21世紀のコンパイラ道しるべ
‥COINSをベースにして

連載
1

「概 要」

中田育男（法政大学）
nakata@k.hosei.ac.jp

渡邊　坦（電気通信大学）
tan@cs.uec.ac.jp

はじめに

今回から「21 世紀のコンパイラ道しるべ‥COINS をベースにして」という題目で，連載を始める．この連載は，今までコンパイラのことをあまり知らなかった人にはコンパイラの概要を知ってもらうことを目的とし，また，すでにコンパイラを知っている人にはコンパイラの教育や研究・開発に有用な道具の情報を提供することを目的とする．読者としては，コンパイラなどの言語処理システムの基礎的な知識がある人を想定しているが，知識がない人にも概要を分かってもらえるように説明をするつもりであるので，より理解を深めるために文献 1）～ 6）なども参照していただければ幸いである．

コンパイラは，C 言語や Java 言語などで書かれたプログラムを機械語のプログラムに翻訳するソフトウェアであり，ソフトウェアの最も基礎的なものとして重要である．翻訳される前のプログラムはソースプログラム，翻訳結果として生成されるプログラムは目的プログラムあるいは目的コードと呼ばれる．

目的コードとしては，実在のマシンの機械語コードでなく，その言語に適した仮想マシンを考えて，その仮想マシンの機械語コードを生成するコンパイラもある．その場合は，目的コードは仮想マシンのインタプリタによって解釈実行される．そのインタプリタを仮想マシンと呼ぶこともある．

実在のマシンの機械語を目的コードとして生成するコンパイラでは，そのマシンの機能を最大限に活かすような効率の良い目的コードとすることが重要である．より効率の良いものとすることは「最適化」と呼ばれる（文字通り「最」適化することは一般には難しいので，「適化」と言うべきかもしれないが）．Fortran 言語とそのコンパイラを最初（1954-1957）に作った人は，熟練のプログラマがアセンブラ（機械語）で書くプログラムに負けない性能の目的コードを生成しようと最適化に心血を注いだ．それ以来，コンパイラの最も重要な技術として最適化の研究と開発が現在まで脈々と続けられている．

コンパイラの中で，字句解析と構文解析と呼ばれる部分の理論は 1960 年代から 1970 年代の初めにかけて盛んに研究され，このためのプログラムは自動生成することが可能になった．一時は，そのまま拡張すれば，コンパイラ全体を自動生成するコンパイラ・コンパイラを作成することも可能ではないか，と期待されたこともある．しかし，字句解析と構文解析以外の部分を自動生成するのは容易ではなく，最近ではコンパイラ・コンパイラという言葉は使われなくなっている．

コンパイラの中で，目的コードを生成する部分（コード生成系とも呼ばれる）は，マシンの（命令セット）アーキテクチャごとに考える必要があり，自動化は困難であるが，アーキテクチャの記述から半自動的にコード生成系を生成する研究は 1980 年から 1990 年代にかけて行われ，実用化されるようになってきた．そのようなシステムはリターゲッタブル・コード生成系とかコード生成系生成系などと呼ばれる．

コンパイラについては以上のように研究・開発が積み重ねられているが，それらがいつでも簡単に利用できるというものではないから，自分でコンパイラを作ろうとすることは，あまり容易ではない．教育用のコンパイラとして，簡単な言語を対象とし，その言語に適した仮想マシンのコードを生成するようなものを作成するのは比較的容易である．しかし，実際のマシンの機械語のコードを生成するコンパイラや，いろいろな最適化を施したコードを生成するコンパイラを開発するのは容易ではない．それを助けるためには，そのようなコンパイラを構成するための部品群を完備したシステムを整備して，い

用語の定義がされているところに下線を付す．

つでも簡単に利用できるようにしておく必要がある．このようなシステムは，いろいろなコンパイラの開発に共通に利用できるので，コンパイラ・インフラストラクチャと呼ばれる．

