21世紀のコンパイラ道しるべ

・・COINSをベースにして

連載 4

「TMDによるコード生成 — SPARC0を例題として」

森公一郎（エル・エス・アイ ジャパン（株））
kmori@lsi-j.co.jp

阿部正佳
abe@coins-project.org

中田育男（法政大学）
nakata@cis.k.hosei.ac.jp

鈴木　貢（電気通信大学）
gian@cs.uec.ac.jp

はじめに

前回は，COINS コンパイラ・インフラストラクチャのバックエンドの概要を説明した．バックエンドの構成は**図 -1** のようになっている．今回は，バックエンドの中の命令選択に使われるマシン記述ファイルの概要を，簡単なマシンのマシン記述ファイルを作りながら説明する．簡単なマシンとしては，SPARC マシンの中から C0 コンパイラに必要な機能だけを取り出したようなマシンを考え，それを SPARC0 マシンと呼ぶことにする．前々回に作った C0 フロントエンドと，今回作る SPARC0 マシン記述ファイルを使うバックエンドによって，SPARC0 マシン用の C0 コンパイラができあがる．

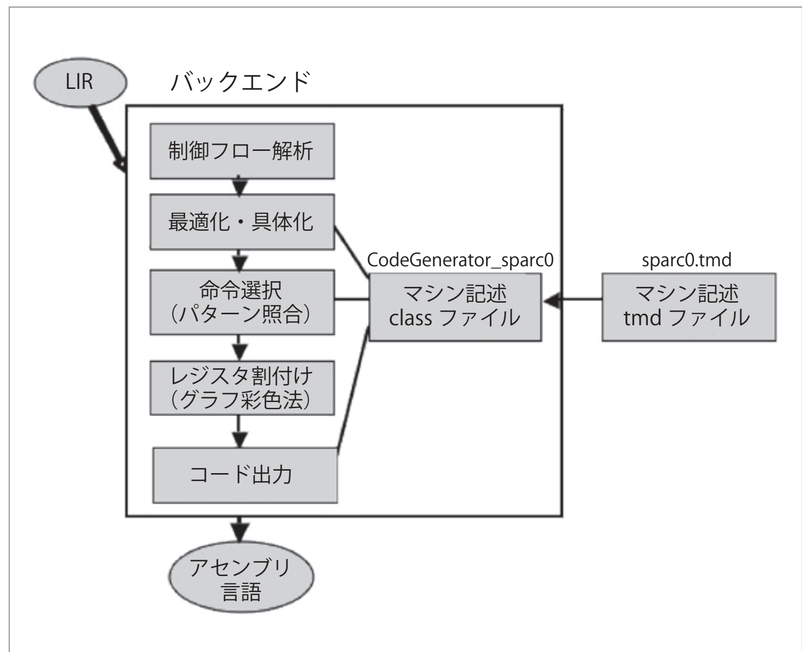

図 -1　バックエンドの構成

C0バックエンド用のSPARC0マシン記述

COINS には，すでに x86 や SPARC など，全部で 6 機種のコード生成系が作られているが，そのマシン記述は 2 千行から 5 千行の大きさであり，1 人で数カ月を要している．それと同程度のものをここで作ってみるのは困難であるため，C0 言語のプログラムの実行に必要となる最小限の命令語を備えたマシンを考え，そのマシン用のバックエンドを作ることにする．そのマシンは，SPARC マシンの命令語セットのサブセットを備えたものとし，SPARC0 マシンと呼ぶことにする．

マシン記述は TMD（Target Machine Description）と呼ばれ，SPARC マシン記述は sparc.tmd というファイルに書かれている．その中から，C0 言語のプログラムの実行に必要となる部分だけを取り出せば，それが SPARC0 マシン記述となる．SPARC0 マシン記述のファイルを sparc0.tmd という名前で作ることにする．

ところで，マシン記述をするときに 1 つ問題になるのは，そのマシンのサブルーチン呼び出し規約に合わせるところである．LIR では，それは PROLOGUE と

用語の定義がされているところに下線を付す．

EPILOGUEで抽象化してあるが，TMDではそのマシンのサブルーチン呼び出し規約に合わせてそれらを具体化するプログラムを書く必要がある．SPARCでは，サブルーチン呼び出しの引数は一般にはレジスタで渡すが，引数の個数が6を超えたらスタック渡しになる．その処理はちょっと面倒なので，ここでは引数の個数は6以下に制限することにする．

以上の方針でsparc0.tmdを書いてみたところ約580行になった．もとのsparc.tmdの行数は約2,000である．以下では，sparc0.tmdに書かれているものとその書き方の説明をする．

まず，sparc0.tmdによってLIR表現に対してマシン命令の選択が行われる様子を説明する．

●SPARC0マシン記述によるコード生成例

バックエンドの主要な部分は，図-1の「命令選択」，「レジスタ割付け」，「コード出力」である．命令選択は，マシン記述に書かれた生成規則と，LIR表現とのパターンマッチングによって行われる．マッチした生成規則に書かれているcode属性が生成すべき機械語コードを表している．ただしこの段階では，レジスタは仮想レジスタになっているものが多い．命令選択の後で，レジスタ割付けをすることによって，生成されるコードが決まる．最後にそれをアセンブラのコードとして出力するのが「コード出力」である．

