21世紀のコンパイラ道しるべ
‥COINSをベースにして

連載 3

「LIRの説明とバックエンドの概要説明」

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

阿部正佳（東京大学大学院情報理工学系研究科）
abe@is.s.u-tokyo.ac.jp

中田育男（法政大学）
nakata@cis.k.hosei.ac.jp

はじめに

前回と，前々回で，コンパイラやCOINSコンパイラ・インフラストラクチャの概要と，C0言語という簡単なプログラミング言語の説明をし，C0言語のコンパイラのフロントエンドの作り方を説明した．それは，**図-1**の上半分の部分である．

C0コンパイラのフロントエンドができたので，今度はバックエンドを作ってみよう．COINSには，すでにいくつかのバックエンドができているので，それを使えばC0のコンパイラは完成するのである（そのことは前回説明した）が，ここでは，コンパイラの作り方を説明するために，それを使わず，新しいバックエンドを作ってみることにする．今回は，図-1の太字の部分である低水準中間表現LIRとバックエンドの概要を説明し，次回に簡単なマシンの機械語のコードを生成するバックエンドの作りかたの説明をする．

バックエンドの構成

COINSでは，ソースプログラムはフロントエンドによって高水準中間表現HIRに変換され，次にそれが低水準中間表現LIRに変換され，最後にLIRから対象とするマシン（対象マシンとか目的マシンと呼ばれる）の機械語のコード（目的コードとも呼ばれる）が生成される．この機械語のコード生成にかかわる部分がバックエンドと呼ばれる部分である．

機械語のコードを生成する部分は，コンパイラの中でも複雑な処理を必要とし，自動化が難しい部分である．

用語の定義がされているところに下線を付す．

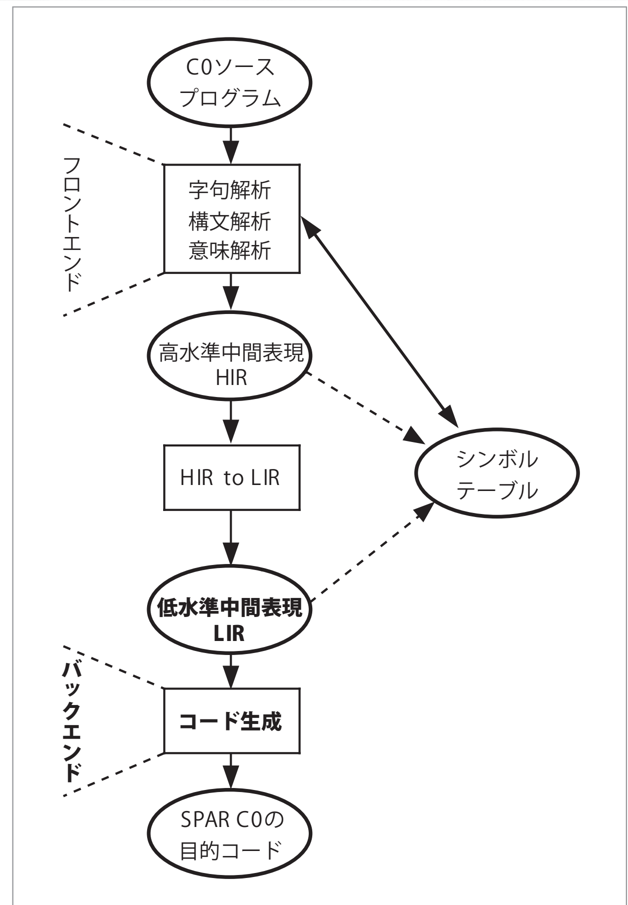


図-1　C0コンパイラの構成

難しい理由の1つは，ソースプログラムの構成要素には，整数型や複素数型，配列や構造体，演算の種類など多様なものがあり，一方，マシンの機械語にも多様な命令があることから，ソースプログラムに対する目的コードとして考え得る組合せの数が膨大なものになり，その中で，できるだけ効率のよい目的コードを選ぶのが容易ではな

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

図 -2 ソースプログラム ex1.c

いからである．

効率のよい目的コードを生成するためには，効率のよい機械語命令を選ぶことと，それらの命令で使うレジスタを適切に割り当てることが必要である．前者は命令選択と呼ばれ，後者はレジスタ割付けと呼ばれる．命令選択については多くの研究がされ，マシン記述からある意味で最適な命令を選択できる方式が実現されている．ある意味でというのは，マシン記述の中に各命令のコストといわれる数値を入れておき，ソースプログラムの 1 つの代入文などに対して，そのコストの和が最小になるような命令列を選択するという意味である．COINS にもその機能が実装されており，マシン記述にしたがって最適な命令を選択することができる．また，レジスタはメモリより高速に動作するので，その割付けは，コンパイラにおける最適化の中でも最も重要なものとして研究が行われてきている．COINS では，その最近の成果に加えて，いくつかの工夫をしたものが実装されている．

