「HIRの説明と簡単な言語のフロントエンド」

中田育男（法政大学）
nakata@cis.k.hosei.ac.jp

渡邊　坦
tan@coins-project.org

はじめに

本連載では，第1回から第4回までで，**図-1**のような構造を持ったコンパイラを作ってみる予定である．前回は，その第1回目として，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，本連載で扱う簡単なC0言語の説明をし，JavaCC[1), 2)]を使ってC0言語のコンパイラのフロントエンドのうち構文解析プログラムを作成するところまで説明した．

今回は，主として図-1の太字の部分の説明をし，さらにコンパイラ・ドライバの説明をする．**図-2**は，その部分をさらに詳細に記述したものである．図-2の黒字の部分を今回説明する．図-2の左右の太枠内が人が記述する部分であり，点線枠が自動生成されるものである．まず，JavaCCを使ってC0フロントエンドの意味解析プログラムを作成し，意味解析の結果をCOINSの高水準中間表現（HIR: High-level Intermediate Representation）に変換する方法を説明する．次回以降にバックエンドの作り方の説明をする予定であるが，今回はとりあえずCOINSにすでにあるバックエンドを使うことにする．そうすれば，C0フロントエンドを作っただけでC0言語のコンパイラができたことになる．

JavaCCによる構文解析はLL構文解析と呼ばれるものであるが，もう1つよく使われる方法としてLR構文解析と呼ばれるものがある．また，今回の前半では，構文解析と意味解析との関係を比較的分かりやすく説明するために，構文解析をすると同時に意味解析をして，直ちにHIRに変換する方法を説明するが，それができるのはC0言語が簡単な言語であるからである．より複雑な通常のプログラム言語に対しては，構文解析をした結果を抽象構文木と呼ばれるかたちに変換し，意味解析はその抽象構文木に対して行うのが一般的である．その方法については，後半でnotavaCCを使って説明することにする．

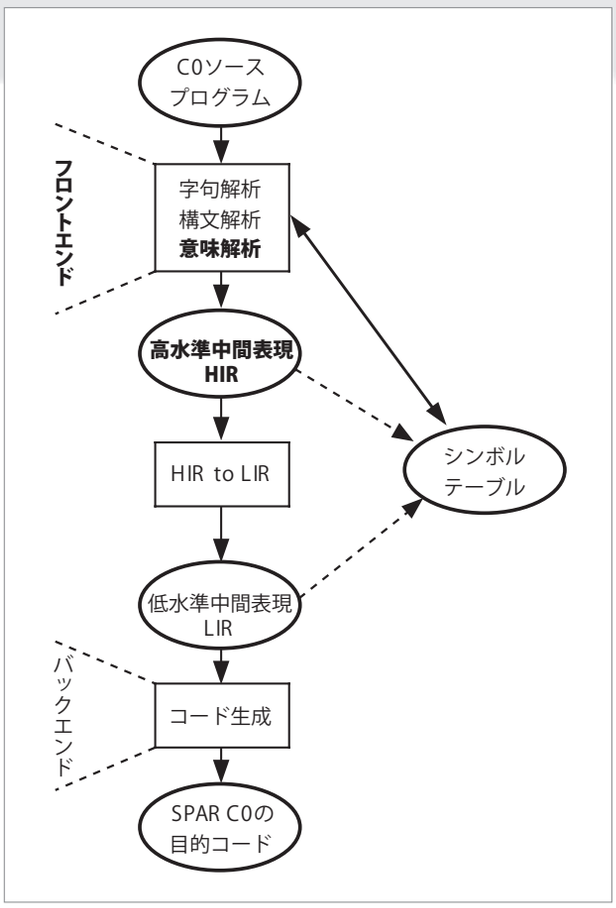

図-1　C0コンパイラの構成

●C0フロントエンドのJavaCC記述（その3：意味解析）

前回に構文解析のプログラムができたから，それに意味解析のプログラムを追加していくことにする．構文規則から構文解析のプログラムが生成されるように，意味

用語の定義がされているところに下線を付す．

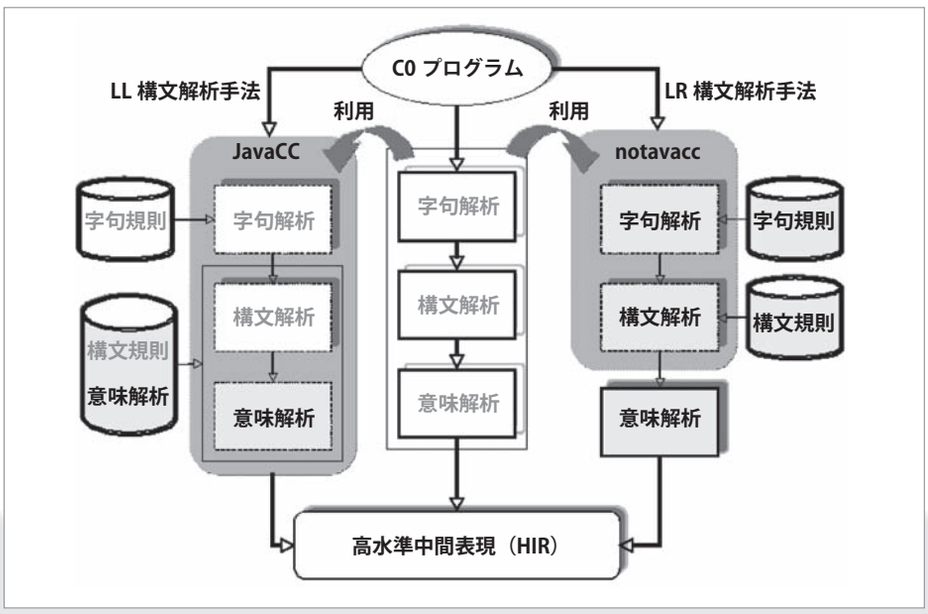

図 -2　フロントエンドの２通りの記述法

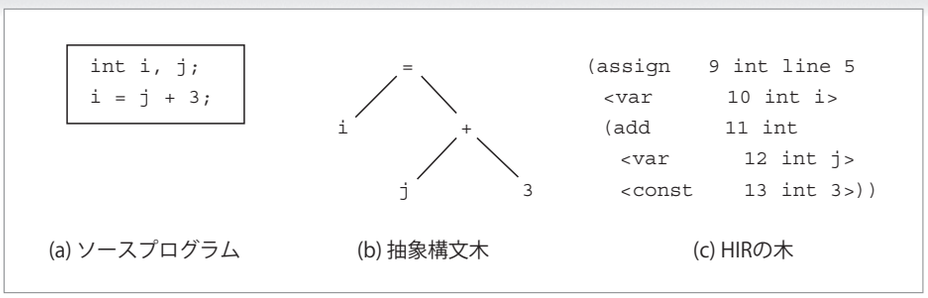

図 -3　ソースプログラムとその HIR 木

規則を書いたら意味解析のプログラムが生成されるシステムもあることはある（構文規則と意味規則を合わせて書く方法として属性文法と呼ばれるものがある[4]）．それを使えば，意味解析のプログラムを書く代わりに意味規則を書けばよい．しかし，実用的なコンパイラではそのようなシステムはあまり使われていないので，ここでは取り上げない．

