■心機一転、環境構築

自宅用に新しいPCを購入しました。日本ヒューレット・パッカードのPavillion 2000という製品で、モニター込みで3万9800円という破格の値段で売られていたのです。そこそこ使えるPCがこんな値段で売られているんだから、IT産業の利益が薄くなっているのもうなずけますね。安売りしていた理由は一世代前のものだったからなのですが、最新(?)機種でも5万円を切る値段で売られていました。私のは古い機種なのでハードディスクの容量が7.5GBytesしか(なんて言葉が出るくらい最近は大容量が当たり前)ありません。USB接続のCD-RWドライブと通信カードインターフェイスSlipperを導入して、とりあえずのハードウェア環境は整いました。

DDI PocketのAirH"なるパケット通信と組み合わせて利用することで(私のところはADSLもフレッツISDNも来ていないし、ケーブルテレビは部屋には引けないので、これが唯一の定額通信手段なのです)、必要な時には大学のマシンにリモートで入ればいいと割り切って、Windows 98SEのまま使うことにします。論理合成関係のツールなどもWindows NT Terminal Serverにインストールしておけば(遅いのを我慢すれば)家からでも何とか利用できます。何がなんでもLinuxにしろとおっしゃる読者の方も多いかと思いますが、私にとってはx86のLinuxもWindows 98SEも必要なアプリケーションさえ安定して動けば、どちらでもかまいません。USB関連のドライバがWindows用しかないものが多いため、とりあえずは現実的な選択をしています。

とはいっても、最低限の使い勝手を確保するために、キーボードを101に交換し、IMEの起動キーをCtrl+SPACEに変更し、標準のエディタをgvimに変更しました。レジストリを書き換えて`notepad.exe`を呼び出しているところをgvimに変更するだけで気持ち良く使えるようになります。ちょっと注意が必要なのはキーボードドライバで、101keyboardドライバを追加するのはうまくいかず、106キー用ドライバのプロパティでドライバの更新を指定した後、手動で101ドライバを指定する必要があります。

余談ですが、プリインストールしてあるmultimedea keyboard maestroというアプリケーションを削除しておかないと、変更したPS/2ポートのデバイスが不安定な動作をするようです。

TeraTerm、TeX、Ghostscript、JDK、Max+Plus2、Cygwinなどを入れていくとアッという間にハードディスクの半分くらいは埋まっています。他に必要なのはドロー系のソフトですが、私としては普段使いなれたTgifを使いたいので、X端末が必要になります。フリーのものだとJavaベースのWeirdXというのがあります(記事末のResource【1】を参照)。

さらにいろいろと探していくと、昔はGNUのツール類程度しかなかったCygwinがずいぶん拡張されていてXFree86まで動くということなので、XFree86+Tgif+Cannaで図を描く環境を整えることができました。不思議なことに、DOSコンソールからCygwinを用いると遅くて文字の取りこぼしもあるのですが、XFree86上のrxvtは快適に動作するのです。メインメモリに余裕があれば用はなくてもXを立ち上げて使ったほうが気持ちよく使えるようです。ネイティブなrxvtは試していませんが、メモリの値段が下がった昨今では、素直にメモリを増設してXで使うほうがいいのではないでしょうか?

また今後、Agenda ComputingのAgenda VR3もいろいろと試したいので、Pavillion 2000か研究室のPCにクロス開発環境を設置しなくてはと思っています。Linuxでのクロス環境はx86であればAgenda ComputingのWebページ([2])からダウンロードできますから簡単ですが、Windowsでクロス環境を作るには、まずCygwinが動作する環境を作成する必要があります。binutilsとgccはCygwinで比較的簡単に構築でき、クロス開発で問題となるglibcはAgendaが配布するパッケージを展開すればいいのでこちらもそれほど大きな問題にはなりません。落ち着いたら試してみます。私が買ったAgenda VR3はDevelopers Editionなので市販品と色が少し違いますが、ROMなどの中身はいくらでもアップデートできるので使い勝手はチューニング次第ですね。

■yacc、インタプリタのソースコード

さて、本題に入りましょう。前号でコンパイラを作る話をしましたが、今回はyaccのソースコードとインタプリタのソースコードを一挙に説明します。CPUの命令を決定して命令コード生成の話をすると書いたのですが、コンパイラをいきなり説明するよりも、yaccのソースコードを見ながら1歩ずつ話を進めたほうが分かりやすいしょう。yacc/lexを利用していますが、Linuxではbison/flex([3])を用います。互換性のある機能しか使っていないので話としてはどちらでも同じです。bisonをyacc互換モードで使うために引数として「-y」を付ける必要がある点が違うくらいですね。先月号のlexのファイルも今月号のyaccのファイルもbison/flexだけでなく、Solaris上のyacc/lexでも動作を確認済みです。

