記号処理・第 9 回

2017 年 6 月 6 日

今回の内容

9.1 MVM におけるメモリ割り当て 9–1
9.2 関数呼び出しの実現法 9–3
9.3 局所変数へのメモリ割り当て 9–5
9.4 フレーム 9–6
9.5 演習問題 9–8

9.1 MVM におけるメモリ割り当て

Minimum C コンパイラが生成する機械語プログラムは MVM のメモリ空間の一部に置かれ実行されます。ソースプログラム中の変数の値も、この同じメモリ空間の一部を使って記憶されます。コンパイラが生成する機械語プログラムは、この部分にアクセスして、その値を変更したり、読み出したりすることになりますので、コンパイラは、メモリ空間のどの位置に何の情報をどのように記憶しておくかに関する方針をあらかじめ決めておき、その方針にしたがった機械語プログラムを生成する必要があります。ここでは、次のような方針で MVM のメモリ空間を使用する機械語プログラムを生成することにします。

1. 関数定義のコンパイル結果は、ソースプログラム内での出現順にメモリ中に配置する。それぞれの関数定義がいくつの機械語命令になるかは、その関数定義をすべてコンパイルしてみないとわからない。まだコンパイルの済んでいない関数の呼び出しが出現した場合、その関数定義のコンパイル結果が置かれるアドレスはまだ決まっていないことになる。よって、仮の (呼び出し先アドレスが空欄の) 関数呼び出しのための機械語命令を生成しておき、後にその関数定義がコンパイルされた時に、この呼び出し先アドレスを修正する必要がある。

メモ

2. 大域変数の値を格納する領域は、関数のコンパイル結果の直後に配置する。そのアドレスはすべての関数定義のコンパイルが終らないと決まらない。このため、機械語プログラム全体の冒頭で、この領域 (大域変数領域) の先頭アドレスを `R11` に格納するようにしておき、機械語プログラム中では、`R11` からの相対位置で大域変数を参照するようにする。`R11` の値は機械語プログラムの実行が開始されると変ることはない。

メモ

3. メモリ空間の最後尾の部分はスタック領域として用いる。この領域は、プログラムの実行にしたがって伸縮する。スタック領域の先頭アドレスはスタックポインタ (R14) に保持する。各関数の引数や局所変数はこのスタック領域にある取り決めにしたがって配置し、**フレームポインタ**と呼ばれる特定のレジスタ (R13) に格納されたアドレスを基準としてアクセスする。

メモ

4. メモリ空間の先頭部分には、各レジスタを初期化し、関数 main に対する関数呼び出し (CALL 命令) を実行し、その後、機械語プログラム全体を終了される命令 (EXIT 命令) を実行するような命令群を置く。

メモ

ソースプログラム

```
int x;

foo()
{
   ⋮
}

bar()
{
   ⋮
}

int a, b;

main()
{
   ⋮
}
```

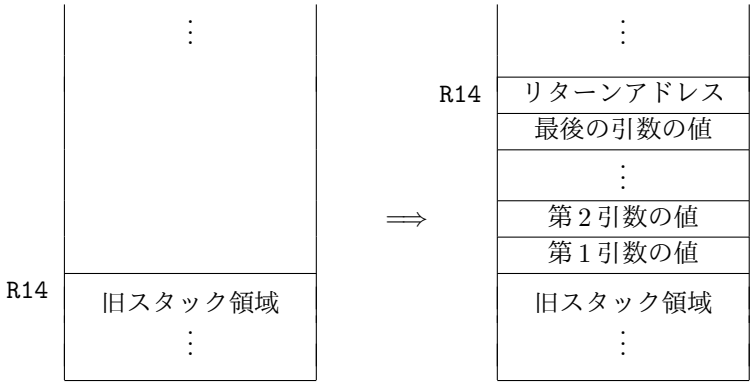
MVM のメモリ空間

9.2 関数呼び出しの実現法

ソースプログラム中の関数呼び出しに対して機械語命令を生成する際には、単に呼び出した関数定義のコンパイル結果 (機械語命令の列) の先頭に分岐する命令を生成するだけではなく、関数呼び出し時の実引数の値や、呼び出された関数定義の実行が終了した後に戻ってくるべきアドレス (リターンアドレス) などを情報を、何らかの形で呼び出された関数に知らせる必要があります。ソースプログラム中に定義された関数は、他の関数や自分自身の定義の中など、いろいろな箇所で呼び出される可能性がありますので、このための一定の手順を定めておかなければなりません。Minimum C コンパイラが生成する機械語プログラムは、次のような手順で関数呼び出しを行うことにします。