ここでは，構文解析をしながら同時に意味解析をし，その結果を COINS の高水準中間表現 HIR のかたちにしていくことにする．それは抽象構文木に近いかたちである．以下，いくつかの項目について意味解析の仕方を説明する．

〈代入文〉

図 -3 で，(a) のソースプログラムの代入文の抽象構文木が (b) である．それを HIR で表現したものもほとんど同じ木のかたちである．それを見やすいようにプリントしたものが (c) である．(c) の中で，括弧（丸括弧または <> の括弧）で囲まれたものは 1 つのノードを表す．ある行が上の行より字下げしてプリントされているものは，上のノードの子供のノードである．同じ位置からプリントされているものは兄弟ノードである．ノードの中の数字 9 ～ 13 はノードに付けられた番号である．assign は代入文のノードであることを表し，代入される型は int 型であること，line 5 はこの代入文がソースプログラムの第 5 行にあるものであることを表している．その代入の左辺はその次に書かれている var ノードで，int 型の変数 i のノードである．右辺は add のノードから始まる木であり，その左側の子供が変数 j のノード，右側の子供が定数 3 のノードである．

図 -3 の (a) から (c) を作り出すプログラムは，次のように書けばよい．

代入文の構文規則は 4 月号の図 -5 で

10: Statement → Variable "=" Expression ";"

であったが，それを JavaCC 用に書き換えたものは

```
void stmt() : { }
{   LOOKAHEAD(2)
    variable() "=" expr() ";"
 | . . .   }
```

となっている．これに意味解析を付けて**図 -4** のようにすればよい（太字の部分が意味解析のために付け加えた部分である）．

```
Stmt stmt() : { Exp lExp, lExp2; }
{   LOOKAHEAD(2)
    lExp=variable() "=" lExp2=expr() ";"
     { return hir.assignStmt(lExp, lExp2); }
  | . . .   }
```

図 -4　代入文の意味解析プログラム

図 -4 で，stmt メソッドは HIR の Stmt 型の値を返すメソッドとし，variable と expr は Exp 型の値を返すメソッドとしている．代入文などの文(ステートメント)のノードを表現する HIR の型は Stmt 型であり，変数や式などのノードは Exp 型である．代入文の左辺として variable メソッドから返される値を lExp とし，右辺の値を lExp2 として

```
hir.assignStmt(lExp, lExp2);
```

を呼ぶと HIR での代入文（assign のノード）が得られる．HIR のノードを生成する各種のメソッドは COINS のパッケージ coins.ir.hir の中の HIR0.java (interface) に書かれている．hir は HIR0 型のオブジェクトであり，HIR のノードを生成するのに使われる（COINS ではオブジェクトの生成を簡便化するために，Java の new は使わずに，hir のようなオブジェクトを生成するためのオブジェクトを使う)．

〈変数宣言〉

意味解析が行う主要な仕事の 1 つは，変数の宣言情報をシンボルテーブルと呼ばれる表に集めておいて，それらの変数が使われているときにシンボルテーブルを調べて，その変数の型などの情報を取り出すことである．COINS にはそれらを行うメソッドがあるから，ここではそれを使って変数の宣言情報をシンボルテーブルに登録する．たとえば，変数宣言の構文規則は

3: VariableDeclaration → TypeId VarDecl
("," VarDecl)* ";"
6: TypeId → "int" | "void"
7: VarDecl → ID ("[" NUM "]")?

であったが，それを JavaCC 用に書き換えたものは

```
void variableDecl() : {}
{ typeId() varDecl() ("," varDecl())* ";" }
void typeId() : {}  {   <INT> | <VOID>  }
void varDecl() : {}
 {  <ID>  ( "[" <NUM> "]" )?  }
```

となっている．これに意味解析を付け加えると**図 -5** のようになる．

図 -5 で，4 行目の Type は HIR で型を表現するための型（Java の interface）であり，6，7 行目の typeInt，typeVoid はそれぞれ int 型，void 型を表す型オブジェクトである．HIR の各種の仕様は Java の interface として定義するという方針で書かれている．2 行目はメソッド typeId で得られた型の情報をメソッド varDecl に引数として渡している．このように情報を下向きに流すことで意味解析が簡単に書けるのは下向き構文解析の特徴である．LR 構文解析のような上向き構文解析では，情報は基本的には上向きに渡されるだけである．メソッド typeId では int か void かの型の情報を返す．基本的な型は symRoot オブジェクトに用意されている（たとえば，int 型には symRoot.typeInt が用意されている）のでそれを使えばよい．

hir が HIR のノードを生成するメソッドを持つオブジェクトであったように，sym はシンボルテーブルの情報を生成するメソッドを持つオブジェクトである．メソッド varDecl では 16 行目で sym.defineVar メソッドを呼んでシンボルを登録するが，その型は，「"[" <NUM> "]"」が付いている場合は，t を要素型とし長さを <NUM> の値とするベクトル型であり，ついていない場合は t 型である．13 行目の num.image は <NUM> の形をしたトークン num の文字列を表すので，Long.parseLong(num.image) が <NUM> の値となる．

〈グローバル変数／ローカル変数／関数宣言〉

シンボルテーブルは，変数などの名前のスコープに合わせた構造を持たなければならない．C0 言語では，ExternalDeclaration で宣言された変数はグローバルなスコープを持ち，関数内で宣言された変数や引数のスコープはその関数内である．たとえば，**図 -6** のプログラムでは 1 行目で宣言された変数 a，b はグローバルなスコープを持ち，func1 でその b が使われ，func2 で a が使われている．関数内で宣言された変数はローカルな変数とも呼ばれ，その関数内だけで使われる．関数内でグローバルな変数と同じ名前のローカル変数が宣言されると，その関数内ではその名前はローカル変数としてしか使えない．