マシン記述の中には，生成規則だけではなく，図-1の「具体化」のための記述や，アセンブラ・コードへの変換の仕方なども書く必要がある．また，生成規則だけでは書きにくいものをJava言語で書く必要もある．

〈マシン記述の中の生成規則〉

LIR表現の命令（以下それをL式と呼ぶ．また，たとえばSETで始まるL式をSET式，JUMPで始まるL式をJUMP式，などと呼ぶ）のパターンは，マシン記述ファイルでは，プログラミング言語の文法の記述にも使われた文脈自由文法のかたちで記述されている．文脈自由文法では，文法は

$A \rightarrow \alpha$

というかたちの規則からなる．ここで，αは非終端記号や終端記号からなる記号列である．この規則は，Aはαのかたちをしているという意味である．Aからαが生成されるともいう．プログラミング言語の場合は，その言語のソースプログラムは，その文法の（1つしかない）開始記号から生成されたかたちをしていなければならない．LR構文解析の場合は，ソースプログラムのある部分がαのかたちをしていたらそれをAに置き換えるということを繰り返して，ソースプログラム全体が開始記号に置き換えられたら，構文解析に成功したことになる．生成規則の右辺のαを左辺のAに置き換える操作は，αをAに還元するといわれる．

マシン記述の文法は，通常のプログラミング言語の文法とは違って，開始記号は複数個あり，SET式やJUMP式を生成する開始記号は別々にある．命令選択はLIR表現を開始記号に還元することによって行われるのであるが，1つのLIRプログラム全体が開始記号に還元されるのではなく，個々のSET式やJUMP式が別々の開始記号に還元されてコードが生成される．たとえば**図-5**の例では，SET式がreglに還元されたところで終わっている．LIR表現に対するパターンマッチングは，マシン記述の文法の右辺のパターンにマッチするものを左辺の記号に還元することの繰り返しで行われる．それはLIR表現の木の下の方から上の方に向かって行われるので，ボトムアップ（上向き）のパターンマッチングであるといわれる．同様に，LR構文解析は，解析木を下から作っていくので，ボトムアップの構文解析であるといわれる．LL構文解析は，開始記号の構文解析をする関数を呼び出すことから始めて順次右辺の解析を進めるので，トップダウン（下向き）の構文解析であるといわれる．マシン記述では，文法を書くときに，主として還元のことを考えて書くことになるので，生成規則のことを還元規則と呼ぶこともある．

図-2にマシン記述ファイルsparc0.tmdの一部を示す．マシン記述ファイルでは，生成規則はdefrule構文で記述される．命令選択では，このようなマシン記述ファイルを直接参照してパターンマッチングをするのではなく，そのファイルをパターンマッチングしやすいかたちに変換したものを使う．その変換の仕方は後で述べるが，sparc0.tmdから変換されたものは，JavaのクラスCodeGenerator_sparc0になる．CodeGenerator_sparc0は，入力L式の一部をdefrule中の右辺のパターンと照合し，マッチする規則をみつけて，それを左辺に還元する．その生成規則中のcode属性（`code`で始まるもの）で指定された命令列が生成される目的コードになるのであるが，実際にそれに置き換えるのは，前回述べたように，バックエンドの最後の段階である．

図-2は，sparc0.tmdから以下の説明に使われるルールだけを取り出したものである．

図-2の左端の数値は，本稿での説明のために付けたルール番号であり，tmdファイルに書く必要はない．`defrule`で始まる生成規則の2番目の項（第1引数ともいう）はその生成規則の左辺の非終端記号である．3番目の項（第2引数ともいう）はその生成規則の右辺であり，そこにはL式を記述する．たとえば，2番のルールは，文脈自由文法の書き方では

```
 1: (defregset regl *reg-I32*)       ;; 非終端記号 regl 用のデフォルトのレジスタセット
 2: (defrule xregl (REG I32))        ;; xregl はレジスタを表す非終端記号（左辺値）
 3: (defrule regl xregl)             ;; regl はレジスタを表す非終端記号（右辺値）
 4: (defrule addr regl)              ;; addr はアドレッシングモードを表す非終端記号
 5: (defrule rc regl)                ;; rc はレジスタまたは小さい整定数を表す非終端記号
 6: (defrule sta (STATIC I32))       ;; sta は static 変数のアドレスを表す非終端記号
 7: (defrule asmcon sta)             ;; asmcon は整定数を表す非終端記号
 8: (defrule regl asmcon             ;; asmcon はレジスタ regl に還元できる．その命令は
      (code (_set $1 $0))            ;; $1（asmcon の値）を $0（regl）にセットする命令である．
      (cost 2))                      ;; その命令のコストは 2 である．
 9: (defrule regl (ADD I32 regl rc) ;; 加算はレジスタに還元できる．その命令 add は
      (code (add $1 $2 $0))          ;; $1（regl）と $2（rc）の和を $0（regl）にセットする
      (cost 1)))                     ;; その命令のコストは 1 である．
10: (defrule regl (MEM I32 addr)     ;; アドレスの指すメモリの内容はレジスタに還元できる．
      (code (ld (mem $1) $0))        ;; その命令は ld 命令である．
      (cost 1)))                     ;; その命令のコストは 1 である．
```