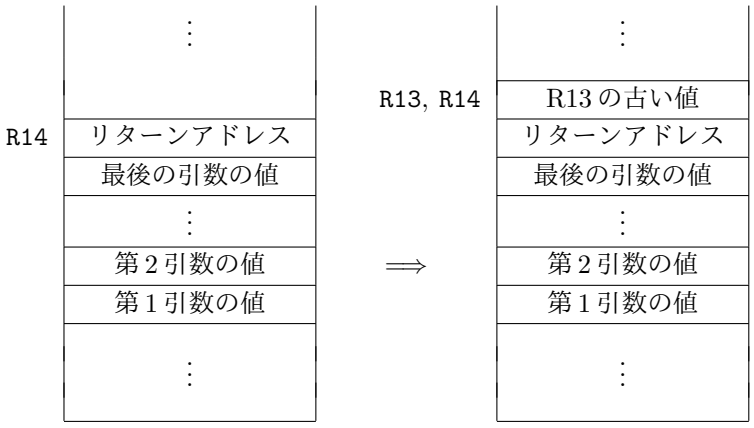
まず、関数を呼び出す側では

1. 引数の値を左から順にスタックに積む。
2. 呼び出す関数のアドレスに対して CALL 命令を実行する。これにより、現在のプログラムカウンタ (R15) の値が、リターンアドレスとしてスタックに積まれた後、呼び出された関数定義 (のコンパイル結果) に分岐する。



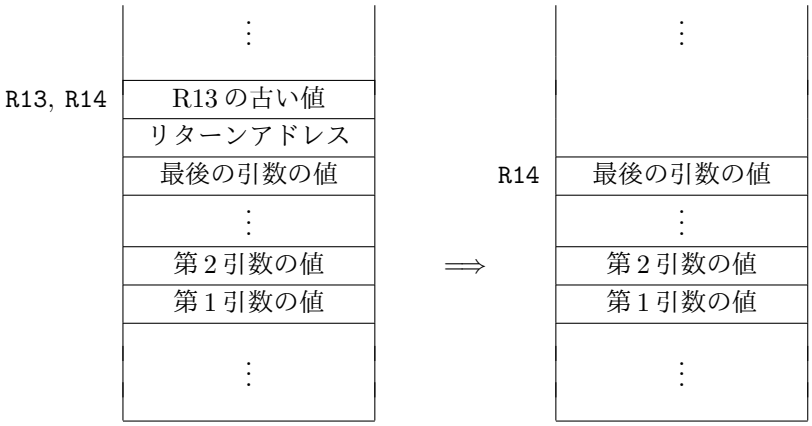
一方、呼び出された関数では次のような手順に従います。

1. フレームポインタ (R13) の値をスタックに積む。
2. スタックポインタ (R14) の値をフレームポインタ (R13) にコピーする。
3. 呼び出された関数定義の本体の実行を開始する。