図 -6 の関数 func1 の代入文で使われている各変数がどこで宣言されたものであるかの解析をするためには，グローバルな変数の宣言情報を集めたグローバルなシンボルテーブルと，関数 func1 内での宣言情報を集めたローカルなシンボルテーブルがあればよい．関数内で使用されている変数がどこで宣言されたものであるかを調べるためには，最初にローカルなシンボルテーブルを調べ，それがそこにあればローカルな変数である．なかっ

```
 1: void variableDecl() : { Type t; }
 2: {   t=typeId() varDecl(t) ("," varDecl(t))* ";"  }
 3:
 4: Type typeId() : {}
 5: {
 6:      <INT>  { return symRoot.typeInt; }
 7:   |  <VOID> { return symRoot.typeVoid; }
 8: }
 9: void varDecl(Type t) : { Token id, num; Var lVar; Type lType; }
10: {
11:   id=<ID>
12:    ( "[" num=<NUM> "]"
13:            { lType = sym.vectorType(t, Long.parseLong(num.image)); }
14:    |       { lType = t; }
15:    )
16:            { lVar = sym.defineVar(id.image.intern(), lType);
17:              lVar.setVisibility(Sym0.SYM_PUBLIC); }
18: }
```

図 -5
変数宣言の意味解析プログラム

```
int a, b;         /* global a, b */
void func1(){
     int a, c;    /* local a, c */
     b = a + c;   /* global b, local a, c */
}
void func2(){
     int b, c;    /* local b, c */
     b = a + c;   /* global a, local b, c */
}
```

図 -6　グローバル変数とローカル変数の例

たときにはグローバルなシンボルテーブルを調べればよい．

関数 func2 の解析にかかるときには，グローバルなシンボルテーブルと関数 func2 のローカルなシンボルテーブルが必要になる．そのようにシンボルテーブルを取り換えるためには，関数（サブプログラム）に入ったときは新たなスコープ用のシンボルテーブルをスタックに積み（プッシュし），出るときはそれを下ろし（ポップし）てもとのテーブルに戻せばよい．そのためのメソッドが以下に出てくる pushSymTable と popSymTable である．

関数宣言の部分の JavaCC 記述は次のようになっていた．

```
void externalDecl() : {  }
{   LOOKAHEAD(3)
     variableDecl()
   | subpDeclarator()
      (   ";"
       | "{"  ( variableDecl() )*
              ( stmt() )* "}"
      )
}

void subpDeclarator() : { }
{
  typeId() <ID>
    "(" ( paramDecl()
          ("," paramDecl())* )? ")"
}

void paramDecl() : {  }
{
   typeId() <ID>
   (  "["  "]"  ) ?
}
```

これに意味解析を付け加えると**図 -7** のようになる．

図 -7 では，subpDeclarator の中で paramDecl を読み始める前に pushSymTable を呼んでいる（25 行目）．引数宣言のところからスコープが変わるからである．27 行目の closeSubpHeader はサブプログラムのヘッダを完結させる（仮引数などの情報を整理する）呼び出しである．subpDeclarator だけで関数の本体がついていない場合は，subpDeclarator の処理が済んだら直ちに popSymTable を呼んでシンボルテーブルをグローバル

```
 1: void externalDecl() : { Subp lSubp; BlockStmt lBlockStmt; Stmt lStmt; }
 2: {    LOOKAHEAD(3)
 3:     variableDecl()
 4:   | lSubp=subpDeclarator()
 5:     (   ";"        { lSubp.setVisibility(Sym0.SYM_EXTERN);
 6:                      symRoot.symTableCurrent.popSymTable(); }
 7:       | "{"
 8:          ( variableDecl() )*
 9:                        { lBlockStmt = hir.blockStmt(null); }
10:          ( lStmt=stmt()   { lBlockStmt.addLastStmt(lStmt); }
11:          )*
12:            { lSubp.setVisibility(Sym0.SYM_PUBLIC);
13:              SubpDefinition lSubpDef
14:                      = hir.subpDefinition(lSubp, symRoot.symTableCurrent);
15:              lSubpDef.setHirBody(lBlockStmt);
16:              ((Program)hirRoot.programRoot).addSubpDefinition(lSubpDef);
17:              symRoot.symTableCurrent.popSymTable();
18:            }
19:      "}"
20:     )
21: }
22: Subp subpDeclarator() : { Type t; Token id; Subp lSubp;}
23: {
24:   t=typeId() id=<ID>   { lSubp = sym.defineSubp(id.image.intern(), t);
25:                          symRoot.symTableCurrent.pushSymTable(lSubp);}
26:     "(" ( paramDecl(lSubp) ("," paramDecl(lSubp))* )? ")"
27:                     { lSubp.closeSubpHeader();
28:                       return lSubp;
29:                     }
30: }
31:
32: void paramDecl(Subp pSubp) : { Type t, lType; Token id; }
33: {
34:   t=typeId() id=<ID>
35:  (  "["  "]"  { lType = sym.pointerType(t); }
36:   |           { lType = t; }
37:  )          { Param lParam = sym.defineParam(id.image.intern(), lType);
38:               pSubp.addParam(lParam);
39:            }
40: }
```

図-7 関数宣言の意味解析

なテーブルに戻す必要がある（6行目）が，本体がある場合は，ローカルなシンボルテーブルを使って本体のローカル変数宣言やステートメントの処理を全部済ませてからpopSymTableを呼ぶ必要がある（17行目）．

