2023 年度
システムプログラミング実験

コンパイラ
実験レポート

実験日：2023 年 12 月 10 日（日）
提出日：2023 年 12 月 18 日（月）
学修番号：22140003
氏名：佐倉仙汰郎

1 実験の目的

C 言語などの高級言語のプログラムは直接 CPU で実行することができない．そのため、CPU で実行できる、機械語にコンパイルする必要がある．本実験では、ソースコードを仮想的なスタック機械の命令コードに翻訳するコンパイラを作ることを目的とする．またそのうえで、コンパイラ作成に必要な、構文解析や字句解析について学んでいく．

2 開発環境

- Central Processing Unit: 11th Gen Intel(R) Core(TM) i7-1165G7[*1], 2.80 GHz
- 主記憶: Double Data Rate 4 Synchronous Dynamic Random-Access Memory[*2], 1200 MHz, 256 bit
- コンパイラ: gcc version 11.4.0[*3]
- コンパイルコマンド: gcc compiler.c (compiler.c はソースプログラムのファイル名である．)
- Operating System: Arch Linux[*4] (カーネルは 5.14.8-arch1-1 を使用)

[*1] https://www.intel.co.jp/content/www/jp/ja/products/sku/208921/intel-core-i71165g7-processor-12m-cache-up-to-4-70-ghz-with-i specifications.html
[*2] https://www.jedec.org/standards-documents/docs/jesd79-4a
[*3] https://gcc.gnu.org/
[*4] https://archlinux.org/

3 作成したプログラム

```
int lexan()
{
    int t;
    while(1) {
        t = getchar();
        if (t == ' ' || t == '\t')
            ;
        else if ( t == '\n')
            lineno = lineno + 1;
        else if (t==':'){
        t=getchar();
        if (t=='='){
        return ASSIGN;}
        }
        else if (isdigit(t)) {
            ungetc(t, stdin);
            scanf("%d", &tokenval); // for gcc
            //scanf_s("%d", &tokenval); // for Visual Studio
            return NUM;
        }
        else if (isalpha(t)) {
            int p,b = 0;
            while (isalnum(t)) {
                lexbuf[b] = t;
                t = getchar();
                b = b + 1;
                if (b >= BSIZE)
                    error("compiler error");
            }
            lexbuf[b] = EOS;
            if (t !=EOF)
                ungetc(t, stdin);
            p = lookup(lexbuf);
            if (p == 0)
                p = insert(lexbuf, ID);
            tokenval = p;
            return symtable[p].token;
        }
        else if (t == EOF)
            return DONE;
        else {
            tokenval = NONE;
            return t;
        }
```

lexan() では、字句解析をおこなっている．標準入力を解析し正しい定数を返り値として渡すのがこの関数の目的である．代入演算子を読み込めるようにするために、条件文を用いて、=が読み込まれまたその次の入力が: だった場合に、新しく定義した定数 ASSIGN を返している．また他には、数かアルファベットかなどを判断したり、行数のカウントもここで行われている．

```
#include <stdio.h>

void parse()
{
    lookahead = lexan();
    while (lookahead != DONE) {
        stmt(); match(';');
    }
}
```

図 2　parse()

parse 関数では構文解析を行っている．parse 関数を呼び出し字句をして常に次のトークンを確認している．stmt 関数を呼び出すことで、構文解析が始まり、トークンが; だった場合に終了するようになっている．

```
void expr()
{
    int t;
    term();
    while(1)
        switch (lookahead) {
        case '+': case '-':
            t = lookahead;
            match(lookahead); term(); emit(t, NONE);
            continue;
        default:
            return;
        }
}
```

図 3　expr()

expr 関数では、まず構文木の仕組みから次に生成される term を呼び出す．そして式として扱う加算と減算を処理する．次のトークンが + もしくは-だった場合には、さらに term へと分岐していき、その後演算子を出力する．

```
void stmt(){
  if(lookahead == ID){
        printf("lvalue ");
        emit(ID,tokenval);
        match(lookahead);
        if(lookahead == ASSIGN){
              match(lookahead);
              expr();
              printf(":=\n");
        }
  }
  else if(lookahead == WHILE){
        match(lookahead);
        printf("label test\n");
        cond();
        printf("go false\n");
        if(lookahead == DO){
              match(lookahead);
              stmt();
              printf("goto test\n");
              printf("label out");
        }
  }
  else if(lookahead == BEGIN){
        while(1){
              if(lookahead == END){
                    break;
              }
              if(lookahead != END){
              match(lookahead);
              stmt();}
        }
  }
}
```

