プログラミング言語処理

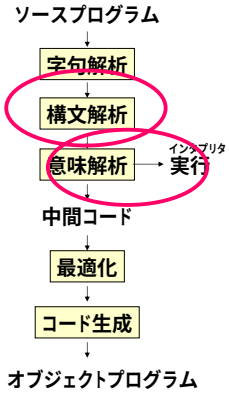
第５回（平成１５年度１０月７日）

tiny Cのインタプリター

筑波大学　佐藤

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。



インタプリター

- tiny-Cのインタープリタを作ってみることにする。
- まず、式の実行から考えてみよう。変数を考えなければ、大体は式の評価でつくったインタープリタと同じである。その後に、言語の重要な機能である関数について考えてみることにする。
- 説明するプログラムは、以下にある。
  - interp.h : インタプリタのheader
  - interp_expr.c : インタプリタの式の処理
  - interp.c : インタプリタの関数、文の処理

変数の扱い

- 変数の値を格納しておくためには、シンボル構造体のvalのフィールドにいれておく。
  - シンボル構造体は以下のようになっていた。

```
typedef struct symbol {
    char *name;
    int val;    /* ← これを用いる */
    AST *func_params;
    AST *func_body;
} Symbol;
```

関数の定義はここにいれておく。

式の評価

式を評価（実行）する関数

```
int executeExpr(AST *p)
{
    if(p == NULL) return 0;
    switch(p->op){
    case NUM:
       return p->val;
    case SYM:
       return getValue(getSymbol(p));
    case EQ_OP:
       return setValue(getSymbol(p->left),executeExpr(p->right));
    case PLUS_OP:
       return executeExpr(p->left) + executeExpr(p->right);
    case MINUS_OP:
       return executeExpr(p->left) - executeExpr(p->right);
    case MUL_OP:
       return executeExpr(p->left) * executeExpr(p->right);
    case LT_OP:
       return executeExpr(p->left) < executeExpr(p->right);
    case GT_OP:
```

これはどうなっていればいいのか？

再帰的に呼ばれていることに注意

変数の値の参照

- 関数を考えなければこれでいい。

```
int getValue(Symbol *var)
{
    return var->val;
}

int setValue(Symbol *var,int val)
{
    var->val = val;
    return val;
}
```

単なる式を評価するだけならば、以上のコードで十分であるが、実際はもうすこし仕掛けが必要となる。それは関数のパラメータ引数や局所変数があるからである

プログラミング言語処理

関数の定義：構文解析とのインタフェース

- 構文解析部において、関数の定義が処理されると、defineFunctionが呼び出される

```
void defineFunction(Symbol *fsym,AST *params,AST *body)
{
    fsym->func_params = params;
    fsym->func_body = body;
}
```

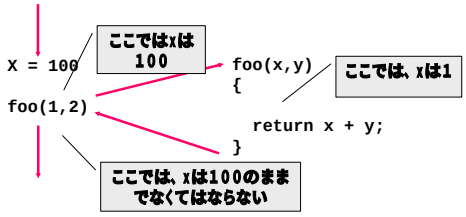
- 変数宣言に対するインタフェースは、declareVariable

```
void declareVariable(Symbol *vsym,AST *init_value)
{
    if(init_value != NULL){
      vsym->val = executeExpr(init_value);
    }
}
```

プログラミング言語処理

環境(environment) ：変数と値の結合(bind)

- どの変数がどのような値と結合されているかという状態のことを *環境(environment)*という
  - 代入とは異なる



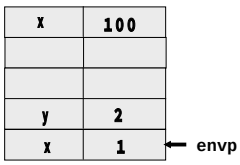
プログラミング言語処理

環境のためのデータ構造

- 環境のためのデータ構造Environment
  - シンボルと値とのペアを記憶しておく
  - envpは、この配列をどこまでつかっているか
  - 値を知りたいときには、最新のものから探す

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

Environment Env[MAX_ENV];
int evnp = 0;
```



プログラミング言語処理

環境へのアクセス、操作

- 変数の値を探す時には、この表を最近に結合された順に探し、この表で見つかった場合にはその値を使い、ない時にはシンボル構造体にある値を使えばよい
- 関数の実行が終わったら、envpの値を元に戻せば結合はなくなる