〈引数〉

引数の受け渡し方法としては，C0コンパイラの目的コードではC言語などと同じようにint型の引数の場合は値を渡し，配列の場合はポインタを渡すことにしている．メソッドparamDeclの中で配列型の引数の型をsym.pointerType(t)としているのはそのためである．それに合わせて，その関数を呼び出すときの実引数の配列については，その配列のポインタを作って渡すようにする必要がある．一方，関数の中で配列の引数を使っているところではポインタから配列を作り出す必要がある．これらを実現するために，配列からポインタに変換する

```
 1: Exp variable():{Token id; Exp lExp; }
 2: {
 3:   id=<ID>    { Sym0 lSym = (Var)symRoot.symTableCurrent.search
 4:                                        (id.image.intern());
 5:                if (!(lSym instanceof Var)) {
 6:                  recoveredError("undeclared variable " + id.image,
 7:                                                 id.beginLine);
 8:                  lSym = sym.defineVar(id.image, symRoot.typeInt);
 9:                }
10:                Exp lVarNode = hir.varNode((Var)lSym);
11:              }
12:     ( "[" lExp=expr() "]"
13:              { if (lSym.getSymType() instanceof PointerType)
14:                  lExp = hir.subscriptedExp
15:                                    (hir.contentsExp(lVarNode), lExp);
16:                else
17:                   lExp = hir.subscriptedExp(lVarNode, lExp);  }
18:      |       {  if (lSym.getSymType() instanceof VectorType)
19:                   lExp = hir.exp(HIR0.OP_ADDR, lVarNode);
20:                else
21:                   lExp = lVarNode;  }
22:     )        { return lExp;  }
23: }
```

図 -8
変数使用の意味解析プログラム

ノードと，ポインタから配列に変換するノードを作成するメソッドがHIRに用意されているので，それらを使えばよい．そのためには，JavaCCの記述では

```
void variable():{}
{
  <ID>
    ( "[" expr() "]" | {} )
}
```

となっているのに対して，意味規則を加えて**図 -8** のようにすればよい．

図 -8 の 3 行目の，

```
symRoot.symTableCurrent.search
                        (id.image.intern());
```

は，現在のシンボルテーブルの中で変数（id）の宣言情報を探しているところである．この search メソッドでは，まず現在のシンボルテーブル（スタックのトップにある）の中を探し，なければさらにスタックの奥のシンボルテーブルを探し，最初に見つかったものを返す．また 14 行目の

```
lExp = hir.subscriptedExp
        (hir.contentsExp(lVarNode), lExp);
```

は，引数のポインタからそれが指している配列を作ってそれに添え字を付ける HIR 表現を作っているところである．これによって，たとえば

```
int func(int y[]){
 ....y[3];
}
int main(){
  int z[5];
  ...func(z);
}
```

の y[3] に対して，

```
(subs     12 int
  (contents 13 <VECT 0 0 int>
    <var      14 <PTR <VECT 0 0 int>> y>)
  <const    15 int 3>)
```

が作られる．subs は配列要素，VECT は 1 次元配列，PTR はポインタであることを示している．また func(z) の z に対しては

```
 1: options {
 2:  STATIC=false;
 3: }
 4: PARSER_BEGIN(C0Parser)
 5: package c0front;
 6:
 7: import ...
 8: public class C0Parser {
 9: ...
10: private Sym0 sym;          // Sym0 object creator
11: private HIR0 hir;          // HIR0 object creator
12: public C0Parser(SymRoot sRoot, HirRoot hRoot, IoRoot iRoot,
13:                              FileReader reader) {
14:      ...
15:      hir = (HIR0)hRoot.hir; sym = (Sym0)sRoot.sym;
16:  }
17: }
18: PARSER_END(C0Parser)
```

図 -9
C0 フロントエンドの JavaCC 記述の先頭部分

```
(addr     28 <PTR <VECT 5 0 int>>
 <var      29 <VECT 5 0 int> z>)
```

が作られる．これは 19 行目の

```
lExp = hir.exp(HIR0.OP_ADDR, lVarNode);
```

によって作られ，配列の先頭番地をポインタとするものである．

なお，recoveredError はエラーがあったけれどリカバリを試みたというメッセージを出力するメソッドである．

以上，いくつかの項目についての意味解析の説明をした．その他の部分もこれにならって書けばよい．

ところで，今までの説明に使われていた hir や sym などを最初に設定しておく必要がある．それは，最初に JavaCC の生成する parser（構文解析プログラム）のクラスを書くところに，**図 -9** の 15 行目のように書いておけばよい．

図 -9 の 5 行目は，C0Parser クラスをおくパッケージを指定する package 文である．7 行目は，意味解析プログラムが使っているクラスの import 文を書いておくところである．また，JavaCC によって生成されるクラスのメソッドはデフォルトではクラスメソッドになる．図 -9 の最初の 3 行はそれをインスタンスメソッドにするためのオプション指定である．

以上の方針ですべてを書けば，C0 コンパイラのフロントエンドができあがる．紙面の関係でここにすべてのプログラムを載せることはできないが，書き上げたものが文献 6）においてあるので参考にして欲しい．また，hir や sym の使い方の詳細については文献 5）の HIR の章を参照されたい．

● C0コンパイラのコンパイラ・ドライバ

フロントエンドができたら，後は COINS の最適化モジュールやバックエンドを使うことにすれば，もう 1 つコンパイラ・ドライバを作るだけでコンパイラが完成する．そのコンパイラ・ドライバは COINS が標準として備える C コンパイラのドライバ（coins.driver.Driver）があるので，それを真似て作ればよい．真似をする簡単な方法はそのサブクラスとして作る方法である．たとえば，**図 -10** のようなクラスを作ればよい（main と makeHirFromSource の 2 つのメソッドだけをオーバライドしている）．

main は図 -10 の 3 行目にあるように Driver の go メソッドを呼ぶだけでよい．それによって，コンパイル時のオプション指定に従ってプレプロセッサ，コンパイラ，アセンブラなどが次々に呼ばれる．コンパイルは Driver クラスにある compile というメソッドで行われ，その中でフロントエンドやバックエンドが呼ばれる．この C0Driver では，compile メソッドからフロントエンドを呼び出すメソッド makeHirFromSource だけをオーバライドしている．JavaCC によって生成された C0Parser クラスのオブジェクト parser が 9 行目で作られる．その parser の program メソッドが C0 言語の開始記号 program から作られているメソッドであり，構文解析の主メソッドである．それが，12 行目で呼び出され，それによって構文解析が行われる．構文解析が完了したら最後に finishHir() メソッドを呼ぶことに

```
 1: public class C0Driver extends Driver {
 2:  public static void main(String[] args) {
 3:    new C0Driver().go(args);
 4:    }
 5:  protected void makeHirFromSource(File sourceFile, HirRoot hirRoot,
 6:                        Suffix suffix, InputStream in, IoRoot io)
 7:  throws IOException, PassException {
 8:    if (suffix.getLanguageName().equals("C")) {
 9:       C0Parser parser = new C0Parser(io.symRoot, hirRoot, io,
10:                                    new FileReader(sourceFile));
11:       try {
12:          parser.program();
13:          ((Program)hirRoot.programRoot).finishHir();
14:          ...
15:       } catch (Exception e) {
16:          io.msgError.put("in C0 parser: "+e);
17:       }
18:    } else {
19:     ...
20:    }
21:  }
22: }
```

