCode1(JUMP,l2);
      genCode1(LABEL,l1);
      compileStatement(else_part);
      genCode1(LABEL,l2);
    } else {
      genCode1(LABEL,l1);
    }
```

条件式のための一時変数

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

(スタックマシンのコンパイラと同じ)

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;
}
```

局所変数を一つづつ取り出し、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、一時変数を作りそれに結果を計算し、コードを生成する

```
void compileReturn(AST *expr)
{
    int r;
    if(expr != NULL){
      r = tmp_counter++;
      compileExpr(r,expr);
    } else r = -1;
    genCode1(RET,r);
}
```

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、引数をコンパイルして、CALL命令を出す。
  - 一時変数ｒを生成して、引数をコンパイルし、ｒに引数の値が格納されるコードを出す。
  - 中間コードARGを生成
  - その後に、 CALL命令を生成する。
    - 中間コードCALLの引数は、引数の数と呼び出す関数名

プログラミング言語処理

関数呼び出しのコンパイル

```
void compileCallFunc(int target, Symbol *f,AST *args)
{
    int narg;
    narg = compileArgs(args);
    genCodeS(CALL,target,narg,f->name);
}
int compileArgs(AST *args)
{
    int r,n;
    if(args != NULL){
      n = compileArgs(getNext(args));
      r = tmp_counter++;
      compileExpr(r,getFirst(args));
      genCode1(ARG,r);
    } else return 0;
    return n+1;
}
```

引数のコンパイル

CALL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

引数の値をrに計算するコードを生成

引数をつむ中間コードARGの生成

プログラミング言語処理

While文、For文

- 考えてみてください。

プログラミング言語処理

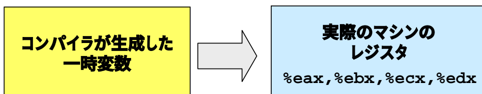
中間コードへのコンパイル（まとめ）

- block文をコンパイルするcompileBlockは、スタックマシンのものと同じ でよい。
- compileReturnでは、一時レジスタを生成し、それに式が計算されるコード を生成した後、RETのコードを生成する。もし、式がない場合にはコードに 対するレジスタを-1にしておく。
- 関数呼び出しの中間コードはCALLである。中間コードのオペランドは、 呼び出した結果をいれるtagetと引数の個数と関数名である。compileCallFunc は、compileArgを使って引数の値を計算するコードを生成し、そのあとでCALL を生成している。
- compileArgでは、 それぞれの引数について、一時変数を作り、その変数に計算結果が引数が入る コードを生成し、中間コード ARG rを生成する。同時に、引数の個数を計算し ていることに注意。
- If文のコンパイルを行うcompileIfでは、条件式のコンパイルをcompileExprで行い、BEQ0のコードを生成している以外は、スタックマシンの ものと同じである。

プログラミング言語処理

中間コードからマシンコードの生成

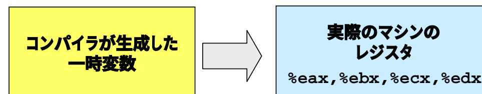
- 実際のコンパイラでは、この中間コードについて様々な最適化をし、最後にこれをマシンコード（アセンブリ言語）を出力する。
- マシンコードに変換するために最低限必要なのは、コンパイラで作り出した一時的な変数（仮想レジスタ） に実際のレジスタを割り当てる作業（register allocation）である。



プログラミング言語処理

レジスタ割り当て

- x86では汎用レジスタとして、6個のレジスタがあるが、このコンパイラでは %eax,%ebx,%ecx,%edxの4つのレジスタを使うことにする。
- 割り当ての過程で、 この実際のレジスタが足りなくなったら、適宜、実際のレジスタ上にある仮想 レジスタの値をメモリに退避して使い回さなくてはならない。
  - このための領域 として４レジスタ分の領域を確保する。
  - 実際は複雑な式を実行するには４つ以上の退避領域が必要になることがあるが、 簡単にするために４つに限定することにする。
  - （４つ以上の退避領域が必要の 場合はコンパイルをあきらめる）



