記号処理・第 11 回

2017 年 6 月 27 日

今回の内容

11.1 式 (Exp, Term, Factor) に対するコード生成

Minimum C のコンパイラは、例えば `x = y + 2;` のような代入文に対して、右辺の値を計算して左辺の変数に格納するような機械語命令の列を生成しなければなりません。このためには、

1. 式の値を計算して、レジスタの 1 つに置く
2. そのレジスタの値をメモリ中のその変数が置かれているアドレスに格納する

という 2 つの作業を行うような機械語命令を生成します。2 に関しては、単に MVM の `ST` 命令を生成すればよいわけですが、1 の式の値の計算はいくらでも複雑になりうるので配慮が必要となります。今回は、どのようにして式の値を計算させるかということについて考えます。

メモ

変数　もっとも単純な場合は、式全体が変数や定数となっている場合です。変数の場合は `LD` 命令 1 つでレジスタにその変数の値を持ってくることができます。例えば、局所変数 `x` が `R13` − 4 というアドレスに割り当てられているとすると、この変数の値をレジスタ `R1` に取ってくるためには、

```
LD R1, R13, -4
```

という命令を生成します。変数のアドレス (基準となるレジスタと変位) は、前回解説したスコープリストとオブジェクトリストを参照して調べることができます。

メモ

定数　定数の場合は `LDI` 命令で、定数をレジスタに置くことができます。例えば、定数 1234 をレジスタ `R3` に置くためには、

```
LDI R3, 1234
```

という命令を生成します。ただし、MVM の LDI 命令は -2^{19} から $2^{19}-1$ までの値しか扱えませんので、絶対値が非常に大きい定数の場合は、まず LDHI 命令を使ってレジスタの上位 16 bit を生成し (下位 16 bit はすべて 0 となる)、さらにそのレジスタに対して ADDI 命令[1]を使用して下位 16bit の部分を足し込みます。例えば、定数 123456789 ($= 1883 \times 2^{16} + 52501$) をレジスタ R2 に置くためには、つぎの 2 つの命令を生成します。

```
LDHI R2, 1883
ADDI R2, R2, 52501
```

メモ

四則演算　式が演算子 (+, -, *, /, %) を含んでいる場合は、演算子の両側の式の値をそれぞれ計算して 2 つのレジスタに置くような機械語命令の列をまず生成し、その 2 つのレジスタ間で ADD, SUB, MUL, DIV, MOD 命令を使った演算を行わせます。例えば 100+200 という式の値を計算してレジスタ R1 に置くためには、右のような 3 つの機械語命令を生成します。

100+200 の計算

```
LDI R1, 100
LDI R2, 200
ADD R1, R1, R2
```

メモ

同じ強さの結合優先度を持つ演算子での演算 (たとえば、+ と -) が繰り返される場合も同様となります。例えば 100+200-300-400+500 という式の値を計算してレジスタ R1 に置くためには、右のような命令列を生成します。このような形の式では、その式がどんなに長くても、2 つのレジスタがあれば十分であることに注意してください。

100+200−300−400+500 の計算

```
LDI R1, 100
LDI R2, 200
ADD R1, R1, R2
LDI R2, 300
SUB R1, R1, R2
LDI R2, 400
SUB R1, R1, R2
LDI R2, 500
ADD R1, R1, R2
```

メモ

[1] ORI 命令を使うこともできます

括弧を含む式や異なる結合優先度を持つ演算子による計算が含まれている場合は、部分式を計算した結果を一時的に記憶しておく必要が出てきます。例えば、100+200-300(400+500) という式の値を計算するためには 300*(400-500) を計算している間も 100+200 の計算結果を記憶しておかなければなりません。例えば、レジスタ R1 にこの式全体の計算結果を置くためには、上図右のように、3 個以上のレジスタを使って計算を行う命令列を生成することになります。この例では R1 から R4 までの 4 つのレジスタが使われています。