図-10
C0コンパイラ・ドライバのプログラム

よって，HIRがちゃんとした木構造をしているか，関連するデータ構造間に不整合がないかがチェックされて，各ノードに番号が付けられる．後のコードはトレース情報を出力したり，エラーチェックをしたりするものであるが，それは省略してある．

このプログラムに，必要なimport文などを付けたファイルが文献6）に置いてある．

C0コンパイラによる実験

C0コンパイラのクラスはCOINS（をダウンロードして展開したディレクトリ）のclassesディレクトリの中にたとえば"c0front"というディレクトリを作ってその中に置けばよい．そのようにしたとすると，C0コンパイラによる実験をするためには，次のようなコマンドを打てばよい．

```
java -cp $COINS/classes c0front.C0Driver
 options c0source.c
```

ここで*$COINS*はCOINSを展開したディレクトリを表す環境変数の参照であり，*options*はいろいろなオプション指定である．

オプション指定には，gccで使われているようなのと同様のオプション指定とCOINS特有のオプション指定がある．前者には`-S`でアセンブラ出力までを指定するものなどがある．後者には

(1) ターゲット・アーキテクチャの指定
(2) 各種の最適化の指定
(3) コンパイル過程のトレース情報の指定

などがある．

(1) ではアーキテクチャだけでなく，たとえば

```
-coins:target=x86-cygwin
```

のように環境などを指定することもできる．`target=x86`だけであればLinuxを仮定する．現在，アーキテクチャとしては`sparc`，`x86`，`ppc`（PowerPC），`sh4`，`arm`，`mips`，などを指定することができる．

(2) には，HIRでのデーフロー解析による各種の最適化，LIRでのSSA最適化などがある．

(3) には，

```
-coins:trace=HIR.1/Sym.1/LIR.1
```

によるHIR, シンボルテーブル, LIRなどの表示（`HIR.3`などのように数値を大きくするとより多くの情報が出力される）のほかに，CoVisと呼ぶツールによるグラフ表

```
java -cp $COINS/classes c0front.C0Driver -coins:trace=HIR.1 ex1.c

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

図-11 ソースプログラム ex1.c

```
(prog     1
 <null 0 void>
 <nullNode 2>
 (subpDef  3 void
  <subp     4 <SUBP < int > false false int> func>
 <null 0 void>
  (labeldSt 5 void
   (list 6
    <labelDef 7 _lab1>)
   (block    8 void
    (return   9 int
     (add      10 int
      <var      11 int x>
      <var      12 int y>))))))
```

図-12 プログラム ex1.c の HIR 表現

示を含んだソース/HIR/LIR の対応表示や，バックエンドでのコード生成過程のブラウザによる表示などがある．詳しくは文献 5）を参照されたい．

たとえば，**図-11** の ex1.c をソースプログラムとして，以下のコマンドを打つと，**図-12** の HIR 表現が得られる．

C0 コンパイラをとりあえず動かしてみるためには，たとえば文献 6）にある C0 言語のプログラム prime.c と C 言語のプログラム print.c を使って

```
java -cp $COINS/classes c0front.C0Driver
 -S target-option prime.c
gcc prime.s print.c
./a.out
```

と打って，2 から 541 までの素数がプリントされたら成功である．print.c では C 言語の printf を使っている．C0 言語のプログラムでは printf を直接呼ぶことができないので，print.c の関数を呼ぶようにしている．*target-option* はこの操作の環境に合わせてマシンを指定する必要がある．

notavacc を使ったフロントエンド

ここまでで C0 コンパイラのフロントエンドが 1 つできあがったが，フロントエンドの作り方としては他の方法も考えられる．構文解析の方法としては，LL 構文解析より適用できる文法の範囲が広い LR 構文解析がある．LR 構文解析では，LL 構文解析での左再帰性の問題もない．また，意味解析は，構文解析と同時に行うより抽象構文木に対して行うほうが複雑な意味規則にも対処できる．

構文解析プログラムを自動生成するシステムとして最初に実用化され，現在でも使われているものに yacc がある．この名前は Yet Another Compiler Compiler からきている．これは，コンパイラを自動生成するコンパイラ・コンパイラの実現は難しいが，自動化できるところだけを自動化し，難しいところは C 言語のプログラムとして表現すればよい，という考えで実用化に成功したものである．yacc による構文解析は LR 構文解析である．しかし，本連載では yacc ではなく，notavacc[3), 9), 10)] を使うことにする．その理由は，プログラム言語が C 言語でなく Java であることと，抽象構文木を作成する機能もあること，筆者たちの身近で最近開発されたものであることなどである．これを使って，LR 構文解析や抽象構文木を対象にして意味解析をする方法を説明する．

●LR 構文解析とは

LR 構文解析は上向き構文解析の一種である．上向き構文解析では，ソースプログラムのトークンを見ながら，ある生成規則（構文規則）の右辺を読み終わっていると判断したときは，読み込んであったもののうちその右辺に対応する部分を左辺に還元する（それで左辺の非終端記号に相当するものが構成されたとする）．そう判断できないときは，そのトークンを読み込んで，その次のトークンを見る，ということを繰り返していく．

読み込んだものはスタックに積んでいく．還元するときは，スタックの上から生成規則の右辺に相当する部分を取り出して，その代わりに左辺の非終端記号に相当するものをスタックに積む．その構文解析の進行状況は

（スタックの現状，ソースプログラムの残り）

というかたちで表現することができる．ここで，スタックは横向きであり，左端がスタックの底で右端がトップである．上記の動作のうち「トークンを読み込む」とい

```
 1: (ExternalDeclaration SubprogramDeclarator "{",                          x = 1; }$)      シフト
 2: (ExternalDeclaration SubprogramDeclarator "{" ID,                        = 1; }$)      シフト
 3: (ExternalDeclaration SubprogramDeclarator "{" Variable,                  = 1; }$)      還元
 4: (ExternalDeclaration SubprogramDeclarator "{" Variable "=",                1; }$)      シフト
 5: (ExternalDeclaration SubprogramDeclarator "{" Variable "=" NUM,            ; }$)      シフト
 6: (ExternalDeclaration SubprogramDeclarator "{" Variable "=" Factor,         ; }$)      還元
 7: (ExternalDeclaration SubprogramDeclarator "{" Variable "=" Term,           ; }$)      還元
 8: (ExternalDeclaration SubprogramDeclarator "{" Variable "=" Expression,     ; }$)      還元
 9: (ExternalDeclaration SubprogramDeclarator "{" Variable "=" Expression ";",  }$)      シフト
10: (ExternalDeclaration SubprogramDeclarator "{" Statement,                    }$)      還元
```

図 -13 LR 構文解析の進行状況

う動作は，そのトークンを「ソースプログラムの残り」からスタックの方へ移す動作になり，シフトと呼ばれる．

たとえば，C0 言語のソースプログラム，

