プログラミング言語処理
講義資料

ver 1.1 (2007) by msato

目次

付録：　tiny C ソースコード

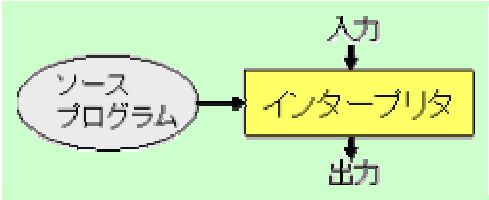
プログラミング言語処理

１．　言語処理系とは

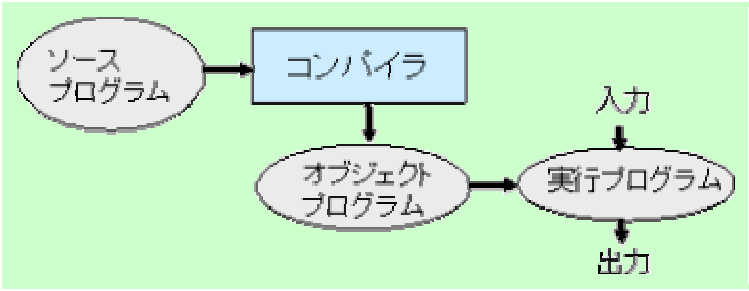
インタプリタとコンパイラ

言語処理系とは、プログラミング言語で記述されたプログラムを計算機上で実行するためのソフトウエアである。そのための構成として、大別して２つの構成方法がある。

- *インタプリタ（interpreter,通訳系）*： 言語の意味を解析しながら、その意味する動作を実行する。



- *コンパイラ（compiler,翻訳系）* 言語を他の言語に変換し、その言語の プログラムを計算機上で実行させるもの。狭い意味でコンパイラは、言語を機械語に変換し、実行するものであるが、他の言語、あるいは仮想機械コードに 変換するものもコンパイラと呼ぶ。他の言語に変換するときには、特に translator と呼ぶ場合もある。



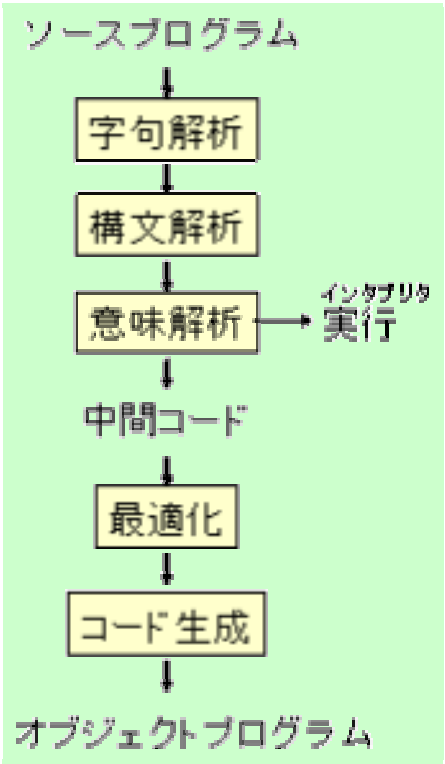
元のプログラムを*ソースプログラム*、 翻訳の結果と得られるプログラムを*オブジェクトプログラム*と呼ぶ。 機械語で直接、計算機上で実行できるプログラム を*実行プログラム*と呼ぶ。オブジェクトプログラムがアセンブリプログラムの 場合には、アセンブラにより機械語に翻訳されて、実行プログラムを得る。他 の言語の場合には、オブジェクトプログラムの言語のコンパイラでコンパイルすることにより、実行プログラムが得られる。仮想マシンコードの場合には、 オブジェクトコードはその仮想マシンにより、インタプリトされて実行される。

言語処理系の基本構成

コンパイラにしてもインタプリタにしても、その構成は多くの共通部分を持 つ。すなわち、ソースプログラムの言語の意味を解釈する部分は共通である。 インタプリタは、解釈した意味の動作をその場で実行するのに対し、コンパ イラではその意味の動作を行うコードを出力する。

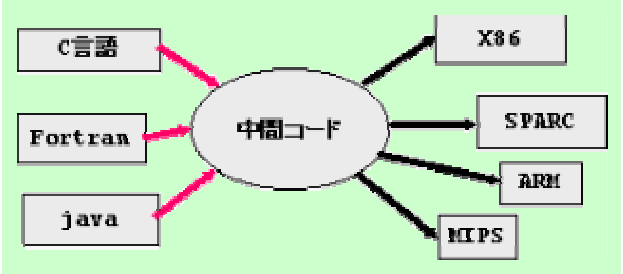
言語処理系は、大きく分けて、次のような部分からなる。

1. 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。
2. 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。
3. 意味解析(semantics analysis): 構文木の意味を解析する。インタプリタでは、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。
4. 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。
5. 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。



コンパイラの性能とは、如何に効率のよいオブジェクトコードを出力できるかであり、最適化でどのような変換ができるかによる。インタプリタでは、プログラムを実行するたびに、字句解析、構文解析を行うために、実行速度はコンパイラの方が高速である。もちろん、機械語に翻訳するコンパイラの場合には直接機械語で実行されるために高速であるが、コンパイラでは中間コードでやるべき操作の全体を解析することができるため、高速化が可能である。

また、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。



例題：式の評価

さて、例として最も簡単な数式の評価について、インタプリタとコンパイラを作ってみることにする。目的は,

```
12 + 3 - 4
```

の式の入力に対し、この式を計算し、

```
11
```

と出力するプログラムを作ることである。これは、式という「プログラミング言語」を処理する言語処理系である。「式」という言語では、token として、数字と"+"や"-"といった演算子がある。

まずは、字句解析ではこれらのトークンを認識する。例えば、上の例では、

```
１２の数字、＋の演算子、３の数字、－の演算子、４の数字、終わり
```

という列に変換する。

token は、token の種類と 12 の数字という場合の 12 の値の 2 つの組で表される。以下に token の種類を定義する exprParser.h を示す。

```
#define EOL 0
#define NUM 1
#define PLUS_OP  2
#define MINUS_OP 3

extern int tokenVal;
extern int currentToken;
```

exprParser.h（一部）

字句解析を行う関数 getToken を示す。

```
void getToken()
{
    int c,n;
again:
    c = getc(stdin);
    switch(c){
    case '+':
         currentToken = PLUS_OP;
         return;
    case '-':
         currentToken = MINUS_OP;
         return;
    case '¥n':
         currentToken = EOL;
         return;
```

```
    }
    if(isspace(c)) goto again;
    if(isdigit(c)){
        n = 0;
        do {
            n = n*10 + c - '0';
            c = getc(stdin);
        } while(isdigit(c));
        ungetc(c,stdin);
        tokenVal = n;
        currentToken = NUM;
        return;
    }
    fprintf(stderr,"bad char '%c'¥n",c);
    exit(1);
}
```

getToken.c

この関数は、字句を読み込み、currentToken に token の種類、NUM の場合に tokenVal に値を返す。これを使うことにより、いわゆる構文解析しなくても、直接実行する（計算してしまう）インタプリタは簡単にできる。その動作は以下のような動作である。

1. 現在の結果を変数 result に覚えておく。また、直前の演算子を変数 op に覚えておく。
2. 関数 getToken を呼んで、数字であれば、現在の結果と今の数字の値との計算を行う。但し、最初の数字（まだ、op がない）の場合には、現在の結果に入力された数字を格納する。
3. 終わりがきたら、現在の数字を出力する。

これが、いわゆる電卓のアルゴリズムである。（この電卓の欠点を考えてみよ！）

```
main()
{
    int op;
    int result;

    op = NUM;
    result = 0;
    while(1){
        getToken();
        switch(currentToken){
        case NUM:
            switch(op){
            case NUM:
                result = tokenVal;
                break;
            case PLUS_OP:
                result = result + tokenVal;
                break;
            case MINUS_OP:
                result = result - tokenVal;
                break;
```

```
            }
            break;
        case PLUS_OP:
        case MINUS_OP:
            op = currentToken;
            break;
        case EOL:
            printf("result = %d¥n",result);
            exit(0);
        }
    }
}
```

cal.c

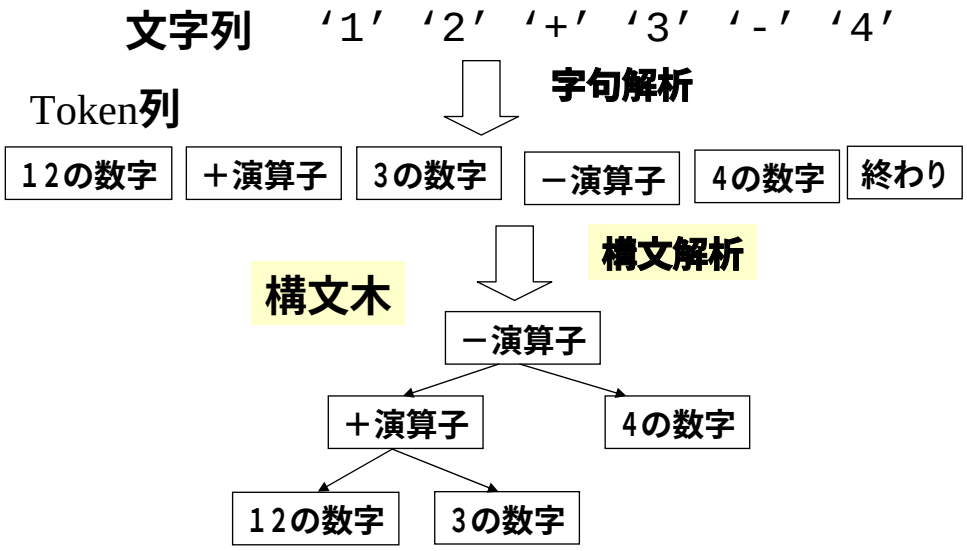
ＢＮＦと構文木

では、この「式」というプログラミング言語の構文とはどのようなものであろうか。例えば、次のような規則が構文である。

```
足し算の式　:=　式　＋の演算子　式
引き算の式　:=　式　－の演算子　式
式　:= 数字　|　足し算の式　|　引き算の式
```

このような記述を、*BNF (Backus Naur Form または Buckus Normal Form)* という。

このような構造を反映するデータ構造を作るのが、構文解析である。図に示す。



構文解析のデータ構造は、以下のような構造体を作る。これを exprParser.h に定義しておく。

```
typedef struct _AST {
    int op;
    int val;
    struct _AST *left;
    struct _AST *right;
} AST;

AST *readExpr(void);
```

exprParser.h（一部）

この構文木を作るプログラムが、readExpr.c である。このプログラムでは、exprParser.h で定義されている AST を使って、構文木を作っている。このデータ構造は式の場合は、演算子とその左辺の式と右辺の式を持つ。数字の場合はこれらを使わずに値のみを格納する。token を読むたびに、データ構造を作っている。

```
AST *readExpr()
{
    AST *e,*ee;

    e = readNum();
    while(currentToken == PLUS_OP || currentToken == MINUS_OP){
        ee = (AST *)malloc(sizeof(AST));
        ee->op = currentToken;
        getToken();
        ee->left = e;
        ee->right = readNum();
        e = ee;
    }
    return e;
}

AST *readNum()
{
    AST *e;
    if(currentToken == NUM){
        e = (AST *)malloc(sizeof(AST));
        e->op = NUM;
        e->val = tokenVal;
        getToken();
        return e;
    } else {
        fprintf(stderr,"bad expression: NUM expected¥n");
        exit(1);
    }
}
```

readExpr.c

解釈実行：インタプリタ

この構文木を解釈して実行する、すなわちインタプリタをつくってみることにする。その動作は、

1. 式が数字であれば、その数字を返す。
2. 式が演算子を持つ演算式であれば、左辺と右辺を解釈実行した結果を、演算子の演算を行い、その値を返す。

このプログラムが evalExpr.c である。evalExpr.c は、構文木 AST を解釈して、解釈する。

1. 数字の AST つまり、op が NUM であれば、その値を返す。
2. 演算式であれば、左辺を評価した値と右辺を評価した値を op に格納されている演算子にしたがって、計算を行う。

これらは再帰的に呼び出しが行われていることに注意しよう。

```
int evalExpr(AST *e)
{
    switch(e->op){
    case NUM:
         return e->val;
    case PLUS_OP:
         return evalExpr(e->left)+evalExpr(e->right);
    case MINUS_OP:
         return evalExpr(e->left)-evalExpr(e->right);
    default:
         fprintf(stderr,"evalExpr: bad expression¥n");
         exit(1);
    }
}
```

evalExpr.c

main プログラムでは、関数 readExpr を呼び、構文木を作り、それを関数 evalExpr で解釈実行して、その結果を出力する。これが、インタプリタである。先のプログラムと大きく違うのは、式の意味を表す構文木が内部に生成されていることである。この構文木の意味を解釈するのがインタプリタである。(readExpr では１つだけ先読みが必要であるので、getToken を呼び出している)

```
int main()
{
    AST *e;
    getToken();
    e = readExpr();
    if(currentToken != EOL){
         printf("error: EOL expected¥n");
         exit(1);
    }
    printf("= %d¥n",evalExpr(e));
    exit(0);
}
```

interpreter.c

コンパイラとは

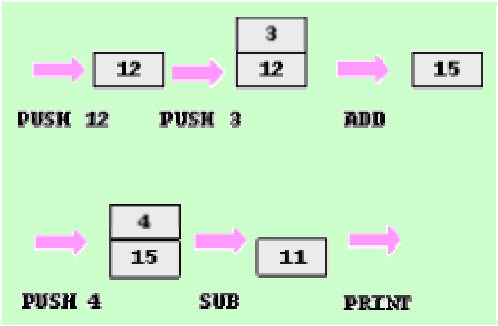
次にコンパイラをつくってみる。コンパイラとは、解釈実行する代わりに、実行すべきコード列に変換するプログラムである。実行すべきコード列は、通常、アセンブリ言語（機械語）であるが、そのほかのコードでもよい。中間コードとして、スタックマシンのコードを仮定することにする。スタックマシンは以下のコードを持つことにする。