100+200−300*(400+500) の計算

```
LDI R1, 100
LDI R2, 200
ADD R1, R1, R2
LDI R2, 300
LDI R3, 400
LDI R4, 500
ADD R3, R3, R4
MUL R2, R2, R3
SUB R1, R1, R2
```

メモ

11.2 仮想レジスタ

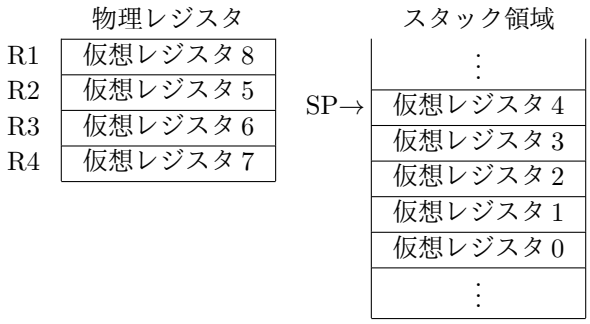
Mimimum C に現れる式はいくらでも複雑になり得ますので、その途中の計算結果を保持するために必要なレジスタの数もいくらでも多くなり得ます。一方 MVM のレジスタは高々 16 個しかありませんので、式によっては、すべての途中結果をレジスタに記憶しておくことはできない場合が出てきます。この問題を解決するため、**仮想レジスタ** という考え方を導入します。仮想レジスタは、コンパイラが無限に存在すると仮想的に考えるレジスタ群です。コンパイラは、MVM に実際に存在する物理的なレジスタ群に、この仮想レジスタ群の一部を割り付けて式の計算を行うような命令群を生成します。

式の値が計算される過程をよく調べると、どんな複雑な計算を行う場合でも同時に計算に関わっているレジスタは高々 2 つでしかないことがわかります。このため、次の計算のために新しい仮想レジスタが必要となった場合には、途中結果を保持していた古い仮想レジスタの内容を、一時的にメモリ空間のスタック領域に退避し、その物理レジスタを新しい仮想レジスタとして使用することにすれば、いくらでも多くの仮想レジスタを使用できるようになります。

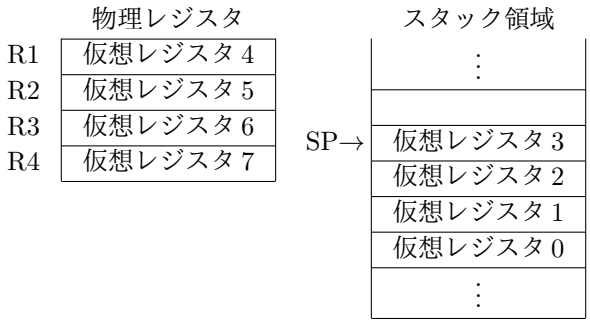
計算が進んで、退避した途中結果が必要となったときには、新しく割り付けた仮想レジスタの方は必要がなくなっている筈なので、その仮想レジスタを割り付けていた物理レジスタに、スタック上に退避していた値を戻して、それを元の仮想レジスタとして使用します。つまり、式の値の計算を行っている過程では、仮想レジスタの一部は物理レジスタに、残りの一部はスタック領域に割り付けられているわけです。

メモ

例えば、MVM の物理レジスタの内 R1 ～ R4 の 4 つのレジスタのみを式の値の計算に使用することにすると、仮想レジスタ 8 を使う状況では、仮想レジスタは右上図のように割り付けられます。この状況では、5 から 8 までの 4 つの仮想レジスタを直接使って計算を行うことができますが、それ以外の仮想レジスタは使用することはできません。



式の計算が進んで仮想レジスタ 4 が必要となる場面で、コンパイラは POP R1 という命令を生成して、右下図のように、この仮想レジスタ 4 を物理レジスタ R1 に割り付け直します。この時点では、仮想レジスタ 8 はすでにその役割を終えているはずですから、その値は廃棄しても構わないわけです。