バックエンドは，上記の命令選択，レジスタ割付けなどをする部分のほかに，それ以外の最適化をする部分や最終的にアセンブラコードを出力する部分などからなる．COINS ではバックエンドで扱われる情報はすべて LIR のかたちで表現されており，バックエンドの各処理は LIR のかたちで表現されたもの（それを以下では LIR 表現と書くことにする）に対する変換処理と見なすことができる．

これらの各処理は，ほとんどマシンに依存しないかたちで書かれている．マシンによって変えなければならないのはマシン記述だけである．

●低水準中間表現 LIR

COINS を使ってバックエンドを作るためには，まず COINS の LIR の仕様を理解する必要がある [1)]．よく知られた低水準中間表現としては gcc の RTL がある [2)]．RTL と LIR は表現レベルはほぼ同等で，命令構成も似ているが，RTL に対する COINS の LIR の特徴は，次の 3 点である．

(1) マシン独立性が高く，SSA 最適化[☆1] などのマシン独立な最適化が可能である．

(2) バックエンドの全フェーズを通じて使用できる中間表現である．

(3) 1 つの独立したプログラミング言語である．

gcc での RTL 表現は始めから対象とするマシンの機械語に対応しており，COINS でいえば命令選択をした後の LIR 表現に相当する．COINS では，最初に作成される LIR 表現は低水準であるとはいっても，ソースプログラムの 1 つの代入文などが機械語命令とは直接対応しない 1 つの木として表現されている．その LIR 表現に対して SSA 最適化などのマシンには依存しない最適化を施すことができる．LIR 表現に対して命令選択をしてターゲットマシンの機械語に変換した後も，その結果もまた LIR のかたちで表現されているから，命令レベルの並列性を活かす命令スケジューラなどもマシン独立なかたちで書くことができる．バックエンドの最終出力はアセンブラ言語のプログラムであり，その時に初めてマシン特有（アセンブラ言語特有）の表現に変換される．

一般に，関数の引数や返却値に使われるレジスタはそのマシンの（あるいはソフトウェアシステムの）関数呼び出し規約によりあらかじめ定められている．gcc ではこのような実レジスタ割り当てを最初の RTL 生成時に行ってしまう．それに対して，LIR では関数引数と返却値を含めて関数の入口と出口が抽象化されているため，関数呼び出し規約に従った実レジスタの割り当てを最初に行う必要がなく，SSA 最適化などのコード最適化においては実レジスタに対する考慮は一切不要である．

また，LIR は，プログラミング言語としての仕様もきちんと定義されているので，別途 LIR プログラムをテキストとして生成する言語処理系を作成し，その生成テキストをバックエンドに入力してターゲットマシンのコードを得るという使い方もできる．

〈LIR のプログラム例〉

前回 HIR の説明の時に使ったプログラムを**図 -2** に再掲する．このソースプログラムを COINS の C フロントエンド，あるいは前回開発した C0 フロントエンドで変換すると**図 -3** の LIR プログラムが得られる（LIR は 1 つのプログラム言語であるので，ソースプログラムから変換されて得られる LIR 表現は LIR プログラムと呼ぶことができる）．図 -3 の表現はコンパイラオプションで

```
-coins:trace=LIR
```

を指定すると得られる．

図 -3 の左端の数字は説明のために付けたものである．LIR プログラムはモジュール（MODULE）と呼ばれる．モジュールは，（グローバル）シンボルテーブル（2 ～ 4 行），いくつかの関数の定義（5 ～ 16 行），データ（17