図-2 生成規則例（sparc0.tmd の一部）

xregl → (REG I32)

に相当する．

入力 L 式がこの生成規則の右辺の L 式とマッチしたとき，そのルールに code 属性があればそこに記述された目的コードが（最終的には）出力される．たとえば，

```
(SET I32 (REG I32 "%i0")
         (ADD I32 (REG I32 "%i1")
                  (REG I32 "%i2"))
```

という入力 L 式があれば，この 2 行目の (REG I32 "%i1") が図-2 の 2，3 番のルールによって regl に還元され，(REG I32 "%i2") が 2，3，5 番のルールによって rc に還元されるので，この 2 行目全体は図-2 の 9 番のルール

(defrule regl (ADD I32 regl rc)

の右辺とマッチすることになる[☆1]．したがって，そこに書かれている code 属性に従って

add %i1, %i2, %i0

のようなコードが生成される．この場合，左辺の regl の値（code 属性では $0 と表現されている．$1，$2 はそれぞれ，右辺の regl，rc の値である）としては，この ADD 式が SET 式の第 2 引数であるときは，その第 1 引数の値 %i0 がとられる．ただし，マッチングをとって LIR 表現を変換するのは命令選択で行われるが，code 属性に従って実際にアセンブラコードが出力されるのは最後の段階である．図-2 の中にコメントとして書いてある左辺値／右辺値については後で説明する．

☆1 1 つ目の (REG I32 "%i1") も rc に還元することは可能であるが，そうしてもそれにマッチする SET 式などの構文がないので，ここではそれは使われないで，マッチするものだけが使われる．

```
int x;
int func(int y){
   return x+y;
}
```

図-3 ソースプログラム ex1.c

〈生成規則による命令選択〉

図-3（6 月号の図-2 の再掲）のプログラムの中の

return x+y;

に対応する LIR 表現から，コードが生成される様子を順を追って眺めてみる．この代入文に対応する L 式は，命令選択直前には次のようになっている．

```
(SET I32 (REG I32 "returnvalue.2%")
     (ADD I32
          (MEM I32 (STATIC I32 "x"))
          (REG I32 "y.1%")))
```

COINS のバックエンドでは，FRAME 変数は単純にフレーム領域に割り当てるのでなく，できるだけレジスタに割り当てるようにするために，最初にすべての FRAME 変数をこのように仮想レジスタ名に変換してしまう．レジスタ割付けの結果，レジスタが足りなくなった場合にだけフレーム領域に割り当てることにしている．なお，レジスタ名には必ず "%" を付けることにしている．

```
*71: regl -> (ADD I32 regl rc) [dest=(REG I32 "returnvalue.2%")] SU=1 ;;9
 *26: regl -> (MEM I32 addr) [dest=(REG I32 ".T1%")] SU=1 ;;10
   3: addr -> regl SU=0 ;;4
    *24: regl -> asmcon [dest=(REG I32 ".T2%")] SU=1 ;;8
      13: asmcon -> sta SU=0 ;;7
       11: sta -> (STATIC I32) SU=0 ;;6
 19: rc -> regl SU=0 ;;5
  *2: regl -> xregl [dest=(REG I32 "y.1%")] SU=0 ;;3
    1: xregl -> (REG I32) SU=0 ;;2
```