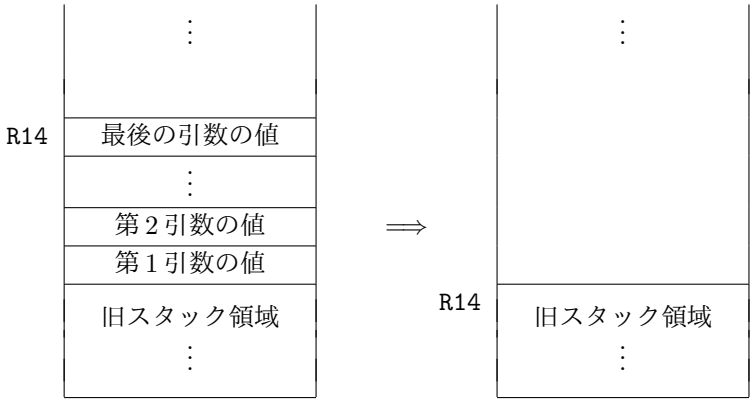
また、呼び出された関数定義の実行が終了して、呼び出し元に戻る場合には

1. 関数の戻り値をレジスタ R1 に格納する。
2. スタック上の値をフレームポインタ (R13) へ POP して、フレームポインタ (R13) とスタックポインタ (R14) の値を、この関数が呼び出された時の状態に戻す。
3. さらにスタック上の値 (リターンアドレス) をプログラムカウンタ (R15) に POP して、この関数の呼び出し元のアドレスから実行が継続されるようにする。



その後、呼び出し元の関数では

1. 関数を呼び出した時に引数を PUSH した分だけ、スタックポインタ (R14) の値を増やして、以前の状態に戻す。
2. 呼び出し元での処理を続行する。呼び出した関数の戻り値が必要な場合は、R1 の値を参照する。



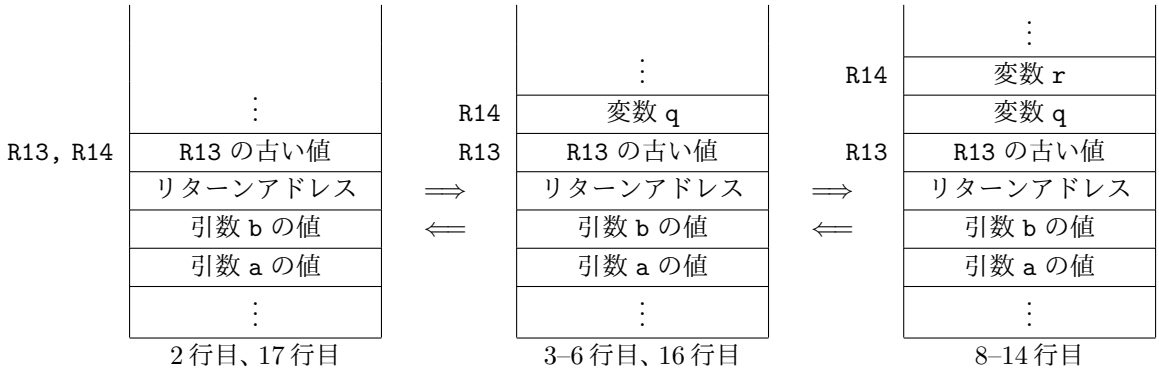
メモ

9.3 局所変数へのメモリ割り当て

Minimum C の局所変数はすべてスタック領域に配置します。Minimum C コンパイラは、局所変数宣言を含むブロックの始まりに対し、スタックポインタ (R14) の値を減少させるような機械語命令を生成し、局所変数の値を保持するための領域をスタックに確保します。また、このブロックの終りに対してスタックポインタの値を増加させるような機械語命令を生成し、局所変数のための領域をスタックから解放します。

例えば右上の Minimum C プログラムの場合、3 行目や 8 行目では、スタックポインタを 4 減じる機械語命令を生成し、15 行目や 17 行目では、逆にスタックポインタを 4 増やすような命令を生成します。

```
 1 div(a, b)
 2 {
 3     int q;
 4
 5     q = 0;
 6     if (b > 0)
 7     {
 8         int r;
 9
10          r = a;
11          while (r >= b) {
12              r = r - b;
13              q = q + 1;
14          }
15      }
16     print q;
17 }
```



2 行目、17 行目　　3–6 行目、16 行目　　8–14 行目

機械語プログラム中で局所変数や関数の引数を参照する場合は、フレームポインタ (R13) のアドレスを基準とした相対的な位置 (オフセット) で参照します。局所変数はフレームポインタよりも小さい側のアドレスに、引数はフレームポインタよりも大きい側のアドレスになります。