このような考え方に基づいて、例えば、

```
100+(200+(300+(400+(500+(600+700)))))
```

という式の値を計算して、その結果を R1 に置く場合に生成される命令列は以下のようなものとなります。

```
LDI R1, 100              仮想レジスタ 0 に 100 を置く
LDI R2, 200              仮想レジスタ 1 に 200 を置く
LDI R3, 300              仮想レジスタ 2 に 300 を置く
LDI R4, 400              仮想レジスタ 3 に 400 を置く
PUSH R1                  仮想レジスタ 0 をスタックに退避する
LDI R1, 500              仮想レジスタ 4 に 500 を置く
PUSH R2                  仮想レジスタ 1 をスタックに退避する
LDI R2, 600              仮想レジスタ 5 に 600 を置く
PUSH R3                  仮想レジスタ 2 をスタックに退避する
LDI R3, 700              仮想レジスタ 6 に 700 を置く
ADD R2, R2, R3
ADD R1, R1, R2
ADD R4, R4, R1
POP R3                   仮想レジスタ 2 をスタックから復帰する
ADD R3, R3, R4
POP R2                   仮想レジスタ 1 をスタックから復帰する
ADD R2, R2, R3
POP R1                   仮想レジスタ 0 をスタックから復帰する
ADD R1, R1, R2
```

メモ

11.3 関数呼び出し

式中での関数呼び出し $Ident$（Exp_1, Exp_2, ..., Exp_n）を実行して関数の返り値を物理レジスタ Ri に置きたい場合は、まずスコープリストを検索して、関数 $Ident$ の置かれているアドレスを調べます。すでに関数 $Ident$ の定義が現れている場合には、そのアドレス L を得ることができますので、第 9 回で解説した関数呼び出しの実現法に沿って、右上のような命令の列を生成します。

仮想レジスタをすべてスタックに退避
$Exp_1 \rightarrow$ R1
PUSH R1
$Exp_2 \rightarrow$ R1
PUSH R1
⋮
$Exp_n \rightarrow$ R1
PUSH R1
CALL R0, L
ADD R14, R14, $4n$
ADD Ri, R1, R0

ただし、$Exp_k \rightarrow$ R1 は、式 Exp_k の値を計算してレジスタ R1 へ格納するために生成される命令の列を表しています。最後の ADD 命令は、関数の返り値を Ri へコピーするためのものですので、i が 1 と等しい場合にはこの命令は省略することができます。

関数を呼び出すと、呼び出された関数の定義の中でも (物理) レジスタを使った処理が行われるため、呼び出し前の (物理) レジスタの内容が壊されてしまう可能性があります。このため、関数呼び出しに先だって、現在使用中の仮想レジスタの内容をすべてスタックに退避しておかなければなりません。呼び出した関数からリターンしてきた後、スタック中に PUSH しておいた仮想レジスタは、その値が必要になった時に物理レジスタに POP されて利用されます。

メモ

例えば、局所変数 x が R13 − 4 というアドレスに割り当てられており、関数 foo の (機械語プログラムの先頭) アドレスが 300 番地であった場合、

```
x = x + 5 + (7 - foo(10, x));
```

という文は、次のような機械語プログラムへコンパイルします。

```
LD R1, R13, -4      x の値を R1 へ
LDI R1, 5           5 を R2 へ
ADD R1, R1, R2      x + 5 の計算結果を R1 へ
LDI R2, 7           7 を R2 へ
PUSH R1             R1 をスタックへ退避
PUSH R1             R2 をスタックへ退避
LDI R1, 10          foo の第 1 引数 10 を R1 へ
PUSH R1             foo の第 1 引数 10 をスタックへ PUSH
LD R1, R13, -4      foo の第 2 引数 x の値を R1 へ
PUSH R1             foo の第 2 引数 x の値をスタックへ PUSH
CALL R0, 300        関数 foo を CALL
ADD R14, R14, 8     2 つの引数の分だけスタックを戻す
ADD R3, R1, 0       foo の戻り値を R3 へコピー
POP R2              スタックへ退避しておいた古い R2 の値 (7) を復帰
SUB R2, R2, R3      7 から foo の戻り値を引いて R2 へ
POP R1              スタックへ退避しておいた古い R1 の値 (x + 5 の計算結果) を復帰
ADD R1, R1, R2      x + 5 の計算結果に 7 - foo(10, x) の計算結果を加えて R1 へ
ST R1, R13, -4      代入分の右辺の値を変数 x へ書き込み
```