図-4　マッチングの結果

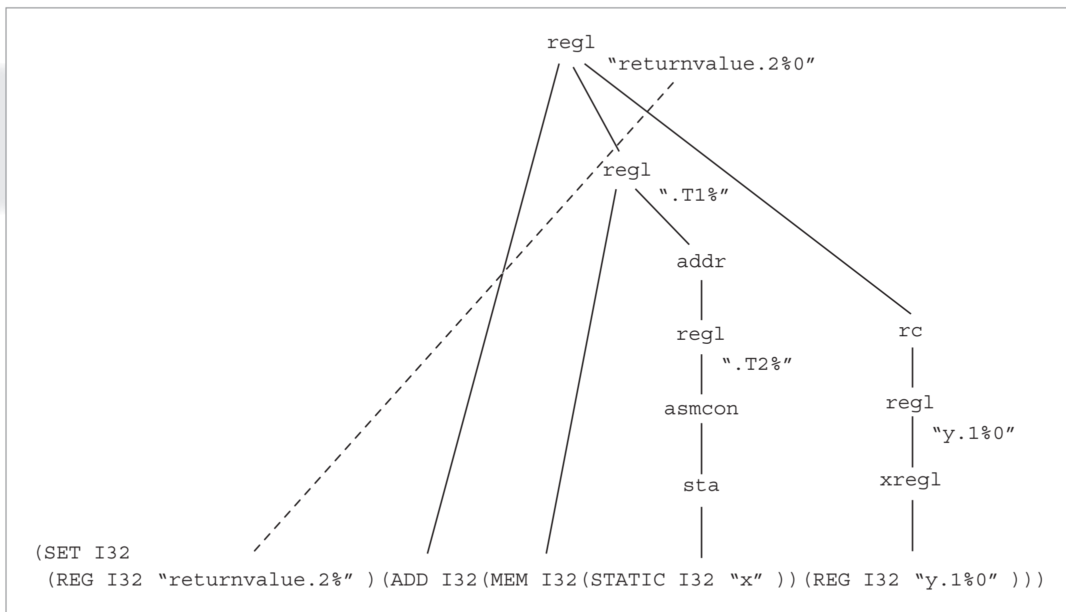

図-5
マッチングの結果の木

L式に対するマッチングは，L式の木の下から上へとボトムアップに行われる．その際は，マッチするすべての可能性を調べて，そのcodeのコストを加算していき，木のトップに達したところで，その中のコスト最小のものが選択される．選択された結果は，後述（**図-18**）のコマンドで見ることができる．今の例のL式に対する結果は**図-4**のようになる．

ここで，左端に付いている数値は，CodeGenerator_sparc0の中で対応するルールに付けられている番号であり，右端の数字は，説明のために付けたものであり，生成規則例のルール番号である．

図-4は少し読みにくいので，それを木の形の図に書いてみると，図-5のようになる．

生成規則の中のLIRのかたちで，一部が省略されているのは，そこに何があっても良いことを表す．たとえば，生成規則例の2番のルールは

```
(REG I32 ..)
```

というかたちの任意のL式にマッチし，6番のルールは

```
(STATIC I32 ..)
```

というかたちの任意のL式にマッチする．それらのルールで還元されたときのルールの左辺の値はマッチしたL式である．

`regl`のような，レジスタを表す非終端記号に還元される場合に，その非終端記号が`defregset`で定義されているか，またはその生成規則がcode属性を持つときは，還元結果のレジスタにレジスタ名がアサインされる（図-4ではそのような生成規則の左端には"*"が付いている）．図-2の3, 8, 9, 10番のルールがそれに相当する．たとえば，`[dest=(REG I32 ".T2%")]`では，`".T2%"`というレジスタ名を還元結果の`regl`にアサインしている．この`(REG I32 ".T2%")`が，マッチングにより変換されて得られるL式の中でその`regl`の値として使われる．レジスタに還元する場合に，還元される右辺がレジスタである場合は，その右辺のレジスタが還元結果のレジスタとなる（図-4では，`[dest=(REG I32 "y.1%")]`）．また，レジスタに還元する場合に，それがSET式の第2引数であるときには，第1引数のレジスタが還元結果のレジスタとなる（図-4では，`[dest=(REG`

```
(SET I32 (REG I32 ".T2%") (STATIC I32 "x"))
(SET I32 (REG I32 ".T1%") (MEM I32 (REG I32 ".T2%")))
(SET I32 (REG I32 "returnvalue.2%")
         (ADD I32 (REG I32 ".T1%") (REG I32 "y.1%")))
```

図 -6　命令選択の結果

```
(SET I32 (REG I32 "%i1") (STATIC I32 "x"))                (a)
(SET I32 (REG I32 "%i1") (MEM I32 (REG I32 "%i1")))
(SET I32 (REG I32 "%i0") (ADD I32 (REG I32 "%i1") (REG I32 "%i0")))
```

図 -7　レジスタ割付けの結果

```
%defbuild(_set x y) {
   return ImList.list(ImList.list("sethi", ImList.list("%hi", x), y),
                      ImList.list("or", y, ImList.list("%lo", x), y));
}
```

図 -8　"_set" マクロの定義

I32 "returnvalue.2%")])．このマッチングの結果，このL式は**図 -6** のように変換される．

なお，前回も述べたように，マッチしたツリーからL式を生成するときは，実際に必要とされるレジスタが少なくなるような順序で生成している（Sethi-Ullman の方法）．図 -4 で，"SU" がその必要レジスタの数を示しているが，その数の大きな方の枝から先に生成している（3 行目のマッチングは，コード生成には直接かかわらないので，"SU" の値は正確には表現せず，0 になっている）．

図 -6 のL式が命令選択の結果である．この 3 つの SET 式はそれぞれ機械語の命令に対応しているが，LIR の形（L式）で表現されている（code 属性はまだ使われない）．そして，それぞれの命令の入力や出力となるレジスタはマシン独立なかたちで表現されているから，命令選択の結果に対して命令スケジューリング（命令レベルの並列性を活かすための命令の並べ替え）をするプログラムなどを，マシン独立なかたちで書くことが可能である．

〈レジスタ割付け〉

レジスタ割付けもマシン独立なかたちで書くことが可能である．実際に COINS のバックエンドではレジスタ割付けがそのように作られているので，ここでもそれを使えばよく，SPARC0 のために新たに何かを書く必要はない．

レジスタ割付けの結果，命令選択の結果のL式は**図 -7** のように変換される．"%i0"，"%i1" は機械語のレジスタを表す．ここでもL式であることは変わらない．