レジスタ割り当てのためのデータ構造

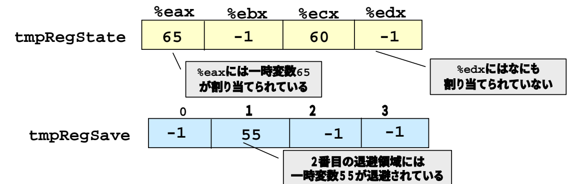
- tmpRegState : 実際のレジスタにどの仮想レジスタ（変数）が割り当て られているかを示す配列
- tmpRegSave : 退避領域にどの仮想レジスタの値が退避されているかを 示す配列

```
#define N_REG 4   /* 一時的な変数に割り当てるレジスタ数 */
#define N_SAVE 4  /* 一時的な変数の退避領域の数 */

#define REG_AX 0
#define REG_BX 1
#define REG_CX 2
#define REG_DX 3

char *tmpRegName[N_REG] = { "%eax", "%ebx", "%ecx", "%edx'
int tmpRegState[N_REG];  /* 実レジスタに割り当てられている仮想レジス
int tmpRegSave[N_SAVE];  /* 退避領域にある仮想レジスタ */
```

番号からレジスタ名を求める時に使う配列

レジスタ割り当てのためのデータ構造

- 例えば、reg番目のレジスタ（つまり、%eaxであれば、0番目）に仮想レジスタ rが割り当てられているときには、tmpRegState[reg]には、rをいれる。
  - 使われていないときには、-1をいれておく。
- tmpRegSaveも同様に、i番目の待避領域に仮想レジスタrの値がある場合には tmpRegSave[i]がrとなる。



関数フレーム内のオフセットの計算

- 退避領域と局所変数の%ebpからのオフセットを計算するマクロが、TMP_OFFと LOCAL_VAR_OFFである。
- %ebpから引数のオフセットを計算する関数が、ARG_OFFで ある。
- 退避領域は、常に4ワード分確保していることに注意

```
#define TMP_OFF(i)       -((i+1)+1)*4
#define LOCAL_VAR_OFF(i)       -(N_SAVE+1+(i+1))*4
#define ARG_OFF(i)       ((i)+2)*4
```


レジスタ割り当ての初期化

- 初期化する関数が、initTempReg
- 全ての値を、-1にする。-1は使われていないことを示す。

```
void initTmpReg()
{
    int i;
    for(i = 0; i < N_REG; i++) tmpRegState[i] = -1;
    for(i = 0; i < N_SAVE; i++) tmpRegSave[i] = -1;
}
```


レジスタの割り当て

- 実際のレジスタを割り当てるのに使われていない実レジスタを探さなくてはならない。
- getRegは、仮想レジスタrに空いている実際のレジスタを割り 当て、その実レジスタの値を返す。

```
int getReg(int r)
{
    int i;
    for(i = 0; i < N_REG; i++){
      if(tmpRegState[i] < 0){
          tmpRegState[i] = r;
          return i;
      }
    }
    error("no temp reg");
}
```

tmpRegStateの値が－1のものを探す

見つかったら、その値をセットしてレジスタ番号を返す

レジスタの割り当て

- saveRegは、実際のレジスタの値を退避するルーチンである。
  - もしも、なにも レジスタがロードされていない(tmpRegState[reg]が-1)の場合は何もしない。
  - それ以外の場合には、使われていない退避領域を探してそこにセーブするコー ドをだす。

```
void saveReg(int reg)
{   int i;
    if(tmpRegState[reg] < 0) return;
    for(i = 0; i < N_SAVE; i++){
       if(tmpRegSave[i] < 0){
           printf("¥tmovl¥t%s,%d(%%ebp)¥n",
                  tmpRegName[reg],TMP_OFF(reg));
           tmpRegSave[i] = tmpRegState[reg];
           tmpRegState[reg] = -1;
           return;
       }
    }
    error("no temp save");
}
```

なにも割り当てられていない場合

あいている退避領域を探す

値を退避領域に退避するストア命令を出す。

レジスタの割り当て

- useRegは、仮想レジスタrがどの実際のレジスタに割りあてられているのかを 調べる。
- もしも、仮想レジスタrが退避領域にある場合には、その値を実レジスタにロードして、その値を返す。

```
int useReg(int r)
{    int i,rr;
     for(i = 0; i < N_REG; i++){
        if(tmpRegState[i] == r) return i;
     }
     /* not found in register, then restore from save area. */
     for(i = 0; i < N_SAVE; i++){
        if(tmpRegSave[i] == r){
            rr = getReg(r);
            tmpRegSave[i] = -1;
            /* load into regsiter */
            printf("¥tmovl¥t%d(%%ebp),%s¥n",TMP_OFF(i),tmpRegName
            return rr;
        }}
```

一時変数rが割り当てられているレジスタを探す

退避領域にないか探す

みつかったら、getRegでレジスタを確保して、ロード命令を出す

プログラミング言語処理

レジスタの割り当て

- assignRegは、仮想レジスタrを実際のレジスタregに強制的に割り当てる関数であ る。
  - もしも、現在仮想レジスタに実際のレジスタが割り当てられていればなに もしないが、それ以外の場合は割り当てるレジスタをsaveRegで空いていることを確認してから、割り当てる。
  - この関数は、命令が特定のレジスタが必要な 場合に用いる。

```
/* assign r to reg */
void assignReg(int r, int reg)
{
    if(tmpRegState[reg] == r) return;
    saveReg(reg);
    tmpRegState[reg] = r;
}
```

