[第７回目２００２・１０・２２]

Stack Machine について

これからは、このインタープリタでつくった tinyC 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。

仮想スタックマシンの命令

tiny C 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。

- 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 して引き算し、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）

なお、この命令を持つ仮想マシンのインタプリタを作ってみた。st_machine.c を参照のこと。POP や、PUSHI, 演算 ADD,SUB などは、１回目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。コンパイルでは、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。

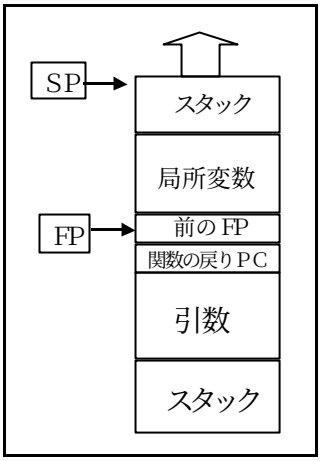
```
...条件文のコード...
BEQ0 L0   /* もし、条件文が実行されて、結果が 0 だったら, L に分岐*/
...then の部分のコード...
JUMP L1
LABEL L0
... else の部分のコード...
LABEL L1
```

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。

なお、else 部がないときには、IF 文の値が 0 とするために、PUSHI 0 をおいておかなくてはならないことを注意。



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。

次回、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。

前回の補足：

前回のプリントにあるプログラムを、wwwに掲載した。コンパイルの方法やまず、yacc を使って、cparser.y をCに変換しておく。

```
% yacc cparser.y
```

生成されたプログラムは、y.tab.c である。このプログラムとインタプリタ interp.c をコンパイルすればよい。

```
% cc –o interp interp.c y.tab.c
```

インタープリタを起動するには、標準入力から tiny-c のコードを入力してやればよい。例えば、このコードを test.tiny-c とすると,

```
% interp < test.tiny-c
```

これで、読み込まれた後、main から実行が始まる。

演習課題６：

前回、説明した tiny C のインタプリタの parser に、while 文と for 文を付け加え、以下のプログラムを実行できるようにしなさい。while 文と for 文は C と同等の構文である。提出は、修正したところのみでよい。実際に実行した結果を提出すること。

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