〈コード生成〉

最後のコード生成では，もう一度図 -7 のL式に対するパターンマッチングが行われ，生成規則の記述のうちの code 属性に従って，コードが生成される．今の例では次のコードが生成される．これはアセンブラ・コードをリスト形式で表現したものである．

```
(sethi (%hi x) %i1)        (b)
(or %i1 (%lo x) %i1)       (c)
(ld (mem %i1) %i1)         (d)
(add %i1 %i0 %i0)
```

ここで，(b)，(c) の 2 つのマシン命令は図 -7 の最初の SET 式 (a) から生成される．命令選択後の SET 式は通常 1 つのマシン命令に対応しているのであるが，このように複数のマシン命令に対応する場合もある．その方が書きやすいのでそうしているが，命令スケジューリングでは 1 つの SET 式を 1 つのマシン命令のように見なしてスケジュールしているから，1 つの SET 式は 1 つのマシン命令に対応するようにルールを書いた方がよりきめ細かいスケジュールができる場合もある．

このような，コード生成時における変換は，マシン記述の中にマクロ命令のかたちで書けばよい．今の例では，(b)，(c) の命令は，生成規則例の 8 番のルールの code 属性 (code (_set $1 $0)) によって生成されるが，この "_set" はマクロの名前であり，sparc0.tmd の中で**図 -8** のように定義されている．クラス ImList はアセンブラ・コードをリスト形式で保持するときに使われる．

最後にアセンブラ・ファイルに出力されるときは，次

```
1: (defrule regl (MEM I32 regl)
    (code "copy memory[$1] to $0"))
2: (defrule void (SET I32 (MEM I32 regl) regl)
    (code "copy $2 to memory[$1]")
3: (defrule void (SET I32 regl regl)
    (code "copy $2 to $1"))
4: (defrule regl (REG I32))
```

図-9 間違ったルールの例

```
3:void <- (SET I32 regl regl)

1:regl <- (MEM I32 regl)   4:regl <- (REG I32 "v")

              4:regl <- (REG I32 "p")
```

図-10 誤ったパターンマッチの例

```
3': (defrule void (SET I32 xregl regl))
4-1: (defrule xregl (REG I32))
4-2: (defrule regl xregl)
```

図-11 訂正後のルール

のコードになる．この場合必要となる変換も，マシン記述の中にマクロ命令のかたちで書けばよい．たとえば，上記の（d）の中の（mem %i1）を［%i1］に変換するのもマクロ定義に従って行われる（後述）．

```
sethi %hi(x),%i1
or    %i1,%lo(x),%i1
ld    [%i1],%i1
add   %i1,%i0,%i0
```

〈生成規則中の左辺値の必要性〉

生成規則例の中のコメントに xregl は左辺値であると書いてあるが，ここで，その必要性を説明する．LIR の MEM 式は，以下のように 2 通りの使われ方をする（t は I32 などの型を表し，t-value は t 型の何かの値を表す）．

```
(SET t (REG I32 "v")
       (MEM t (REG I32 "p")))
(SET t (MEM t (REG I32 "p")) t-value)
```

上の MEM 式は，変数 p の指すメモリの内容を意味しているのに対し，下の MEM 式は左辺値，すなわち p の指すアドレスそのものを意味している．

仮に，**図-9** のようなルールを書いたとする．これらのルールのもとで，

```
(SET I32 (MEM I32 (REG I32 "p"))
         (REG I32 "v"))
```

からは，

```
copy v to memory[p]
```

というコードを生成したいのであるが，**図-10** のように誤ったパターンとマッチしてしまう可能性がある．このマッチングの結果，次のような誤ったコードが生成されてしまう．

```
copy memory[p] to tempreg1
copy v to tempreg1
```

誤りの原因は，1 番のルールにマッチした結果が，3 番のルールの中の左側の regl として使われているからである．このようなことをなくすためには，3 番のルールの左側のレジスタとして使えるものを制限する必要がある．そのために xregl という非終端記号を導入し，**図-11** のようにルールを変更すれば，このような誤ったマッチングの可能性はなくなる．図-11 の xregl はレジスタ変数の「左辺値」を意味する．これで，「右辺値」にしか還元されない 1 番のルールが SET の左辺に現れる可能性はなくなり，上のようなマッチングは起こらなくなる．

●マシン記述ファイルの書き方

tmd ファイルには以下のものを書く必要がある．

(1) マシンのデータ型の定義
(2) マシンのレジスタの定義
(3) LIR のパターンと機械命令との対応関係の記述
(4) PROLOGUE などの書き換え規則
(5) Java による記述

以下の各節でそれらの書き方と sparc0.tmd における記述例を説明する．

〈データ型の定義〉

アドレスを表現するデータ型と，テスト命令の演算結果（真か偽）のデータ型を定義する．sparc0.tmd では次のように書けばよい．

```
(def *type-address* I32)
(def *type-bool* I32)
```