```
int x;
int main(){ x = 1; }
```

の構文解析をしていて，"{" まで読み込んだときの進行状況の表現は，`int x;` が ExternalDeclaration として認識され，`int main()` が SubprogramDeclarator として認識されているので，

```
(ExternalDeclaration SubprogramDeclarator "{",
                                    x = 1; }$)
```

となっている．ここで，`$` はソースプログラムの終わりを示す記号である．たとえば，それはファイルの終わりに相当する記号である．これ以後の構文解析の進行状況を**図 -13** に示す．

次に，ソースプログラムの残りの先頭にある x を見てそれをトークン ID としてシフトすると図 -13 の 2 行目のようになり，さらにその ID が C0 の文法（前回の図 -5 の 30 行目の構文規則で Variable に還元されて 3 行目）のようになる．

このように，構文解析は，シフトか還元かの繰り返しで進行する（1 行目は "{" をシフトした結果である）．いつの時点でも，シフトすべきか還元すべきかが決定でき，還元するときはどの構文規則で還元すべきかが決定できれば，構文解析ができることになる．しかし，もとの文法によっては，それが簡単に決定できない場合がある．ある時点で，シフトと還元の両方の可能性がある場合には，シフト／還元競合（コンフリクト）があるといわれる．また，複数の還元の可能性があるときは還元／還元競合があるといわれる．そのような場合には，もとの文法をより詳しく調べて決めることを試みる．比較的簡単な調べ方でそれが決められるような文法は SLR(1) 文法（Simple LR）と呼ばれる．その決め方で構文解析するのが SLR(1) 構文解析である．カッコの中の「1」は，次のトークンを 1 つ調べるだけで決められるという意味である．LR 文法のうち，1 番詳細に調べて決められるような文法が LR(1) 文法であり，その中間的な方法で調べて決められるような文法は LALR(1) 文法（Look Ahead LR）と呼ばれる．これらの詳細については参考文献[4), 7), 8)]を参照されたい．

なお，LR 構文解析では，通常は次の 1 つのトークンを見ることしかしない．LL 構文解析の場合は，右辺の先頭部分だけを見て決めてしまおうとするので，1 つのトークンだけでは決まらない場合のことを考える必要があるが，LR 構文解析では右辺を全部読んでから（どの生成規則の右辺であったかを）決めようとするので，1 つのトークンだけで十分と考えられている．

● C0フロントエンドのnotavacc記述（その1：字句規則）

notavacc では，C0 言語の字句規則に相当するものをたとえば，**図 -14** のように書けばよい．

図 -14 で，`$subtoken` は JavaCC の字句規則での "#" に相当する．すなわち，他のトークンの定義に使われる名前で構文解析プログラムには渡されない．1 行目の `'¥u0000'` は Unicode の文字コードである．3 行目の `$white` は JavaCC での `SKIP` に相当し，無視されるトークンを表す．notavacc の特徴的な機能として，正規表現に差集合が使える．これを使うとコメントは 8 行目のように書ける．その中の

```
( ANY_CHARACTER* - ( ANY_CHARACTER* "*/"
                     ANY_CHARACTER* ) )
```

は，任意文字の列で，途中に "*/" がないものを表す．

```
 1: $subtoken ANY_CHARACTER = '¥u0000'..'¥uFFFF' ;
 2: $subtoken WS     = ' ' | '¥t' | '¥r' | '¥n' ;
 3: $white $token WSS = WS+ ;
 4:
 5: $subtoken DIGIT  = '0'..'9' ;
 6: $subtoken LETTER = 'a'..'z' | 'A'..'Z' ;
 7: $white $token TRADITIONAL_COMMENT =
 8:  "/*" ( ANY_CHARACTER* - ( ANY_CHARACTER* "*/" ANY_CHARACTER* ) ) "*/" ;
 9:
10: $token ID = LETTER ( LETTER | DIGIT )* - KEYWORDS ;
11: $subtoken KEYWORDS = "int" | "void" | "if" | "else" | "while" |"return";
12: $token NUM = DIGIT+ ;
```

図 -14
C0 言語の
字句規則の notavacc 記述

```
1: $abstract Stmnt { }
2: stmnt = AssignStmnt -> Stmnt { left:Variable "=" right:expr ";" }
3:     | IfStmnt -> Stmnt { "if" "(" condExpr:conditionalExpr ")"
4:                     thenPart:stmnt ( "else" elsePart:stmnt )? }
5:     ;
```

図 -15　代入文と if 文の構文記述

● C0フロントエンドのnotavacc記述（その2：構文規則）

notavacc に構文規則を与えると LALR(1) 構文解析プログラムを生成する．JavaCC の場合は LL 構文解析ができるようにするために一部文法を書き直す必要もあったが，notavacc ではそのような必要はない．ただし，notavacc で決められているかたちにしなければならない．また，抽象構文木を作るための書き換えも必要になる．たとえば，

A → B D

に対しては

A { B D }

の形に書かなければならない．また 4 月号の図 -5 の

9: ParamDecl → TypeId ID ("[" "]")?

に対しては

```
ParamDecl { typeId ID  (  "["  "]"  ) ? }
```

と書けばよい．

ただし，notavacc が生成する構文解析プログラムは，構文解析するだけでなく解析結果から抽象構文木を作っていくので，抽象構文木にアクセスする方法を構文規則の中に書かなければならない．アクセスするものには名前（label）を付けなければならない．たとえば，

7: VarDecl → ID ("[" NUM "]")?

に対して