```
int getValue(Symbol *var)
{
    int i;
    for(i = envp-1; i >= 0; i--){
      if(Env[i].var == var) return Env[i].val;
    }
    return var->val;
}
```

逆順に探す

あったら、その値を返す

なかったら、valの値 大域変数の場合

プログラミング言語処理

環境へのアクセス、操作

- 代入で、変数の値を変える場合もこの表にある場合には、その値を変更

```
int setValue(Symbol *var,int val)
{
    int i;
    for(i = envp-1; i >= 0; i--){
      if(Env[i].var == var){
          Env[i].val = val;
          return val;
      }
    }
    var->val = val;
    return val;
}
```

最新のbindから探す

あったら、その値を変更 局所変数への代入

なかったら、valの値を変更 大域変数の値

プログラミング言語処理

関数呼び出し

- executeExprの残り

```
int executeExpr(AST *p)
{
    if(p == NULL) return 0;
    switch(p->op){

    /* 上に示した演算式、代入の実行 */

    case CALL_OP:
      return executeCallFunc(getSymbol(p->left),p->right);
    case PRINTLN_OP:
      printFunc(p->left);
      return 0;

    /* あと、配列についての式の実行は後で説明する */

    }
```

関数の呼び出しをするする関数がexecuteCallFunc

関数名のシンボル

引数のリスト

関数呼び出しの実行

- 引数に書かれた式を実行して、その値をパラメータにbindして、関数の中の文を実行する

```
int executeCallFunc(Symbol *f,AST *args)
{
    int nargs; int val;
    AST *params;
    nargs = 0;
    for(params = f->func_params; params != NULL;
       params = getNext(params)){
       Env[envp+nargs].var = getSymbol(getFirst(params));
       Env[envp+nargs].val = executeExpr(getNth(args,nargs));
       nargs++;
    }
    envp += nargs;
      … /* 省略しているところあり */
    executeStatement(f->func_body);
      … /* 省略しているところあり */
    envp -= nargs;
}
```

関数のパラメータの部分を取り出す

パラメータを1つつづとりだす

環境に積んでいく

引数を実行

一挙に、環境に積んだものを有効にする

関数本体の実行

環境を実行前に戻す

println

- システム関数

```
static void printFunc(AST *args)
{
    printf((char *)executeExpr(getNth(args,0)),
         executeExpr(getNth(args,1)));
    printf("¥n");
}
```

第二引数がないとまずい？

動的結合と静的結合

- 説明した環境の作り方は、コンパイラで実行するCなどの言語とはちょっと違った振舞を示すことがある

```
var x;

addx(y) {   return x + y; }

addxy(x,y) {  return addx(y); }

main()
{
    x = 10;
    println("%d",addx(2)); /* ここは、12 */
    println("%d",addxy(2,3)); /* ここは 5 */
}
```


動的結合と静的結合

- どのような順番で呼び出されるかに依存してしまう。このような実現の方法を動的結合(dynamic binding)と呼ぶ（動的束縛と呼ぶこともある）。
- どのような順番でよびだされても、プログラム中にかかれた参照範囲できまった変数を参照する方式を静的束縛という。
  - コンパイラでは静的束縛になるのが普通である

配列と文字列の処理

- 構文解析部で変数宣言が入力されると、declareArrayが呼び出される
- tiny Cのインタープリタでは配列はシンボルのvalの部分に、配列のアドレスをいれることによって実現している
- 文字列も同様

```
void declareArray(Symbol *a, AST *size)
{
    a->val = (int)malloc(sizeof(int)*executeExpr(size));
}
```

```
int getArray(int a, int index)
{
    int *ap;
    ap = (int *)a;
    return ap[index];
}
```

```
int setArray(int a,int index,
             int value)
{
    int *ap;
    ap = (int *)a;
    ap[index] = value;
    return value;
}
```


文の実行

- executeCallFuncの中で、本体の実行するためにexecuteStatementを呼び出している

```
void executeStatement(AST *p)
{
    if(p == NULL) return;
    switch(p->op){
    case BLOCK_STATEMENT:
        executeBlock(p->left,p->right);
        break;
    case IF_STATEMENT:
        executeIf(p->left,getNth(p->right,0),getNth(p->right,1));
        break;
    case WHILE_STATEMENT:
      ...