本連載で扱う COINS（COmpiler INfraStructure）は，コンパイラの技術を集積し，新しいコンパイラの研究・開発を容易にすることを目的として，2000 年度から 5 年間の文部科学省科学技術振興調整費による研究プロジェクトとして開発し，現在は COINS コンパイラ・インフラストラクチャ協会として保守・改良を続けているものである．その詳細は Web のページ[20]で見ることができる．

コンパイラ・インフラストラクチャとしてすでに多くの実績があるものとしては，GCC[25] がある．しかし，複雑な GCC を理解して新しいコンパイラを開発するのは必ずしも容易ではない．このため，我々は，COINS を GCC よりもコンパイラの開発がしやすいシステムとすることを心がけた．大きな違いは，GCC が C 言語で記述されているのに対して，COINS はすべて Java 言語（以下単に Java と略す）で記述されていることである．システムを改良したり新しい機能を追加したりすることは，Java の方がずっと容易である．COINS の開発は，10 カ所以上の離れた場所で同時に行われ，開発用のマシンも環境もばらばらでありながら，約 23 万行のシステムが大した問題もなく稼働できたのは，Java であったからこそだと実感している．Java を使ったことによりコンパイル時間が長くなったことは欠点であるが，最近のハードウェアの進歩を考えれば，それほど障害にはならないと考える．

もう 1 つの違いは，COINS は中間表現として，機械語に近い低水準中間表現だけでなく，高級言語に近い高水準中間表現を最初から設計の中心に置いたおかげで，入力言語に近いレベルでの処理が行いやすいことである（GCC では最近になってそのような構成が採り入れられた）．

この連載では，この COINS を使って実用的なコンパイラが比較的容易に開発できることを解説していくが，紙面の都合もあるので，簡単な例で示すことにする．最初に簡単な言語のフロントエンド（コンパイラの先頭部分）[☆1] の作り方と，簡単なマシンのバックエンド（コンパイラの後尾部分）[☆1] の作り方を説明し，その後で各種の最適化について説明する．なお，フロントエンドのうち字句解析と構文解析の部分については，自動化が可能であり，そのためのソフトウェアがいろいろ開発されているので，それらを使うことにする（その使い方の説明もする）．連載の最初から 4 回目まででこれらの説明をした後で，いろいろな最適化の方法やその使い方の説明をする予定である．

第 1 回目は，一般的なコンパイラの構造と，COINS のインフラストラクチャの構造を説明し，次回以降で作成するコンパイラのソース言語である C0 言語と，その構文解析プログラムの 1 つの作り方を説明する．