- PUSH n ：　数字 n をスタックに push する。
- ADD ：　スタックの上２つの値を pop し、それらを加算した結果を push する。
- SUB ：スタックの上２つの値を pop し、減算を行い、push する。
- PRINT: スタックの値を pop し、出力する。

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。例えば、図で示した例の式 12+3-4 は下のようなコードになる。

```
PUSH 12
PUSH 3
ADD
PUSH 4
SUB
PRINT
```

スタックマシンでの実行は以下のように行われる。



stackCode.h には、コードとその列を格納する領域を定義してある。

```
#define PUSH 0
#define ADD 1
#define SUB 2
#define PRINT 3

#define MAX_CODE 100

typedef struct _code {
    int opcode;
    int operand;
} Code;

extern Code Codes[MAX_CODE];
extern int nCode;
```

stackCode.h

コンパイルの手順は、以下のようになる。

1. 式が数字であれば、その数字を push するコードを出す。
2. 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。
3. 式のコンパイルしたら、PRINT のコードを出しておく。

この中間コードを生成するのが、compileExpr.c である。構文木を入力して、再帰的に上のアルゴリズムを実行する。コードは Codes という配列に格納しておく。

```
void compileExpr(AST *e)
{
    switch(e->op){
    case NUM:
         Codes[nCode].opcode = PUSH;
         Codes[nCode].operand = e->val;
         break;
    case PLUS_OP:
         compileExpr(e->left);
         compileExpr(e->right);
         Codes[nCode].opcode = ADD;
         break;
    case MINUS_OP:
         compileExpr(e->left);
         compileExpr(e->right);
         Codes[nCode].opcode = SUB;
         break;
    }
    ++nCode;
}
```

compileExpr.c

コード生成では、ここではスタックマシンのコードを C に直して出力することにしよう。C で実行させるために、main にいれておくことにする。このプログラムが、codeGen.c である。

```
void codeGen()
{
    int i;
    printf("int stack[100]; ¥nmain(){ int sp = 0; ¥n");
    for(i = 0; i < nCode; i++){
         switch(Codes[i].opcode){
         case PUSH:
             printf("stack[sp++]=%d;¥n",Codes[i].operand);
             break;
         case ADD:
             printf("sp--; stack[sp-1] += stack[sp];¥n");
             break;
```

```
        case SUB:
            printf("sp--; stack[sp-1] -= stack[sp];¥n");
            break;
        case PRINT:
            printf("printf(¥"%%d¥¥n¥",stack[--sp]);¥n");
            break;
        }
    }
    printf(")¥n");
}
```

codeGen.c

コンパイラの main プログラムであるが、readExpr まではインタプリタと同じである。標準出力に出力されるプログラムに適当に名前をつけ（たとえば、output.c）これを C コンパイラでコンパイルして実行すればよい。（assembler のファイルの場合は as コマンドでコンパイルする。）

```
int main()
{
    AST *e;

    getToken();
    e = readExpr();
    if(currentToken != EOL){
        printf("error: EOL expected¥n");
        exit(1);
    }
    nCode = 0;
    compileExpr(e);
    Codes[nCode++].opcode = PRINT;
    codeGen();
    exit(0);
}
```

codeGen.c

電卓のプログラムに比べて、構文木を作るなど、ずいぶん遠周りをしたようであるが、その理由は演算の優先度や、括弧の式など、通常の数学で使われる式を正しく処理するためである。例えば、

```
12*3 + 3*4
```

の場合には、掛け算を最初にして、それらを加算しなくてはならない。この処理を反映した構文木を作ることによって、正しく処理する「言語処理系」を作ることができるようになる。

ソースコード　（付録参照）

- 式のインタプリタ

  - exprParser.h
  - interpreter.c
  - evalExpr.c
  - getToken.c
  - readExpr.c
  - コンパイルの方法

```
% cc -o interpreter interpreter.c evalExpr.c getToken.c readExpr.c
```

  - 実行の方法
    1. ファイル(input)に式を書いておく。
    2. それを標準入力にして実行

```
% ./interpreter < input
```

- 式のコンパイラ
  - exprParser.h
  - compiler.c
  - compileExpr.c
  - codeGen.c
  - getToken.c
  - readExpr.c
  - コンパイルの方法

```
% cc -o compile compile.c compileExpr.c codeGen.c getToken.c readExpr.c
```

  - 実行の方法
    1. ファイル(input)に式を書いておく。
    2. それを標準入力、標準出力として output.c に出力

```
% ./compile < input > output.c
```

    3. output.c をコンパイル

```
% cc output.c
```

    4. 実行

```
% a.out
```

- Makefile

プログラミング言語処理

２．字句解析の基礎：正規表現によるパターンマッチ

字句解析とは、文字列として入力されるプログラムを token の列に分解するフェーズである。前回のプログラムにおいて、字句解析を行う関数 getToken は、文字列を数字(NUM)、演算子(PLUS_OP,MINUS_OP)などの token に分けて、返す関数である。

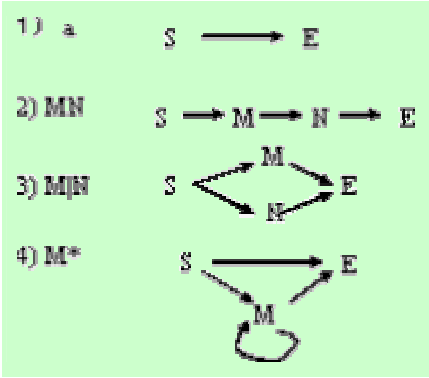
字句解析と正規表現

どのような文字列がどのような token になるかについては、*正規表現(regular expression)*で定義することができる。アルファベット A 上の正規表現とは、

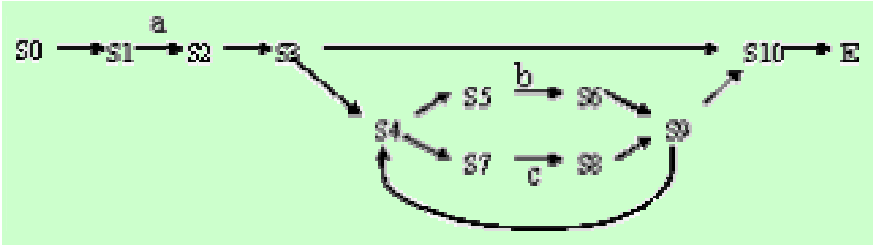
1. ε(空列記号)は正規表現である。
2. A の要素 a は正規表現である。
3. R と S が正規表現であれば、M|N、MN,M*は正規表現である。M|N とは、M もしくは N、MN は M の次に N がくる列、M*とは、M の 0 回以上繰り返しを意味する。

なお、（S）は、S と同等であることを意味する。

例えば、正規表現 a(b|c)*は、最初に a があり、b または c が続く文字列を表現する。abbc も abcbcc も、abcc もこの正規表現で表現される文字列である。



正規表現は、図のような規則で*非決定性有限オートマトン（NDA:nondeterministic finite automaton）* に変換できる。 *有限オートマトン（finite automaton）*とは、有限の内部状態を持ち、与えられた記号列を読みこみながら状態遷移し、その記号列があるパターンをもつ列であるかを決定するものである。非決定性有限オートマトンとは、入力によらない状態遷移（空列記号に対する状態遷移）をもち、それは非決定的に遷移してもしなくてもよいと状態をもつものである。図にそれぞれの規則に対応するオートマトンを図示する。



先にあげた a(b|c)*について、この規則で、NDA に変換した結果も示す。

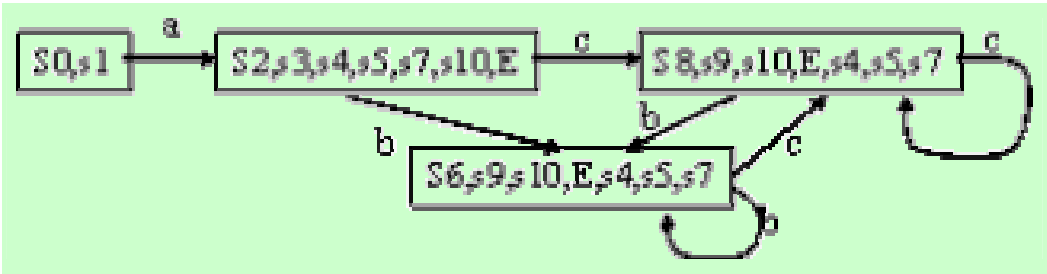
NDA では、空の状態遷移に対して、状態遷移するかしないかの両方の可能性をしらべなくてはならないので、実際にそのまま実装すると効率がわるい。そのため、非決定的な遷移を取り除き、*決定性有限オートマトン(DFA:deterministic finite automaton)* に変換する。DFA とは、以下の条件を満たす有限オートマトンである。

1. ε による遷移がない。
2. 一つの状態から同じ記号による異なった状態への遷移はない。

変換には次の規則を適用し、上の条件にあったオートマトンに変換する。

1. 初期状態から、ε による遷移を一まとめにした集合を初期状態とする。
2. 状態の集合からの遷移は、その集合からの遷移の集合の合併とする。つまり、状態の集合 D={x1,x2,...}からの a による遷移の行き先は、x1 から a で遷移した状態 y もしくは、y から ε で遷移する状態の集合になる。
3. ２を繰り返し新しい遷移が得られなくなるまで繰り返す。

このアルゴリズムで得られる FDA は必ずしも、最小のオートマトンとはならない。最小にするには、同じ遷移が２つあった場合には、冗長なので１つにまとめることができ、これを繰り返すことにより、最小化ができる。



自動字句解析生成プログラム：lex

正規表現が与えられた時に、その言語（文字のパターン）を認識する DFA を機械的に作り出すことができる。そのアルゴリズムをプログラムにしたものが字句解析器生成系(lexical analyzer generator)である。このプログラムとして、有名なのが、lex である。lex では、定義ファイルは以下の形式でかく。

```
%{
任意の C プログラム。定数や C のマクロの宣言をここにかく。
%}
下の定義で使う lex のマクロの定義
%%
正規表現による入力パターンの定義
正規表現パターン　アクション　という形で書く
%%
任意の C プログラム
```

例えば、a(b|c)*の正規表現を認識する字句解析は、

```
%{
%}
%%
a(b|c)*  { printf("OK¥n"); }  /*このパターンを入力したら OK を出力する*/
.    {printf("NG¥n");}  /*.は以上以外のパターンの場合のアクションを指定 */
%%
/*   なにもなし  */
```

でよい。

これをコンパイルするには、

```
% lex test.l
% cc lex.yy.c -ll
```

とする。これでできた a.out を実行すると、abc にたいしては OK, xxx には NG と 出力される。
Linux の flex のときには、-ll の代わりに、-lfl をつかうこと。

前回示した数式の字句解析の部分は以下のように書ける。

```
%{
#include "expr.h"
%}
digit   [0-9]  /* マクロの定義, digit を 0-9 の数字の集合と定義する  */
%%
{digit}+  return NUM;    /*0-9 の１回以上の繰り返しは、NUM と認識する  */
"+"     return PLUS_OP; /*+は PLUS_OP +は繰り返しの意味なので、""で囲む*/
"-"     return MINUS_OP, /* -は MINUS_OP */
[ ¥t]                        /* 空白は無視  */
.       printf("error?¥n");   /* error */
%%
main(){
    yystdin = stdout;
    ....
    r = yylex(); /* token の列は yytext に入る  */
    printf(" token is %d,'%s'¥n",r,yytext);
}
```

lex は、字句解析ルーチンとして、yylex を生成する。このルーチンは、action で指定された return 文で返された値を返す。

lex の使い方については、

```
 % man lex
```

として、マニュアルを参照のこと。

プログラミング言語処理

３．数式の構文解析：top-down parser の作り方

構文規則とは

前章では、簡単な数式の処理系を解説した。構文は、以下のような BNF(Backus nur form, Backus normal form)による構文規則で記述される。

```
足し算の式　:= 式　＋の演算子　式
引き算の式　:= 式　－の演算子　式
式　:= 数字　| 足し算の式 |　引き算の式
```

ここで、構文の最終的な要素に現れるものを*終端記号(terminal symbol)*、それ以外のほかの構文規則によって定義される記号を*非終端記号(non-terminal symbol)*と呼ぶ。ここでは、非終端記号を<>で囲んで表すことにする。

1. 構文規則は,非終端記号<T>に対して、<T> := e (e は構文規則)で、表現される。これは、非終端記号<T>は、構文規則 e によって置き換えられることを意味する。
2. e は空でもよい。
3. e は、非終端記号、終端記号、もしくは e1 | e2、e1 e2、e1* のいずれか。 e1 | e2 は、e1 もしくは e2 であることを意味し、e1 e2　は e1 の次に e2 が現れる ことを意味し、e1* は、e1 の 0 個以上の繰り返しを意味する。

(...)は、構文規則のまとまりを示す。e1|e2|e3 は、((e1|e2)|e3)を、e1 e2 e3　は　((e1 e2) e3) と同じである。正規表現と似ていることを注意しよう。