    default:
        executeExpr(p);
    }
}
```

ASTのopによりそれぞれの文の処理に分岐する。

文でない場合のdefaultの処理式も文として扱う。値は捨てる

プログラミング言語処理

IF文の処理

- if文のASTは、左に条件式、右に条件が成立した時に実行される文(then部)のASTと条件式が成立しなかった時に実行される式(else部)のリストが入っている

```
case IF_STATEMENT:
      executeIf(p->left,getNth(p->right,0),
                        getNth(p->right,1));
      break;
```

```
void executeIf(AST *cond, AST *then_part, AST *else_part)
{
    if(executeExpr(cond))
      executeStatement(then_part);
    else
      executeStatement(else_part);
}
```

プログラミング言語処理

複文と局所変数

- 関数の本体は、複文
- ブロック文のASTは、左に局所変数のリスト、右に実行するべき文のASTのリストが入っている。これらをとりだして、executeBlockを呼び出す

```
case BLOCK_STATEMENT:
      executeBlock(p->left,p->right);
      break;
```

- 局所変数が現れた場合には、ブロック文で宣言された局所変数を参照しなくてはならない。しかし、このブロック文が終わった時には、元の値に戻さなくてはならない。
  - つまり、有効範囲（スコープ）を持つことになる。
  - 局所変数として宣言されていない変数は、大域変数として、シンボルテーブルの中のシンボルのvalの値が参照される。

プログラミング言語処理

複文と局所変数

- excuteBlockでは、局所変数をつくり、リストの中の文を順番に実行する。
  - 局所変数をつくるのはbindするということ
  - 関数呼び出しに似ている

```
void executeBlock(AST *local_vars,AST *statements)
{
    AST *vars;
    int envp_save;
    envp_save = envp;
    for(vars = local_vars; vars != NULL; vars = getNext(va
      Env[envp++].var = getSymbol(getFirst(vars));
    for( ; statements != NULL; statements = getNext(statem
      executeStatement(getFirst(statements));
    envp = envp_save;
    return;
}
```

あとで環境を戻すためにenvpの値を覚えておく

局所変数をとりだしbindしておく。

文を順番に実行

終わったら環境をもとに戻す

プログラミング言語処理

return文：setjmp/longjumpの使い方

- return文は、関数の実行を終了し、関数の戻り値を返す文
- return文のASTは、左に戻り値の式が入っている。これをとりだして、executeReturnを呼び出す

```
case RETURN_STATEMENT:
      executeReturn(p->left);
      break;
```

- インタープリターでは関数本体を実行する時に、executeStatementで再帰的に呼び出しを行って実行をしている
- 途中で、return文が実行されたときには、最初のexecuteCallFuncのところに戻ってこなくてはならない。

この動作を行うためにsetjmp/longjmpを使わなくてはならない

プログラミング言語処理

setjmp/longjumpの使い方

- setjmp/longjmpは関数の現在の状態を記録しておき、呼び出された先から戻る機能である

```
#iclude <setjmp.h>
jmp_buf env;

foo()
{   ...
    setjmp(env);
    ...
    goo1();
    ...
}

goo1()
{
    ...
    goo2();
    ...
}

goo2()
{
    ...
    longjmp(env,1);
}
```

はじめは0を返す

longjmpから返るとlongjmpで与えた値が返る、つまり1

プログラミング言語処理

return文とexecuteCallFunc

- return文が実行されると実行中の関数を実行しているexecCallFuncにもどらなくてはならない
  - executeCallFuncで、戻るところを記録しておく。

```
jmp_buf *funcReturnEnv;
int funcReturnVal;

...
 ret_env_save = funcReturnEnv; /* 元の値をとっておく */
 funcReturnEnv = &ret_env;      /* 今度戻ってくるところにセット */
 if(setjmp(ret_env) != 0){      /* longjmp で戻ってきたとき */
    val = funcReturnVal;       /* returnからの値をとる */
 } else {                  /* はじめにセットしたとき,本体を評価 */
    executeStatement(f->func_body); /* 本体の実行 */
 }
 funcReturnEnv = ret_env_save; /* 前の値に戻す*/
 ...
```

戻る時点を記憶しておくjmpbufへのポインタ(大域変数)

executeCallFuncの中で、局所変数としてjmp_bufが確保されている。

プログラミング言語処理

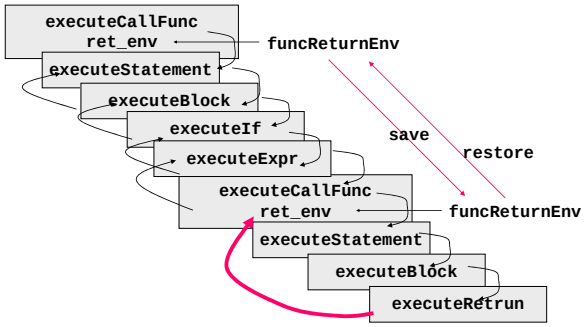
return文とexecuteCallFunc

- return文を実行するexecuteReturnでは、返す値をfuncReturnValにいれて、 funcReturnEnvでしめされている場所にもどればよい