```
VarDecl  { varId:ID
             ( "[" arraySize:NUM "]" )? }
```

と書くと，VarDecl から作られる抽象構文木の ID と NUM の内容はメソッド varId() と arraySize() で取り出せるようになる．ソースプログラムに "[" NUM "]" のかたちがなかったときは arraySize() の値は null になる．

ところで，できあがった抽象構文木では各ステートメントが代入文であるか if 文であるかなどの区別をしたいので AssignStmnt とか IfStmnt とかの名前を導入したい．またそれらはすべて文であるので共通な名前，たとえば Stmnt で扱えるようにしたい．そのためには**図 -15** のように書けばよい．

図 -15 の 1 行目の $abstract は，Stmnt 型のオブジェクトが作られることはないという意味である．2 行目の「stmnt =」はエイリアス（別名）の定義を表す．stmnt が現れたところは以下の AssignStmnt や IfStmnt で置き換え得る（言い換え得る）という意味である．「AssignStmnt -> Stmnt」は AssignStmnt が Stmnt のサブ interface であること，すなわち Java の書き方で

```
interface AssignStmnt extends Stmnt
```

を意味する．また 2 行目の AssignStmnt 以下は

AssignStmnt → Variable "=" expr ";"

という構文規則を表している．なお，エイリアスの定義の最後には ";" が必要である（5 行目）．

ところで，JavaCC の場合にも述べたように，この if 文の定義はあいまいである．notavacc の場合も，当然 warning メッセージが出る．それは "else" を見たときにシフトすべきか還元すべきか分からない（シフト／還元競合がある）からである．"else" を見たときにシフトの方を採るとすると，その "else" はそれに最も近い

```
 1: $abstract Expr { }
 2: BinExpr -> Expr {operand1:expr operand2:expr}
 3:
 4: expr = Add -> BinExpr { operand1:expr "+" operand2:term }
 5:      | Sub -> BinExpr { operand1:expr "-" operand2:term }
 6:      | term
 7:      ;
 8: term = Mul -> BinExpr { operand1:term "*" operand2:factor }
 9:      | Div -> BinExpr { operand1:term "/" operand2:factor }
10:      | factor
11:      ;
```

図 -16 2 項演算の notavacc 記述

"if" に対応付けられることになって，言語の仕様に合うことになる．

一般に，LR 構文解析プログラム生成系では，シフト／還元競合があった場合は，デフォルトの動作としてはシフトを優先することになっている．notavacc の場合もそうなっているので，この warning メッセージは無視すればよい．

四則演算については加減乗除の 4 つの生成規則がある．

19: Expression → Expression "+" Term
20: | Expression "-" Term
21: | Term
22: Term → Term "*" Factor
23: | Term "/" Factor
24: | Factor

ここで，演算ノードは演算子の種類ごとに作るとしたら上記のエイリアスの定義を使えばよいが，さらに，意味解析ではそれらをまとめて扱いたいとしたら，たとえば，2 項演算のノードを扱うための interface として `BinExpr` を導入して，**図 -16** のように書けばよい．

これで，`Add` も `Mul` も `BinExpr` として扱えるようになる．

プログラムとして表現するときには必要であるが，抽象構文としては必要のないものもある．たとえば，

25: Factor → "(" Expression ")"

は，演算の優先順位を示すためにカッコをつけるという構文規則であるが，抽象構文木ではそれは必要がないから，新たな名前を作る必要はなく，右辺の Expression の構文木についている名前をそのまま使えばよい．それを示すにはこの右辺を次のように書けばよい．

```
"(" $label:expr ")" ;
```

ここで `$label` は，`expr` についている名前（label）をそのまま使うという意味である．

以上の方針に従って C0 フロントエンドを記述したものは文献 6）に C0Parser.notavacc として置いてある．C0Parser.notavacc では，生成される構文解析プログラムのクラス名を指定しなければならないので，

```
$parser c0front2.C0Parser;
```

としている．また，生成規則の開始記号は `$parsable` で指定しなければならない．それを

```
$parsable
Program { externalDecls:externalDecl+ }
```

と指定している．これで，`parseProgram` という構文解析の主メソッドが生成される．なお，この例のように繰り返し記号の付いたものにつけられた名前 `externalDecls` に対しては，java.util.List 型のノードが作られる．

● C0フロントエンドのnotavacc記述（その3：意味解析）

意味解析プログラムは，抽象構文木に対して書くことになる．ソースプログラムと抽象構文木との対応関係は notavacc 記述として書かれている．今の場合，notavacc 記述の生成規則の最初の部分は**図 -17** のようになっている．これは，C0 言語の文法の

1: Program → ExternalDeclaration
(ExternalDeclaration)*
2: ExternalDeclaration → VariableDeclaration |
SubprogramDeclaration | SubprogramDefinition
3: VariableDeclaration → TypeId VarDecl
(“,” VarDecl)* “;”
4: SubprogramDeclaration →
SubprogramDeclarator “;”
5: SubprogramDefinition → SubprogramDeclarator
“{” (VariableDeclaration)* (Statement)* “}”

のかたちの通りに書かれている．それに対する意味解析プログラムは**図 -18** のように書けばよい．

C0Parser.notavacc で，生成される構文解析プログラム

```
 1: $parser c0front2.C0Parser;
 2: .....
 3: $parsable Program { externalDecls:externalDecl+ }
 4:
 5: $abstract ExternalDecl { }
 6:
 7: externalDecl =
 8:   VariableDecl -> ExternalDecl
 9:     { typeName:TypeId varDecls:VarDecl ("," varDecls:VarDecl )* ";" }
10:  | SubpDecl -> ExternalDecl
11:     { subpSpec:SubpDeclarator ";"  }
12:  | SubpDef -> ExternalDecl
13:      { subpSpec:SubpDeclarator
14:        "{" ( variableDecls:VariableDecl )* ( stmnts:stmnt )* "}" }
15:  ;
```

図 -17
notavacc 記述の生成規則の最初の部分