プログラミング言語処理

レジスタの割り当て

- saveAllRegは、全てのレジスタの値を退避する。これは関数呼び出しの場合に 用いる。
- freeRegは、実レジスタregを開放する。

```
void saveAllRegs()
{
    int i;
    for(i = 0; i < N_REG; i++) saveReg(i);
}

void freeReg(int reg)
{
    tmpRegState[reg] = -1;
}
```

プログラミング言語処理

レジスタの割り当ての例

- 以上の関数を使ってたとえば、ADD　r, r1, r2の中間コードについては以下の ようにしてコードを生成する。
  1. r1,r2について、useRegで現在割り当てられているレジスタを求める。これをR1,R2とする。
  2. R1、R2をfreeRegで開放する。
  3. assignRegで、ｒに、R1を割り当てる。
  4. addl R1,R2のコードを生成する。
- なお、中間コードの生成では変数は一回しか使われないようにしている。従って、使ってしまえば、開放してよい。
- しかし、実際のコンパイラではこのよう な条件は必ずしも成立しないことがあるので、レジスタの開放はこの命令以降、レジスタが使われないことを確かめなくてはならない。

プログラミング言語処理

アセンブリ言語への変換

- genFuncCodeでは、生成された命令を上の手順を使って、実際の命 令を生成している。
- まず、関数のはじめの部分を生成して、本体のコードを生成し、最後のreturnの部分のコードを生成する。
- あらかじめ、ret命令が埋め 込まれるようにして、RETではここにJUMPするようにするため、ret_labにラベ ルを作っておく。

プログラミング言語処理

アセンブリ言語への変換

```
void genFuncCode(char *entry_name, int n_local)
{
     ... 宣言省略 ...
    /* function header */
     puts("¥t.text");                          /* .text         *
     puts("¥t.align¥t4");                 /* .align 4      */
     printf("¥t.globl¥t%s¥n", entry_name);    /* .globl  */
     printf("¥t.type¥t%s,@function¥n", entry_name);/* .type ,@func
     printf("%s:¥n", entry_name);              /* :             *
     printf("¥tpushl¥t%%ebp¥n");
     printf("¥tmovl¥t%%esp,%%ebp¥n");

     frame_size = -LOCAL_VAR_OFF(n_local);
     ret_lab = label_counter++;

     printf("¥tsubl¥t$%d,%%esp¥n",frame_size);
     printf("¥tmovl¥t%%ebx,-4(%%ebp)¥n");
```

関数headerの出力

局所変数、退避領域の確保

%ebxの退避

プログラミング言語処理

関数のheader

- 関数の最初は、以下のコードである

```
.text
        .align 4
        .globl 関数名
        .type 関数名, @function
関数名:
        pushl %ebp
        movl  %esp,%ebp
        subl  フレームのサイズ, %esp
        movl  %ebx,-4(%ebp)
```

- 関数の最初では、%ebp,%espのセットの他、callee save のレジスタである%ebx を退避しておく。本当は、%ebx が使われない限り、セーブする必要はないが、 簡単のために常にセーブすることにする。

プログラミング言語処理

アセンブリ言語への変換

```
initTmpReg();
 for(i = 0; i < n_code; i++){
   opd1 = Codes[i].operand1; opd2 = Codes[i].operand2;
   opd3 = Codes[i].operand3; opds = Codes[i].s_operand;
   switch(Codes[i].opcode){
   case LOADI:
        if(opd1 < 0) break;
        r = getReg(opd1);
        printf("¥tmovl¥t$%d,%s¥n",opd2,tmpRegName[r]);
        break;
   case LOADA:   /* load arg */
        if(opd1 < 0) break;
        r = getReg(opd1);
        printf("¥tmovl¥t%d(%%ebp),%s¥n",ARG_OFF(opd2),tmpRegNa
        break;
   case LOADL:   /* load local */
        if(opd1 < 0) break;
        r = getReg(opd1);
        printf("¥tmovl¥t%d(%%ebp),%s¥n",LOCAL_VAR_OFF(opd2),tm
```

