プログラミング言語処理

第２回（平成１５年度９月９日）

tiny Cの概要とデータ構造

筑波大学　佐藤

Tiny C

- この講義では、具体的なコードを解説しながら、講義を進めていく。コードの例として使うのがC言語風の極簡単なプログラミング言語tiny Cである。
- 講義の中では、tiny Cに対して
  - インタプリタ
  - スタックマシンへのコンパイラ
  - インテルx86プロセッサへのコンパイラ

  をつくる

tiny Cの言語仕様

- 使えるデータ型は、integerのみ。
- integer型の配列は１次元のみ。
- 関数の中では、局所変数を宣言できる。
- if文の他、while文、for文をサポート。
- 使える演算子は+,-,*の他、比較は<,>。
- システム関数は出力するためのprintln関数。printfを呼び出す。ここでのみ、フォーマットを指定するために文字列を使える。
- 分割コンパイルはなし。
- もちろん、ポインターもなし。
- C言語と同じようにmain
- pre-processorは通してないので、#includeなどはできない。

例１

- 例１

```
main()
{
    var i,s;
    s = 0;
    i = 0;
    while(i < 10){
        s = s + i;
        i = i + 1;
    }
 println("s = %d",s);
}
```


例２

- 例２

```
var A[10];

main()
{
    var i;
    for(i = 0; i < 10; i = i + 1) A[i] = i;
    println("s = %d",arraySum(A,10));
}

arraySum(a,n)
{
    var i,s;
    s = 0;
    for(i = 0; i < 10; i = i + 1) s = s + a[i];
    return s;
}
```


tiny Cの構文規則

- BNF記法による

```
program :=  {external_definition}*

external_definition:=
        function_name '(' [ parameter {',' parameter}* ] ')' compound_statement
      | VAR variable_name ['=' expr] ';'
      | VAR array_name '[' expr ']' ';'

compound_statement:=
      '{' {local_variable_declaration}* {statement}* '}'

local_variable_declaration: =
       VAR variable_name [ {',' variable_name}* ] ';'

statement :=
        expr ';'
      | compound_statement
      | IF '(' expr ')' statement [ ELSE statement ]
      | RETURN [expr ] ';'
      | WHILE '(' expr ')' statement
      | FOR '(' expr ';' expr ';' expr ')' statement

expr:=   primary expr
```

tiny Cの構文規則

- BNF記法による

```
expr :=   primary_expr
        | variable_name '=' expr
        | array_name '[' expr ']' '=' expr
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '<' expr
        | expr '>' expr

primary_expr :=
          variable_name
        | NUMBER
        | STRING
        | array_name '[' expr ']'
        | function_name '(' expr [{',' expr}*] ')'
        | PRINTLN  '(' STRING ',' expr ')'
```


tiny C処理系のデータ構造

- tiny Cの処理系で使われる基本的なデータ構造について説明する。プログラムでは、
  - AST.h : 構文木などの基本的なデータ構造の定義ファイル
  - AST.c ： 構文木のデータ構造、その他の基本的なデータ構造

  の2つのファイルに定義されている。

構文木(AST)のデータ構造

- opには、PLUS_OPやMINUS_OPなどのノードの演算子を表すコードが入る。
- 木構造を持つものついては、leftとrightに木の左、右のASTノードへのポインタをいれる。
- opがNUMの時には、valにその値をいれる。
- opがSYM(シンボル)の場合には、シンボル構造体へのポインターをいれる。

```
typedef struct abstract_syntax_tree {
    enum code op;
    int val;
    struct symbol *sym;
    struct abstract_syntax_tree *left,*right;
} AST;
```

シンボルの時を追加

Unionをつかえば、メモリが節約できる

ASTの生成

- ASTのノードを作る関数が、makeAST

```
AST *makeAST(enum code op,AST *left,AST *right)
{
    AST *p;
    p = (AST *)malloc(sizeof(AST));
    p->op = op;
    p->right = right;
    p->left = left;
    return p;
}
```

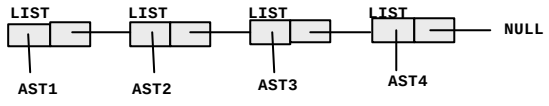

ASTの生成

- NUMの数値定数のASTノードを作る関数がmakeNum

```
AST *makeNum(int val)
{
    AST *p;
    p = (AST *)malloc(sizeof(AST));
    p->op = NUM;
    p->val = val;
    return p;
}
```


