情報科学実験II 第4回

コンパイルの4つのステップ

1、字句解析(lexical analysis)
2、構文解析(syntax analysis)
3、意味解析(semantic analysis)
4、コード生成(code generation) ⇐ ここに関係する話

目標
・簡単なプログラミング言語のインタプリタを作る
・コンパイラとインタプリタの違いを理解する

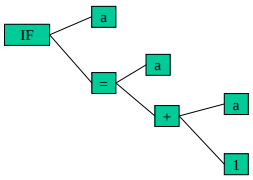
C言語風の簡単なインタプリタの例

算術式の評価では、構文解析の過程で同時に部分式の値が決まり、その値を返すことができるが、一般の文では「式の評価」ではなく「文の実行」が必要である。一般の文を扱うには、構文解析をしながらまず構文木を構築し、その後葉から順に実行をしていくという方法をとる。このような処理系をインタプリタという。

インタプリタにおける解析木の生成とその実行

例） if (a) a=a+1;

構文解析のプロセスを考えてみよ。



解析木は葉から生成し、実行は根から行う。実行文に複文や入れ子が含まれる(実行文にさらにIF文があるなど)場合、解析木はあらかじめ全て展開され、葉に近い方の演算要素から順に実行される。

インタプリタのしくみ

評価ではなく木の生成をする

・yy.lex.cのyylex()からトークンを受け取って、y.tab.cのyyparse内で構文解析を行い構文木を作るようにする。

・各演算要素（例えば前頁のIF）に対応する動作を「インタプリタを記述している言語（オブジェクト言語）」のプログラムとして記述する。つまり、ソース言語の条件式の評価や実行文の実行をその演算要素の中の条件式や実行文に置き換えて全て記述する。

・IFの中に他の文がある（例えば、IF文の実行文にIF文やWHILE文がある）場合の動作は、再帰呼び出しで実現する。

プログラムの実行法・・・

・構文木をトラバースし、演算要素に応じた処理を再帰的に実行していく。

C言語風の小さなインタプリタ

共通ヘッダファイル tci.h

```
typedef enum {typeCon, typeId, typeOpr} nodeEnum;

typedef struct{
  nodeEnum type;
  int value;
}conNodeType;

typedef struct{
  nodeEnum type;
  int i;
} idNodeType;

typedef struct{
  nodeEnum type;
  int oper;
  int nops;
  union nodeTypeTag *op[1];
}oprNodeType;

typedef union nodeTypeTag{
  nodeEnum type;
  conNodeType con;
  idNodeType id;
  oprNodeType opr;
}nodeType;
```

構文木のノードの種類を区別する列挙型 nodeEnumの定義。これによりそのノードが定数か識別子か演算要素かを区別する。ここでは、IFや+や=などを全て演算要素としてみているので注意。

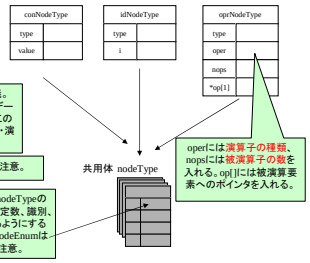
構造体 conNodeTypeの定義。ノードが定数の場合のデータ構造。

構造体 idNodeTypeの定義。ノードが識別子の場合のデータ構造。

構造体 oprNodeTypeの定義。ノードが演算要素の場合のデータ構造。4番目のメンバは、この中で実行される定数・識別子・演算要素へのポインタ

型名と構造体名は違う点に注意。

共用体 nodeTypeTag型のnodeTypeの定義。ノードの型に応じて、定数、識別、演算要素として読み分けるようにする仕掛け。どんな場合でも、nodeEnumは必ず先頭に来る点に注意。



operには演算子の種類、nopsには被演算子の数を入れる。op[]には被演算要素へのポインタを入れる。

tci.l

```
%{
#include <stdlib.h>
#include "tci.h"
#include "y.tab.h"
void yyerror(char *s);
%}

%%

[a-z] { yylval.vIndex = *yytext - 'a';
        return VARIABLE;
      }

[0-9]+ { yylval.iValue = atoi(yytext);
         return INTEGER;
       }

[-()=+*/;{}.¥[¥]] { return *yytext; }

"if" return IF;
"else" return ELSE;
"printf" return PRINTF;

[ ¥t¥n]+ ;

. yyerror("Bad character");

%%

int yywrap(void){ return 1; }
```

yytextの先頭要素の値からaのASCIIコードを引いてyylvalのメンバvIndex(後述)に代入。これは、変数の格納されている配列上の添え字を求めて代入している点に注意。

yylvalを共用体としているので、yylval. vIndexにより変数に用いる配列の添え字が代入できる。vIndexはyaccのトークンの定義で記述する。

同様に、yylval.iValueを使って整数定数を取り込める。

演算子、区切り記号、各括弧などはそのまま返す

入力の終端で呼び出される関数

tci.y

定義部

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "tci.h"

nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
int exe(nodeType *p);
void yyerror(char *s);

int var[65536];
%}

%union{
 int iValue; /* integer value */
 char vIndex; /*  variable table index*/
 nodeType *nPtr; /* node pointer */
};