[☆1] SSA 最適については，簡単な説明が第 1 回（2006 年 4 月号）にある．詳細は次々回に説明する予定である．

```
1:(MODULE "ex1.c"
2:  (SYMTAB                          //（グローバル）シンボルテーブル
3:    ("func" STATIC UNKNOWN 4 ".text" XDEF) // 関数 func
4:    ("x" STATIC I32 4 ".bss" XDEF))       // static 変数 int x
5:  (FUNCTION "func"                   // 関数 func の定義本体
6:    (SYMTAB                          //（ローカル）シンボルテーブル
7:      ("y.1" FRAME I32 4 0)              // 引数 int y
8:      ("returnvalue.2" FRAME I32 4 0))    // 返り値用変数
9:    (PROLOGUE (0 0) (MEM I32 (FRAME I32 "y.1")))   // 関数の入口，引数 int y
10:   (DEFLABEL "_lab1")                 // ラベル定義 _lab1
11:    (SET I32 (MEM I32 (FRAME I32 "returnvalue.2"))  // returnvalue.2 =
12:            (ADD I32 (MEM I32 (STATIC I32 "x"))     //                 x +
13:                   (MEM I32 (FRAME I32 "y.1"))))    //                     y.1
14:    (JUMP (LABEL I32 "_epilogue"))     // goto _epilogue
15:   (DEFLABEL "_epilogue")             // ラベル定義 _epilogue
16:    (EPILOGUE (0 0) (MEM I32 (FRAME I32 "returnvalue.2")))) // 関数の出口，返り値
17:  (DATA "x" (SPACE 4)))             // int x の領域
```

図 -3　ソースプログラム ex1.c から得られる LIR プログラム

行）などからなる．関数は，その名前（5 行），（ローカル）シンボルテーブル（6 ～ 8 行），命令列（9 ～ 16 行）からなる．関数の命令列は，PROLOGUE 式で始まり，EPILOGUE 式で終わる．

シンボルテーブルは，いくつかのエントリからなる．各エントリの最初の要素（"func"，"x" など）は名前である．2 番目の要素はクラスと呼ばれ，クラスが STATIC である名前は静的に割り付けられるオブジェクトを表す．クラスが FRAME である名前（7 行の "y.1" など）はスタックフレームに割り付けられるオブジェクトを表す（関数のローカル変数や引数は通常はスタックフレームに割り付けられる）．3 番目の要素は型を表す．I32 は 32 ビット整数型である．4 番目以降はアラインメント，セグメント，リンケージの情報である．ローカル変数名には，他の関数などに同じ名前があっても区別できるようにサフィックスが付けられている（それで，引数の "y" が "y.1" になっている）．8 行目の "returnvalue" は関数の戻り値を入れる変数として導入されたものである．

9 行目の PROLOGUE 式の 2 番目の (0 0) はここでは無視してよい．16 行目の EPILOGUE 式の 2 番目の要素も同様である．PROLOGUE 式の残りの要素は関数の仮引数を示している．EPILOGUE 式の残りの要素は関数の返す値を表す式のリストである（複数個の値を返す表現もできる）．MEM はメモリの内容を表す．11 行目の SET は代入を表す．11 ～ 13 行は

```
returnvalue.2 = x + y.1;
```

という代入文に相当する．

LIR の命令の詳細については，紙面の都合で説明を省略する（文献 1）の第 4 章に詳細な説明がある）．

●バックエンドの概要

〈バックエンドの処理の流れ〉

COINS のバックエンドの処理は，以下のような流れで行われる．

1. 外部形式の LIR（たとえば図 -2 の文字列表現）から内部 LIR 表現への変換
2. 内部 LIR 表現から CFG（Control Flow Graph：制御フローグラフ）への変換
3. 各種解析を行い，解析データを抽出
4. 簡単な最適化と具体化
5. 機種依存な命令列に変換（命令選択）
6. レジスタ割付け
7. アセンブリ言語に変換

その 1. を省略し，2. と 3. を 1 つにまとめて表現したものが**図 -4** である．

2. の制御フローグラフは，if 文，while 文，goto 文などによる，分岐や合流をグラフのかたちで表現したものであり，プログラムのどの部分がどんな順序で実行される可能性があるかを示すグラフである．コンパイラが行