これで，アドレスを表現するデータ型も，テスト命令の

```
 1: (def *real-reg-symtab*
 2:  (SYMTAB
 3:   (foreach @gl (g l)
 4:    (foreach @n (0 1 2 3 4 5 6 7)
 5:     ("%@gl@n" REG I32 4 0)))
 6:   (foreach @oi (o i)
 7:    (foreach @n (0 1 2 3 4 5)
 8:     ("%@oi@n" REG I32 4 0)))
 9:   ("%sp" REG I32 4 0)
10:   ("%fp" REG I32 4 0)))
```

図 -12 実レジスタの登録

```
(def *reg-I32* (
         (foreach @io (i o)
             (foreach @n (0 1 2 3 4 5)
                (REG I32 "%@io@n")))
         (foreach @n (0 1 2 3 4 5 6 7)
            (REG I32 "%l@n"))
         (foreach @n (2 3 4 5 6 7)
            (REG I32 "%g@n")) ))
```

図 -13 レジスタ割付けに使える汎用レジスタの定義

```
(def *reg-call-clobbers*
    ((foreach @n (0 1 2 3 4 5 6 7)
           (REG I32 "%g@n"))
     (foreach @n (0 1 2 3 4 5)
           (REG I32 "%o@n"))
    ))
```

図 -14 サブルーチン呼び出しで壊されるレジスタの定義

演算結果のデータ型も I32 であることを定義する．

〈レジスタの定義〉

まず，全実レジスタをシンボルテーブル SYMTAB に登録するための記述を**図 -12** のようにする．3 行目などにある `foreach` はマクロであり，2 番目の項が引数，3 番目の項が引数のとり得る値のリスト，4 番目の項がその適用対象を表す．たとえば 3 行目は，「`@gl` の値を `g` または `l` としてそのそれぞれについて，4 行目に適用する」という意味であり，その次の行も同様である．`@gl` の値が `g` で，`@n` の値が `0` であるときに 5 行目は

```
("%g0" REG I32 4 0)
```

となる．すなわち，3 ～ 5 行で `%g0` から `%g7` までと，`%l0` から `%l7` までのレジスタを登録することになる．6 ～ 8 行も同様で，`%o0` から `%o5` までと，`%i0` から `%i5` までのレジスタを登録することになる．スタックポインタ `%sp` とフレームポインタ `%fp` は最後に登録してある☆2．

次に，レジスタ割付けに使える汎用レジスタとサブルーチン呼び出しをしたときに壊される可能性のあるレジスタを**図 -13**，**図 -14** のように定義する．

次に，これから述べる生成規則の非終端記号にデフォルトで割り当てられるレジスタ集合を定義する．図 -2（生成規則例）の 1 行目がその例であり，非終端記号 `regl` に割り当てられるレジスタ集合を図 -13 で定義した `*reg-I32*` であるとしている．

最後に，レジスタ変数（演算の中間結果でなく，フレーム変数をレジスタ変数に変換したもの）にデフォルトで割り当てられるレジスタ集合を以下のように定義する．

```
(defregsetvar (I32 *reg-I32*) )
```

〈LIR のパタンと機械命令との対応関係の記述〉

L 式のパターンと機械命令との対応関係は生成規則とそれに付随する属性によって記述する．先にも述べたように

```
(defrule A B)
```

は「A → B」という生成規則に相当する．

JUMP 式やメモリへのストア命令になる SET 式などを生成する開始記号は次のように `defstart` で定義する．

```
(defstart void)          void が開始記号
```

先にも述べたように，1 つの LIR プログラム全体が開始記号に還元されるのではなく，個々の SET 式や JUMP 式が別々に還元されてコードが生成される．実行結果がレジスタに得られる L 式の生成規則の開始記号はレジスタであるが，そうでない場合の開始記号を定義する必要がある．結果が使われないという意味で `void` という名前にしている．

以下はレジスタを表す非終端記号である．2 行目は，左辺値が右辺値としても使えることを表している．

```
(defrule xregl (REG I32))
                          xregl は左辺値レジスタ
(defrule regl xregl)      regl は右辺値レジスタ
```

図 -15 はアドレッシングモードを表す非終端記号である．

`defrule` の 1 番目の引数は生成規則の左辺であり，2 番目の引数は右辺である．それ以後にあるものは生成規則に付随している属性である．また属性に書かれている `$1` と `$2` は，それぞれ生成規則の右辺の 1 番目と

☆2 `%sp`(=`%o6`)，`%fp`(=`%i6`)，リターンアドレスレジスタ（`%o7` および `%i7`）は特殊な用途だけに使われるので，ここでは除いてある．

```
(defrule addr regl)
(defrule addr con13)
(defrule addr (ADD I32 regl regl) (value (+ $1 $2)))
(defrule addr (ADD I32 regl con13) (value (+ $1 $2)))        (*)
(defrule addr (SUB I32 regl negcon13) (value (+ $1 (minus $2))))
```

図-15 アドレッシングモードを表す非終端記号

```
(foreach (@op @code) ((ADD add) (SUB sub) (BAND and) (BOR or) (BXOR xor))
 (defrule regl (@op I32 regl rc)
  (code (@code $1 $2 $0))
  (cost 1)))
```