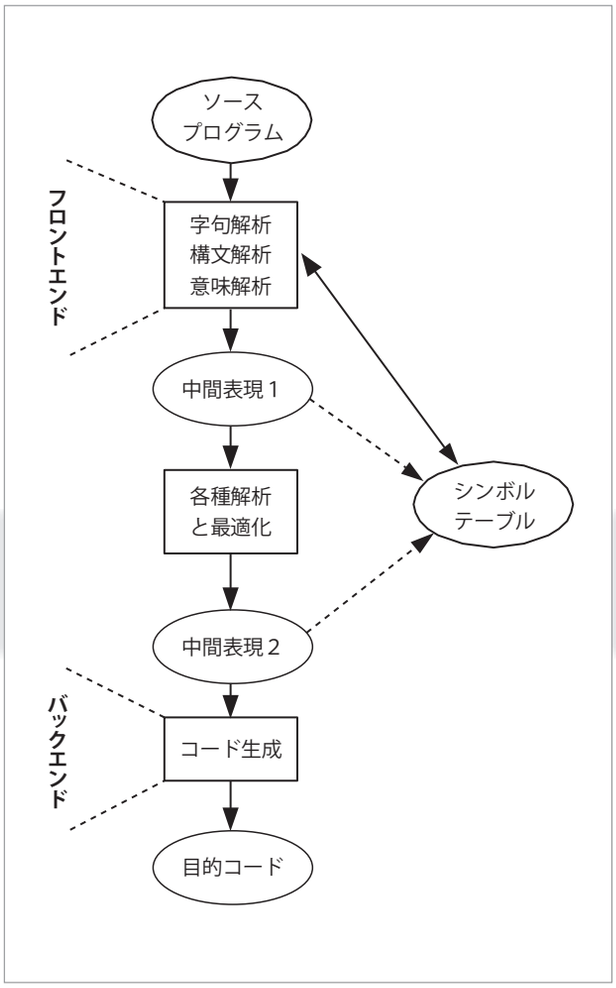

図 -1 コンパイラの一般的な構造

コンパイラの一般的な構造

コンパイラは一般に**図 -1** のような構造をしている．以下の各節で，図の各部（モジュール）の説明をする．図には書かれていないが，コンパイラ全体の制御をするコンパイラ・ドライバと呼ばれるものも必要である．コンパイラ・ドライバは，コンパイラの各モジュールを次々に呼び出したり，コンパイラを呼び出すコマンドからソースファイル名や各種のオプション情報を取り込んで，それを各モジュールに伝えたりする役目を担う．

● 字句解析

字句解析は，ソースプログラムから，意味のある最小

☆1 これらの用語のより正確な意味は以下の章で説明する．

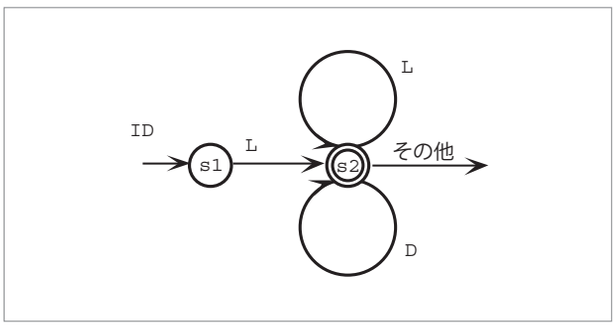

図 -2 ID を切り出す状態遷移図

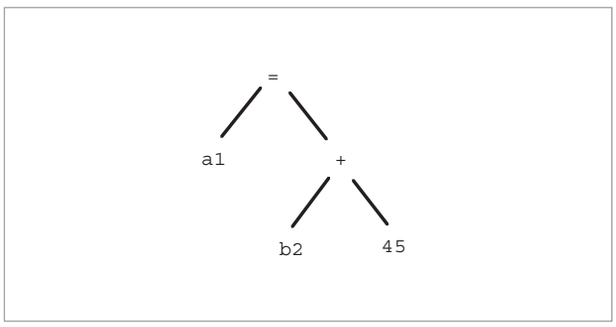

図 -3 a1 = b2 + 45; の抽象構文木

単位を切り出していくことである．

たとえば，C 言語のソースプログラムの中にある

```
int a1, b2;
a1 = b2 + 45;
```

から，'i'，'n'，'t' の 3 文字で構成される 'int' という予約語を切り出したり，'a1' という変数名を切り出したり，';' や '+' が 1 文字からなる区切り記号や演算記号であることを認識したりするのが字句解析である．これらの予約語，変数名，演算記号などが意味のある最小単位であり，これらは字句とかトークンと呼ばれる．字句解析のプログラムは lexer，scanner，lexical analyzer などと呼ばれる．

それらの字句の形は正規表現によって定義することができる．正規表現は，文字に対して，連結，選択（'|' で表現する），繰り返し（'*' で 0 回以上の繰り返しを表す）を使って構成される表現である．たとえば，

```
L: 'a'|'b'|'c'      L は a または b または c の形
D: '1'|'2'          D は 1 または 2 の形
ID: L(L|D)*         ID は先頭が L の形，その後に L
                    または D の形のものの 0 回以
                    上の繰り返し
```