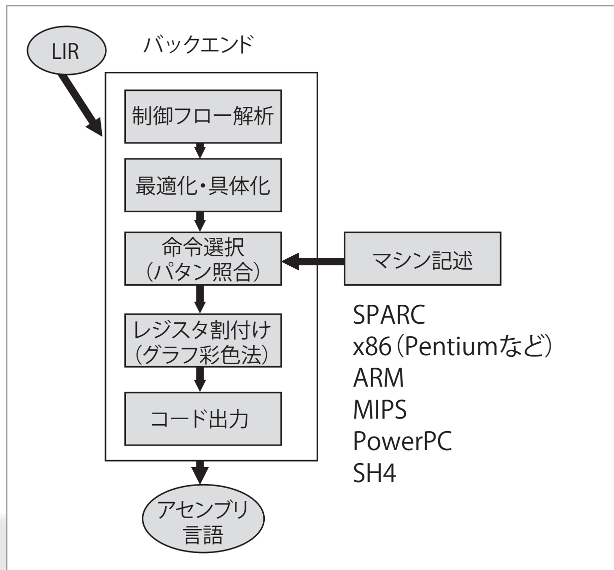


図 -4　バックエンドの構成

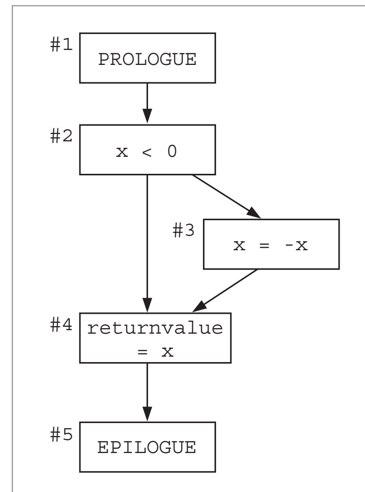


図 -6
ソースプログラム ex2.c
の制御フローグラフ

```
int abs(int x){
    if (x < 0)
        x = -x;
    return x;
}
```

図 -5
ソースプログラム ex2.c

```
1: (PROLOGUE (0 0) (REG I32 "y.1%"))
2:
3: (SET I32 (REG I32 "returnvalue.2%")
4:        (ADD I32 (MEM I32 (STATIC I32 "x"))
5:                 (REG I32 "y.1%")))
6:
7: (EPILOGUE (0 0) (REG I32 "returnvalue.2%"))
```

図 -7　簡単な最適化後の LIR 表現

う最適化のほとんどは，この制御グラフ上で，変数の値がどこで定義されどこで使われているかなどの解析をして行われる．そのような解析をするのが，3. である．

制御フローグラフのノードは，基本ブロックと呼ばれ，分岐のない直線的な命令の列の最後に分岐命令があるかたちである．たとえば，**図 -5** のプログラムに対して，trace=LIR のオプション指定をすると，制御フローグラフに関する詳細な情報が得られる．それをグラフのかたちにしてみたのが**図 -6** である．図 -6 では，実際に得られる情報を簡略化して書いてある．COINS には，制御フローグラフとソースプログラム，HIR，LIR などとの対応を表示する CoVis という名前のツールがある [1]．

4. の最適化としては，無駄な分岐命令の削除や基本ブロック内の定数畳み込みなどを行う．具体化とは，抽象化されているかたちの LIR 表現を対象機種に合わせてより具体化することである．たとえば，CALL，PROLOGUE，EPILOGUE，JUMPN 等の命令をより単純な命令に分解する．しかし，まだ機械語の生成はしない．

COINS の LIR は全バックエンドフェーズを通じて使用できる中間言語であるので，上記の 2. ～ 6. の処理をするプログラムはすべて LIR 表現に対するものとして，マシン独立なかたちで書くことができる（扱うデータはマシンごとに異なる）．

たとえば，5. の命令選択をして，特定のマシンの機械語が選択されたとしても，それが LIR のかたちで表現されているし，さらに，6. のレジスタ割付けをした後でもそれが保たれている．したがって，マシン語の命令の順序を最適なものに変更する命令スケジューラもマシン独立なかたちで書くことができる．

〈コード生成例〉

図 -3 の LIR 表現から最後にアセンブラコードが出力されるまでの様子を上記の処理の流れに沿って，眺めてみる．

⇩ 簡単な最適化と具体化

最初に，LIR 内部表現への変換，制御フローグラフへの変換，各種解析，などを行った後で，簡単な最適化を行った結果，図 -3 の 9 行目と 11 ～ 13 行目と 16 行目はそれぞれ**図 -7** の 1 行目, 3 ～ 5 行目, 7 行目のようになる．

ここで，REG はレジスタを表す．ただし，この段階ではマシンに実際にあるレジスタではなく，後でマシンのレジスタに割り当てて欲しい仮想レジスタを表す．最初の図 -3 のかたちでは引数や局所変数はフレームと呼ばれるスタック上の領域に割り付けられるものとしており，それが通常のコンパイラでとられる方法であるが，

```
(SET I32 (REG I32 ".T2%") (STATIC I32 "x"))
(SET I32 (REG I32 ".T1%") (MEM I32 (REG I32 ".T2%")))
(SET I32 (REG I32 "returnvalue.2%0")
         (ADD I32 (REG I32 ".T1%") (REG I32 "y.1%0")))
```