図-16 演算命令の生成規則

```
 1: (defrule label (LABEL _))
 2:
 3: (defrule void (JUMP label)
 4:  (code (ba $1)
 5:        (delayslot))
 6:  (cost 1))
 7:
 8: (foreach (@op @b) ((EQ be) (NE bne)
 9:                    (LTS bl) (LES ble) (GTS bg) (GES bge)
10:                    (LTU blu) (LEU bleu) (GTU bgu) (GEU bgeu))
11:  (defrule void (JUMPC (TST@op I32 regl rc) label label)
12:    (code (cmp $1 $2)
13:          (@b $3)
14:          (delayslot))
15:    (cost 3)))
```

図-17 分岐命令の生成規則

2番目の非終端記号の持つ値を表す．図-15の(*)の行のvalue属性は，たとえばreglが%g2でcon13が16であるとき左辺のaddrの値は"%g2 + 16"になることを意味する．また，con13は13ビットの整数定数を表すものであるが，それは次の生成規則で定義されている．

```
(defrule con13 (INTCONST _)
 (cond "is13bitConst
  (((LirIconst)$0).signedValue())" ))
```

ここで，cond属性はその値（「"」で囲まれたJavaの式で表される値）が真のときだけこの生成規則によって還元されることを表す．$0は生成規則の右辺全体（整数定数を表すLIRノード）を表し，(LirIconst)$0はそれがLirIconstクラスのオブジェクトであることを示している．is13bitConstメソッドは後に述べる「Javaによる記述」の中で述べる．

レジスタでも13ビット定数でもよいものをrcと書くとすれば，それは次のように定義できる．

```
(defrule rc regl)
(defrule rc con13)
```

演算命令は，たとえば，**図-16**のように書けばよい．

図-16は，たとえば@opと@codeの組合せとしてADDとaddをとったときは

```
(defrule regl (ADD I32 regl rc)
  (code (add $1 $2 $0))
  (cost 1))
```

となることを意味する（それが生成規則例の9番のルールであった）．code属性が機械語の命令を表している．cost属性はその命令のコストである．命令選択の際はこのコストの和が1番小さくなるような命令列を選択する．したがって，コストとして命令のサイズ（バイト数のようなもの）を与えておけば，サイズの小さな命令列が選択され，コストとして命令の実行時間を与えておけば，実行時間の短いものが選択される．

分岐命令については，**図-17**のように書けばよい．

図-17の5，14行のdelayslotは遅延分岐のために実行される命令を入れる場所である．命令スケジューラによってここに適当な命令を挿入することができる（その方法は文献2）に書かれている）．挿入されなかった場合は，最後のアセンブラ出力のときにnop命令が挿入される．11行のルールの右辺は，reg1とrcを比較してその結果によって2つのlabelのどちらかにジャンプするLIR命令である．

〈PROLOGUEなどの書き換え規則〉

EPILOGUEやPROLOGUEのように，すこし抽象的に表現されているL式はターゲットマシンに合わせて具体化する必要がある．その変換規則を書くのがdefrewrite構文である．

たとえば，6月号の図-2のソースプログラムex1.cから最初に得られるPROLOGUE式は

```
(PROLOGUE (0 0)
 (MEM I32 (FRAME I32 "y.1")))
```

であった．これは引数がフレーム変数yであることを表しているが，SPARCでは最初の引数はレジスタ%i0に渡されることになっているので，これを次のように変換する必要がある．

```
(PROLOGUE (0 0) (REG I32 "%i0"))
(SET I32 (REG I32 "y.1%")
         (REG I32 "%i0"))
```

その変換はsparc0.tmdの中で次のように記述されている．

```
(defrewrite (PROLOGUE)
 (to (norescan
     (eval "rewritePrologue($0, post)")))
 (phase late))
```

ここで，

```
(defrewrite pattern (to new-pattern))
```

のかたちで，*pattern*にマッチしたL式を*new-pattern*に置き換えることを指示している．上の例では，PROLOGUEのノードがrewritePrologue($0, post)の結果で置き換えられる．

*pattern*中に非終端記号で書かれた部分は，*new-pattern*中で$1,$2...の形式で引用することができる．$0はマッチしたL式全体を意味する．今の例では$0はマッチしたPROLOGUE式を意味する．デフォルトでは，変換後の新しいL式はもう1度マッチングの対象となり何度でも変換される．しかし，上記の例のように，norescanが指定されていると2度と変換されなくなる．PROLOGUE命令は，変換した後もPROLOGUE命令のままなので，この指定がないと何度でも上のマッチングを繰り返して停止しなくなる．

phaseは，バックエンドの変換フェーズのどこでこの変換を行うかの指定である．lateはLateRewritingのフェーズを指定する．このフェーズは図-1の具体化の中にあり，命令選択のフェーズ（InstSel）より前である．

evalは文字列中のJavaコードを実行し，その戻り値と置き換えよという指令である．pre，postは，それぞれマッチしたものの前の命令列，後の命令列を意味する．上の例では，postはマッチしたPROLOGUE式の直後の命令列を意味する．

rewritePrologue($0, post)は，まず$0に当たる

```
(PROLOGUE (0 0)
 (MEM I32 (FRAME I32 "y.1")))
```