一方、まだ関数 *Ident* の定義が現れていない場合は、その関数のアドレス L を知ることができないので、CALL 命令を置いたアドレスを覚えておき、その関数の定義が現れた時点で、CALL 命令の L の部分を設定します。このような CALL 命令は何度でも現れる可能性があるため、修正の必要な CALL 命令の置かれたアドレスをリストとして (オブジェクトリスト中の) その関数オブジェクトの中に保持して置きます。その関数の定義が現れた時点で、このリスト中のすべての CALL 命令を修正しなければなりません。

メモ

11.4 演習問題

1. 局所変数 x と y が、それぞれ R13 − 4 と R13 − 20 というアドレスに割り当てられているとする。次のような式の値を計算して、レジスタ R1 に置くための機械語の命令列を作りなさい。その時、物理レジスタとしては R1 〜 R4 の 4 つのレジスタだけを使うようにしなさい。

   (a) `x`

   (b) `x + y + 123456789`

   (c) `123456789 - x*y`

   (d) `(x+1)*(x-1)+y*y`

   (e) `x-(123456789-(x+(y-(123+(y-(x+(456-(x+y)))))))))`

2. 関数 foo のアドレスが R0 + 16 であるとき、次のような式の値を計算して、レジスタ R1 に置くための機械語の命令列をそれぞれ作りなさい。

   (a) `foo(100, 200)`

   (b) `foo(100+200, 300+400)`

   (c) `100+(200+foo(300, 400))`

   (d) `100+(200+foo(300, 400+(500+(600+700))))`

   (e) `foo(100+foo(200 300), foo(400, 500))`

記号処理・第 11 回・終わり

第 9 回の演習問題の解答例

1.

```
  0  ADD R13,R14,R0
  4  LDI R11,136
  8  CALL R0,68
 12  EXIT R1
 16  PUSH R13
 20  ADD R13,R14,R0
 24  LD R1,R13,12
 28  LD R2,R13,8
 32  SUB R0,R1,R2
 36  JPGE R0,44
 40  ST R2,R13,12
 44  LD R1,R13,12
 48  POP R13
 52  POP R15
 56  PUSH R13
 60  ADD R13,R14,R0
 64  SUBI R14,R14,8
 68  IN R1
 72  ST R1,R13,-4
 76  IN R2
 80  ST R2,R13,-8
 84  PUSH R1
 88  PUSH R2
 92  CALL R0,16
 96  ADDI R14,R14,8
100  OUT R1
104  ADDI R14,R14,8
108  POP R13
112  POP R15
```

2.

```
  0  ADD R13,R14,R0
  4  LDI R11,164
  8  CALL R0,112
 12  EXIT R1
 16  PUSH R13
 20  ADD R13,R14,R0
 24  LD R1,R13,8
 28  LDI R2,0
 32  SUB R0,R1,R2
 36  JPNE R0,52
 40  LDI R1,1
 44  POP R13
 48  POP R15
 52  LD R1,R13,8
 56  LDI R2,1
 60  SUB R1,R1,R2
 64  PUSH R1
 68  CALL R0,16
 72  ADDI R14,R14,4
 76  LD R2,R13,8
 80  MUL R1,R2,R1
 84  POP R13
 88  POP R15
 92  PUSH R13
 96  ADD R13,R14,R0
100  SUBI R14,R14,4
104  IN R1
108  ST R1,R13,-4
112  PUSH R1
116  CALL R0,16
120  ADDI R14,R14,4
124  OUT R1
128  ADDI R14,R14,4
132  POP R13
136  POP R15
```