図 -8 代入文の命令選択後の LIR 表現

```
(SET I32 (MEM I32 (FRAME I32 "a.1")) // MEM[a.1] = MEM[b.2 + MEM[j.3] * 4]
         (MEM I32 (ADD I32 (FRAME I32 "b.2")
                           (MUL I32 (MEM I32 (FRAME I32 "j.3"))
                                    (INTCONST I32 4)))))
```

図 -9 `a = b[j];` の最初の LIR 表現

```
(SET I32 (REG I32 "a.1%") // a.1% = MEM[%fp - 40 + j.3% << 2]
         (MEM I32 (ADD I32 (ADD I32 (REG I32 "%fp")
                                    (INTCONST I32 -40))
                           (LSHS I32 (REG I32 "j.3%") (INTCONST I32 2)))))
```

図 -10 `a = b[j];` の簡単な最適化後の LIR 表現

COINS では，このようにそれらをまず仮想レジスタに割り付けておいて，後のレジスタ割付けの際に実際のレジスタに割り付けられなかったものだけをフレームに割り当てることにしている．フレームは関数の入口で確保され，出口で解放される．なお，COINS では，レジスタ名には必ず "%" をつける．

ところで，関数呼び出しの際に引数や返り値をどこに置くかといった約束はマシンやソフトウェアシステムによって異なるので，関数の入口と出口に置くべき機械語の命令列も異なる．LIR ではそれを抽象化して PROLOGUE と EPILOGUE というかたちで表現しているが，命令選択の前にはそれを具体的な表現に変更する必要がある．それは前にも述べたように図 -4 の 2 番目のボックスに書かれている具体化の中で行われる．その結果，SPARC CPU 搭載マシンでは，PROLOGUE と EPILOGUE は次のようになる．

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

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

これは，SPARC マシンでは第 1 引数の値は %i0 レジスタ，関数の返り値も %i0 レジスタに置くという約束になっているからである．そのために PROLOGUE では，引数を受け取った %i0 の値を引数変数（引数変数 y は y.1% という名前のレジスタになっている）に代入し，EPILOGUE では返り値の変数の値を %i0 に代入している．このような変換の仕方は，次回に述べるマシン記述（Target Machine Description: TMD）ファイルに書いておけばよい．

⇩ 命令選択

命令選択の結果，PROLOGUE の直後と EPILOGUE の直前の SET 式は変わらない（すでに SPARC の命令に対応している）が，図 -7 の 3 ～ 5 行目が**図 -8** のように 3 つの SET 式になる．

図 -8 のそれぞれの SET 式は SPARC の 1 つの命令を表している．最初の SET 式は，x のアドレスを .T2% レジスタに載せる命令である（.T2% レジスタは仮想レジスタである）．2 番目の SET 式は，.T2% レジスタの値の番地から .T1% レジスタにロードする命令である（これで，x の値が .T1% にロードされる）．3 番目の SET 式は，.T1% レジスタと y.1%0 レジスタの和の値を returnvalue.2%0 レジスタに置く命令である．このような命令選択は，SPARC のマシン記述ファイルに従って行われる（その命令選択の様子は次回に説明する）．

命令選択では，コスト最小の命令列が選ばれる．それを簡単な例で説明する．たとえば，

```
int a, b[10], j;
a = b[j];
```

の代入文に対する LIR 表現は，最初は**図 -9** に示すものであるが，命令選択の直前では，配列や構造体以外のフレーム変数の仮想レジスタへの変換と簡単な最適化の結果，掛け算命令がシフト命令に変換されて**図 -10** に示す

```
(SET I32 (REG I32 ".T1%")            // .T1% = %fp -40          cost 1
         (ADD I32 (REG I32 "%fp") (INTCONST I32 -40)))
(SET I32 (REG I32 ".T2%")            // .T2% = j.3% << 2        cost 1
         (LSHS I32 (REG I32 "j.3%") (INTCONST I32 2)))
(SET I32 (REG I32 "a.1%0")           // .a.1%0 = [.T1% + .T2%]   cost 1
         (MEM I32 (ADD I32 (REG I32 ".T1%") (REG I32 ".T2%"))))
```

図 -11　a = b[j]; のコスト3の命令列

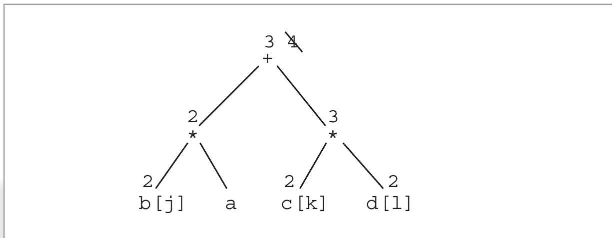


図 -12　最小限必要なレジスタ数

ものになる．これはフレーム変数 b.2 のアドレス（%fp レジスタの値－40）と，j.3% レジスタを左に 2 ビットシフトしたものを足してできる番地の内容を a.1% レジスタにセットすることを意味する．