図 4　stmt()

stmt 関数は構文木上で文を表すために実装した関数である．stmt 関数では、Identifier、ループ文、begin と end をもちいた文を処理する．まず次のトークンが ID だった場合、まずはそのトークンを ”*lvalue* ” として出力する．その次のトークンに進みそのトークンが代入演算子の ASSIGN だった場合、expr 関数を呼び出し数字を出力した後に、:=を出力する．トークンが WHILE だった場合、まず ”*labeltest*” と出力し、次のトークンを読み込む．cond 関数を呼び出し”go false” を出力し、条件の中

身の出力が終了する．次のトークンが DO である場合、次のトークンに移動し、stmt 関数を呼び出すことで命令内容の翻訳が完了する．最後に”goto test” と”label out” を出力し完了する．begin と end を用いた文では、while ループの中で、END が読み込まれるまで、次のトークンを読み込む →stmt 関数を呼び出すというサイクルを続けることで、構文木が作成される．END が読み込まれたら break で終了する．

```
        void term()
{
          int t;
          factor();
          while(1)
                    switch (lookahead) {
                    case '*': case '/': case DIV: case MOD:
                              t = lookahead;
                              match(lookahead); factor(); emit(t, NONE);
                              continue;
                    default:
                              return;
                    }
}
```

図 5　term()

term 関数ではまず factor 関数を呼び出し、構文木を作っていく．term 関数の中では、乗算除算を処理するので、while 文の中で、乗除算を表す演算子が読み込まれた場合、次のトークンへ移動、factor 関数を呼び出す、演算子を出力という順番で処理する．

```
        void cond()
{
          int t;
          expr();
          while(1)
                    switch (lookahead) {
                    case '<':
                              t = lookahead;
                              match(lookahead); expr();emit(t,NONE);
                              break;
                    case '>':
                              t = lookahead;
                              match(lookahead); expr();emit(t,NONE);
                              break;
                              //printf(">\n");
                    case '=':
                              t = lookahead;
                              match(lookahead); expr(); emit(t,NONE);
                              break;  //printf("=\n");
                    default:
                              return;
                    }
}
```

図 6　cond()

cond 関数では比較演算子を処理している．stmt 関数で WHILE 文だと判断されたときに、WHILE 文の条件式にこの比較演算子は出てくるので cond 関数が呼ばれてる．ここではほかの関数 expr などと似たような手順を踏んでいる．次のトークンが $>, <, m, =$ のいずれかの場合新しいトークンの呼び出し →expr 関数の呼び出し → 数値の出力を行ってる．

```
void factor()
{
        switch(lookahead) {
                case '(':
                        match('('); expr(); match(')'); break;
                case NUM:
                        printf("push ");emit(NUM, tokenval); match(NUM
                            ); break;
                case ID:
                        printf("rvalue ");emit(ID, tokenval); match(ID);
                            break;
                default:
                        error("syntax error"); break;
        }
        //printf("factor finished\n");
}
```

図 7　factor()

factor 関数は構文木の葉にあたるものになる．stmt 関数から呼び出され様々な形で分岐するが最終的にはこの factor 関数にたどりつく．ここでは、数、ID、括弧を処理する．括弧を読み込んだ時には括弧内の式を処理するために expr 関数を呼び出しペアになる括弧をマッチさせ終了させる．数が読み込まれた場合、まず"push" を出力する．そして数を emit 関数を用いて出力し、終了する．ID を読み込まれた場合には、まず"rvalue" と出力し、その後数が読み込まれたときと同様な手順をたどる．その他の場合には"syntax error" となる．

```
        void match(t)
        int t;
{
        if (lookahead == t)
                lookahead = lexan();
        else error("syntax error");
}
```

図 8　match()

match 関数では、lexan 関数を用いて新しいトークンを読み込む.

```
        void emit(t, tval)
        int t, tval;
{
        switch(t) {
                case '+': case '-': case '*': case '/': case '>': case '<' : case '=':
                        printf("%c\n", t); break;
                case DIV:
                        printf("DIV\n"); break;
                case MOD:
                        printf("MOD\n"); break;
                case NUM:
                        printf("%d\n", tval); break;
                case ID:
                        printf("%s\n", symtable[tval].lexptr); break;
                default:
                        printf("token %d, tokenval %d\n", t, tval); break;
        }
}
```

図 9　emit()

emit 関数は、主に出力をする関数である．変更点は、比較演算子を読み込めるように、case に追加した．case を使って様々な場合において適切に出力ができるようになっている．

```
        int lookup(s)
        char s[];
{
        int p;
        for (p = lastentry; p > 0; p = p −1)
                if (strcmp(symtable[p].lexptr, s) == 0)
                        return p;
        return 0;
}
```