ここでは、構文規則の左辺が、一つの終端記号<T>だけという文法を考える。これはすなわち、どのような場合でも<T> := e の規則を使って、右辺に置き換えることができることを意味し、このような文法を*文脈自由文法*とよぶ。この制限を取り払って、例えば、

```
e1 <T> e2 := ...
```

というような、e1 e2 の間に構文要素<T>が現れたときだけ、置き換えることができるというような文法が考えられるが、このような文法を*文脈依存文法*と呼ぶ。

top-down parser の作り方

構文規則に対し、構文解析を作る方法を紹介しよう。例えば、

```
<A> := a <B> c
```

という構文規則があったとすれば、<A>のための構文解析関数 readA は以下のように作ることができる。

```
readA() {
   a を読み込む;
   readB();      /* B を読み込むための関数を呼ぶ*/
   b を読み込む;
}
```

これは、これから読み込むものの形を先に仮定してしまってから、それに合致するかを調べていく構文解析法である。このような構文解析法を*再帰的下向き構文解析(recursive decent parsing)*あるいは *top-down parser* と呼ぶ。以前、解説した数式の構文解析もこの方法によるものである。

さて、前回の数式を再度考え、括弧や乗算をいれて考えてみることにする。構文規則を再度定義してみると、

```
<expression> := <expression> <expr_op> <term>
<term> := <term> <term_op> <factor>
<factor> := number | '(' <expression> ')'
<expr_op> := '+' | '-'
<term_op> := '*'
```

ここの定義で、加減算の優先順位を考慮して、生成される構文木を反映して構 文規則が作られていることに注目。しかし、これを上の方法で書くと

```
readExpr() {
    readExpr();
    readExprOp();
    readTerm();
}
```

となってしまって、readExpr が再帰的に呼ばれて，うまく行かない。この問題を、*top-down parser の　左再帰性の問題* と呼ばれている。 すなわち、最終的に

```
 <T> := <T> e
```

となる、文法規則ではうまくいかないのである。 このために、前回のプログラムでは、

```
readExpr() {
   readTerm();
   while(readExprOp() is OK)
        readTerm();
```

```
}
```

とした。これは、

```
<expresssion> := <term> <expression1>
<expression1> := <expr_op> <term> <expression1> | e
```

と書き換えたのと同等である。e は空を示す。

このほかに、

```
 <T> := b ( c| e )
```

はうまく行かない。 (c|e)は、c を読むか、それともなにもしないかという意味であるが、この場合は、<T>の次に何がくるかによって、c を読むかどうかが決まるので、top-down parser では、処理ができないことになってしまう。

プログラミング言語処理

４．構文解析の基礎

top-down parser と bottom-up parser

前章までで、式のインタプリタでは構文解析に top down parser を使って解説した。top down parser は再帰的下方構文解析の代表的な手法であり、次に何が来るのかを推定しながら構文解析を進めていく方法である。比較的構成がわかりやすく、人手で書いていく場合などには適した方法とされている。

このほかにも構文解析には、*上方構文解析法*（*bottom-up parser*、上昇型ともいう）という方法がある。この方法は人手で直接実現するには向かない方法であるが、理論的に構成されており、構文解析のプログラムを自動的に生成するためには重要な方法になっている。今回は、この上向き構文解析法についてみていく。

簡単に説明するためにいつもの式の構文解析を考えてみる。文法は以下のものを考える。F,T,E は非終端記号であり、id は変数のようなシンボルを仮定する。

```
(1) E := E + T
(2) E := T
(3) T := T * F
(4) T := F
(5) F := ( E )
(6) F := id
```

上向き構文解析と還元

構文解析の重要な役割は、入力がこの文法にあっているかどうかを認識することである。下方構文解析では、まず、Ｅであることを仮定して解析をはじめ、それぞれの非終端記号に対応する関数を呼び出し、最終的に必要な終端記号列になっているかを認識する方法であった。つまり、構文木という観点からみれば、構文木の根から葉に向かって解析を進めていく。（ここで、この文法は左再帰で書いてあるため、そのままでは top-down parser ができないことは以前に説明したとおり）

これに対し、上方構文解析では葉すなわち終端記号から、根すなわち非終端記号へ向かって文法を適用して、最終的にＥになっているかを認識する。例えば,a+c*d を考えてみる。これを token の列にしてみると、

```
 id + id*id
```

さらに、(6)の文法を適用して、

```
F+F*F
```

(3)(4)の文法を適用して

```
F+T*F  ⇒  T + T*F  ⇒  T + T
```

さらに、(1)(2) を適用して

E + T ⇒ E

となって、認識される。この適用して、非終端記号に置き換えていくことを*還元(reduction)*と呼ぶ。上方構文解析で、構文木を構成する過程は、文字列を非終端記号に還元していく過程である。この例では、順番を考えずにできるところから還元していったが、これをするためには入力を全部みてからやることになるため、あまり現実的ではない。

上向き構文解析では、入力を左から右に見ながら（つまり、一文字づつ入力しながら）還元していく。入力の右側から適用できる構文規則を逆順にたどって最終的に最後の構文規則まで還元できる部分列を handle(把手)という。上方構文解析は handle を見つけて還元する過程とみなすことができる。

右文形式	handle	還元につかった規則
a + b * c	a	(6) (4) (2)
E + b * c	b	(6) (4)
E + T * c	c	(6)
E + T * F	T*F	(3)
E + T	E+T	(1)
E	---	---

このような構文解析を（自動的に）構成するために、現在の構文解析の状態を記憶するためのスタックと入力の動作として以下のものを考える。

1. 移動(shift)：次の入力記号をスタックの上段に移動する。
2. 還元(reduce)：handle の右の記号がスタックの一番上にあり、適用できる構文規則をみつけて、その非終端記号に置き換える。
3. 受理(accept)：構文解析が終了
4. エラー：適用できる構文規則がみつからず、誤りを発見。

これを図示すると以下のようになる。

スタックの状態	入力	動作
$	a + b * c $	shift
$a	+ b * c $	(6)(4)(2)による reduce
$E	+ b * c $	shift
$E +	b * c $	shift
$E + b	* c $	(6)(4)による reduce
$E + T	* c $	shift
$E + T*	c $	shift
$E + T*c	$	(6)による reduce
$E + T*F	$	(3)による reduce
$E + T	$	(1)による reduce
$E	$	accept

演算子順位構文解析法

さて、上方構文解析のためのアルゴリズムについて、考えていくことにしよう。演算子順位構文解析法は、演算子順位文法(operator precedence grammar)に対する簡単な上方構文解析法である。演算子文法とは、どの構文生成規則の右辺も空ではなく、しかも、隣接する２つの非終端記号を持たないという文法である。つまり、算術式 E := E + T のように、必ず非終端記号の間には演算子（終端記号）が入るものである。演算子順位文法とは、終端記号について、優先度＞＜ =が定義されている文法のことである。

1. A:= ...st... または、A:= ...sBt..なる構文規則があれば、s=t
2. A:= ...sB...なる構文規則があり、さらに B⇒t または B⇒Ct なる規則が導かれるならば、s ＜ t
3. A:= ...Bt...なる構文規則があり、さらに B⇒...s または B⇒...sC なる規則が導かれるならば、s ＞ t

例についていえば括弧(、)に関しては=、E:= E+T と T:=T*F により、+ ＜ *である。 つまり、数式の直感的な優先度に対応している文法とおもえばよい。２，３の規則は構文規則で、構文木を作ったとき下流にある演算子が強いことを示す。このような関係にしたがって、構文規則より、演算子順位行列を作ることができる。

	+	*	(	)	id
+	＞	＜	＜	＞	＜
*	＞	＞	＜	＞	＜
(	＜	＜	＜	=	＜
)	＞	＞		＞	
id	＞	＞		＞	

これを使って、以下のアルゴリズムをつかえばよい。

1. スタックに空記号$をつんでおく
2. 入力記号 a をよむ
3. スタック上の演算子 s に対し、s ＞ a であるかぎリ、還元する
4. そうでなければ、a をスタック上につみ、２へ
5. 全部認識されたら終わり。

最後の reduction のために、便宜的に優先度が一番低い最後の記号を導入する必要がある。また、１項演算子を扱う場合には工夫が必要となる。

ＬＲ構文解析法

演算子文法に関しては比較的簡単なアルゴリズムで構成することができたが、一般の文法には使えない。ここで説明する方法は入力を左から右へ走査し、最右の規則を導くので、*LR(left-to-right scanning right most derivation)構文解析法* と呼ばれるものである。

LR 構文解析は入力とスタック、構文解析表からなる。入力は１記号ずつ左から右に読む。スタックには、

s0 X1 s1 X2 s2 X3 s3.... Xm sm

という記号列を積む。ｓは状態に対応した記号である。Ｘは文法記号で、実際は必要ないが説明のために加えてある。構文解析表は構文解析動作関数 ACTION と行き先関数 GOTO の２つの部分からな

る。LR 構文解析のプログラムは現在のスタックの最上段の状態 sm と入力記号 ai をもちいて、ACTION[sm,ai]を引いて、以下の動作のどれかをとる。

1. shift s: 入力記号 ai と GOTO[sm,ai]で求めた次の状態 s をスタックに積む。次の入力に進む。
2. reduce A := b: 文法規則 A:=b で還元する。すなわち、最上段にある X の列が b であるはずなので、これに対応する Xs のペアをスタックから取り除き、最後の状態 sm と A で、GOTO[sm,A]=s を次の状態とし、As をスタックに積む。還元の動作は現在の入力記号は変わらない。
3. accept
4. error

例に挙げた数式の文法について番号をつける。構文解析表は次のようになる。

ACTION							GOTO (非終端記号)		
state	**id**	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

ここで、si は shift で状態 i をスタックに積む動作を意味する。また、rj は文法 j による reduce 動作を意味する。ここで、a*b+c$を入力として考えてみよ。

1. まず、はじめの状態は０から始まる。
2. state 0 で、id が入力されると s5 となっているので、shift。入力記号 id と状態５をつむ。
3. 次に*が入力になるが、state 5 で、*の欄は、r6 である。これは文法（６）による reduce 操作である。スタックの上にある id 5 のペアを取り除き、最上段が 0 になるので、state０とＦを GOTO で引き、3 となっているため、Ｆと３がスタックに積まれる。
4. 入力記号はそのままである。state3 において、入力が*であれば、r4 である。文法規則(4)での reduce をする。T となり、state 0 とＴで GOTO を引き 2 となるため、T 2 を積む。
5. state 2 で、入力が*の時には s7 となる。したがって、*と７をつみ、次の入力に移る。
6. 以下、省略。

	スタック	入力
(1)	0	id*id+id $
(2)	0 id 5	*id+id$
(3)	0 F 3	*id+id$
(4)	0 T 2	*id+id$
(5)	0 T 2 * 7	id+id$
(6)	0 T 2 * 7 id 5	+id$
(7)	0 T 2 * 7 id F 10	+id$
(8)	0 T 2	+id$
(9)	0 E 1	+id$
(10)	0 E 1 + 6	id$
(11)	0 E 1 + 6 id 5	$
(12)	0 E 1 + 6 F 3	$
(13)	0 E 1 + 6 T 9	$
(14)	0 E 1	$

このような表を作ることによって、ＬＲ構文解析ができる。この表の作りかたについてはこの講義では説明しないが、重要な点は字句解析のところでオートマトンから字句解析を自動的に生成できると同様に、この表を自動的に作る方法があり、構文解析ルーチンを自動的に生成できることである。

構文解析生成プログラム yacc

LR 構文解析ルーチンを自動生成するプログラムの一つが yacc である。実際、構文解析ルーチンは top-down parser で書くことがあるが、複雑になると手に負えなくなるため、yacc のような自動生成プログラムを使う。（linux のフリーな構文解析は実際 bison というプログラムであるが、yacc というコマンドになっている）実際の場面では yacc の使い方を習得しておくことが重要になる。

yacc は、LR 構文解析に一文字の先読み機能を付け加えた LALR(Look-ahead LR)という文法のクラスを扱うことができる。yacc の入力（文法の定義）は、例として以下のように書く。

```
%token NUM  /* yylex から返す token の定義、文字を直接返してもよい*/
%token SYM
%token STRING
%{
#include   /*Cのプログラムのヘッダー、なんでもかける*/
%}
%start expression /* yyparse で何の認識をするかの指定*/
%% /* 文法の定義の始まり*/
expression: term
        | expression '+' term
        ;
term: factor
        | term '*' factor
        ;
factor: NUM | SYM ;
%% /* 文法の定義の終わり*/

#include "lex.c" /* ここからは何のCのプログラムをかいてもいい*/
```

token で定義されているものは、define されるので、lex.c のなかでそのまま使うことができる。lex.c では、これまででやった字句解析のルーチンが定義する。構文解析から呼び出される字句解析のルーチンは、yylex という名前で定義しなくてはならない。これは、lex を使っても生成できる。

このプログラムを expression.y とすると、

```
% yacc expression.y
```

で、構文解析プログラムが y.tab.c という名前で生成される。このプログラムには、構文解析ルーチン yyparse が含まれており、、main プログラムを以下のようにして、リンクすればよい。

```
main()
{
     yyparse();
     printf("ok¥n");
}

void yyerror()
{
     printf("syntax error!¥n");
     exit(1);
}
```

yyerror は構文解析でエラーになったときに呼び出される関数で、ユーザが与える。

プログラミング言語処理

５．tiny C について

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

を紹介する。

tiny C の言語仕様

以下に、tiny C の言語仕様を示す。

- 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。

以下に、tiny-C で書いた 0 から、9 までの加算し、それをプリントするプログラムを示す。

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

配列と関数を使う場合には、以下のようにして使う。

```
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;
}
```