リスト1はコンパイラ・インタプリタの構文を記述するyaccファイルです。yaccの文法はマニュアルやいろいろな書籍を参照してください。簡単な説明は、前号で紹介したePaperPress([4])にも英文で出ています。生成したCに含まれるコードとトークンとその優先順位の定義に続いて、「%%」で囲まれた部分が文法記述の本体です。この本体について少し詳しく説明していきましょう。

yaccでは、文法記述に従って入力を解析して解析木を作成します。解析木で使うノードの型は`toyp.h`で定義しています(リスト2)。`union`で複数の型をオーバーラップさせて定義していますが、要素`type`はすべての型で必ず先頭の要素として共通に持っているため、`nodeType.type`の形で参照できるようにしています。このインクルードファイルはePaperpressのものをそのまま使わせていただいています。定義している型の内、`oprNodeType`は解析の木を保存する一番重要な型です。`conNodeType`は定数を保存し、`idNodeType`は変数参照のときに用います。解析木は分岐することがあるのですが、インクルードファイルではオペランドの

【リスト1】toyp.y

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

/* prototypes */
nodeType *opr(int oper, int nops, ...);
nodeType *id(int i);
nodeType *con(int value);
void freeNode(nodeType *p);
void sinit(int val);
int ex(nodeType *p, int reg, int pres);

void yyerror(char *s);

int sym[65536];                    /* symbol table + memory*/
%}

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

%token <iValue> INTEGER RI EI
%token <sIndex> VARIABLE
%token WHILE IF PRINT MRD MWT OUT IN MEM IO LO FUNCNAME FDEF RETURN FUNC
%token DEF FDEFA
%token <nPtr> ARG ARGV
%nonassoc IFX
%nonassoc ELSE

%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%right UMINUS

%type <nPtr> stmt expr stmt_list

%%

program:
        init function               { exit(0); }
         ;

init:
        /* NULL */                  { sinit(SPTOP);}
         ;

function:
          function stmt                { ex($2,1,0); freeNode($2); }
        | /* NULL */
         ;

stmt:
          ';'                           { $$ = opr(';', 2, NULL, NULL); }
        | expr ';'                      { $$ = $1; }
        | DEF FUNCNAME '(' ARG ')' stmt     { $$ = opr(FDEFA,1, $6); }
        | DEF FUNCNAME '(' ')' stmt         { $$ = opr(FDEF, 1, $5); }
        | PRINT expr ';'                { $$ = opr(PRINT, 1, $2); }
        | EI ';'                        { $$ = opr(EI, 1, con($1)); }
        | RI ';'                        { $$ = opr(RI, 1, con($1)); }
        | IO '[' expr ']' '=' expr ';'  { $$ = opr(OUT, 2, $3, $6); }
        | MEM '[' expr ']' '=' expr ';' { $$ = opr(MWT, 2, $3, $6); }
        | VARIABLE '=' expr ';'         { $$ = opr('=', 2, id($1), $3); }
        | WHILE '(' expr ')' stmt       { $$ = opr(WHILE, 2, $3, $5); }
        | RETURN '(' expr ')' ';'       { $$ = opr(RETURN, 1, $3); }
        | RETURN ';'                    { $$ = opr(RETURN, 0); }
        | IF '(' expr ')' stmt %prec IFX { $$ = opr(IF, 2, $3, $5); }
        | IF '(' expr ')' stmt ELSE stmt { $$ = opr(IF, 3, $3, $5, $7); }
        | '{' stmt_list '}'             { $$ = $2; }
         ;

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

expr:
          INTEGER               { $$ = con($1); }
        | VARIABLE              { $$ = id($1); }
        | ARG                   { $$ = opr(ARGV, 0); }
        | '-' expr %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
        | IO '[' expr ']'       { $$ = opr(IN, 1, $3); }
        | MEM '[' expr ']'      { $$ = opr(MRD, 1, $3); }
        | FUNCNAME '(' expr ')' { $$ = opr(FUNC, 1, $3); }
        | FUNCNAME '(' ')'      { $$ = opr(FUNC, 0); }
        | expr '+' expr         { $$ = opr('+', 2, $1, $3); }
        | expr '-' expr         { $$ = opr('-', 2, $1, $3); }
        | expr '*' expr         { $$ = opr('*', 2, $1, $3); }
        | expr '/' expr         { $$ = opr('/', 2, $1, $3); }
        | expr '<' expr         { $$ = opr('<', 2, $1, $3); }
        | expr '>' expr         { $$ = opr('>', 2, $1, $3); }
        | expr GE expr          { $$ = opr(GE, 2, $1, $3); }
        | expr LE expr          { $$ = opr(LE, 2, $1, $3); }
        | expr NE expr          { $$ = opr(NE, 2, $1, $3); }
        | expr EQ expr          { $$ = opr(EQ, 2, $1, $3); }
        | '(' expr ')'          { $$ = $2; }
         ;