命令選択では，この木のパターンにマッチする命令列をマシン記述ファイルの命令のパターンの中から探すのであるが，一般にはマッチするものは複数個あり得る．その中で最適なものを選ぶために，LIR の木の下の方からマッチするものすべてを調べながら，それらの命令のコストを加えていき，木のトップまでマッチしたところで，それまでの命令のコストの和が最小のものを選び，トップから下に向かって和が最小のものを構成している命令列をとり出す．今の例では**図 -11** がコストの和が最小 (=3) のものである．その図で "//" 以降はコメントである．ただし，パターンマッチングの際には，図 -11 の最後の命令の代わりに

```
.T3% = .T1% + .T2%      cost 1
.a.1%0 = [.T3%]         cost 1
```

に相当する命令を選んだもの（コストの和 =4）も調べた上で最小のものを選んでいる．

また，COINS の命令選択では，演算の途中結果を保持するレジスタの個数ができるだけ少なくなるように命令の順序を選んでいる（Sethi-Ullman の方法 [3),4)]）．それは，ある式の計算をするのに，途中結果をメモリに退避せずに計算するのに必要になるレジスタ数を最小にする方法である．その最小数を

```
b[j]*a+c[k]*d[l];
```

という式で求めたのが**図 -12** である．先の例で見たように，b[j] の値をレジスタにロードするのには途中で 2 つのレジスタ（先の例では .T1% と .T2%）が必要になる．b[j] の値を .T1% と .T2% のどちらかと同じレジスタにロードし，それに a.1%0 レジスタの値を掛けた値をその同じレジスタに置くとすれば，結局，b[j]*a の計算をするのに 2 つのレジスタを必要とし，結果が 1 つのレジスタに得られることになる．

右側の c[k]*d[l] について，同じことを考えると，c[k] の値を 1 つのレジスタに置いておいて，さらに d[l] の値をとり出すのに 2 つのレジスタを必要とするから，結局 c[k]*d[l] の計算をするのに 3 つのレジスタを必要とする．

そこで，b[j]*a+c[k]*d[l] 全体の計算をするのに，b[j]*a の計算を先にしたとすると，その値を保持するレジスタ 1 つと，c[k]*d[l] の計算をするレジスタ 3 つの合計 4 つのレジスタが必要になる．逆に，c[k]*d[l] の計算を先にすれば，3 つのレジスタを使ってその計算をして，その結果をそのうちの 1 つのレジスタに保持しておき，先に使った 3 つのレジスタのうちの残りの 2 つのレジスタを使って b[j]*a の計算をすればよいから，結局，3 つのレジスタですむことになる．

一般的には，必要とするレジスタの少ないほうを先に計算するように命令の順序を選べばよい．COINS の命令選択では，LIR の木のパターンマッチングをやりながら，それぞれの命令を実行するのに必要となるレジスタの個数も数えていき，コスト最小の命令を選んだときに，必要となるレジスタ数が多いほうの計算を先に実行するように命令を並べていっている．

Sethi-Ullman の方法では，上記のように，すでに使ったレジスタを使って次の計算をすることで，必要とするレジスタを少なくすることを考えており，そのレジスタの個数が実際のマシンの持つレジスタ数を超える場合の処置（どの時点でどのレジスタの値をメモリに移すか）も考えられているが，それは，変数はメモリに割り付け，レジスタを使うのは計算と計算の途中結果の保持のためだけである，という仮定に基づいている．COINS では，変数もレジスタに割り付けるので，命令選択では，実際のレジスタ割付けの時にレジスタの個数が少なくてすむように命令の順序を並べているだけである．レジスタ数が足りなくなったときの処置はレジスタ割付けの時に考えている．

```
(PROLOGUE (0 0) (REG I32 "%i0"))
(SET I32 (REG I32 "%i1") (STATIC I32 "x"))
(SET I32 (REG I32 "%i1") (MEM I32 (REG I32 "%i1")))
(SET I32 (REG I32 "%i0") (ADD I32 (REG I32 "%i1") (REG I32 "%i0")))
(EPILOGUE (0 0) (REG I32 "%i0"))
```