なお、for 文については、作っていないので注意。この他にも、演算子など必要な機能については各自演習で作ること。

tiny C の文法

以下に、簡単な BNF 書式での文法を示す。

- 小文字は、非終端記号を示す。但し、variable_name, function_name, array_name, parameter は識別子
- 大文字は、終端記号。大文字で示してある IF,RETURN,WHILE,FOR、PRINTLN は、キーワード NUMBER は、10 進数。STRING は、"..."の文字列。
- ' 'は、１文字の終端記号。
- {...}*は、...の部分の 0 回以上の繰り返し示す。
- [ ]は省略可能の部分を示す。

```
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
        | 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}*] ')'
        | function_name '(' ')'
        | '(' expr ')'
        | PRINTLN  '(' STRING ',' expr ')'
```

tiny C の文法

ソースコード　（付録参照）

注意：ソースコードは、完全ではない。説明のためにいろいろと簡単化してある。「正しいプログラムは正しく」動作するが、間違ったプログラムについてのチェックは完全ではない。

- 構文解析部（インタプリタ、コンパイラ共通）
  - 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
  - cparser.y ： parser の yacc プログラム
  - clex.c: 字句解析部分
  - parser の作り方（cparser.o を作る）

```
% yacc cparser.y
% mv y.tab.c cparser.c
% cc -c cparser.c
```

  - AST.o を作っておくこと。

```
% cc -c AST.c
```

- インタプリタ
  - interp.h ： インタプリタの header
  - interp_main.c ： インタプリタの main

- interp.c : インタプリタの関数、文の処理
- interp_expr.c : インタプリタ の式の処理
- インタプリタ(tiny-c-run) の作り方
  - 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
```

- スタックマシンへのコンパイラ
  - 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
  - st_machine.c : スタックマシンのインタプリタ
  - スタックマシン(st_machine)の作り方

```
% cc -o st_machine st_machine.o st_code.o
```

- コンパイラ(tiny-cc-st) の作り方
  - それぞれの*.c をコンパイル
  - 以下でリンク

```
% cc -o tiny-cc-st compiler_main.o AST.o cparser.o st_compile.o ¥
        st_compile_expr.o st_code_gen.o st_code.o
```

- コンパイラ(tiny-cc-st) でコンパイル
  - tiny C のプログラム(sample.c)を準備
  - 標準入力から入力し、オブジェクトコード output.st に出力

```
% tiny-cc-st < sample.c > output.st
```

  - スタックマシン st_machine で実行

```
% st_machine < output.st
```

- レジスタマシン(x86)へのコンパイラ
  - 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
- println.c : println ライブラリ関数
- コンパイラ(tiny-cc-x86) の作り方
  - それぞれの*.c をコンパイル
  - 以下でリンク

```
% cc -o tiny-cc-x86 compiler_main.o AST.o cparser.o reg_compile.o ¥
      reg_compile_expr.o x86_code_gen.o
```

- コンパイラ(tiny-cc-x86) でコンパイル
  - tiny C のプログラム(sample.c)を準備
  - 標準入力から入力し、オブジェクトコード output.s に出力

```
% tiny-cc-x86 < sample.c > output.s
```

  - オブジェクトコードをアセンブル、ライブラリ(println.c) とリンク

```
% cc output.s println.c
```

  - 実行

```
% ./a.out
```

- Makefile

プログラミング言語処理

６．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

最初の講義で解説した通り、構文木は最も基本的なデータ構造である。以下のように定義する。

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

AST.h

- 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。

前に解説した AST のものとの違いは、シンボルについての定義が加わっている。なお、val と sym と left,right は同時には使わないので、union を使ってメモリを節約することができる。

AST の生成

AST のノードを作る関数が、makeAST である。

```
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.c

また、NUM の数の定数の AST ノードを作る関数が makeNum である。

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

AST.c

AST のリスト

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なことがある。そのために、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,NULL))
#define makeList3(x1,x2,x3)
makeAST(LIST,x1,makeAST(LIST,x2,makeAST(LIST,x3,NULL)))
```

AST.h

リストの最後に要素(AST)を加える関数が、addList である。

```
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;
}
```

AST.c

最後を見付けて、LIST の AST ノードを付け加える。

リストについてのアクセスは、n 番目の要素を取り出す getNth で行う。

```
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;
}
```

AST.c

これを使えば、1 番目をとるのは、getFirst は以下のように定義できる。

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

AST.h

また、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;
}
```

AST.c

この関数はリストの要素を１つずつ、アクセスするのに用いる。

```
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
}
```

op コード

AST 構造体の op に入るコードについては以下の通りである。

```
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
};
```

AST.h

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。

シンボル構造体とシンボルテーブル

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。

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

AST.h

- 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。

この構造体を使って、シンボルテーブル、すなわち表にして管理する。

```
Symbol SymbolTable[MAX_SYMBOLS];
int n_symbols = 0;
```

AST.c

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。

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

AST.c

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である。関数 makeSymbol は名前を与えて、それに対応する AST を作る。

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

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

AST.c

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 である。

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

AST.c

プログラミング言語処理

７．構文解析の実際：yacc の使い方

これまで、yacc の基本的な使い方について解説した。さて、これから yacc を使って tiny C のインタプリタを作ることにする。yacc のクローンである bison のマニュアルは、

http://www.gnu.org/manual/bison/

に解説してあるために、参考にするとよい。

yacc の動作

yacc は、以下のように動作する。

1. yylex を呼び出して、token を読み込み、その token から始まる文法規則を探す。
2. その文法規則が終るまで、token を読み、遷移(shift)を続ける。
3. 文法に非終端記号がある場合は、その文法規則をスタックに積み、1 からやり直す。
4. 文法規則の最後まで遷移したら、その規則を還元(reduce)する。
5. スタックから一つ前に処理していた規則に戻り、3. で還元した非終端記号をつかって、さらに shift する。
6. 2 に戻る。

実際に yacc が出力する parser のコードを解読するのは無理であるが、参考として、-v を付けて yacc を起動することによって、y.output というファイルができるので、これを見るとどのように shift、reduce しているかを見ることができる。

yacc の action と意味値(semantic value)

以前、つくった式の yacc のプログラムでは、単に構文が定義した文法にあっているかをチェックするものであったが、構文解析の仕事は定義した構文にあっているかとチェックするとともに、構文を表現する構文木(抽象構文木：abstract syntax tree, AST)を作ることである。構文木は意味解析でその意味に従った処理が行われる。

yacc では、構文解析の途中で、何らかの動作を行う action の指定ができる。構文木を作る作業はこの action の中で行う。action は構文規則の中に{ }で囲んで,C 言語で記述する。例えば、

```
term:  factor { printf("factor is coming"); }
     | term '*' factor { printf("factor is added"); }
     ;
```

この例では、term の各規則が reduce されたときに、{}の中の action が実行される。通常、action は各構文規則の最後に書き、reduce された時に実行されるようにするが、途中に書いてもよい。その場合には、そこまで、shift されたときに action が実行されるようになる。

構文規則の主な仕事は、構文木を作ることである。yacc では各構文規則で生成される値を意味値(semantic value)を持つことができ、その構文で認識された構文木を意味値として、action でその意味値を生成（計算）する。例えば、上の例では

```
term: factor { $$ = $1; }
     | term '*' factor { $$ = makeAST(PLUS_OP,$1,$2); }
     ;
```

$1,$2,...は、右に現れる非終端記号の意味値であり、これを使って、$$は右の記号 term の意味値を計算する。この値のデータ型は構文木すなわち AST へのポインタ型にする。なお、makeAST は２つの AST から新しい AST を作る関数である。

宣言部に、以下の記号のデータ型を定義する。

```
%union {
  AST *val;
}
%type <val> term factor
```

%union は、意味値に使うデータ型を定義するもので、この中のデータ型は複数でもよい。この union のメンバーを使って、%type で構文規則の記号の意味値を定義する。

さて、factor の定義では、終端記号の意味値を使う。

```
%type <val> NUM SYM
 ...
factor: NUM  | SYM ;
```

{ $$ = $1; }の場合は省略してもよい。終端記号に対しては、字句解析ルーチン yylex からは、yylex の値を NUM,SYM を返すとともに、意味値を yylval という変数（これは yacc から生成されるルーチンの中で定義されている）を介して、意味値を返す。

```
int yylex(){
   .... /* NUM の時  */
   yylval.val = makeNum(n);
   return NUM;
  .... /* SYM の時*/
  yylval.val = makeSymbol(yytext);
  return SYM;
  ....
}
```

なお、意味値のデータ型は、yacc の中では YYSTYPE という名前になっており、

```
#define YYSTYPE ...
```

として、直接定義する方法もある。

優先度の定義

yacc は LALR parser であり、一文字先読みをしているため、演算子の結合規則と、優先度を定義できる。%left は左結合規則を持つ演算子であることを指定する。例えば

```
%left '+'
```

と指定すると

```
expr: expr '+' expr { ...} ;
```

の文法規則を使って、x + y + z に対して((x + y) + z)のように処理される。 %right は右結合規則を持つもので、例えば代入の'='は右結合規則を持つものである。%left、%right は同時に演算子の優先度も指定する。後から、指定したほうが高い優先度を持つものと解釈される。これを使うと優先度をもつような規則を簡単に書くことができる。

```
%left '+' '-'
%left '*'
%left UMINUS
...
expr:   factor
       |  expr '+' expr { $$=addSymbol(plusSym,makeList2($1,$3)); }
       | exp '-' exp { $$=addSymbol(minusSym,makeList2($1,$3)); }
       | exp '*' exp { $$=addSymbol(mulSym,makeList2($1,$3)); }
       | '-' exp %prec UMINUS { .... }
       ;
```

なお、最後の%prec は単項演算子を最も優先度の高い処理をするための指定である。

あいまいな文法と shift/reduce conflict, reduce/reduce conflict

文法にあいまいさがあると、LR 構文解析ができなくなるので、yacc は警告メッセージをだす。メッセージには２種類あり、*shift/reduce conflict*, *reduce/reduce conflict* がある。shift/reduce conflict とは、文法規則が shift（つまり、さらに長い非終端記号に reduce できる）なのか、reduce(そこで打ち切って、非終端記号にしてしまう)か、解釈ができることを示す。この conflict は一概に文法定義が間違っているということではない場合がある。有名な例として、IF 文の定義がある。

```
statement : IF '(' expression ')' statement
          | IF '(' expression ')' statement ELSE statement
           ....;
```

これは次の場合にあいまいになる。

```
if (a > 0)
 if (b > 0) c = 100;
  else
```

```
c = 2000;
```

else を読んだとき、この token は内側の if 文の一部であると考え、遷移すればよいのだろうか、それとも、内側の if 文は完了したと考え、還元して、読みこんだ else は外側の if 文の一部であるとして遷移すればよいのだろうか？一般に yacc は、shift/reduce conflict がおきたときには、例外条件として、遷移(shift)を優先させる。したがって上の else は内側の if 文の一部と解釈される。この解釈は、C 言語を始めほとんどの言語の仕様と一致するので、一般に if 文にまつわる shift/reduce conflict はそのままにしておいて問題ない。

他方、reduce/reduce conflict は、同時に還元できる文法規則が複数あることを意味する。便宜上、yacc でははじめに現れた文法規則を優先させるが、これは望ましいことではないので、この conflict がないように文法を作る必要がある。例えば、良くある例として 0 個以上の word 列を読む場合を考えてみる。

```
sequence: /*  */  { printf ("empty sequence¥n"); }
        | maybeword
        | sequence word  { printf ("added word %s¥n", $2); }
        ;
maybeword: /*  */  { printf ("empty maybeword¥n"); }
        | word  { printf ("single word %s¥n", $1); }
        ;
```

この場合は、word は mayword に reduce でき,sequence でも reduce できてしまう。この場合は単に、以下のように定義してやればよい。

```
sequence: /*  */  { printf ("empty sequence¥n"); }
        | sequence word  { printf ("added word %s¥n", $2); }
        ;
```

もう一つの注意点として、再帰的な定義がある。例えば、','で区切られた列を表現する場合に次の２つの方法がある。

```
seq:  item |  seq ',' term ;  /* left recursion */
seq:  item | term ',' seq ;   /* right recursion */
```

yacc では、right recursion では、途中の状態をスタックにとっておく必要があるため、なるべく、left recursion で書いておくべきである。

エラー回復処理

通常使っているコンパイラでは、途中で文法エラーを見つけたとしてもなるべく、他の部分も parse して一度に多くの文法エラーを見つけることができるようにしてある。文法エラーを見つけたときに、次にどこから構文解析を再開するかの処理をエラーからの回復処理という。どこから処理を再開するか、どうやって再開するかについてはコンパイラの使いやすさの要素の一つにもなり、結構むずかしい問題である。ここでは、yacc での簡単なエラー処理だけについて述べておく。

yacc では、予約の非終端記号として、error という予約語があり、yyerror が呼び出されて、これが終了(retrun)すると、error という記号に reduce されるように処理してある。例えば、

```
statement: ....
         | error ';'
         ;
```

とすることによって、statement の構文解析で文法エラーが起きた場合には、';'がくるまで読みとばす処理をすることになる。

プログラミング言語処理

８．tiny C の構文解析

前に解説したデータ構造を使って、parser を作る。構文解析に関連するのは、以下の２つのファイルである。

- clex.c: 字句解析部分
- cparser.y : parser の yacc プログラム

プログラムでは、AST を作り、それを処理ルーチンに渡すということにしてある。

字句解析のプログラム： yylex (clex.c)