図 10　lookup()

lookup 関数ではシンボルテーブル内で引数に一致するものがあるかないかを判断している．あったら p を返し、ない場合には 0 を返却する．

```
int insert(s, tok)
char s[];
int tok;
{
    int len;
    len = strlen(s);
    if (lastentry +1 >= SYMMAX)
        error("symbol table full");
    if (lastchar + len +1 >= STRMAX)
        error("lexemes array full");
    lastentry = lastentry +1;
    symtable[lastentry].token = tok;
    symtable[lastentry].lexptr = &lexemes[lastchar + 1];
    lastchar = lastchar + len + 1;
    strcpy(symtable[lastentry].lexptr, s);
    return lastentry;
}
```

図 11　insert()

insert 関数は新しいシンボルをシンボルテーブルに追加する．シンボルテーブルの容量を確認し満杯であればエラーを吐く．また、lexemes 配列の容量も確認して、満杯であればこれもエラーを吐くようになっている．

```
void init()
{
    struct entry *p;
    for (p = keywords; p -> token; p++)
        insert(p -> lexptr, p -> token);
}
```

図 12　init()

init 関数ではシンボルテーブルを初期化している．

```
void error(m)
char *m;
{
    fprintf(stderr, "line %d: %s\n", lineno, m);
    exit(1);
}
```

図 13　error()

error 関数はエラーメッセージを表示してプログラムを終了するためのエラー処理関数である.

```
void main()
{
        init();
        parse();
        exit(0);
}
```

図 14　main()

main 関数ではまずシンボルテーブルを初期化して、その後字句解析を始めるここですべての入力を処理して最後に終了する.