図-13 関数 func の命令列

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

図-14 関数 func の命令列のアセンブラコード

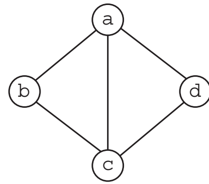


図-16 図-15 のプログラムの干渉グラフ

```
0: while (...){
1:   a = c * d;
2:   b = d + a;
3:   c = a - 5;
4:   d = b * a;
5: }
```

図-15 プログラム例

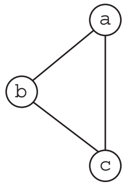


図-17 図-16 から d ノードを取り除いた干渉グラフ

⇩ レジスタ割付け

レジスタ割付けの結果，図-2 のソースプログラムに対する LIR 表現は**図-13** のようになる（ただし，PROLOGUE から EPILOGUE までを記述している）.

命令選択した直後には，PROLOGUE の後に

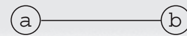
```
(SET I32 (REG I32 "y.1%0")
         (REG I32 "%i0"))
```

があったが，レジスタ割付けで y.1%0 に %i0 を割り付けたので，この命令が不要になって削除されている．EPILOGUE の直前にあった命令がなくなっているのも同様の理由である．

最後に出力されるアセンブラコードでは，PROLOGUE と EPILOGUE 以外の部分は**図-14** のようになる．

レジスタ割付けのアルゴリズムとしては，変数の生存区間の干渉グラフを使う方法がよく知られている．その方法を**図-15** の例を使って説明する．

図-15 の a は 1 行目で定義されて 4 行目まで使われているから，a の生存区間は 1 行目から 4 行目の直前までである．それを [1-4) と表現することにする（「[」は端点を含み「)」は端点を含まないことを表す記法である）．同様に，b の生存区間は [2-4)，c の生存区間は [3-1)（ループの繰り返しにまたがっている），d の生存区間は [4-2) である．a と b の生存区間には重なりがあるからその 2 つに同じレジスタを割り付けることはできない．このとき a と b はお互いに干渉しているとも言われる．そのことを a と b をエッジで結んだグラフのかたちで



と表現する．図-15 のすべての変数についてのグラフは**図-16** のようになる．このグラフは干渉グラフと呼ばれ，エッジの先の 2 つのノードが干渉していることを表している．

レジスタ割付けの問題は，この干渉グラフに対して，互いに干渉しているノードには異なるレジスタを割り付け，全体としてマシンの持つレジスタの個数（実際には，そのうちのレジスタ割付けに使えるレジスタの個数）の範囲内に収める問題である．異なるレジスタは異なる色で表現されると考えて，この問題はグラフの彩色問題とも呼ばれる．

この問題を最適に解く効率のよいアルゴリズムは存在せず，いろいろなヒューリスティックなアルゴリズムが考えられているが，基本的には次のような方法が使われている．

図-16 のグラフで，割り当てに使えるレジスタが 3 つあるとする．a は b，c，d と違う色にしなければならないから，b，c，d に先に色をつけてしまえば，色の数（レジスタ数）が足りなくなって，それらと異なる色が付けられなくなるかもしれない．しかし，d は a，c と違う色にすればよいので，a，c にどんな色がつけられても，それと違う色をつけることができる．したがって，グラフから d を取り除いて，残りのグラフ（**図-17**）に彩色ができれば，全部のグラフに彩色できる．図-17 からは，同様に考えて，a を取り除くことができる．さらにそれから b，最後に c を取り除くことができる．そのようにして，グラフのノードがなくなったら，取り除いたのと逆の順番に割り付けることで全部のノードに彩色できる．今の場合は，c に色 1，b には c の色とは違う色 2，a には b，c の色とは違う色 3，d には a，c の色とは違う色 2 を割り付ければよい．

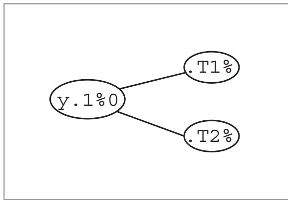


図 -18　関数 func の干渉グラフ

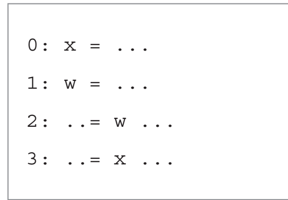


図 -19　プログラム例

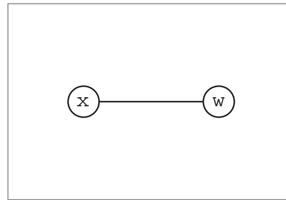


図 -20　干渉グラフ

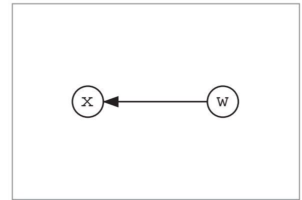


図 -21　妨害グラフ

一般には，あるノードから出ているエッジの数が割り当てに使えるレジスタの個数より少ないものがあったらそれを取り除くということを繰り返していって，ノードが 1 つもなくなれば彩色できる．しかし，ノードが残ってしまった場合が問題である．その場合は，残ったすべてのノードについて，それから出ているエッジの数は割付けに使えるレジスタの個数以上である．