を

```
(PROLOGUE (0 0) (REG I32 "%i0"))
```

に置き換え，ついで引数postが指す命令列も書き換えて，後の命令列に

```
(SET I32 (REG I32 "y.1%0")
         (REG I32 "%i0"))
```

を加える．このrewritePrologue($0, post)で実行されるメソッドは，次の「Javaによる記述」のところに書けばよい．

同じように，

```
(EPILOGUE (0 0)
 (MEM I32 (FRAME I32 "returnvalue.2")))
```

は

```
(defrewrite (EPILOGUE)
 (to (norescan
     (eval "rewriteEpilogue($0, pre)")))
 (phase late))
```

によって，

```
(SET I32 (REG I32 "%i0")
         (REG I32 "returnvalue.2%"))
(EPILOGUE (0 0) (REG I32 "%i0"))
```

に変換される．

〈Javaによる記述〉

文法定義のかたちのみでターゲットマシンの記述の全てが完了するわけではない．Javaでコードを書かなければならない部分がある．tmdファイルには，文法定義部のほかに，Javaでコードを記述する部分がある．行頭に%%という印があらわれると，それ以降はJavaのコードとなる．tmdファイルの全体構造は以下のようになっている．

```
grammar definitions
%%
```

```
$ java -cp $COINS/classes c0front.C0Driver -S
       -coins:target=sparc0,debuginfo,trace=InstSel/RegisterAllocation/TMD
       ex1.c > trace
$ perl trace2html.pl -o trace -c trace
```

図 -18　コード生成のトレースをとるコマンド

imports
`%State methods`
Methods of class State
`%CodeGenerator methods`
Methods of class CodeGenerator

Java による記述は，*imports*，*Methods of class State* および *Methods of class CodeGenerator* の 3 つの部分に分けられる．1 つ目の *imports* には，以下の記述に必要な import 文を書く．2 つ目の *Methods of class State* には主に，cond 属性中で引用されるメソッドを置く．その例としては，先に述べた is13bitConst メソッドがある．それは以下のものである．

```
private boolean is13bitConst(long value)
{
  return -4096L <= value && value < 4096L;
}
```

3 つ目の *Methods of class CodeGenerator* には，class CodeGenerator_*target*（今の場合は CodeGenerator_sparc0）のメソッドを書く．その例としては，先に述べた `rewritePrologue`，`rewriteEpilogue` のほかにメソッド `rewriteFrame`，`rewriteCall` などがある．`rewriteFrame` は，与えられた FRAME 式をターゲットマシンでのフレーム変数のアドレスを表す式に変換する．また，`rewriteCall` は CALL 式をターゲットマシンでの呼び出し規約に合わせる変換をする．

Methods of class CodeGenerator にはまたアセンブラ命令を出力するときのマクロ `%defbuild` や `%defemit` を書くことができる．`%defbuild` はリスト形式のアセンブラ命令を作るときに使われる．`%defemit` は最終出力のアセンブラ命令を作るときに使われる．`%defbuild` の例には `%defbuild(_set x y)` があった．`%defemit` には次の例がある．

```
%defemit(mem x) { return "[" + x + "]"; }
```

これは，以前の例にあった `(ld (mem %i1) %i1)` から `"ld [%i1],%i1"` を出力するのに使われる．

〈COINSへのマシン記述の組み込み〉

以上のようにして記述した sparc0.tmd は文献 1）に置いてある．それを COINS に組み込むためには，まず，sparc0.tmd を COINS の src/coins/backend/gen/ に入れ，そのディレクトリにある Makefile に `CodeGenerator_sparc0.java` に関する行を書き込む必要がある（Makefile を見てそこにあるものをまねすればよい．書き込んだかたちのものも文献 1）に置いてある）．次に，COINS のルートディレクトリで

```
$ ./build.sh
```

を実行すると `CodeGenerator_sparc0.java` が生成され，それが javac でコンパイルされる．それで組み込み完了である．

〈COINSでの使い方〉

COINS に組み込んだ sparc0.tmd を使うには

```
-coins:target=sparc0
```

というオプション指定をすればよい．これによって sparc0 マシンのコードが生成される．そのコード生成の過程を見るためには，図 -18（*$COINS* は COINS を展開したディレクトリを表す環境変数）のコマンドを打ち，得られた `trace.html` をブラウザで見ればよい．

ブラウザで見て，"Before InstSel" をクリックすると，命令選択直前の L 式を見ることができる．その後に図 -4 のようなマッチングの結果が表示されている．"After InstSel" をクリックすると，命令選択後の L 式を見ることができる．

おわりに

今回は，簡単なマシンを対象として，マシン記述ファイルの作り方を説明した．COINS を使えば，マシン記述ファイルを作るだけでそのマシン用のバックエンドが得られる．

次回以降は，いろいろな最適化について説明する予定である．

参考文献

1）http://www.coins-project.org/IPSJ-mitisirube/
2）http://www.coins-project.org/COINSdoc/

（平成 18 年 6 月 2 日受付）