```
 1: public void makeHirFromC0(File sourceFile, HirRoot hirRoot)
 2:                          throws ParseException, IOException
 3: {
 4:   C0Parser.Program program = parseProgram(sourceFile, null);
 5:
 6:   Iterator it0 = program.externalDecls().iterator();
 7:   while (it0.hasNext()) {
 8:     C0Parser.ExternalDecl lExternalDecl = (C0Parser.ExternalDecl)it0.next();
 9:     if (lExternalDecl instanceof C0Parser.VariableDecl) {
10:       C0Parser.VariableDecl lVarDecl = (C0Parser.VariableDecl)lExternalDecl;
11:       processVariableDecl(lVarDecl);
12:     }else if (lExternalDecl instanceof C0Parser.SubpDecl) {
13:       ...
14:     }else if (lExternalDecl instanceof C0Parser.SubpDef) {
15:       ...
16:     }
17:   } // End of external declaration sequence
18: }
19:
20: void processVariableDecl( C0Parser.VariableDecl pVarDecl )
21: {
22:    Type lDeclType = makeType(pVarDecl.typeName().getImage().intern());
23:    Iterator it = pVarDecl.varDecls().iterator();
24:    while (it.hasNext()) {
25:       variableDeclaration(lDeclType, (C0Parser.VarDecl)it.next());
26:    }
27: }
```

図 -18　図 -17 に対応する部分の意味解析プログラム

のクラス名を C0Parser としてある (図 -17 の 1 行目) と，生成された C0Parser の中では，C0Parser.notavacc での非終端記号 xxx に対応するものは，C0Parser.xxx という interface 名になっている．たとえば，図 -18 の 4 行目の C0Parser.Program は図 -17 の 3 行目の Program に対応する interface 名である．図 -18 では，最初に 4 行

目の parseProgram で構文解析をし（2 番目の引数は文字セットの名前を示すものであるが，デフォルトのセットでよいから null としてある），その結果が program に得られる．そのかたちは，図 -17 の 3 行目により externalDecl の列である．その externalDecl は図 -17 の 7 行から 12 行により ExternalDecl であり，それは VariableDecl か SubpDecl か SubpDef である．それに対応して，図 -18 の 6 行目で program に対して externalDecls メソッドを呼ぶと ExternalDecl のリストが得られる．そのリストの中身を 8 行目で lExternalDecl に取り出して，それが VariableDecl（9 行目），SubpDecl（12 行目），SubpDef（14 行目）のどれであるかによって，それぞれの処理をしている．VariableDecl の場合は，図 -17 の 9 行目によって型名 TypeId とそれに続く VarDecl のリストがあるはずであるので，図 -18 では，20 行以降にあるメソッドでそれを処理している．

```
private Exp makeExp(C0Parser.Expr pExpr)
  {
   Exp lExp = null;
   int operator = 0;
   if (pExpr instanceof BinExpr){
     C0Parser.BinExpr lBinExpr = (C0Parser.BinExpr)pExpr;
      if (pExpr instanceof Add) {
         operator = HIR0.OP_ADD;
      } else if (pExpr instanceof C0Parser.Sub) {
        operator = HIR0.OP_SUB;
      } else if (pExpr instanceof C0Parser.Mul) {
        operator = HIR0.OP_MULT;
      } else if (pExpr instanceof C0Parser.Div) {
        operator = HIR0.OP_DIV;
      }
      lExp = hir.exp(operator, makeExp(lBinExpr.operand1()),
                               makeExp(lBinExpr.operand2()));
    } else if . . .
```

図 -19　四則演算の意味解析部分

意味解析の内容は，JavaCC の場合に述べたものと同じであるので，省略する．

なお，最初の JavaCC の場合は，構文規則と意味解析のプログラムを一緒に書くことができたのに，今度の場合は 2 つに分けて書いているのを煩わしいと感ずるかもしれない．しかし，C0 の場合は簡単な言語であるので，前者のように簡潔に書くことができたが，複雑な言語になると，後者のように分けて書いたほうが書きやすいことが多い．

図 -16 に書いた四則演算の部分を HIR に変換する部分は**図 -19** のように書けばよい．

このようにして完成したプログラムは文献 6）に C0toHIR.java として置いてある．

●C0コンパイラのコンパイラ・ドライバ

コンパイラ・ドライバは JavaCC の場合とほとんど同じで，呼び出すメソッド名などが違うだけである．そのプログラムは文献 6）に C0Driver.java として置いてある．ただし，C0Driver.java は JavaCC を使った場合の同名のものとは違うから注意して欲しい．

以上の C0toHIR.java，C0Driver.java と，C0Parser.notavacc から notavacc によって生成される C0Parser.java を c0front2 パッケージに入れることによって C0 コンパイラのフロントエンドができあがる．これを使っても，「C0 コンパイラによる実験」に述べたことをやってみることができる．

おわりに

前回と今回で，簡単な言語を使って，コンパイラのフロントエンドの作り方を説明した．COINS を使えば，フロントエンドを作るだけでその言語のコンパイラが得られる．

次回は，バックエンドの概要を説明し，次々回に，あるマシンのマシン記述ファイルを作るだけで，そのマシンのコードを生成するコンパイラが得られることを，簡単なマシンを使って説明する．それ以降の回ではさらに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://javacc.dev.java.net/
2）五月女健治：JavaCC コンパイラ・コンパイラ for Java，テクノプレス (2003).
3）http://www.notava.org/notavacc/
4）佐々政孝：プログラミング言語処理系，岩波書店 (1989).
5）http://www.coins-project.org/COINSdoc/
6）http://www.coins-project.org/IPSJ-mitisirube/
7）中田育男：コンパイラ，産業図書 (1981).
8）中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
9）小藤哲彦，河野健二，竹内郁雄：¬＜＞＜∪∪（notavaCC），オブジェクト指向抽象構文木を生成するコンパイラ・コンパイラ，情報処理学会論文誌，Vol.44, No.SIG13(PRO18), pp.84-99, (Oct. 2003)
10）小藤哲彦，竹内郁雄：曖昧な文法を扱うコンパイラ・コンパイラ，情報処理学会論文誌 , Vol.45, No.SIG12(PRO23), pp.39-51, (Nov. 2004)

（平成 18 年 4 月 10 日受付）

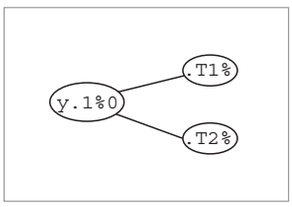

図 -18　関数 func の干渉グラフ

```
0: x = ...
1: w = ...
2: ..= w ...
3: ..= x ...
```

図 -19　プログラム例

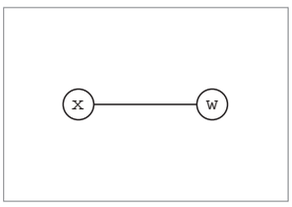

図 -20　干渉グラフ

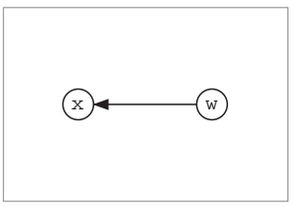

図 -21　妨害グラフ

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/