%%

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

    /* allocate node */
    if ((p = malloc(sizeof(conNodeType))) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeCon;
    p->con.value = value;

    return p;
}

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

    /* allocate node */
    if ((p = malloc(sizeof(idNodeType))) == NULL)
        yyerror("out of memory");

    /* copy information */
    p->type = typeId;
    p->id.i = i;

    return p;
}

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

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

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

void freeNode(nodeType *p) {
    int i;

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

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

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

要素を1要素分しか取っていません。分岐して複数のオペランドに解析が及ぶときには新たにメモリを割り当ててオペランドの配列を拡張しています。toyp.yの*opr()の関数の記述を参照してください。こうすると、配列の範囲を超えてアクセスするのですが、C言語ではレンジチェックをしないためこのようにしても問題なくプログラムができます。

このコンパイラでは1文を入力するごとに構文木を作成し、その実行をex()関数で行った後、構文木に割り当てたメモリをfreeNode()関数で開放します。プログラムはprogramという構文がトップレベルになります。この構文はinitとfunctionという2つの要素からなります。トップレベルの構文の解析が終わったらプログラムを終了するようにexit(0)を定義しています。initは構文要素は何もないのですが、始めに実行するsinitを呼び出すためにおいてあります。コンパイラではsinitの呼び出しでスタックポインタの調整を行うコードを出力します。インタプリタでも同様にすればいいのですが、static変数の初期化で同じことをやらせているので、この関数は何もしないでリターンするようにしています。

functionはfunctionとstmtか空文となります。実はstmtが入力文に該当するので、この記述はstmtの繰り返しを定義しているだけです。

構文解析の処理はexprやstmtなどの非終端記号がトークンで置き換えられるまで再帰的に置き換えを繰り返していきます。そのときにopr()などの関数を用いて構文木を作成していくのです。構文要素は左から$1、$2、$3……として参照されます。elseのないif文だけは1文を読み込んだだけでは判定できないので、次の文を読み込んでelseでないと分かったときにコード生成に移ります。そこで、手動でコンパイラを動かしていると一瞬戸惑いますが、バグではありません。

yacc自身は構文を解析して当てはまる文法規則の実行ルールを呼び出しているだけで、構文木を作成するのは呼び出された関数の責任になります。構文木を作成する関数はopr、id、conの3つを使っています。oprは操作一般を受け持つ関数でほとんどの構文はこの関数で処理します。idは変数を扱う関数になります。lexのファイルで変数名を数値に変換しているので、これをsym[]というメモリの実体を参照する構文木のノードと結びつけます。conは数値定数を扱う関数で、こちらもlexの中で定数が定義されるとその定数値をもつ構文木のノードを返すような関数になっています。これら3つの関数は呼び出されるごとに必要なノードをヒープ領域から取り出し、そこへのポインタを返します。コンパイラ・インタプリッタのメイン関数はこのyaccの定義の最後にあります。

次はインタプリタのコードです（**リスト2**）。中心となるのは構文解析木を実行するex関数です。解析木を引数として再帰的に実行を続けていきます。解析木のノードは定数(typeCon)、変数（typeId)、演算操作（typeOpr）の3つのタイプがあり、p->typeのメンバによって処理を切り分けています。定数の参照はノードのメンバの値を関数値として戻します。変数の参照はノードのメンバに記憶した変数名によって決まる数値から参照する変数テーブルの添え字を決定して配列の値を関数値として戻します。演算操作ではyaccのファイルにおいて記述したopr関数が作成した解析木の内容に従って処理を進めます。演算操作の種別はp->opr.operに書かれていますので、この値を使ってswitch文で場合分けをしています。多くの演算操作についてはソースコードを見ていくと直感的に分かると思いますので、説明が必要なものに絞っていくつか説明を加えます。

まずは、関数定義(FDEF,FDEFA)です。関数は定義した後に利用されるため、インタプリタでは実際の関数呼び出しよりも先に定義をしておきます（次の号で説明するコンパイラでは、