字句解析は lex.c で定義されている yylex である。(yacc からは、yylex のインタフェースで呼び出される)yylex からは、yacc の終端記号が返され、意味値がある場合(終端記号が値を持つ場合)には、yylval.val にセットして返す。

```
int yylex()
{
    int c,n;
    char *p;
again:
    c = getChar();
    if(isspace(c)) goto again;
    switch(c){
    case '+':
    case '-':
    case '*':
    case '>':
    case '<':
    case '(':
    case ')':
    case '{':
    case '}':
    case ']':
    case '[':
    case ';':
    case ',':
    case '=':
    case EOF:
        return c;
    case '"':
        p = yytext;
        while((c = getChar()) != '"'){
            *p++ = c;
        }
        *p = '¥0';
```

```
        yylval.val = makeNum((int)strdup(yytext));
        return STRING;
    }
    if(isdigit(c)){
        n = 0;
        do {
            n = n*10 + c - '0';
            c = getChar();
        } while(isdigit(c));
        ungetChar(c);
        yylval.val = makeNum(n);
        return NUMBER;
    }
    if(isalpha(c)){
        p = yytext;
        do {
            *p++ = c;
            c = getChar();
        } while(isalpha(c));
        *p = '¥0';
        ungetChar(c);
        if(strcmp(yytext,"var") == 0)
            return VAR;
        else if(strcmp(yytext,"if") == 0)
            return IF;
        else if(strcmp(yytext,"else") == 0)
            return ELSE;
        else if(strcmp(yytext,"return") == 0)
            return RETURN;
        else if(strcmp(yytext,"while") == 0)
            return WHILE;
        else if(strcmp(yytext,"for") == 0)
            return FOR;
        else if(strcmp(yytext,"println") == 0)
            return PRINTLN;
        else {
            yylval.val = makeSymbol(yytext);
            return SYMBOL;
        }
    }
    fprintf(stderr,"bad char '%c'¥n",c);
    exit(1);
}
```

- 入力から、文字を読み込み、空白(isspace)ならば、読み飛ばす。
- 次の switch 文では、１文字の終端記号を読み込む。
- もしも、'"'ならば、文字列定数を読み込み、STRING を返す。意味値としては、読み込んだ文字列をセーブして(strdup)、そのアドレスを NUM の AST ノードとして返す。
- もしも、数字ならばそれに続く数を読み込み、NUMBER を返す。読み込んだ数を十進数として解釈し、NUM の AST ノードを意味値とする。

- もしも、英文字ならば、まず英文字の列を読み込み、それを yytext にいれる。次に、それがキーワードならばキーワードに対応する終端記号を返す。キーワードでなかったら、SYM を返し、意味値として SYM の AST を作る。
- なお、このルーチンは cparser.y に include されて、コンパイルされる。終端記号に使われる名前は cparser.y で自動的に生成されていることに注意。getChar と ungetChar は、デバック用のもので、getc と ungetc に変えてもよい。

構文解析のプログラム: cparser.y

parser は、yacc で記述されている。部分部分に分けて解説する。

まずは、終端記号の token の定義を行う。1 文字で表される token はその文字自体をつかってよい。例えば、+などの記号は、'+'と記述する。token の定義は、%token で行う。

```
%token NUMBER
%token SYMBOL
%token STRING
%token VAR
%token IF
%token ELSE
%token RETURN
%token WHILE
%token FOR
%token PRINTLN
```

ここで、token として定義された名前は、生成された C プログラムの中で、#define で適当な値に定義されている。よって、最後に include されている clex.c の中では値として使うことができる。

次に、生成された C プログラムの中で必要となるファイルの include を行う。ここでは、stdio.h と AST.h を include する。

```
%{
#include <stdio.h>
#include "AST.h"
%}
```

意味値として返される値のデータ型を定義する。このプログラムでは、AST を意味値として返すために、AST へのポインタを意味値の値としている。

```
%union {
    AST *val;
}
```

clex.c の中では、終端記号についての意味値は yylval.val で参照する。このプログラムでは、一つのデータ型しかつかわないため、AST *val だけである。この val は%type でデータ型を示す名前として用いられる。

次に、token の優先度と token に対する意味値のデータ型を指定する。意味値のデータ型の指定は%type で行う。

```
%right '='
%left '<' '>'
%left '+' '-'
%left '*'

%type <val> parameter_list block local_vars symbol_list
%type <val> statements statement expr primary_expr arg_list
%type <val> SYMBOL NUMBER STRING
```

なお、意味値を持たない token についてはデータ型を定義しない。
最後に、入力全体を表す token を%start で指定する。このプログラムでは、program である。

```
%start program
```

構文の定義は%%で始める。どのような順番でもいいが、通常、top-down に定義していく。program は空であるか(入力がなし)、もしくは外部定義の列 external_definitions である。

```
%%
program: /* empty */
         | external_definitions
         ;
```

列の定義は以下のように定義する。left recursion に定義することに注意。

```
external_definitions:
           external_definition
         | external_definitions external_definition
         ;
```

外部定義 external_definition は、関数定義、大域変数の定義、もしくは配列の定義である。

```
external_definition:
           SYMBOL parameter_list block  /* fucntion definition */
         { defineFunction(getSymbol($1),$2,$3); }
         | VAR SYMBOL ';'
         { declareVariable(getSymbol($2),NULL); }
         | VAR SYMBOL '=' expr ';'
        { declareVariable(getSymbol($2),$4); }
         | VAR SYMBOL '[' expr ']' ';'
         { declareArray(getSymbol($2),$4); }
         ;
```

この外部定義が認識された時点で、それぞれの処理する関数を呼び出す。

- 関数定義の場合

```
void defineFunction(Symbol *fsym,AST *params,AST *body)
```

fsym に関数名のシンボル、params にパラメータの AST、body に関数本体の AST を与える。

- 大域変数の定義の場合

```
void declareVariable(Symbol *vsym,AST *init_value);
```

vsym に変数名のシンボル、init_value に初期値の AST を与える。初期値がない場合は、NULL

- 大域配列の定義の場合

```
void declareArray(Symbol *asym,AST *size);
```

asym に配列名のシンボル、size にサイズの AST を与える。

シンボルを引数にするために、getSymbol を使っていることに注意。これらの関数はインタプリタ、コンパイラによって違うものをリンクする。

パラメータの並びの定義。パラメータは、シンボルの並びを'('と')'ではさんだもの。

```
parameter_list:
          '(' ')'
           { $$ = NULL; }
         | '(' symbol_list ')'
           { $$ = $2; }
         ;

symbol_list:
           SYMBOL
           { $$ = makeList1($1); }
         | symbol_list ',' SYMBOL
           { $$ = addLast($1,$3); }
         ;
```

シンボルの並びはシンボルを','で区切ったもの。まず、最初に makeList1 で最初のリストを作り、それに addLast で続くシンボルを最後に加えて生成している。

block すなわち複文は、'{'ではじまり、'}'で終る。最初に局所変数の定義 local_vars があり、文の並びが続くもの。複文を表す AST は、左に局所変数のリスト、右に文のリストをいれたものである。

```
block: '{' local_vars statements '}'
         { $$ = makeAST(BLOCK_STATEMENT,$2,$3); }
```

```
        ;

statements:
          statement
         { $$ = makeList1($1); }
        | statements statement
         { $$ = addLast($1,$2); }
        ;

local_vars:
          /* NULL */ { $$ = NULL; }
        | VAR symbol_list ';'
          { $$ = $2; }
        ;
```

local_vars は、なくてもよい。キーワード VAR がある場合には、局所変数の定義なので、シンボルの並びを読み込む。文の並びの処理の仕方は、シンボルのならびと同じように、最初に makeList1 で最初のリストを作り、それに addLast で続くシンボルを最後に加えてつくる。

以下が、文の定義である。それぞれの文に対応した AST を作り返す。

```
statement:
         expr ';'
         { $$ = $1; }
        | block
         { $$ = $1; }
        | IF '(' expr ')' statement
         { $$ = makeAST(IF_STATEMENT,$3,makeList2($5,NULL)); }
       | IF '(' expr ')' statement ELSE statement
         { $$ = makeAST(IF_STATEMENT,$3,makeList2($5,$7)); }
        | RETURN expr ';'
         { $$ = makeAST(RETURN_STATEMENT,$2,NULL); }
        | RETURN ';'
         { $$ = makeAST(RETURN_STATEMENT,NULL,NULL); }
        | WHILE '(' expr ')' statement
         { $$ = makeAST(WHILE_STATEMENT,$3,$5); }
        | FOR '(' expr ';' expr ';' expr ')' statement
         { $$ = makeAST(FOR_STATEMENT,makeList3($3,$5,$7),$9); }
        ;
```

- まず、式はそれ自身で文になる。だだし、文の終りを表す';'が必要
- もちろん、複文は文である。
- if 文の AST は、左に条件式、右に then 部と else 部の 2 つの AST を持つリストを持つ AST である。else がない場合には NULL とする。
- RETURN 文の AST は、右にリターン値の式をいれた AST である。ない場合は NULL とする。
- WHILE 文の AST は、右に条件式、左の本体をいれたもの。
- FOR 文の AST は、左に初期式、繰り返し条件式、繰り返し式の 3 つの要素を持つ AST、右に本体をいれたもの。

以下が、式の定義である。

```
expr:     primary_expr
        | SYMBOL '=' expr
          { $$ = makeAST(EQ_OP,$1,$3); }
        | SYMBOL '[' expr ']' '=' expr
          { $$ = makeAST(SET_ARRAY_OP,makeList2($1,$3),$6); }
        | expr '+' expr
          { $$ = makeAST(PLUS_OP,$1,$3); }
        | expr '-' expr
          { $$ = makeAST(MINUS_OP,$1,$3); }
        | expr '*' expr
          { $$ = makeAST(MUL_OP,$1,$3); }
        | expr '<' expr
          { $$ = makeAST(LT_OP,$1,$3); }
        | expr '>' expr
          { $$ = makeAST(GT_OP,$1,$3); }
        ;
```

- primary_expr とは、以下に示すように変数や配列参照など。
- 2 項演算の式については、左には左辺の式の AST、右には右辺の式の AST をいれた AST を作る。
- 代入については左には変数、右には代入する式の AST をいれた EQ_OP の AST を作る。

  配列の代入の場合は、左に配列のシンボルとインデックスの式のリストをいれた SET_ARRAY_OP の AST を作る。
  なお、２項演算子の優先度については、最初に%right,%left などを使って定義してある。

次の定義が、primary_expr である。変数や定数の他に、配列要素の参照、関数呼び出しが定義してある。関数呼び出しでは、左に関数名、右に引数のならびのリストを持つ CALL_OP の AST ノードを作る。唯一のシステム関数である println はここで定義してある。

```
primary_expr:
          SYMBOL
        | NUMBER
        | STRING
        | SYMBOL '[' expr ']'
          { $$ = makeAST(GET_ARRAY_OP,$1,$3); }
        | SYMBOL '(' arg_list ')'
          { $$ = makeAST(CALL_OP,$1,$3); }
        | PRINTLN  '(' arg_list ')'
          { $$ = makeAST(PRINTLN_OP,$3,NULL); }
        ;

arg_list:
          expr
          { $$ = makeList1($1); }
        | arg_list ',' expr
          { $$ = addLast($1,$3); }
        ;
```

引数のリストは、','で区切った expr の並びになるので、順次、expr の AST のリストをつくる。

最後に、%%で終る。そのあとは、任意のCのプログラムが書けるので、ここにlex.cをincludeする。このようにすることのメリットは、clex.cのなかで、tokenの名前をマクロで定義されているものとして、使うことができる点である。

```
%%
#include "clex.c"
```

yyerror

yaccで生成される構文解析ルーチンでは、エラーを起こした時に、つまり間違った構文が入力されたときに、yyerrorという関数が呼び出されるようになっている。このプログラムでは、yyerrorは以下のように、メッセージをプリントしてプログラムを停止する、簡単なものにしている。

```
void yyerror()
{
    printf("syntax error!¥n");
    exit(1);
}
```

プログラムではエラーの処理については考慮されていない。本格的なプログラムにするには、エラー処理について考慮する必要がある。

構文解析のプログラムのコンパイル

cparser.yから、yaccを使ってparserを生成する。

```
% yacc cparser.y
```

生成されたプログラムは、y.tab.cになっているので、これを適当な名前(cparser.c)に変えて、Cコンパイラで、コンパイルする。

```
% mv y.tab.c cparser.c
% cc -c cparser.c
```

なお、clex.cはcparser.cにincludeされているので、別にコンパイルする必要はない。

プログラミング言語処理

９．インタプリタ(1) 式、変数、関数

これから、tiny-C のインタ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;
```

AST.h

式を実行する関数は、以下のようになる。

```
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:
         return executeExpr(p->left) > executeExpr(p->right);

        /* 残りは省略 */
    }
}
```

interp_expr.c

1. まず、AST が数値 NUM であれば、その値(p->val)を返す。
2. AST が 2 項演算子(PLUS_OP,MINUS_OP など)であれば、左右の AST を executeExpr を再帰的に呼び出して、実行し、その結果を演算子にしたがって、計算してその結果を返す。
3. シンボルの場合は、変数と解釈する。変数の値は、最初に説明したとおり、シンボル構造体の val のところにはいっているので、これを返す。それを行う関数が getValue である。
4. 変数の代入（EQ_OP）では右の式の値を求めて、それを左の変数のシンボルの val にセットすればよい。それを行う関数が setValue である。
5. また、AST がない(NULL)場合には、0 を返す。

いろいろなところで、executeExpr を再帰的に呼び出していることに注意しょう。

getValue は変数シンボルの val の値をかえし、setValue はその val に値をセットする。したがって、以下のようになるはずである。