```
void executeReturn(AST *expr)
{
    funcReturnVal = executeExpr(expr); /* 戻り値の式を実行 *
    longjmp(*funcReturnEnv,1);   /* 最近のsetjmpにかえる !!
    error("longjmp failed!¥n");  /* もしも、飛べなければエラー
}
```

プログラミング言語処理

return文とexecuteCallFunc



プログラミング言語処理

executeCallFuncの完成版

```
int executeCallFunc(Symbol *f,AST *args)
{
    int nargs;
    int val;
    AST *params;
    jmp_buf ret_env;
    jmp_buf *ret_env_save;
    nargs = 0;
    for(params = f->func_params; params != NULL;
        params = getNext(params)){
        Env[envp+nargs].var = getSymbol(getFirst(params));
        Env[envp+nargs].val = executeExpr(getNth(args,nargs));
        nargs++;
    }
    ret_env_save = funcReturnEnv;
    funcReturnEnv = &ret_env;
    envp += nargs;
    if(setjmp(ret_env) != 0){
        val = funcReturnVal;
    } else {
        executeStatement(f->func_body);
    }
    envp -= nargs;
    funcReturnEnv = ret_env_save;
    return val;
}
```

プログラミング言語処理

while文

- while文のASTは、左の条件式、右に条件が成立している間実行される文が入っている

```
case WHILE_STATEMENT:
      executeWhile(p->left,p->right);
      break;
```

- executeWhileでは、executeExprで条件式を実行し、これが真、すなわち0でない間、本体の文を実行すればよい

```
void executeWhile(AST *cond,AST *body)
{
    while(executeExpr(cond))
      executeStatement(body);
}
```

プログラミング言語処理

For文

- 自分で考えてみること

プログラミング言語処理

インタプリタのmainプログラム

- mainでは、まず、構文解析ルーチンであるyyparseを呼び出す。
- yyparseはEOFが入力されるまで、構文解析を行い、外部定義に対して、defineFunctionやdeclareVariableを呼び出す。
- その後で、executeCallFuncを使ってmainプログラムを呼び出す。

```
int main()
{
    int r;
    yyparse();
    r = executeCallFunc(lookupSymbol("main"),NULL);
    return r;
}
```

構文解析を行う。
ここで、定義などは処理される

mainを探して、callする

インタプリタのコンパイルと実行

- それぞれの*.cをコンパイル
- 以下でリンク

```
  % cc -o tiny-c-run interp_main.o AST.o cparser.o
interp.o interp_expr.o
```

- インタプリタ(tiny-c-run) の実行。
  - tiny Cのプログラム(sample.c)を準備
  - 標準入力から入力。

```
% tiny-c-run < sample.c
```


次回は

- 小テストをします
- スタックマシンの説明
- tiny Cのスタックマシンへのコンパイラ

課題４

- 課題３の言語のインタプリターを作りなさい。

例えば、以下のようなプログラムをかくことができる。

```
x = 1+2;
y = 100;
z = (x+y)*10+34;
print z+1;
```


課題５：
tiny-Cインタープリタによる8クイーン問題の実行

- tiny Cで、8クイーン問題のプログラムを書き、インタプリターを用いて、この問題を解きなさい。
  - 8クイーン問題とは、8×8のチェス版に8個のクイーン（縦横斜めに移動できる）をおいて、お互いにぶつからない位置にクイーンを置く問題である。なお、問題は解が何個あるか求めるだけでよい（実際の位置を出力する必要はない）。
  - この問題を解くために必要な機能があれば、適宜追加製作すること。
- 以下のものを提出すること：
  - 実行した８クイーン問題のtiny-Cプログラムと８クイーン問題の解答（いくつ解があったか）
  - インタプリターのプログラムとどのような機能を追加したかの説明