【リスト2】 toypi.c

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include "toyp.h"
#include "y.tab.h"

static nodeType *foo;
static int sp=SPTOP;
static jmp_buf funbuf[SPTOP];
static int val, jv;
int sinit(int init) {return 0;}
int ex(nodeType *p) {
    int tmpv;
    if (!p) return 0;
    switch(p->type) {
    case typeCon:       return p->con.value;
    case typeId:        return sym[p->id.i + 1];
    case typeOpr:
        switch(p->opr.oper) {
        case FDEF:
        case FDEFA:     foo=p->opr.op[0];
                        p->opr.op[0] = NULL;
                        return 0;
        case ARGV:
                        return sym[sp+1];
        case FUNC:      if(p->opr.nops>0) {
                        tmpv = ex(p->opr.op[0]);
                        sp -=2;
                        sym[sp+1] = tmpv;
                        jv = setjmp(funbuf[sp]);
                        if(jv == 0)
    val=ex(foo);
                        sp +=2;
                        return val;
                        } else
                           {
                        jv = setjmp(funbuf[sp]);
                        if(jv == 0)
                             val = ex(foo);
                        return val;
                           }
        case RETURN:    if(p->opr.nops>0) {
                        val = ex(p->opr.op[0]);
                        } else
                           {
                        val = 0;
                           }
                        longjmp(funbuf[sp], -1);
        case WHILE:     while(ex(p->opr.op[0])) ex(p->opr.op[1]); return 0;
        case IF:        if (ex(p->opr.op[0]))
                            ex(p->opr.op[1]);
                        else if (p->opr.nops > 2)
                            ex(p->opr.op[2]);
                        return 0;
        case PRINT:     printf("%d\n", ex(p->opr.op[0])); return 0;
        case ';':       ex(p->opr.op[0]); return ex(p->opr.op[1]);
        case '=':       return sym[p->opr.op[0]->id.i + 1] =
ex(p->opr.op[1]);
        case OUT:       printf("Port[%d] <- %d\n",
    sym[ex(p->opr.op[0])] , ex(p->opr.op[1]));
   return 0;
        case MWT:       return sym[ex(p->opr.op[0])] = ex(p->opr.op[1]);
        case MRD:       return sym[ex(p->opr.op[0])];
        case IN:        printf("Get random value for Port[%d]: ",
ex(p->opr.op[0]));
   return rand();
        case EI:
        case RI:        return 0;

        case UMINUS:    return -ex(p->opr.op[0]);
        case '+':       return ex(p->opr.op[0]) + ex(p->opr.op[1]);
        case '-':       return ex(p->opr.op[0]) - ex(p->opr.op[1]);
        case '*':       return ex(p->opr.op[0]) * ex(p->opr.op[1]);
        case '/':       return ex(p->opr.op[0]) / ex(p->opr.op[1]);
        case '<':       return ex(p->opr.op[0]) < ex(p->opr.op[1]);
        case '>':       return ex(p->opr.op[0]) > ex(p->opr.op[1]);
        case GE:        return ex(p->opr.op[0]) >= ex(p->opr.op[1]);
        case LE:        return ex(p->opr.op[0]) <= ex(p->opr.op[1]);
        case NE:        return ex(p->opr.op[0]) != ex(p->opr.op[1]);
        case EQ:        return ex(p->opr.op[0]) == ex(p->opr.op[1]);
         }
    }
}
```

命令コード生成の都合で、関数定義は呼び出しの後に記述します。そのためこの部分互換性がなくなりますが、手抜きコンパイラ・インタプリタなのでご勘弁を)。

定義された関数の本体はその先頭の解析木のノードを関数用にnodeType *fooの変数に退避しておきます。定義を実行した結果、解析木の開放が呼び出されますが、退避した関数本体の解析木が開放されることを防ぐために、本体の解析木の関数本体へのリンクをNULLにしておきます。関数引数がある場合には、スタック上にその引数をおきますので、引数の参照はスタックポインタの値に引数のオフセットを足してメモリ上の引数の値を返すようにしています。

関数本体の実行は、関数からのリターン文があるので少しやっかいです。リターン文で値を指定したときには関数値はその値になるという処理をしなくてはなりません。これにはsetjmp/longjmpの仕組みを使います。まず、引数がある場合には引数の評価を行って、スタックに積みます。スタックポインタと連動する関数バッファにsetjmpのバッファを指定しlongjmpで戻るポイントを指示しています。その後、退避しておいた関数本体の実行を行います。returnが実行されるとlongjmpでsetjmpの次に戻ります。戻ったときにはsetjmpの返り値が0でなくなるので、if文で判定を行うことで戻ってきたケースかどうかを判断します。setjmpの一回目の呼出し後は返り値は0となるため、ここではex文で関数本体を評価します。その結果は大域変数のvalに格納していますが、returnの発生時にはvalに返り値を格納しているので、その値が関数の値になります。exの関数は再帰的に呼ばれるので、大域変数での受け渡しに不安を感じる方もいるかもしれません。しかし、setjmp後スタックポインタで示されたジャンプバッファのところに戻るまでにこの返り値は利用されないため、再帰関数の中でおかしな値を受け渡すことにはなりません。もしどうしても心配なら、valを局所変数にして、longjmpのときに受け渡す返り値をスタック渡しに変更してみてください。スタックは関数呼出しごと2つのエントリを確保していますから、空いている側を使って受け渡しができます。

whileやif文の処理はプログラムを見ればすぐに分かると思います。if文はelseがあるかどうかで処理が異なりますが、elseがあることをオペランドの数で判断しています。

print文はコンパイラでは何もしないのですが、インタプリタでは引数を画面表示します。これも手抜きなのでフォーマットなど何も指定するようには作っていません。単に10進で表示がされるだけです。割り込みや入出力などもインタプリタでは実装してもあまり意味がないので、いいかげんな実装になっています。

その他必要なファイルはincludeファイルとMakefileですが、これも次のリストに示します(**リスト3**、**リスト4**)。エラー処理も何もないし、標準入力から入力したものを標準出力に表示するだけの手抜きインタプリタですが、エディタでソースファイルを作って