```
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。

関数の定義：構文解析部とのインタフェース

さて、関数がどのように処理されるかの説明をすることにしょう。まず、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。これは、インタプリタでは、以下のように定義されている。

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

interp.c

関数名のシンボルの func_params にパラメータのシンボルのリスト、func_body に関数本体の AST をいれておく。これらは、関数呼び出しを実行する時に参照する。次に関数呼び出しを考える。
ちなみに、変数宣言に対するインタフェースは、declareVariable である。

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

interp_expr.c

インタプリタでは、プログラム中に変数が現れた時点で、シンボルが作られるために、変数宣言で特になにもする必要はない。ただし、初期化の式が与えられた時には、その式の値をもとめ、シンボルの val にセットしておく。

環境(environment)：変数と値の結合(bind)

例えば、tiny-C の次の関数を考える。

```
foo(x,y) { return x + y; }
```

プログラム中で、foo(1,2)を実行し、関数呼び出しをした場合には、関数本体を実行している間は x の値は 1、y の値は 2 になっていなくてはならない。このために、関数の実行をはじめに 1,2 を x,y にいれておくことが考えられるが、注意しなくてはならないのはパラメータは局所変数なので、関数の実行を終了したときには、x と y の値は元にももどしておかなくはならない。これは単なる代入と異なり、このように動的に変数と値を対応させることを*結合(bind)*するという。このためのデータ構造として、結合した変数と値のペアを記録しておくデータ構造を用意する。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。

以下に環境のためのデータ構造 Environment を示す。

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

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

interp.h

Env は変数と値のペアの配列で、パラメータの変数に値が結合されるごとにこの配列に記録しておく。この配列をどこまで使っているかを示すために、envp という変数を使う。

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。代入で、変数の値を変える場合もこの表にある場合には、その値を変更しなくてはならない。したがって、setValue,getValue は以下のようになる。

```
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;
}

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;
}
```

interp.c

関数呼び出し

executeExpr の省略している部分は、関数呼び出しの部分である。式のなかで、foo(x+1,2)のような式がある場合にこの部分が実行される。

```
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 */

    }
}
```

interp_expr.c

関数の呼び出しをする関数が executeCallFunc である。この関数は、一番目の引数が呼び出す関数のシンボル、第二引数が引数の AST のリストである。
では、executeCallFunc を見てみよう。一部省略しているところがあるが、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;
}
```

interp.c

1. 関数名のシンボルから、パラメータの並びを取り出す。パラメータの並びは、シンボルの AST のリストである。
2. それぞれのパラメータのリストと引数のリストから、パラメータとそれに対応する式をとりだし、式を実行して、その値とパラメータを結合する。ここでは、Env にセットするだけで、envp は変えない。
3. 引数の評価が終わったら、結合したところまで、envp を移動させる。これによって、パラメータの変数の値は env にある値になる。
4.
5. 関数から本体の AST を取り出し、executeStatement で実行する。なお、executeStatement は後で説明する。
6. おわったら、envp の値を元にもどし、結合を解く。

関数が再帰的に呼び出される場合にも、最近の結合されたものから探す（つまり、Env を逆順に探す）ことにより、最も最近に結合された値を参照できることになる。
ちなみに、システム関数である println の処理は、以下のようになる。

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

interp_expr.c

第１引数は文字列のアドレス、第２引数はその値である。これを printf で呼び出す。

動的結合と静的結合

上で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。例えば、

```
var x;

addx(y) {   return x + y; }

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

ここで、x = 10; として、addx(2) を呼び出してみると、その値は 12 になる。なぜなら、addx の中の x は大域変数の x を参照するからである。しかし、x = 10; としても、addxy(2,3)を実行すると、その値は 5 となる。

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

addxy(2,3)が 5 になるわけは、addxy では、x に 2 が結合され、そこで addx が呼び出されるとその中の x は、この束縛された x が参照されてしまうからである。つまり、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）。それに対して、addx が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大域変数の x 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

最後に配列の処理を完成させて、executeExpr を完成させることにしよう。

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。

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

interp_expr.c

tiny C のインタ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。declareArray では、size 部分の式を実行してサイズを求め、そのサイズ分の integer の領域を malloc で確保し、その配列を val にセットする。このインタプリタは、32bit マシンで実行することを仮定しているので、int とポインタのサイズは同じであるので、キャストすることによって、int の変数に無理矢理代入している。

なお、文字列についても、文字列のアドレスを値として、同じような処理をしている。println の処理を参照のこと。

次に、その配列の要素について、参照と代入する関数 getArray と setArray を示す。

```
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;
}
```

interp_expr.c

int の値をキャストして int へのポインタにして、その要素にアクセスしていることに注意しよう。

これで、いままで説明した部分をあわせて、executeExpr を完成させる。

```
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:
        return executeExpr(p->left) > executeExpr(p->right);
    case GET_ARRAY_OP:
        return getArray(executeExpr(p->left),executeExpr(p->right));
    case SET_ARRAY_OP:
        return setArray(executeExpr(getNth(p->left,0)),
                        executeExpr(getNth(p->left,1)),
                        executeExpr(p->right));
    case CALL_OP:
        return executeCallFunc(getSymbol(p->left),p->right);
    case PRINTLN_OP:
        printFunc(p->left);
        return 0;
    default:
        error("unknown operater/statement");
    }
}
```

interp_expr.c

1. 配列の要素の参照は、GET_ARRAY_OP の AST になっている。左は配列名であるが、これを executeExpr で実行すると配列のアドレスが返される。これを引数にして、getArray を呼び出している。
2. SET_ARRAY では、左に配列と要素のインデックスの式のリストが入っていることに注意。これらを executeExpr で実行して、setArray を呼び出す。
3. これ以外のコードが式として処理されることはないはずなので、error とする。

次に、文の実行について解説することにする。

プログラミング言語処理

１０．インタプリタ(1) 関数と文

前章では、インタプリタの主に式に関連する部分を説明した。今回は、文の処理を加え、本格的なプログラムが書けるように拡張してみよう。前回は、以下の機能をつくった。

- 組み込みの四則演算子による整数の計算
- 変数が使える。変数の代入と参照。
- 関数定義と関数呼び出し、関数の実行の一部
- 配列

プログラミング言語として、これ以外に必要な機能は何であろうか。Ｃや Java など、近代的なプログラミング言語にはいろいろな機能がある。

- while 文や for 文、switch 文などの豊富な制御構文
- 局所変数とブロック（複文）
- 関数の途中から関数の値を返す return 文
- 豊富なデータ型と構造体
- ポインター

このほかにもいろいろあるが、tinyC に加える機能は以下の通りである。

- if 文
- 局所変数とブロック文
- 関数の値を返す return 文
- while 文(と for 文)

なお、goto 文はない。繰り返しは while や for 文で行う。

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.c : インタプリタの関数、文の処理
- interp_main.c : インタプリタの main

文の実行

前に、説明した executeCallFunc の中で、本体の実行するために executeStatement を呼び出している。executeStatement は、文の AST の op を見て、それぞれの処理の関数を呼び出す。

```
void executeStatement(AST *p)
{
    if(p == NULL) return;
    switch(p->op){
    case BLOCK_STATEMENT:
         executeBlock(p->left,p->right);
         break;
```

```
    case RETURN_STATEMENT:
        executeReturn(p->left);
        break;
    case IF_STATEMENT:
        executeIf(p->left,getNth(p->right,0),getNth(p->right,1));
        break;
    case WHILE_STATEMENT:
        executeWhile(p->left,p->right);
        break;
    case FOR_STATEMENT:
        executeFor(getNth(p->left,0),getNth(p->left,1),getNth(p->left,2),
                   p->right);
        break;
    default:
        executeExpr(p);
    }
}
```

interp.c

1. AST が NULL の場合はなにもしない。
2. 文に関しては、それぞれの文の処理をする関数を呼び出す。
3. もしも、式も文の一つなので、文でなければ式を実行するために、executeExpr を呼び出す。

IF 文の実行

まず、文の処理でも最も簡単な if 文の処理から説明をしよう。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。これを、取り出して、if 文を実行する executeIf を呼び出す。

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

executeIf は以下のようになる。

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

interp.c

まず、executeExpr で条件式を実行し、その結果によって、then 部か else 部の式を executeStatement を呼び出して実行する。

複文と局所変数

関数の本体は、複文である。複文を実行する executeBlock は関数が呼び出されると最初に実行されることになる。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;
```

文を実行している間に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。なお、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。

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。

```
void executeBlock(AST *local_vars,AST *statements)
{
    AST *vars;
    int envp_save;

    envp_save = envp;
    for(vars = local_vars; vars != NULL; vars = getNext(vars))
        Env[envp++].var = getSymbol(getFirst(vars));
    for( ; statements != NULL; statements = getNext(statements))
        executeStatement(getFirst(statements));
    envp = envp_save;
    return;
}
```

interp.c

局所変数に対する処理は基本的には関数のパラメータ変数に対する式と同じである。block 文に現れた局所変数について、現在の環境に登録しておく。パラメータの場合には引数と結合しておいたが、局所変数に関しては何の値でもかまわない（つまり、未定義）。上の関数では以下のことを行う。

1. 現在の環境を示す envp を envp_save に保存しておく。
2. 局所変数のリストから、局所変数のシンボルを取り出す。
3. 環境に登録する。Env[envp++]=局所変数のシンボル
4. これを、局所変数の全部に繰り返す。
5. このあとで、リスト中の文を実行する。この間で、変数が参照される場合には、関数 getValue/setValue で、値を参照するために Env にある変数の値が参照されることになる。
6. 式が全部実行しおわったら、envp に env_savep を代入して、環境を呼び出し前に戻しておく。これによって、局所変数は取り消されることになる。

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である。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;
```

インタプリタ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。例えば、executeStatement で、IF 文を実行する場合には executeIf を呼び出し、その中で then 部や else 部を実行するのに executeStatement を呼び出している。さらに、then 部が複文の場合には、executeBlock が呼び出され、中の文を実行するために executeStatement を呼び出し、実行が進んでいく。その途中で、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。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。例えば、

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

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

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

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

この例では、foo の setjmp で env に状態を覚えておき、goo1,goo2 と呼び出されたときに、goo2 で longjmp をすることよって、setjmp の後に戻ってくる。setjmp では、はじめにセットしたときに 0 を、longjmp で戻ってきたときには longjmp で指定された値を返す。したがって、longjmp では第 2 引数に 0 以外の値を与える。

この setjmp/longjmp の機能を使って return 文をつくる。まず、戻るべき最も最近の状態 jmp_buf を覚えておくために、変数 funcReturnEnv を使う。funcReturnVal は return 文から返される値を格納する変数である。

```
jmp_buf *funcReturnEnv;
int funcReturnVal;
```

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。execCallFunc の中では、executeStatment を実行する時に以下のプログラムように setjmp を実行し、return 文が実行された時に戻る準備をしておく。

```
    ...
    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 */
         val = evalObject(getNth(func_def,1)); /*なにもなければその値*/
    }
    funcReturnEnv = ret_env_save;  /* 前の値に戻す*/
    ...
```

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 */
}
```

interp.c

以下が、上の setjmp を組み込んだ execCallFunc である。

```
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;
}
```

interp.c

return 文が実行されて setjmp のところにもどってきた時にも envp が元にもどされていることに注意。

while 文：制御文

制御文である 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。これを取り出して、executeWhile を実行する。

```
    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);
}
```

interp.c

このほかにも、for 文などの制御構造も、シンタックスを決め、その意味にしたがってどの部分を実行するかを制御すれば実装することができる。(for 文については、わざとぬいてあるので、作ってみること)

インタプリタの main プログラム

最後に、インタプリタの main プログラムを作る。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;
}
```

interp_main.c

なお、インタプリタのコンパイルの方法については、５章を参照のこと。

プログラミング言語処理

１１．スタックマシン

このインタプリタでつくった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、初回に紹介した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。コンパイラのターゲットの仮想マシンの解説からはじめることにしよう。

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

なお、このスタックマシンの作り方については、５章を参照のこと。

スタックマシンの命令

tinyC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。

命令コード	説明
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して乗算し、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。
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。
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、

1. 式が数字であれば、その数字を push するコードを出す。
2. 式は変数であれば、その値を push するコードをだす。
3. 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。

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。これを組みああわせて IF 文をコンパイルする。コードは次のようになる。

```
...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 文のコンパイルは以下のようになる。

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 部分の式をコンパイルする。
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 部の式をコンパイル。
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。

関数呼び出しの構造

式だけを評価するならば、これでいいが、関数が使えるようにするためには、スタックの使い方を工夫しなくてはならない。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 ： 関数の呼び出し側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。

関数の呼び出しの手順は、以下のようにする。

1. スタック上に引数を積む。
2. 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）
3. 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)
4. 式の評価のための stack はここから始まる。
5. 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)
6. 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)
7. 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)
8. 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）

このような構造を、*関数フレーム*と呼ぶ。このような規則を*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。

さて、関数定義に対するコードは以下のようになる。