%token <iValue> INTEGER
%token <vIndex> VARIABLE
%token IF PRINTF
%token <nPtr>
%nonassoc ELSE

%left '+' '-'
%left '*' '/'
%right UMINUS

%type <nPtr> stmt expr stmt_list
```

引数の数が可変の関数を扱うためのヘッダファイル。oprで使う。

opr、id、conはそれぞれ演算要素、識別子、定数をyylexから受け取った場合にcallされ、ノードを作成し、演算などを格納し、そのポインタを返す。返すポインタはnodeType型へのポインタ。oprは可変引数の関数である点に注意。

木構造を作る関数

exeは、木の根へのポインタを引数として、木を解釈しながら実行する関数。

実際に使う変数はaからzまで。

%unionで、yylvalを共用体として宣言できる。yylvalの読み替えに対応。

yylvalの共用体のメンバ、<iValue>などによって、トークンの型を指定できる

%left、%rightで左右の結合性を指定できる。%nonassocは結合性無しの指定。

%typeで非終端記号の型を指定。stmt expr stmt_listはノードへのポインタ。

tci.y

ルール部

```
%%

program:
function {exit(0); }
;

function:
function stmt {exe($2); freeNode($2);}
| /* NULL */
;

stmt:
';' {$$=opr(';', 2, NULL, NULL);}
| expr ';' {$$ = $1;}
| PRINTF expr ';' {$$ = opr(PRINTF, 1, $2);}
| VARIABLE '=' expr ';' {$$ = opr('=', 2, id($1), $3);}
| IF '(' expr ')' stmt { $$ = opr(IF, 2, $3, $5); }
| IF '(' expr ')' stmt ELSE stmt {$$ = opr(IF, 3, $3, $5, $7); }
| '{' stmt_list '}' {$$ = $2;}
;

stmt_list:
stmt {$$ = $1;}
| stmt_list stmt {$$ =opr(';', 2 ,$1, $2);}
;

expr:
INTEGER   {$$ = con($1);}
|VARIABLE {$$ = id($1);}
|'-' expr %prec UMINUS {$$ = opr(UMINUS, 1, $2); }
|expr '+' expr {$$ = opr('+', 2, $1,$3);}
|expr '-' expr {$$ = opr('-', 2, $1,$3);}
|expr '*' expr {$$ = opr('*', 2, $1,$3);}
|expr '/' expr {$$ = opr('/', 2, $1,$3);}
| '(' expr ')' { $$ = $2;}
;
```

C言語的な構文。
「programはfunctionである」

$2には実行すべきノードへのポインタが返るので、それを引数としてexeをcallする。ノードの実行が終わったらfreeNodeによりメモリを開放。exeする前に、構文解析（木の生成）は終わっている点に注意。

「;」も一応opr扱い。演算対象はNULL。

右辺が合致したら、oprをcallしてノードを作成する（演算要素の場合）。$$には、作成したノードへのポインタが返る。

代入文のノードを作成。

IFのノードを作成。

IF ELSEのノードを作成。

複文の左再帰的定義

定数の場合はcon、識別の場合はidをそれぞれcallしてノードを作成。$$にはノードへのポインタが返る。

%precは単項演算子を最優先で結合することを指定する。

---

tci.y

Cコード部

```
%%

nodeType *con(int value){
  nodeType *p;

  if((p = malloc(sizeof(conNodeType))) == NULL)
    yyerror("out of memory");

  p->type = typeCon;
  p->con.value = value;
  return p;
}

nodeType *id(int i){
  nodeType *p;

  if((p = malloc(sizeof(idNodeType))) == NULL)
    yyerror("out of memory");

  p->type = typeId;
  p->id.i = i;

  return p;

}
```

トークンが定数の場合、yyparseはconをcallする。conでは、conNodeTypeからノードのサイズを求め、ヒープからメモリを切り出しそのポインタをpに得る。pが指すオブジェクトのメンバ'typeには ノードの型typeConを、メンバ'con.valueには定数値を格納した状態でpを返す。

p

conNodeType	
type	typeCon
value	value

トークンが識別子の場合、yyparseはidをcallする。idでは、idNodeTypeからノードのサイズを求め、ヒープからメモリを切り出しそのポインタをpに得る。pが指すオブジェクトのメンバ'typeにはノードの型typeIdを、メンバ'con.valueには変数を格納している配列の添え字を格納した状態でpを返す。

p

idNodeType	
type	typeId
i	i

---

tci.y

Cコード部の続き