コードを一つ一つ順番に翻訳していく

仮想レジスタに実際のレジスタを割り当てる

movl $n,rを出力

ARG_OFFでオフセットを計算

プログラミング言語処理

アセンブリ言語への変換

現在、一時変数opd1が割り当てられているレジスタを求める

そのあとで、開放する

```
case STOREA: /* store arg */
     r = useReg(opd1); freeReg(r);
     printf("¥tmovl¥t%s,%d(%%ebp)¥n",tmpRegName[r],ARG_OFF(opd2)
     break;
case STOREL: /* store local */
     r = useReg(opd1); freeReg(r);
     printf("¥tmovl¥t%s,%d(%%ebp)¥n",tmpRegName[r],LOCAL_VAR_OFF
     break;
```

プログラミング言語処理

アセンブリ言語への変換

- 関数の最初のコードを生成した後は、格納されている中間コードを取り 出し、実際の命令コードを生成する。
- 引数をロード、ストアするLOADA,STOREAについては、ARG_OFFマクロを使っ て、オフセットを計算してコードを生成する。
- ローカル変数をロード、ストアするLOADL,STORELについては、 LOCAL_VAR_OFFマクロを使って、オフセットを計算してコードを生成する。
- 既に実際にロードされている仮想レジスタについては、useRegを使って 探し、新たに確保するレジスタについてはgetRegで割り当てを行っている。 １度使ったレジスタについてはfreeRegで解放していることに注意。

プログラミング言語処理

アセンブリ言語への変換

- 条件分岐命令では、cmpl命令で0との比較をし、je命令で分岐している。 GT,LTのコードについては、分岐命令を使って、dstに0か１をセットする命令 列を生成している。
  - x86では直接0,1をセットするsetcc命令があるが、ここで はあえて使わなかった。
- ラベル、JUMP命令については、以下の通り。

```
case BEQ0:   /* conditional branch */
     r = useReg(opd1); freeReg(r);
     printf("¥tcmpl¥t$0,%s¥n",tmpRegName[r]);
     printf("¥tje¥t.L%d¥n",opd2);
     break;
 case LABEL:
     printf(".L%d:¥n",Codes[i].operand1);
     break;
 case JUMP:
     printf("¥tjmp¥t.L%d¥n",Codes[i].operand1);
     break;
```

プログラミング言語処理

アセンブリ言語への変換

- ARGコードは、push命令で生成される。使った後はfreeしておく。
- CALLコードでは、saveAllRegsで現在 使われているレジスタを退避させなくてはならないことに注意。
  - call命令を使っ て生成した後は、addlを使って、pushした分、スタックポインタを元に戻す。 返り値は、%eaxに入っているはずなので、ターゲットがある場合には、強制的 に、assignRegを使ってREG_AXに割り当てを行う。

```
case CALL:
     saveAllRegs();
     printf("¥tcall¥t%s¥n",opds);
     if(opd1 < 0) break;
     assignReg(opd1,REG_AX);
     printf("¥tadd $%d,%%esp¥n",opd2*4);
     break;
case ARG:
     r = useReg(opd1); freeReg(r);
     printf("¥tpushl %s¥n",tmpRegName[r]);
     break;
```

プログラミング言語処理

アセンブリ言語への変換

- RETに関しては、assignRegをrを%eaxにセットして、プログラムの最後に 生成されているreturnのところにjumpするようにしている。

```
case RET:
    r = useReg(opd1); freeReg(r);
    if(r != REG_AX)
      printf("¥tmovl¥t%s,%%eax¥n",tmpRegName[r]);
    printf("¥tjmp .L%d¥n",ret_lab);
    break;
```

プログラミング言語処理

アセンブリ言語への変換

- 演算に関しては、x86は2オペランド命令なので、片方のオペランドになっ たものは、assginRegでターゲットにわり当てる。

```
case ADD:
    r1 = useReg(opd2); r2 = useReg(opd3);
    freeReg(r1); freeReg(r2);
    if(opd1 < 0) break;
    assignReg(opd1,r1);
    printf("¥taddl¥t%s,%s¥n",tmpRegName[r2],tmpRegName[r1]
    break;
case SUB:
    r1 = useReg(opd2); r2 = useReg(opd3);
    freeReg(r1); freeReg(r2);
    if(opd1 < 0) break;
    assignReg(opd1,r1);
    printf("¥tsubl¥t%s,%s¥n",tmpRegName[r2],tmpRegName[r1]
```

プログラミング言語処理

アセンブリ言語への変換

- MULに関しては、片方のオペランドが%eaxにいれておく必要がある。

```
case MUL:
    r1 = useReg(opd2); r2 = useReg(opd3);
    freeReg(r1); freeReg(r2);
    if(opd1 < 0) break;
    assignReg(opd1,REG_AX);
    saveReg(REG_DX);
    if(r1 != REG_AX)
      printf("¥tmovl %s,%s¥n",
          tmpRegName[r1],tmpRegName[REG_AX]);
    printf("¥timull¥t%s,%s¥n",
          tmpRegName[r2],tmpRegName[REG_AX]);
    break;
```