ASTによるリスト構造

- 演算子の構文木は2分木であるが、いくつかの要素をならべて表現するリスト構造があればいろいろと便利
- ASTのopがLISTの場合にリストとしてみなすことにする



リストの生成

- リストの生成は、makeASTを使って、マクロで定義してある

```
#define makeList1(x1) makeAST(LIST,x1,NULL)
#define makeList2(x1,x2) makeAST(LIST,x1,makeAST(LIST,x2
#define makeList3(x1,x2,x3)¥
    makeAST(LIST,x1,makeAST(LIST,x2,makeAST(LIST,x3,NULL)
```


リストの最後に要素を加える

- 最後を見付けて、LISTのASTノードを付け加える

```
AST *addLast(AST *l,AST *p)
{
    AST *q;

    if(l == NULL) return makeAST(LIST,p,NULL);
    q = l;
    while(q->right != NULL) q = q->right;
    q->right = makeAST(LIST,p,NULL);
    return l;
}
```


リストへのアクセス

- n番目の要素を取り出すgetNth

```
AST *getNth(AST *p,int nth)
{
    if(p->op != LIST){
      fprintf(stderr,"bad access to list¥n");
      exit(1);
    }
    if(nth > 0) return(getNth(p->right,nth-1));
    else return p->left;
}
```

- １番目をとるgetFirst

```
#define getFirst(p) getNth(p,0)
```


リストへのアクセス

- 関数getNextは、最初の要素をとったリストを返す

```
AST *getNext(AST *p)
{
    if(p->op != LIST){
      fprintf(stderr,"bad access to list¥n");
      exit(1);
    }
    else return p->right;
}
```

- 以下のようにして要素を順次アクセスする

```
AST *list,x;
for(list = ...; list != NULL; list = getNext(list)){
     x = getFirst(list); /* 要素の取り出し */
     xについての処理
}
```


ASTのコード

- PLUS_OP, MINUS_OPなどの演算子の他に、IF_STATEMENTなどの文のためのコードが定義しておく
- 前は、#defineで定義したが、enumを使っておけば、デバックに便利である

```
enum code {
    LIST,
    NUM,
    SYM,
    EQ_OP,
    PLUS_OP,
    MINUS_OP,
    MUL_OP,
    LT_OP,
    GT_OP,
    GET_ARRAY_OP,
    SET_ARRAY_OP,
    CALL_OP,
    PRINTLN_OP,
    IF_STATEMENT,
    BLOCK_STATEMENT,
    RETURN_STATEMENT,
    WHILE_STATEMENT,
    FOR_STATEMENT
};
```


シンボル構造体

- シンボルは同じの名前のシンボルを１つのデータ構造で管理するもので、以下の様に定義する
  - nameは、シンボルの名前である。
  - 他のメンバーについては、あとで使うときに説明する。

```
typedef struct symbol {
    char *name;
    int val;
    AST *func_params;
    AST *func_body;
} Symbol;
```

シンボルテーブル

- シンボル構造体を使って、シンボルテーブル、すなわち表にして管理する
- 同じ名前(識別子)は同じ構造体で管理するが、それを見付けるためにこのプログラムでは、単純サーチを使っている。大域変数n_symbolは、その数を数える変数である

```
Symbol SymbolTable[MAX_SYMBOLS];

int n_symbols = 0;
```


シンボルの生成検索

- 関数lookupSymbolは、名前を引数にして、それに対応するシンボル構造体を返す
- もしも、名前のシンボルがなかったら、それに対するシンボルを作る。

```
Symbol *lookupSymbol(char *name)
{
    int i;
    Symbol *sp;

    sp = NULL;
    for(i = 0; i < n_symbols; i++){
      if(strcmp(SymbolTable[i].name,nam
           == 0){
          sp = &SymbolTable[i];
          break;
      }
    }
    if(sp == NULL){
      sp = &SymbolTable[n_symbols++];
      sp->name = strdup(name);
    }
    return sp;
}
```


シンボルとAST

- シンボルを表すASTは、opがSYMで、symにシンボルへのポインタをいれたもの
- 関数makeSymbolは名前を与えて、それに対応するASTを作る
- 逆に、シンボルのASTのノードから、シンボルを取り出すのが関数getSymbol

```
AST *makeSymbol(char *name)
{
    AST *p;

    p = (AST *)malloc(sizeof(AST));
    p->op = SYM;
    p->sym = lookupSymbol(name);
    return p;
}

Symbol *getSymbol(AST *p)
{
    if(p->op != SYM){
      fprintf(stderr,"bad access t
      exit(1);
    }
    else return p->sym;
}
```