```
nodeType *opr(int oper, int nops, ...){
  va_list ap;
  nodeType *p;
  size_t size;
  int i;

  size =sizeof(oprNodeType) + (nops -1) * sizeof(nodeType*);
  if((p = malloc(size)) == NULL)
    yyerror("out of memory");

  p->type =typeOpr;
  p->opr.oper = oper;
  p->opr.nops = nops;
  va_start(ap, nops);
  for(i = 0 ; i < nops; i++)
    p->opr.op[i] = va_arg(ap,nodeType*);
  va_end(ap);
  return p;
}
```

トークンが演算要素の場合、yyparseはoprをcallする。oprでは、その先にある演算の数が不定（つまり、文の中に文がいくつ含まれるかは分からない）なので、conNodeTypeからだけではノードのサイズを求められないが、第二引数のオペランドの数nopsを使って、ノードのサイズをそのつど求めてmallocしている。pが指すオブジェクトのメンバ'typeにノードが演算であることを示すtypeOprを、メンバ'oprのoperに演算の種類を、nopsには続く演算数を格納し、その演算要素を表すノードへのポインタを配列op[i]に順に格納する。

oprは引数の数が可変なので、受け取った引数をva_argで読み出す。nopsに続く引数をnodeType型へのポインタとして順に読み出す。あらかじめ、va_startで初期化し、終わったらva_endをcallする。

p

oprNodeType	
type	typeOpr
oper	oper
nops	nops
*op[0]	
*op[1]	
*op[nops-1]	

葉から構文木を作る。葉にはoprはなく、定数か変数しかない点に注意。

---

tci.y

Cコード部の続き

```
void freeNode(nodeType *p){
  int i;

  if (!p) return;
  if(p->type == typeOpr){
    for(i = 0; i< p-> opr.nops; i++)
      freeNode(p->opr.op[i]);
  }
  free(p);
}
```

ノードの実行が終わったら、freeNodeによりメモリを開放する。ノードへのポインタpを引数として、メンバ'typeが演算の場合は確保した演算数だけ開放を再帰的に繰り返す。最後にオブジェクトそのものを開放してreturnする。

p

oprNodeType	
type	typeOpr
oper	oper
nops	nops
*op[0]	
*op[1]	
*op[nops-1]	

再帰的にfreeNodeをcall

---

tci.y

Cコード部の続き

```
int exe(nodeType *p){
  if(!p) return 0;
  switch(p->type){
  case typeCon:
    return p->con.value;
  case typeId:
    {
      return var[p->id.i + 1];
    }
  ･･･つづく
```

exeは、ノードへのポインタを引数として木構造を解釈実行する。

ノードの型によりスイッチ

定数ノードならメンバ'conのvalueをそのまま返す

識別子ノードならメンバ'conのiに1加えた数を添え字として配列の値を返す

conNodeType	
type	typeCon
value	value

idNodeType	
type	typeId
i	i

---

tci.y

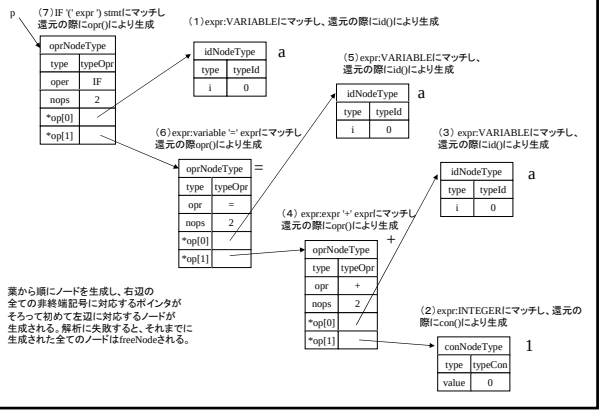
Cコード部の続き

```
case typeOpr:
  switch(p->opr.oper){
  case IF: if( exe(p->opr.op[0]))
             exe(p->opr.op[1]);
           else if( p-> opr.nops >2)
             exe(p->opr.op[2]);
    return 0;
  case PRINTF: printf("%d¥n", exe(p->opr.op[0])); return 0;
  case ';': exe(p->opr.op[0]); return exe(p->opr.op[1]);
  case '=':return var[p->opr.op[0]->id.i +1] = exe(p->opr.op[1]);

  case UMINUS: return -exe(p->opr.op[0]);
  case '+':
    {
      return exe(p->opr.op[0]) + exe(p->opr.op[1]);
    }
  case '-': return exe(p->opr.op[0]) - exe(p->opr.op[1]);
  case '*': return exe(p->opr.op[0]) * exe(p->opr.op[1]);
  case '/': return exe(p->opr.op[0]) / exe(p->opr.op[1]);
  }
 }
 return 0;
}