```
ENTRY foo
FRAME ローカル変数の個数
  .... 関数本体のコード....
RET
```

また、関数呼び出しは、

```
引数１の push ...
引数２の push ...
  ....
CALL foo
POPR  push した引数の個数
```

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数えておかなくてはならない。

次章で、スタックマシンに対するコンパイラ全体について、説明することにする。

プログラミング言語処理

１２．スタックマシンへのコンパイラ

前章では、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 tinyC 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

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;
}
```

compiler_main.c

なお、parser の cparse.y や字句解析の lex.c はインタプリタと同一のものを使う。

コードの生成ルーチン

コンパイラの説明を始める前に、コード生成部のルーチンについて説明しておく。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以下のようになる。

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

int n_code;
```

st_code_gen.c

n_code はいくつコードが生成されたかを管理するカウンタである。

コード生成は以下の関数を使って行う。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する。

```
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;
}
```

st_code_gen.c

オペランドが文字列のアドレスなどのポインタの場合は、キャストして無理矢理、代入している。

関数のコンパイルが終ったら、genFuncCode でコードを出力する。これについては、関数のコンパイルで説明する。

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

なお、スタックマシンの命令コードは、st_code.h に定義されている。

スタックマシンの関数呼び出しの構造

前回説明したとおり、スタックマシンは関数呼び出しのために、SP と FP をつかう。SP スタックポインタは、スタックの top（の上）を指しているレジスタで、フレームポインタ FP は関数の呼び出し側の情報を保存しているところを指すようにする。ここからの相対で、引数や局所変数にアクセスする。関数の呼び出しの手順は、以下のようにする。

1. スタック上に引数を積む。
2. 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)
3. 式の評価のための stack はここから始まる。
4. 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)
5. 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)
6. 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)
7. 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)

このような構造を、*関数フレーム*と呼ぶ。

さて、関数定義に対するコードは以下のようになる。

```
ENTRY foo
FRAME ローカル変数の個数
 ... 関数本体のコード...
RET
```

また、関数呼び出しは、

```
引数１の push ...
引数２の push ...
 ....
CALL foo
POPR  push した引数の個数
```

コンパイラのための環境

これまでみたように、関数を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。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数えておかなくてはならない。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。したがって、コンパイラでは環境は以下のようなデータ構造である。

```
#define VAR_ARG 0
#define VAR_LOCAL 1

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

st_compile.c

var_kind は、変数がパラメータなのか(VAR_ARG)、局所変数か(VAR_LOCAL)を区別するフィールドで、pos にフレーム上の位置を記録しておく。

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 */
}
```

st_compiler.c

1. まず、genCodeInit を呼び出し、コード領域をクリアする。

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。この時、ローカル変数であることを示す VAR_ARG をセットしておく。
3. 関数の本体をコンパイルする。本体の文がコンパイルされると、本体の実行のためのコードが生成され、コード領域に格納される。
4. 最後に、genFuncCode を実行し、コード領域にあるコードを出力する。

コードを一時的にコード領域に格納しておいて、genFuncCode で出力する理由の一つは、コードを全部コンパイルしてみないとローカル変数などに必要な関数フレームのサイズが分からないためである。関数の本体に block 文（C の {} にあたる）がある場合には、局所変数が定義される可能性がある。block 文をコンパイルするときには、local_var_pos という変数を使って数えて、これでスタック上の何番目に割り当てるかを決める。本体のコンパイルが終わると、局所変数が何個あったかがわかるので、これを使って関数の先頭で、FRAME 命令を生成しなくてはならない。そのため、生成されたコードを配列(Codes)にとっておき、ENTRY 命令の後に FRAME 命令を生成し、とっておいた残りの命令を出力する。必要なローカル変数のカウンタである local_var_pos を引数にして、genFuncCode が呼び出される。genFuncCode を下に示す。

```
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:
        case LOADL:
        case STOREA:
        case STOREL:
        case POPR:
            printf("%d",Codes[i].operand);
            break;
        case BEQO:
        case LABEL:
        case JUMP:
            printf("L%d",Codes[i].operand);
            break;
        case CALL:
            printf("%s",(char *)Codes[i].operand);
            break;
        case PRINTLN:
            printf("¥"%s¥"",(char *)Codes[i].operand);
            break;
        }
        printf("¥n");
    }
    printf("RET¥n");
}
```

st_code_gen.c

既に生成されているコードに関数の始めの ENTRY とローカル変数を指定する FRAME 命令、RET のコードを加えて、コードを出力する。

式のコンパイル

さて、式のコンパイルから説明することにする。インタプリタでは、executeExpr という関数を使って式を実行したが、コンパイラでは compileExpr という関数で式に対するコードを生成する。式をコンパイルすると「実行すると式の値がスタック上につまれる」コードが生成される。コードは executeExpr と非常に良くにている。

```
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);
         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:
         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);
        return;

    /* 省略 */

    default:
        error("unknown operater/statement");
    }
}
```

st_compile_expr.c

1. 式が数字であれば、その数字を push するコードを出す。
2. 式が変数であれば、compileLoadVar を呼び出して、その値を push するコードをだす。
3. 代入式の場合は、compileStoreVar を呼び出す。
4. 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。
5. 関数呼び出しに対しては、compileCallFunc を呼び出す。
6. println については、printFunc を呼び出し、PRINTLN のコードを生成する。

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。compileLoadVar ではまず、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。逆に、ローカル変数やパラメータ変数に代入する関数が、compileStoreVar である。compileStoreVar では、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:
                genCode1(STOREA,Env[i].pos);
                return;
            case VAR_LOCAL:
                genCode1(STOREL,Env[i].pos);
                return;
            }
        }
    }
    error("undefined variable¥n");
}


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:
                genCode1(LOADA,Env[i].pos);
                return;
            case VAR_LOCAL:
                genCode1(LOADL,Env[i].pos);
                return;
            }
        }
    }
    error("undefined variable¥n");
}
```

st_compile.c

文のコンパイル

文をコンパイルする compileStatement は、executeStatement と同じく、文の AST の op から、それぞれの構文をコンパイルする関数を呼び出す。

```
void compileStatement(AST *p)
{
    if(p == NULL) return;

    switch(p->op){
    case BLOCK_STATEMENT:
        compileBlock(p->left,p->right);
        break;
    case RETURN_STATEMENT:
        compileReturn(p->left);
        break;
    case IF_STATEMENT:
        compileIf(p->left,getNth(p->right,0),getNth(p->right,1));
        break;
    case WHILE_STATEMENT:
        compileWhile(p->left,p->right);
        break;
    case FOR_STATEMENT:
        compileFor(getNth(p->left,0),getNth(p->left,1),getNth(p->left,2),
                   p->right);
        break;
    default:
        compileExpr(p);
        genCode(POP);
    }
}
```

st_compile.c

但し、式が文になる場合には、式をコンパイルする compileExpr を呼び出すが、compileExpr では、スタック上に式の結果を残すので、それを POP してとりのぞいておかなくてはならない。これを忘れるとスタックが尽きてしまう。compileStatement では、文の実行が終ったら、スタックは元にもどっていることに注意。

制御文のコンパイル

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。これを組みああわせて IF 文をコンパイルする。コードは次のようになる。

```
...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 文のコンパイルは以下のようになる。

```
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);
    }
}
```

st_compile.c

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. ラベルを l1 を作って、BEQ L0 を生成。
3. then 部分の文をコンパイルする。
4. else 部分がある場合には、
    1. then 部のコードの後に、ラベル l2 を作って、ここに JUMP する命令を生成する。

2. 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 l1 を生成する。
3. else 部の文をコンパイル。
4. その後に、then 部の実行が終わったときに飛ぶ先 LABEL l2 をここにおいておく。

5. else 部がない時に、LABEL l1 を生成するだけでよい。

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、その後で、引数をスタックから pop して、結果を push する POPR 命令を生成しておく。

```
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;
}
```

st_compile.c

スタックに積む順番は、引数の最後からなので、compileArgs では再帰を使って、引数の最後からコンパイルしている。また、この関数は同時に引数の個数を数えていることに注意。

局所変数のコンパイル

block 文では、局所変数が宣言されることがあるが、以下のようにしてコンパイルする。

```
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;
}
```

st_compile.c

1. 局所変数について、どこに割り当てるかを決める。割り当てるスタック上の場所の番号をつけるための数えている変数が、local_var_pos である。割り当てるときには、local_var_pos を１つ加えてこの値がスタック上の局所変数の位置になる。
2. これがきまったら、変数のシンボルとこのスタック上をペアにして、Env に登録しておく。登録は、1 で決めたスタックの位置と、var_kind を VAR_LOCAL にしておく。
3. block の本体の文をコンパイルする。
4. 本体のコンパイル中に局所変数が現れた場合には、Env を探して、どこに割り当てられているかによって、LOADA/LOADL/STOREA/STOREL 命令を生成される。
5. block の全部のコンパイルが終わったら、局所変数について変化させた Env を元にもどしておく。これによって、局所変数に使われた領域は参照されなくなる。

return 文のコンパイル

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);
}
```

st_compile.c

1. 式をコンパイル。結果は、スタック上に結果が残るはずである。
2. これで、RET 命令を生成する。

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

while 文についてはプログラムのソースコードを参照のこと。for 文は自分でつくってみること。

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。

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 < foo.c > foo.i
```

とすればよい。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
```

とすればよい。もしも、連続して動かす場合には、

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

としてもよい。

プログラミング言語処理

１３．レジスタマシンへのコンパイラ

前回は、スタックマシンにコンパイルする方法を解説した。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

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。

- 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個あるが、tiny C では、整数レジスタのみをつかう。このほかに、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%ebp がある。すべてのレジスタに e がついているのは extend の意味で、32 ビットで使用するときに用いる。これは、x86 が 16 ビットマシンであったときの名残である。即値は$n と$をつけて記述する。
- オペランドの記述については、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。

- 命令セット：

命令	記法	説明
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

スタックマシンではコンパイラに都合が良いように呼び出し規則を考えたが、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。

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

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。なお、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。

さて、関数のフレームは右の図のようになっている。関数の先頭では、まず最初に前の関数のフレームポインタをスタックに push し、ここに現在のフレームポインタをセットする。次に、レジスタの待避領域、局所変数の領域を確保して、スタックポインタをセットする。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。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 を使わないので、まずはじめに%ebx を待避しておく。

```
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
```

関数から戻る場合には、待避していた%ebx を元にもどし、leave 命令で、%ebp,%esp をもどして、ret 命令で呼び出し側に戻る。

スタック上に確保する領域は、%ebx の待避領域、レジスタの待避領域、局所変数の領域の合計したものである。レジスタには数に限りがあるのでレジスタが足りなくなったり、関数呼び出しがある場合には、レジスタの退避領域に保存しておく。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。そのため、例えば、１番目の局所変数は、(1+4)*4=20 から、さらに-4 のところ、つまり-24(%ebp)でアクセスすることになる。第一の引数は、逆にフレームポインター待避領域、戻り番地の後であるから、8(%ebp)でアクセスすることができる。

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。

tinyC のターゲットとして考える中間コードは、以下のコードである。以下の説明で、*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仮想的なレジスタ*というべきものである。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。

中間コード	説明
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。
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。
BEQO 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。

なお、このように

```
op dst,src1,src2
```

というような形式のコードを、*四つ組*と呼ばれる。
このほかに、命令に近い形に表現する *RTL(Register Transfer Language)*という形式もある。

中間コードは、reg_code.h に定義してある。スタックマシンの場合と同じように、関数ごとにコンパイルされるが、中間コードは一時的な領域に保存しておき、関数のコンパイルが終るごとに実際の x86 のコードに変換して出力される。中間コードには以下のルーチンを使う。

```
struct _code {
    int opcode;
    int operand1,operand2,operand3;
    char *s_operand;
} Codes[MAX_CODE];

int n_code;

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;
}
```

x86_code_gen.c

スタックマシンのコンパイラと同様に、関数のコードを生成する前に、initGenCode を呼び出し、領域をクリアする。それぞれの中間コードは、genCode1,genCode2,genCode3,genCodeS を使って生成する。スタックマシンとは異なり、オペランドは最大 3 つまで持つことに注意。

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。式のコンパイルから考えてみることにしよう。

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している。compileExpr は、引数の AST の式に対して、「コードを実行すると、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。

```
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:
         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;
    case LT_OP:
         r1 = tmp_counter++; r2 = tmp_counter++;
```

```
        compileExpr(r1,p->left);
        compileExpr(r2,p->right);
        genCode3(LT,target,r1,r2);
        return;
    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");
    }
}
```

reg_compile_expr.c

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. 式が変数であれば、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。

ここでは、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である。なお、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;
```

は大丈夫であるが、

```
x = (y=1)+1;
```

のように式のなかで、y=1 は使えない。(代入式は代入文として扱ってもいいが、そうするとfor文の中にはつかえなくなってしまう。)

変数の割り当ての情報を示す環境は、スタックマシンと同じである。

```
#define VAR_ARG 0
#define VAR_LOCAL 1

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

reg_compile.h

スタックマシンと同様に、var_kind にはパラメータ変数であるか(VAR_ARG)、局所変数であるか(VAR_LOCAL)を示す var_kind と関数フレーム上の割り当て位置を示す pos がある。

compileLoadVar と compileStoreVar はこの環境を調べて、適当なロードとストアのコードを生成する。

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

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");
}

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");
}
```

reg_compile.c

compileStoreVar は、レジスタ r を変数にストアするコード、compileLoadVar は、変数を target にロードするコードを生成する。

関数のコンパイル

コンパイラの main プログラムは、スタックマシンのコンパイラとまったく同じである。

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

compiler_main.c

parser の cparse.y や字句解析の lex.c はインタプリタと同一のものを使い、yyparse からは関数定義が入力されるごとに、defineFunction や declareVariable が呼び出される。

defineFuction も、スタックマシンのものとまったく同じである。