プログラミング言語処理

アセンブリ言語への変換

- LTやGTについては、ターゲットに0か1が残るようにコードを生成している。しかし、分岐命令を中間コードにして出力するようにすれば、もっと効率 的なコードを出力することができる。

```
case LT:
    r1 = useReg(opd2); r2 = useReg(opd3);
    freeReg(r1); freeReg(r2);
    if(opd1 < 0) break;
    r = getReg(opd1);
    l1 = label_counter++;
    l2 = label_counter++;
    printf("¥tcmpl¥t%s,%s¥n",tmpRegName[r2],tmpRegName[r1]);
    printf("¥tjl .L%d¥n",l1);
    printf("¥tmovl¥t$0,%s¥n",tmpRegName[r]);
    printf("¥tjmp .L%d¥n",l2);
    printf(".L%d:¥tmovl¥t$1,%s¥n",l1,tmpRegName[r]);
    printf(".L%d:",l2);
    break;
 case GT:
```

プログラミング言語処理

アセンブリ言語への変換

- RPINTLNでは、外部関数であるprintlnを呼び出すコードを生成する。

```
case PRINTLN:
        r = useReg(opd1); freeReg(r);
        printf("¥tpushl¥t%s¥n",tmpRegName[r]);
        printf("¥tpushl¥t$.LC%d¥n",opd2);
        saveAllRegs();
        printf("¥tcall¥tprintln¥n");
        printf("¥taddl¥t$8,%%esp¥n");
        break;
```

プログラミング言語処理

アセンブリ言語への変換

- 最後に、ret_labを生成して、ここに関数の戻りのコード列を生成する。

```
/* return sequence */
    printf(".L%d:¥tmovl¥t-4(%%ebp), %%ebx¥n",ret_lab);
    printf("¥tleave¥n");
    printf("¥tret¥n");
}
```

```
ret_lab:
      movl -4(%ebp),%ebx
      leave
      ret
```

文字列の確保

- 文字列については、以下のコードを生成して、文字列を確保して おく。

```
int genString(char *s)
{
     int l;
     l = label_counter++;
     printf("¥t.section¥t.rodata¥n");
     printf(".LC%d:¥n",l);
     printf("¥t.string ¥"%s¥"¥n",s);
     return l;
}
```


変数と配列の宣言 （最終課題）

- 大域変数と配列宣言については、あえてつくっていない
- 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。最終課題の１つである課題の8-1では、これを作ってもらう。少なくとも、以下の機能が必要である。
  - declareVariableとdeclareArrayでは、大域変数や配列を確保する命令列 を生成する。適当なプログラムをつくってみて、-Sのオプションを付けてコンパイルして、どのようなコード変換されるかを調べること。
  - Cの大域的な宣言 int a[10]は、.comm a,40, 32のようにコンパイルされている
  - 大域変数や配列を扱うための中間コードが必要である。例えば、以下の コードが必要となるであろう。
    - 変数をロード/ストアする中間コード
    - 配列のアドレスをロードするコード
    - 配列の要素をロード/ストアするコード
- これらのコードをコンパイラが生成できるように拡張すること。

コンパイラとプログラムの実行

- さて、説明したコードをコンパイルして tiny-cc-x86を作る。 tiny-cc-x86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.sを作るには、

```
% tiny_cc_x86 < foo.c > foo.s
```

  - printlnはライブラリ関数なので、println.cにある。 実行ファイルをつくるには、これをリンクして、コンパイルする。

```
% cc foo.s println.c
% a.out
```

  - ccの代わりに、アセンブラas、リンカldを直接使って もよい。

最終課題

以下の２つの課題のどちらかを選択し、レポートを提出すること。

- 課題 8-1：これまで説明したtiny Cのコンパイラでは大域変数や配列宣言を処理していない。配列宣言と配列参照を処理できるように拡張して、8 queenのプログラムを書き、コンパイル、実行しなさい。
  ヒント：
  - まずは,適当なプログラムを作ってみて、-Sのオプションを付けてコンパイルして、どのようなコード変換されるかを調べること。
  - Cの大域的な宣言 int a[10]は、.comm a,40, 32のようにコンパイルされている
  - 適当な中間コードを加えて、それに対するgenFuncCodeのルーチンを書けばよい。
- 課題8-2： これまで説明したtiny Cのコンパイラに対して、自分なりの拡張をして、その拡張を用いたプログラムを実行しなさい。

次回

- 最適化、その他