void yyerror(char *s){
  fprintf(stdout, "%s¥n",s);
}

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

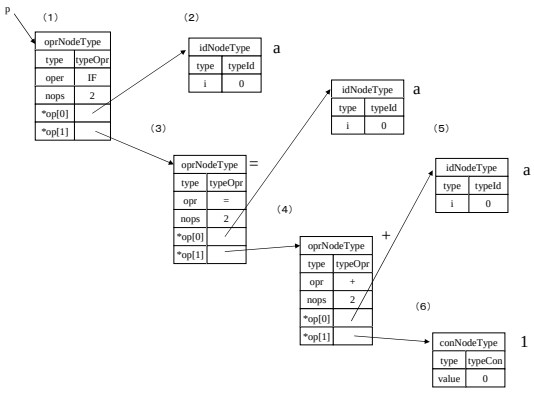
演算ノードなら演算の種類によりさらに分岐

IFの場合は、メンバ'oprのop[0]つまり条件文を引数としてexeをcallし、その後oprのop[1]つまり実行文を引数としてexeをcall。"if文で書かれている条件文と実行文を抜き出して、C言語のif文として実行している"という点に注意せよ。elseの場合は、演算数nopsをチェックしてもしそれが2を超えるならelseつきと判断する。以下同様に、演算の種類に応じて、各文やオペランドへのポインタを引数としてexeをcallし結果をreturnする。

解析木とその生成順 例） if (a) a=a+1;



解析木のノードが実行(exeされる)順序 例） if (a) a=a+1;



Makefile

インタプリタinterを作る
Makefile

```
#
# Makefile for Tiny C Interpreter
#

CC = gcc
YACC = bison -y
LEX = flex

CFLAGS = -Wall -g


CLEANS = inter y.tab.c lex.yy.c *~
OBJI = inter.o y.tab.o lex.yy.o
SRCS = inter.c tci.y tci.l tci.h

all: inter

inter:y.tab.c lex.yy.c
	$(CC) $(CFLAGS) lex.yy.c y.tab.c -o inter

y.tab.c:tci.y tci.h
	$(YACC) -d tci.y

lex.yy.c: tci.l tci.h
	$(LEX)tci.l

y.tab.h: tci.y
	$(YACC) -d tci.y

clean:
	-rm $(CLEANS)
```

実行例

```
[tanaka@minerva tci]$ yacc -d tci.y
yacc: 1 shift/reduce conflict
[tanaka@minerva tci]$ make all
flex tci.l
gcc -Wall -g lex.yy.c y.tab.c -o inter
lex.yy.c:1022: 警告: 'yyunput' defined but not used
tci.y: In function 'main':
tci.y:168: 警告: implicit declaration of function 'yyparse'
y.tab.c: In function 'yyparse':
y.tab.c:392: 警告: implicit declaration of function 'yylex'
[tanaka@minerva tci]$ ls
Makefile  inter  lex.yy.c  sample  tci.h  tci.l  tci.y  y.tab.c  y.tab.h
[tanaka@minerva tci]$ inter
a=1;
if(1) a=a+1;
printf(a);
2
[tanaka@minerva tci]$
```

最初に一回だけyacc -dを実行してy.tab.hを作る

まとめ

- 構文解析で木構造を作り、順に実行するのがインタプリタ、実行するかわりに等価なオブジェクトプログラムを生成するのがコンパイラである。そこでは再帰的な実行のかわりに再帰的な生成が行われる。UNIX上のfortranやpascalのコンパイラも、実際はC言語で記述されたオブジェクトプログラムを生成している。新しいプログラミング言語に関するアイデアがあれば、yaccとlexを用いて比較的簡単にインタプリタやコンパイラを作ることが出来る。

課題1（インタプリタ)

まず、インタプリタのソースを熟読し、データ構造と動作を完璧に理解せよ。
続いてインタプリタに以下の機能を追加せよ。

1) 関係演算子 <、 >、 <=、 >=、==、 !=

これにより、 if(a !=3) b=c; といった記述が出来るようになる。

2) 繰り返し while for
例） while(i != 10) { i=i+1; printf(i); }
例） for(i=1; i < 10 ; i=i+1;){printf(i); }

ソース全体に詳しいコメントをつけること。

課題2（インタプリタ)

課題1の2)のfor文について、先の例にならって解析木を描いてみよ。また、各ノードがexeにより実行される順序をトレースしてみよ。

複文(statement_list)の場合の木の生成と実行の順序はどうなっているか述べよ。

※今回の課題には考察や図の作成が含まれるので、pdfもしくはwordで提出すること。

課題3（木構造の出力)

先のイタプリタにおいて、exeをcallする直前に実行しようとするstatementの木構造を以下のように出力する機能を追加せよ。

例） if (a) a=a+1; ⇨ (IF, a,(=, a, (+, a, 1)))