```
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,param_pos);
    envp = 0;  /* reset */
}
```

reg_compile.c

パラメータの変数を環境に登録し、本体を compileStatement でコンパイルする。

文のコンパイル

文のコンパイルは、compileStatement で行う。compileStatement もスタックマシンのものとほとんど同じである。

```
void compileStatement(AST *p)
{
    if(p == NULL) return;
    switch(p->op){
    case BLOCK_STATEMENT:
         compileBlock(p->left,p->right);
         break;
    case RETURN_STATEMENT:
         compileReturn(p->left);
         break;
    case IF_STATEMENT:
         compileIf(p->left,getNth(p->right,0),getNth(p->right,1));
         break;
    case WHILE_STATEMENT:
         compileWhile(p->left,p->right);
         break;
    case FOR_STATEMENT:
         compileFor(getNth(p->left,0),getNth(p->left,1),getNth(p->left,2),
                    p->right);
         break;
    default:
         compileExpr(-1,p);
    }
}
```

reg_compile.c

但し、最後の式を文としてコンパイルする時は、ターゲットを-1 にして compileExpr を呼び出し、コード生成を行う。

それぞれの文の処理は、スタックマシンのものと非常によく似ている。

```
void compileBlock(AST *local_vars,AST *statements)
{
    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;
}
```

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

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;
}

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);
    }
}
```

reg_compile.c

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。

なお、while 文についてはプログラムのソースコードを参照のこと。for 文は自分でつくってみること。

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）に変換して出力する。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。

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）

それぞれに、0 から 3 の番号を割り当て、

- 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

の２つの配列を準備する。

```
#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タ */
int tmpRegSave[N_SAVE];  /* 退避領域にある仮想レジスタ */
```

x86_code_gen.c

tmpRegName は、番号からレジスタ名を求める時に使う配列である。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。

退避領域と局所変数の bp 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。bp から引数のオフセットを計算する関数が、ARG_OFF である。

```
#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
```

x86_code_gen.c

退避領域は、常に 4 ワード分確保していることに注意。

まず、これらの情報を初期化する関数が、initTempReg である。

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

x86_code_gen.c

全ての値を、-1 にする。-1 は使われていないことを示す。

次に、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");
}
```

x86_code_gen.c

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;
}
```

x86_code_gen.c

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[rr]);
            return rr;
        }
    }
    error("reg is not found");
}
```

x86_code_gen.c

退避されている値を実レジスタにロードする場合には、

```
movl TMP_OFF(i)(%bp),レジスタ
```

が、出力される。

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");
}

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

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

x86_code_gen.c

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。

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。
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。

```
void genFuncCode(char *entry_name, int n_local)
{
    int i;
    int opd1,opd2,opd3;
    int r,r1,r2;
    char *opds;
    int ret_lab,l1,l2;
    int frame_size;

    /* 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 ,@function */
    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");
```

```
    initTmpReg();

    for(i = 0; i < n_code; i++){
        /*debug*//* printf("%s %d %d %d¥n",code_name(Codes[i].opcode),
                Codes[i].operand1,Codes[i].operand2,Codes[i].operand3); */
        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),tmpRegName[r]);
            break;
        case LOADL:       /* load local */
            if(opd1 < 0) break;
            r = getReg(opd1);
            printf("¥tmovl¥t%d(%%ebp),%s¥n",LOCAL_VAR_OFF(opd2),tmpRegName[r]);
            break;
        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(opd2));
            break;
```

x86_code_gen.c

1. 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)
```

2. 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。
3. LOADI に対しては、

```
movl $n,reg
```

を生成する。

4. 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。
5. 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。
6. 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。
7. 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。

```
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;
```

x86_code_gen.c

1. 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。
2. ラベル、JUMP 命令については、上の通り。

```
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;
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;
```

x86_code_gen.c

1. ARG コードは、push 命令で生成される。使った後は free しておく。
2. 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。
3. 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 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]);
        break;
    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;
    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:
            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("¥tjg .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;
```

x86_code_gen.c

1. 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。
2. 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。
3. 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 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;
        }
    }

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

x86_code_gen.c

1. RPINTLN では、外部関数である println を呼び出すコードを生成する。
2. 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。

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

```
        leave
        ret
```

最初に退避した%ebx を復帰し、leave 命令で%ebp,%esp を戻し、ret 命令で戻る

なお、最後に文字列については、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;
}
```

x86_code_gen.c

変数と配列の宣言、大域変数

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。
    1. 変数をロード/ストアする中間コード
    2. 配列のアドレスをロードするコード
    3. 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。

コンパイラと実行

以上説明したコンパイラ tiny-cc-x86 を使ってプログラムをコンパイル、実行する。tiny-cc-x86 は、これまでと同じく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 < foo.c > foo.s
```

とすればよい。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
```

とすれば、実行できる。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。

プログラミング言語処理

１４．コード最適化

最適化とは、効率のよい目的プログラムを生成することである。

「効率のよい」

とはいろいろな意味がある。例えば、なるべくサイズの小さいコードを生成するのも「効率のよい」という意味にもなる。コードを小さくするためには、なるべく小さい命令コードですむスタックマシン（大抵、１バイトで表現されるためバイトコードとも呼ばれる）にし、それを仮想スタックマシンで実行する方法もこの一つである。しかし、一般的には「効率のよい」とは、速いコード、すなわち実行時間が短いコードのことをいう。また、「最適化」といっているが、「最適」は事実上、不可能であるため、ここではなるべく効率を改善するという意味である。

実行時間を短くするには、大別して以下のことを考える

1. 命令の実行回数を減らす。より早い命令（もしくは命令の組み合わせ）を使う。
2. メモリ階層を効率的に使う。
3. 並列度の高い命令を使う。

命令の数を減らしても必ずしも、速いとはいえない。RISC マシンでは大抵の命令は一定のサイクル（１サイクル？）で実行されるために命令数を減らすことは実行時間の短縮になるが、CISC マシンではそうはいえない。命令数を減らす最適化は、基本的には無駄な計算を取り除くことである。例えば、ループ内で同じ計算している場合には、これをループの外で計算することによって、大幅に命令数を減らすことができる。

現在のマイクロプロセッサはレジスタ、キャッシュ（１次、２次）、メモリ、そしてディスクというメモリ階層をもっている。特にコンパイラではレジスタを効率的に使うことは重要な最適化になっている。もっと進んだコンパイラでは、ループの実行順序を入れ替えて、なるべくキャッシュを効率的に使う最適化を行うものもある。

プロセッサの中で命令は並列に実行されるのが普通である。現在のマイクロプロセッサではスーパースケラ（SuperScalar）機構があるが、この機構を効率的に使うために命令をいれ変える。また、数値計算を効率的に行うベクトルマシンに対しては、この機構を利用するコードを生成するが、これも並列度を持つ命令を使う最適化の一つである。

コンパイラで最も重要な最適化は、ループに関する最適化である。このループ最適化は上に挙げた最適化の組み合わせで行われる。多くのプログラムで、比較的小さい部分が実行時間のほとんどを占めるといわれている。つまり、数個のループを最適化することで実行時間を多くを改善することができる。

コンパイラでできない（できることもある？）最適化は、アルゴリズムの最適化である。例えば、ソートする部分をバブルソートからもっとよい quick ソートにするだけで大幅に性能が改善する。しかし、バブルソートから自動的に quick ソートに変換することはコンパイラではできない。コンパイラは、ひどいアルゴリズムを救うことはできない。このような最適化はプログラムの本質の部分であり、プログラマの力量が問われるところでもある。

命令の実行回数を減らす最適化

命令の実行回数を減らす最適化は以下のものがある。

1. １度計算した結果を再利用する。（共通部分式の削除）
2. コンパイル時に実行できるものは実行（計算）しておく。（定数の畳込み）
3. 命令をより、実行頻度が低い部分に移す。（ループ最適化：ループ不変式の削除）
4. 実行回数を減らすように、プログラムを変換する。（ループ変換）
5. 式の性質を利用して、実行を省略する。（帰納変数の削除、演算子の強さの低減）
6. 冗長な命令を取り除く。（死んだコードの削除、複写の削除）
7. 特殊化する。（手続き呼び出しの展開、判定の展開）

共通部分式の削除(common sub-expression elimination)

```
a=b+c;          (1)
 ....
x = (b+c)*e;  (2)
```

というコードがあった時に、b+c に関して、(1)が先に実行されていて、(1)から(2)の間に b+c が変わらない時、(2)の b+c は、a に置き換えることができる。これを、*共通部分式の削除*という。

定数の畳込み(constant folding)、定数伝播(constant propagation)

```
a=3+4    (1)
....
b = a*2   (2)
```

に対して、(1)を a=7 にしてしまう最適化を、*定数の畳み込み*と呼ぶ。共通部分式の削除と同様に、(1)の後に(2)が実行され、(1)から(2)の間に a の値がかわらなければ、b=14 にしてしまうことができる。この最適化を、*定数伝播*と呼ぶ。

ループ不変式の削除（loop invariant motion)

```
for(i=0; i < 10; i++){
    ....
    a=b*c;
    ...
}
```

ループのなかで、b と c の値が変わらなければ、a=b*c は、ループの外に出しておくことできる。

```
a=b*c;
for(i=0; i < 10; i++){
    ....
    ...
}
```

ループ内で変わらない式を*ループ不変式*で、これをみつけ移動すること*(code motion)*を*ループ不変式の削除*という。このためには、ループ内で、代入されていなく、かつ関数呼び出しがあった場合そこでも代入されていないことを確かめる必要がある。

帰納変数の削除(reduction variable elimination)、 演算子の強さの低減(strength reduction)

```
for(i=0; i < 10; i++){
    ....
    k = 10*i+123;
    ...
}
```

ループを制御するｉのような変数を*ループ変数（loop variable）*と呼ぶ。ループ内に、i=i+c しか代入がない変数を*基本帰納変数(basic induction variable)*という。ループ変数は、基本帰納変数である。この基本帰納変数に対し、*帰納変数(reduction variable)*とは、基本帰納変数もしくは、変数に代入されるたびにｉの線形の関数になる変数である。k は帰納変数である。

この帰納変数に対し、

```
k=123;
for(i=0; i < 10; i++){
    ....
    k = k+10;
    ...
}
```

と変形できる。この場合、i*10 という乗算を加算というより簡単な演算に変形しているが、これを演算子の*強さの低減(strength reduction)*と呼ぶ。一般に乗算よりも加算は速く実行できるので、効率的になる。この最適化は帰納変数 x に対し、線形の計算 a*x+b に適用できる最適化である。

Ｃ言語では配列を使った計算を

```
for(i = 0; i < 10; i++){
    ....
    x = a[i];
    ...
}
```

と書くことができるが、この最適化を使って

```
p = a;
for(i = 0; i < 10; i++){
   ....
   x = *p;
   ...
   p++;
}
```

と変形され、余計なアドレス計算を削除する最適化が行われれることがある。多次元配列などについては、そのままコード生成すると乗算が必要となるが、この最適化で加算のみで計算されるようになる。

なお、演算子の強さの軽減とは、一般的には x*2 を x+x としたり、x**2(べき乗)を x*x としたり、より簡単な演算に置き換えることをいう。

ループ展開(loop unrolling)、ループ融合(loop fusion)

```
 for(i = 0; i < 100; i++){
   a[i]= ...;
}
```

について、これを刻み幅を 2 にして、

```
 for(i = 0; i < 100; i+=2){
   a[i]=....;
   a[i+1]=...
}
```

にすることを*ループ展開*という。これにより、条件判定が半分になるほか、ループ内のレジスタの割り当てが効率的になることがある。

また、

```
for(i = 0; i < 10; i++){
     .... a[i] = ...; ...
}
 for(i = 0; i < 10; i++){
     .... b[i] = ...; ...
}
```

を

```
 for(i = 0; i < 10; i++){
    ... a[i] = ...; ...
    ... b[i] = ...; ...
}
```

としてしまうことを*ループ融合*という。この場合、ループ間でデータの依存がないことを確認しなくてはならない。

死んだ命令の削除(dead code elimination)

不要な命令を dead code という。例えば、

```
...
goto L;
x = ...
L: ...
```

では、x への代入は実行されない。この場合は dead code であり、削除できる。また、

```
x=100;   (1)
 ...
x = i+j;  (2)
```

というように、(1)の後に(2)が実行され、(1)から(2)の間に x が使われなければ、(1)は不要な命令であり、削除できる。

複写の伝播(copy propagation)

```
  a= b; (1)
  ...
  c=a+d; (2)
```

で、(1)の後に(2)が実行され、a も b も代入されなければ、(2)は、c=b+d にすることができる。これを*複写の伝播*という。

コードの巻き上げ(code hosting)

```
if(...) {
   ...
   T=a+b;
   ...
} else {
    ...;
    T=a+b;
    ...
}
```

のように、分岐先で同じ計算をする場合、

```
T=a*b;
if(...) {
  ...
} else {
  ...
}
```

とコードを移動することを*コードの巻き上げ(code hosting)*という。

手続き呼び出しの特殊化、式の性質の利用

```
foo(i)
{
   if(i < 10) return i+2;
   else return i*2;
}
```

という関数呼び出しに対して

```
x = foo(5);
```

の foo を展開して、

```
  {  i= 5; if(i < 10) x= i+2; else x=i*2;}
```

さらに、x=7 としてしまうことができる。

また、

```
if(a && b) x = 100;
```

で、a が true であることがわかれば、条件を取りぞのいて、

```
x=100;
```

にしてしまうことができる。 また、x*1 は x, y+0 は、y というように式の性質を利用して計算を省略できる。
