[第７回目２００２・１０・２９]

Stack Machine へのコンパイラ

前回は、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コンパイラについて解説する。これからは、このインタープリターでつくった tiny C について、コンパイラを作っていくことにする。スタックマシンのシミュレータはWWWに掲載してある st_machine.c を参照のこと。コンパイラでは、関数ごとにコンパイルしていく。そのため、yyparse で関数が定義されたときに、関数をコンパイルするルーチン compileFuncDef を呼び出し、関数をコンパイルする。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。parser や字句解析の lex.c はインタプリターと同一のものである。

```
#include "object.h"

main()
{
        initEval();
        yyparse();
}

void compile_error(char *msg)
{
        fprintf(stderr,"compiler_error:  %s",msg);
        exit(1);
}
```

www 上にあるプログラムのうち、コンパイラの中心である compile.c と compile_func.c について、説明する。

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）

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。

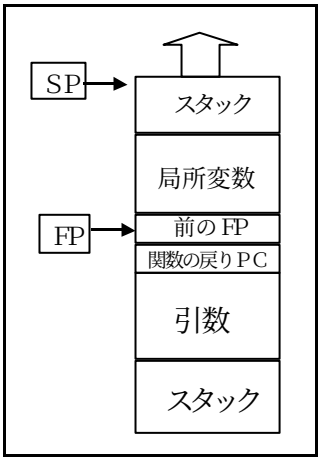
５、引数にアクセスするためには、FP から２つ離れたところにあるので、ここからとればよい。(LOADA/STOREA 命令)

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。



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(CompileFuncDef)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。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。

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

var_kind には、ローカル変数か引数かを格納しておくフィールドである。パラメータ変数の場合は、VAR_ARG をいれ、何番目に詰まれているかを pos にいれておく。

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、局所変数が何個合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。

式のコンパイル

インタープリタでは、evalObject という関数を使って式を実行したが、コンパイラでは compileObject という関数で式に対するコードを生成する。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。

さて、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。compileGetVar 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。なお、逆に、ローカル変数やパラメータ変数に代入する関数が、compileSetVar である。

制御文や block 文などについては、それぞれに対応するコンパイルのための関数を呼び出す。

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。

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 する命令 POPR 命令を生成しておく。スタックに積む順番は、引数の最後からなので、引数の最後からコンパイルしなくてはならないことを注意しよう（CompileArgs, この関数は同時に引数の個数を数えている）。

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（comileBlock）。

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。

3、block の本体をコンパイルする。各文に対応する式をコンパイルしたコードは、実行されると結果をスタック上に置くので、途中の文（式）に関しては、結果を POP しておくために、POP 命令を生成しておかなくてはならない。最後の文は block 文の値になるので、POP 命令は入れない。

4、本体のコンパイル中に局所変数が現れて場合には、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 命令を生成する。

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。

なお、代入文（式）も、値を持つ。したがって、STOREA/STOREL はスタックの top の値を変数（の位置）に格納するだけで、値はそのままスタックに残しておいていることに注意。

return 文のコンパイル

return 文のコンパイルは

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。

だけで、よい。

課題で、WHILE 文を考えてみよう。

出力文の println は、ちょっと変則なので、特別に扱ってある。

コンパイラとスタックマシンの実行

さて、web 上にあるプログラムをコンパイルするとコンパイラ tiny_cc とスタックマシンのインタプリターst_machine ができる。st_machine は、関数 main の最初、つまり、ENTRY main のあるところから、実行を開始する。

tiny_cc 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。

演習課題７：

www 上においてある tiny C のコンパイラでは while 文をわざと抜いてある。while 文をコンパイルする部分を付け加え、以下のプログラム(fac.c)をコンパイルできるようにし、実行せよ。提出は、修正したところのみでよい。

余裕のある人は、for 文のコンパイルを考えてみよ。

```
main()
{
    println("fac is %d", fac(10));
}

fac(n){
    var i,s;
    i = 1;
    s = 1;
    while(i < n){
       s = s * i;
       i = i + 1;
    }
    return s;
}
```