[第９回目２００２・１１・5]

レジスタのある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。

Pentium プロセッサ

MIPS アーキテクチャについては他の講義で学習していると思うが、x86 については今年度から導入されているマシンなので、簡単に解説しておく。x86 はいわゆる CISC マシンであるが、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。

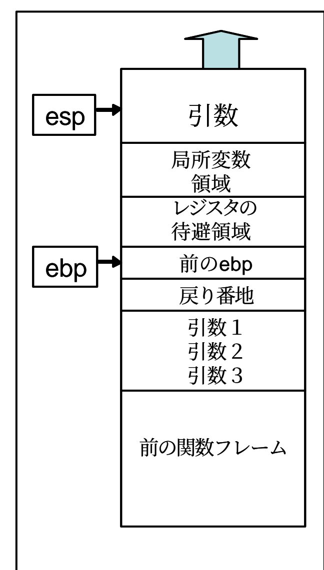
- レジスタの構成：プロセッサには、整数レジスタが%eax,%ebx,%ecx,%edx,%edi,%esi の６個、浮動小数点レジスタが８個あるが、tiny c では、整数レジスタのみをつかう。このほかに、プログラムカウンタ%pc，スタックポインタ%esp，フレームポインタ%ebp がある。すべてのレジスタに e がついているのは extend の意味で、32 ビットで使用するときに用いる。これは、x86 が 16 ビットマシンであったときの名残である。即値は$n と$をつけて記述する。
- オペランドの記述については、レジスタの時には%をつけて、%レジスタ名と記述する。メモリ参照は、offset(%レジスタ)と記述する。
- 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　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 と src2 を減算し、condition code　をセットする。
- 条件分岐命令：je label 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。

tiny C 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。

- 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。
- BEQ0 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)をいう形式もある。変数 r といっているのは、いわゆる局所変数ではなく、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。

中間コードへの変換

さて、構文木を変換することを考える。Lisp からは離れて、これからは文と式を区別考えることにす

る(Lisp では、式と文の区別がなく、文でも値が必要であったが、これからは通常のＣと同じように、式と文は区別する)。関数のコンパイルする関数 compileFuncDef はスタックマシンのものとほとんど同じである。だだし、本体は block のはずなので、compileBlock を呼び出している。compileBlock では、compileStatement を呼び出している。compileStatement では、if 文や while 文、return 文などの処理を呼び出している。制御文などで、分岐命令のコードを出すのはスタックマシンの場合とほとんど同じである。

式のコンパイルは、compileExpression で行う。この関数では、呼び出す側でターゲットとなる変数を作って、これを引数にして呼び出している。文の toplevel から呼び出され、値を必要としない場合には、ターゲットを-1 としている。変数を作るのは、tmp_counter を使って新しい変数の番号を生成する。式のコンパイルは以下のような手順である。

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 コードを出す。
２、式は変数であれば、その値をロードする命令を出す。
３、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、ターゲットに演算をするコードを出す。

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）を出力する。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。レジスタ割り当てには、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 tmpRegState という配列と変数がレジスタになくレジスタ退避領域にある変数を示す tmpRegSave という配列を用いている。tmpRegState は%eax,%ebx,%ecx,%edx のレジスタ、tmpRegSave は退避領域に対応している。reg 番目のレジスタ（つまり、%eax であれば、0 番目）の仮想レジスタ r が割り当てられているときには、tmpRegState[reg]には、r がはいっている。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 がはいっているときには、tmpRegSave[i]が r となる。

以下の関数を用意した。

- initTmpReg()　レジスタ割り当ての初期化,関数の最初で行う。
- getReg(r) 仮想レジスタ r に実際のレジスタを割り当て、そのレジスタ番号を返す。
- assignReg(r,reg)　仮想レジスタ r に実際のレジスタ reg を割り当てる。
- useReg(r) 仮想レジスタ r に割り当てられているレジスタ番号を返す。もしも、退避領域にあるのであれば、その変数をレジスタに復帰させ、そのレジスタ番号を返す。
- saveReg(reg) レジスタ reg を待避領域に保存し、reg を使えるようにする。
- freeReg(reg) レジスタ reg を開放する。
- saveAllRegs() レジスタに割り当てられている変数すべてを退避領域に格納する。

これを使ってたとえば、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。
gencode.c の 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。

ARG コードは、push 命令で生成される。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。RET に関しては、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。

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。

コンパイラと実行

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 ができる。tiny_cc 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.i 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。

最終演習課題：

以下の２つの課題のどちらかを選択し、レポートを提出すること。

課題１：

これまで説明した tiny C のコンパイラでは大域変数や配列宣言を処理していない。配列宣言と配列参照を処理できるように拡張して、8 queen のプログラムを書き、コンパイル、実行しなさい。

ヒント：

- まずは,適当なプログラムを作ってみて、-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
- 適当な中間コードを加えて、それに対する genFuncCode のルーチンを書けばよい。

課題２：

これまで、取り上げてきた Lisp(tiny C)のインタプリタ、スタックマシンのコンパイラ、x86 のコンパイラのいずれかについて、実行速度を向上させるための工夫、技法を調べ、どのように適用できるかについてレポートを提出しなさい。レポートの枚数の目安はＡ４で、５枚程度。E-mail で送付のこと。

（コンパイラの最適化の一部については、次回解説する）