という正規表現で定義される ID の形をしているものとしては，

```
c, a1, b2, c2b, aabb12, a1b2c3
```

などがある．ある文字列がこの ID の形をしているかどうかを判定する有限オートマトンから得られる状態遷移図は**図 -2** のようになる．

図 -2 は次のように解釈される．ID を読み始める状態は s1 である．状態 s1 で L（a または b または c）を読んだら状態 s2 に遷移する．状態 s2 で L を読んだら状態 s2 に遷移する．すなわち，状態 s2 にとどまる．D（1 または 2）を読んでも同様である．状態 s2 で L，D 以外のものを読んだら，その直前までに読んだものが ID であると認識できる．

正規表現から有限オートマトンを生成する方法に従って字句解析のプログラム（上記の状態遷移図のかたちのもの）が自動的に生成できることはよく知られている [1) ~ 6)]．自動生成のソフトウェアとしては，lex[7), 8)]，flex（lex の GNU 版 [9)]），JFlex（flex の Java 版 [10)]）などがある．また，次の節で述べる構文解析プログラム生成系の中には，字句解析プログラム生成系を含んでいるものもある（sableCC，JavaCC，notavacc）．

字句解析のプログラムはそれらの生成系を使って作ればよい．正規表現から有限オートマトンが生成される原理を知らなくても，それらの生成系で規定している正規表現の書き方さえ分かれば字句解析のプログラムを得ることができる．

● 構文解析

構文解析はプログラム言語の文法（構文規則）に従ってソースプログラム（字句の列）を解析し，抽象構文木や中間表現に変換することである．

たとえば，前節の a1 = b2 + 45; からは**図 -3** のような抽象構文木が作られる．

構文解析のプログラムは parser，syntax analyzer などと呼ばれる．

構文解析の方法には

(1) 演算子順位構文解析
(2) LL 構文解析
(3) LR 構文解析

の 3 つの方法が知られている [1) ~ 6)]．(1) は最初に知られた方法で，算術式などの式の構文解析を効率良く行うことができるが，適用範囲が狭いので最近はあまり使われていない．(2)，(3) については，文法を与えると構文解析のプログラムを自動生成するシステムがいくつかある．(2) については JavaCC[11), 12)]，(3) については yacc[7), 8)]，bison（yacc の GNU 版 [13)]），jay（yacc の Java 版 [14)]），sableCC（構文木と Visitor を生成する [15)]），notavacc（構文木を生成する [16), 17)]），kmyacc（C, Java, JavaScript, Perl 対応 [18)]）などがある．

構文解析のプログラムはそれらの生成系を使って作ればよい．それが生成される原理を知らなくても，それら

の生成系で規定している文法の書き方さえ分かれば構文解析のプログラムを得ることができる．ただし，それらの生成系からメッセージが出されたときに，それに対処できるためには，ある程度原理を知っている必要がある．その例については後で実例を扱ったときに説明する．

●意味解析

意味解析はソースプログラムに対して意味付けをすることである．たとえば，ソースプログラムに

```
x + y
```

という式があったとき，xとyがどのように宣言されていたかを調べて，ともに整数型であればこの式は整数型の加算を表し，ともに実数型であれば実数型の加算を表す，というような解析をするのが意味解析である．意味解析の主たる仕事は，宣言された情報をもとに，それらが使われている場所での意味を解析することである．

変数などの宣言された情報は，通常シンボルテーブルと呼ばれる表にまとめられる．変数などが式の中に現れたら，シンボルテーブルを探索（サーチ）して，該当する宣言情報をとり出して上記のような解析をすることになる．

●中間表現

中間表現は，構文解析した結果から最後に機械語の目的コードなどを生成するまでの間，対象となっているプログラムをコンパイラの内部で表現するものであり，計算機処理のしやすい論理的な表現であることが望まれる．