```
./toypi < hoge | tee hoge.out
```

のように使うことで手抜きの部分を補えます(笑)。いろいろ試してみたりコードを変更して機能を追加したり遊んでみてください。

次回は、いよいよCPUの命令の定義とコンパイラおよびアセンブラの設計を行います。もちろん、例によって手抜きコンパイラ・アセンブラなのはあらかじめ予告しておきます(笑)。

Resource

[1] WiredX
http://www.jcraft.com/weirdx/

[2] Agenda COMPUTING
http://www.agendacomputing.com/

[3] bison
http://www.gnu.org/software/bison/bison.html

[4] A Compact Guide to Lex & Yacc
http://www.epaperpress.com/lexandyacc/

【リスト3】toyp.h

```
#define SPTOP 50
typedef enum { typeCon, typeId, typeOpr } nodeEnum;

/* constants */
typedef struct {
    nodeEnum type;              /* type of node */
    int value;                  /* value of constant */
} conNodeType;

/* identifiers */
typedef struct {
    nodeEnum type;              /* type of node */
    int i;                      /* subscript to ident array */
} idNodeType;

/* operators */
typedef struct {
    nodeEnum type;              /* type of node */
    int oper;                   /* operator */
    int nops;                   /* number of operands */
    union nodeTypeTag *op[1];   /* operands (expandable) */
} oprNodeType;

typedef union nodeTypeTag {
    nodeEnum type;              /* type of node */
    conNodeType con;            /* constants */
    idNodeType id;              /* identifiers */
    oprNodeType opr;            /* operators */
} nodeType;

extern int sym[65536];
```

【リスト4】Makefile

```
#
# Makefile for Toyp Compiler
# Gnu C Version
# N. Shimizu 12 Aug 2001
#

CC = gcc
YACC = bison -y
LEX = flex

CFLAGS = -Wall -g

CLEANS = toypc.o toypi.o y.tab.o lex.yy.o y.tab.c lex.yy.c y.tab.h
OBJS = toypc.o y.tab.o lex.yy.o
OBJI = toypi.o y.tab.o lex.yy.o
SRCS = toypi.c toypc.c toyp.y toyp.l toyp.h

all: toypc toypi

toypi: $(OBJI)
 $(CC) $(CFLAGS) $(OBJI) -o toypi

toypc: $(OBJS)
 $(CC) $(CFLAGS) $(OBJS) -o toypc

toypc.o: toypc.c toyp.h y.tab.c
 $(CC) $(CFLAGS) -c toypc.c

toypi.o: toypi.c toyp.h y.tab.c
 $(CC) $(CFLAGS) -c toypi.c

y.tab.o: y.tab.c toyp.y
 $(CC) $(CFLAGS) -c y.tab.c

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

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

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

clean:
 -rm $(CLEANS)

tar:
 tar cvf toyp.tar Makefile Readme.html $(SRCS)
```