4　プログラムの実行結果

4.1　代入文

図 15　代入文を実行したときの結果

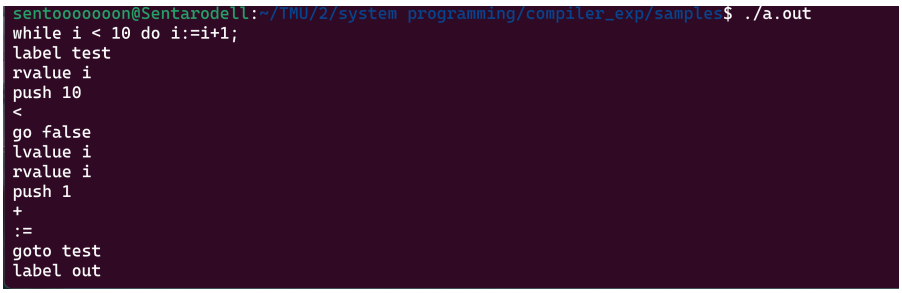

図 16　WHILE 文を実行したときの結果

図 17　begin を実行したときの結果

5 まとめ

今回の実験は非常に私にとって大変なものであった．最初に字句解析と構文解析をテキストで理解しようと思ったがうまくいかずコードをいじりながら理解していった．またこのレポートを書くことでよりコンパイラの理解が深まった．このコンパイラ実験で関数の有用性を強く感じた．関数を分けて置くとエラーが起こった時の修正がしやすい．また構文木の再現などやれることの幅もとても広がるのだと感じた．

```
// global.h
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define BSIZE 128
#define NONE −1
#define EOS '\0'

#define ASSIGN 261
#define NUM 256
#define DIV 257
#define MOD 258
#define ID 259
#define DONE 260
#define WHILE 261
#define DO 262
#define BEGIN 263
#define END 264

int tokenval;
int lineno;

struct entry {
        char *lexptr;
        int token;
};

struct entry symtable[];

int lexan();
void parse();
void parse();
void expr();
```

```
void stmt();
void term();
void cond();
void factor();
void match(int);
void emit(int, int);
int lookup(char[]);
int insert(char[], int);
void init();
void error(char*);


// lexer.c
char lexbuf[BSIZE];
int lineno = 1;
int tokenval = NONE;

int lexan()
{
        int t;
        while(1) {
                t = getchar();
                if (t == ' ' || t == '\t')
                        ;
                else if ( t == '\n')
                        lineno = lineno + 1;
                else if (t==':'){
                t=getchar();
                if (t=='='){
                return ASSIGN;}
                }
                else if (isdigit(t)) {
                        ungetc(t, stdin);
                        scanf("%d", &tokenval); // for gcc
                        //scanf_s("%d", &tokenval); // for Visual Studio
                        return NUM;
                }
                else if (isalpha(t)) {
                        int p,b = 0;
                        while (isalnum(t)) {
                                lexbuf[b] = t;
```

```
                        t = getchar();
                        b = b + 1;
                        if (b >= BSIZE)
                                error("compiler error");
                }
                lexbuf[b] = EOS;
                if (t !=EOF)
                        ungetc(t, stdin);
                p = lookup(lexbuf);
                if (p == 0)
                        p = insert(lexbuf, ID);
                tokenval = p;
                return symtable[p].token;
        }
        else if (t == EOF)
                return DONE;
        else {
                tokenval = NONE;
                return t;
        }
    }
}



// parser.c
int lookahead;

void parse()
{
    lookahead = lexan();
    while (lookahead != DONE) {
        stmt(); match(';');
    }
}

void expr()
{
    int t;
    //printf("expr naka\n");
    term();
```

```
        while(1)
                switch (lookahead) {
                case '+': case '-':
                        t = lookahead;
                        match(lookahead); term(); emit(t, NONE);
                        continue;
                default:
                        return;
                }
}

void stmt(){
        if(lookahead == ID){
                printf("lvalue ");
                emit(ID,tokenval);
                match(lookahead);
                if(lookahead == ASSIGN){
                        match(lookahead);
                        expr();
                        printf(":=\n");
                }
        }
        else if(lookahead == WHILE){
                match(lookahead);
                printf("label test\n");
                cond();
                printf("go false\n");
                if(lookahead == DO){
                        match(lookahead);
                        stmt();
                        printf("goto test\n");
                        printf("label out");
                }
        }
        else if(lookahead == BEGIN){
                while(1){
                        if(lookahead == END){
                                break;
                        }
                        if(lookahead != END){
                        match(lookahead);
```

```
                stmt();}
            }
        }
}

void term()
{
        int t;
        //printf("term naka\n");
        factor();
        while(1)
            switch (lookahead) {
            case '*': case '/': case DIV: case MOD:
                t = lookahead;
                match(lookahead); factor(); emit(t, NONE);
                continue;
            default:
                return;
            }
}

void cond()
{
        int t;
        //printf("cond yobareta");
        expr();
        //printf("expr success");
        while(1)
            switch (lookahead) {
            /*case '*': case '/': case DIV: case MOD:
                t = lookahead;
                match(lookahead); factor(); emit(t, NONE);
                continue;*/
            case '<':
                t = lookahead;
                match(lookahead); expr();emit(t,NONE);
                //printf("rmit owari\n");
                break;
            case '>':
                t = lookahead;
                match(lookahead); expr();emit(t,NONE);
```

```
                    break;
                    //printf(">\n");
                case '=':
                    t = lookahead;
                    match(lookahead); expr(); emit(t,NONE);
                    break;  //printf("=\n");
                default:
                    return;
                //printf("cond seikou");
                }
}

void factor()
{
        switch(lookahead) {
                case '(':
                    match('('); expr(); match(')'); break;
                case NUM:
                    printf("push ");emit(NUM, tokenval); match(NUM); break;
                case ID:
                    printf("rvalue ");emit(ID, tokenval); match(ID); break;
                default:
                    error("syntax error"); break;
        }
        //printf("factor finished\n");
}

void match(t)
        int t;
{
        if (lookahead == t)
                lookahead = lexan();
        else error("syntax error");
}



// emitter.c
void emit(t, tval)
        int t, tval;
{
        switch(t) {
```

```
                case '+': case '−': case '∗': case '/': case '>': case '<' : case '=':
                        printf("%c\n", t); break;
                case DIV:
                        printf("DIV\n"); break;
                case MOD:
                        printf("MOD\n"); break;
                case NUM:
                        printf("%d\n", tval); break;
                case ID:
                        printf("%s\n", symtable[tval].lexptr); break;
                default:
                        printf("token %d, tokenval %d\n", t, tval); break;
        }
}


// symbol.c
#define STRMAX 999
#define SYMMAX 100

char lexemes[STRMAX];
int lastchar = −1;
struct entry symtable[SYMMAX];
int lastentry = 0;

int lookup(s)
        char s[];
{
        int p;
        for (p = lastentry; p > 0; p = p −1)
                if (strcmp(symtable[p].lexptr, s) == 0)
                        return p;
        return 0;
}

int insert(s, tok)
        char s[];
        int tok;
{
        int len;
        len = strlen(s);
```

```
        if (lastentry +1 >= SYMMAX)
                error("symbol table full");
        if (lastchar + len +1 >= STRMAX)
                error("lexemes array full");
        lastentry = lastentry +1;
        symtable[lastentry].token = tok;
        symtable[lastentry].lexptr = &lexemes[lastchar + 1];
        lastchar = lastchar + len + 1;
        strcpy(symtable[lastentry].lexptr, s);
        return lastentry;
}


// init.c
struct entry keywords[] = {
        "div", DIV,
        "mod", MOD,
        "while", WHILE,
        "do", DO,
        "begin", BEGIN,
        "end", END,
        0, 0
};

void init()
{
        struct entry *p;
        for (p = keywords; p -> token; p++)
                insert(p -> lexptr, p -> token);
}


// error.c
void error(m)
        char *m;
{
        fprintf(stderr, "line %d: %s\n", lineno, m);
        exit(1);
}
```

```
// main.c
void main()
{
        init();
        parse();
        exit(0);
}
```

参考文献

[1] 中田育男，コンパイラ: 作りながら学ぶ，オーム社，2017