メモ

この関数 div の定義をコンパイルすると以下のような命令群が生成されます。

アドレス	機械語命令	
16	PUSH R13	フレームポインタの古い値をスタックに PUSH する
20	ADD R13,R14,R0	スタックポインタをフレームポインタにコピーする
24	SUBI R14,R14,4	スタックポインタを 4 減じて局所変数 q 用の領域確保
28	LDI R1,0	R1 に 0 を代入 (q = 0; の準備)
32	ST R1,R13,-4	その 0 を q に代入 (q = 0; の実行)
36	LD R1,R13,8	引数 b を R1 に代入 (b > 0 のチェックの準備)
40	LDI R2,0	定数 0 を R2 に代入 (b > 0 のチェックの準備)
44	SUB R0,R1,R2	b − 0 を計算して CPU のフラグに反映 (b > 0 のチェック)
48	JPLE R0,120	もし b ≦ 0 の場合は if 文の中身をスキップ
52	SUBI R14,R14,4	スタックポインタを 4 減じて局所変数 r 用の領域確保
56	LD R1,R13,12	R1 に引数 a の値を代入 (r = a; の準備)
60	ST R1,R13,-8	その値を r に代入 (r = a; の実行)
⋮	⋮	
112	JP R0,64	while 文の始まりへジャンプ
116	ADDI R14,R14,4	スタックポインタを 4 増やして局所変数 r 用の領域解放
120	LD R1,R13,-4	q の値を R1 に代入 (print q; の準備)
124	OUT R1	その値を OUT で出力 (print q; の実行)
128	ADDI R14,R14,4	スタックポインタを 4 増やして局所変数 q 用の領域解放
132	POP R13	フレームポインタの元の値を回復
136	POP R15	呼び出し元のアドレスをプログラムカウンタに回復

メモ

9.4 フレーム

前述のように、関数が別の関数 (あるいは自分自身) を呼び出す度に、実引数や、リターンアドレス、R13 の古い値、呼び出された関数内で宣言されている局所変数を記憶するための領域がスタック内に確保されます。この領域のことを一般に**フレーム (frame)**[1] と呼びます。関数がある関数を呼び出し、その関数がまた別の関数を呼び出し ... といったことが起こると、関数呼び出しのたびにスタック領域は伸びていき、そこに置かれているフレームの数が増えていきます。逆に、関数呼び出しから戻るたびに、そのとき使われていたフレームは破棄されて、スタック領域は縮んでいきます。

メモ

[1] Activation record と呼ぶこともあります

引数の数や、関数定義内で宣言されている局所変数の数によって、1 つのフレームの大きさは変ってくることに注意してください。また、もっとも新しいフレーム、つまり、現在実行されている関数呼び出しに対応するフレームの大きさは、実行がブロックに出入りすることで、ブロック内で新たに宣言されている変数の数に応じて伸び縮みすることもあります。

Minimum C の場合、ある関数の定義が実行されている状態では、そこでアクセスすることのできる変数は、大域変数であるか、さもなくばこの関数呼び出しによって作られたフレーム中に置かれている引数か局所変数となります。引数や局所変数にアクセスする際には、それらが記憶されているアドレスを知っていないといけませんが、R13 の古い値を記憶している位置[2] (アドレス)、つまり、R13 の新しい値を基準にして考えると、どの変数 (や引数) がどれだけずれた位置にあるのかは、ソースプログラムでの変数 (や仮引数) 宣言がどうなっているかで決めることができます[3]。例えば、最後の引数のアドレスは R13 + 8 ですし、もっとも外側のブロックで宣言された変数のアドレスは R13 − 4 となります。

メモ

[2]もちろん、リターンアドレスを記憶している位置や、最後の引数を記憶している位置などを基準にしてもよいのですが、こちらを基準にする方が簡単です

[3]MVM の R13 をフレームポインタと呼んでいるのはこのためです

9.5 演習問題

1. 今回解説した MVM におけるメモリ割り当ての方針にしたがって、次の Minimum C プログラムと同じことを実現する MVM の機械語プログラムを作りなさい。

```
max(x, y)
{
    if (x < y)
        x = y;
    return x;
}

main ()
{
    int a, b;

    input a;
    input b;
    print max(a, b);

}
```

2. 同様に、次の Minimum C プログラムと同じことを実現する MVM の機械語プログラムを作りなさい。

```
fact (n)
{
    if (n == 0)
        return 1;

    return n * fact(n-1);
}

main ()
{
    int a;

    input a;
    print fact(a);
}
```

記号処理・第 9 回・終わり