そのような場合にはどれかのノード（変数）をレジスタ割付けから外してメモリに割り当てる（スピルするといわれる）ことにして，干渉グラフを作り直してレジスタ割り当てをやり直すことになる．スピルするノードの選び方としては，スピルのコスト（スピルしたことによって必要になるロード／ストア命令のコスト）が最小のものを選ぶのが通常の方法である．

COINS でのレジスタ割付けでも，このような干渉グラフによる彩色の方法を使っている．その詳細な方法についてはいろいろ研究されているが，その中で Iterated Register Coalescing[5] と呼ばれる方法をもとにしている．Coalescing は合併と訳される．それは，

```
a = b;
```

のような代入文があったときに，この 2 つの変数を合併してしまって同じレジスタに割り付けるようにすることである．そうすると，この代入文は削除できる．先に述べた例で，PROLOGUE の直後や EPILOGUE の直前の SET 式が削除されていたのは，この合併による効果である．

図 -7 と図 -8 に対して実際にレジスタ割付けをしてみると以下のようになる．まず，干渉グラフとしては**図 -18** が得られ，%i0 や returnvalue.2%0 と干渉するものはない．y.1%0 と returnvalue.2%0 はどちらも %i0 と合併することができ，ハードウェアレジスタである %i0 に割り付けられる．図 -18 から .T1% と .T2% はいずれも y.1%0 と違うレジスタであればよいので，同じ %i1 に割り付ければよい．その結果として図 -13 が得られる．

COINS では，スピルするノードの選び方に新しい工夫をしている．たとえば，**図 -19** のプログラムでは x の生存区間は [0-3) であり，w の生存区間は [1-2) である．生存区間に共通部分があるから干渉グラフでは x と w は干渉し，両者は同格である（**図 -20**）．しかし，スピルに関しては同格ではない．もし，x をスピルした場合は，w の生存区間に x の定義や参照は入っていないから，w にレジスタを割り付ける際に x の影響はない（x と w のためにレジスタを 1 個使うだけですませることもできる）．逆に w をスピルした場合は，x の生存区間に w の定義や参照が入っているので，w のために使うレジスタが x のレジスタと干渉してしまう（x と w のためにレジスタが 2 個必要になる）．このことを w が x へのレジスタ割付けを妨害すると考えて，**図 -21** のような妨害グラフで表現し，このグラフを使って，妨害され方が大きく，妨害の仕方が小さいものを選んで，その中で（通常の方法である）スピルコストの小さいものをスピルの対象として選んでいる．結局この例では x をスピルする．

おわりに

今回は，LIR の概要とバックエンドの概要を説明した．次回は，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する．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．次々回以降は，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/COINSdoc/
2）http://gcc.gnu.org/onlinedocs/gccint/RTL.html
3）中田育男：コンパイラ，産業図書（1981）．
4）中田育男：コンパイラの構成と最適化，朝倉書店（1999）．
5）George, L. and Appel, A. W. : Iterated Register Coalescing, 23rd POPL, pp.208-218, (1996). TOPLAS, Vol.18, No.3, pp.300-324, (1996).

（平成 18 年 5 月 12 日受付）

前回の連載で notavacc は LALR(1) と説明したが，ソースファイルの最後まで読んで，競合するシフトや還元の中から適切なものを選択する．この手法は，LALR(1) 文法に限定されず，任意の曖昧でない文脈自由文法を構文解析できるが，if 文は文献 [Clar] のように曖昧でない文法で定義しなければならない．この点を修正した C0Parser.notavacc を文献 [www] に置いた．なお，曖昧な文法を扱える notavacc も試験公開中である（前回の文献 3)）．

[Clar] Chris Clark: What to Do with a Dangling Else, ACM SIGPLAN Notices, Vol.34, No.2, pp.26-31(1999).
[www] http://www.coins-project.org/IPSJ-mitisirube/