図-1では，中間表現1，中間表現2となっているが，実際のコンパイラでは，中間表現の形式は1つしかない場合もあるし，2つより多い場合もある．その形式としては，抽象構文木のようなソースプログラムに近い高水準の形式のものから，機械語の形式に近い低水準の形式のものまである．また，ある中間表現に対して最適化などの変換をした結果をまた同じ中間表現形式にする場合もある．ソースプログラムから最初の中間表現までの変換を行う部分は，フロントエンドと呼ばれる．

中間表現形式は一般には公表されないことが多いので，どのような表現が使われているかは明確ではないが，公表されている例にはGCCの中間表現RTL[19]がある．

高水準の中間表現では，ソースプログラムの各文（ステートメント）は木構造の形で表現され，その末端には定数や変数に相当するものが置かれる．それらの変数の宣言情報などはシンボルテーブルにまとめられ，木構造上での変数などはシンボルテーブルへの参照のかたちをとるのが普通である．

中間表現が特定のプログラム言語や特定のマシン向きに作られていれば，その特定のものに対しては都合が良いかもしれない．しかし，そうすると，別の言語やマシンに対しては新たに作らなければならなくなる．たとえば，いろいろな言語に対して共通に使える中間表現であれば，新たなコンパイラはフロントエンドを作るだけでできるので，以前からそのような試みはいろいろ行われている．低水準の中間表現は機械語の形式に近いものになるので，言語に対する独立性は高くなり，どの言語に対しても使える．しかし，中間表現を低水準のものだけにすれば，ソース言語から中間表現までの距離は長くなるからフロントエンドの仕事量が増える．したがって，フロントエンドの立場からは，できるだけ高水準であることが望ましい．

●各種解析と最適化

コンパイラは，できるだけ効率の良い目的コードを生成するために各種の解析をし，その結果を使って最適化を行う．

最適化によってどれくらいの違いがあるかを簡単な例で説明する．例題としては以下の行列の掛け算のプログラムを取り上げる．

```
for (i = 0; i < N; i++)
      for (j = 0; j < N; j++) {
            s = 0;
            for (k = 0; k < N; k++)
                  s += a[i][k] * b[k][j];
            c[i][j] = s;
      }
```

このプログラムの実行時間は最内側のkのループのそれに左右されるから，その部分を最適化するのが効果が大きい．その部分の目的コードの命令数を調べてみると，あまり最適化をしないコンパイラでは，20から30くらいであるが，十分な最適化をすれば，SPARCのようなRISC（Reduced Instruction Set Computer）で8，CISC（Complex Instruction Set Computer）のメインフレームマシンでは5になる（文献5）の233ページ）．この場合の最適化は大きな効果がある．

このような最適化は，ループの中で毎回実行しなくてもよい命令はループの外に出す，同じ計算は省略する，ハードウェアの持つレジスタをうまく使い回す，などの多くの技術を適用することによって可能となるが，そのためにはソースプログラムの各変数の値がどこで定義（代入）されどこで使われるかを詳細に解析する必要がある．

その方法としては，従来はデータの流れの解析と呼ばれる方法が使われていたが，最近はSSA最適化と呼ばれる方法がよく使われるようになっている．

従来のデータの流れによる方法は，プログラムを制御

フローグラフで表現し，各変数について変数の値が制御フローグラフのどこで定義（代入）され，それがどこまで流れて使用されるか，といった解析をもとに最適化する方法である．

SSA最適化は，プログラム（中間表現）をSSA形式（Static Single Assignment form: 静的単一代入形式）で表現して，その上で最適化する方法である．その形式は，すべての変数の使用に対して，その値を定義（代入）している場所は1カ所しかないように変数の名前替えをしたものである．従来の方法では変数の使用に対してその値を定義している場所が複数個あり得るから，それに比べればいろいろな最適化が見通し良くできるようになる．変数の名前替えは，変数名に対して，定義されている場所ごとにサフィックスをつけることで行われる．たとえば，

```
a = x + y;
a = a + b;
z = x + y;
```

をSSA形式に変換すると

```
a1 = x0 + y0;
a2 = a1 + b0;
z1 = x0 + y0;
```

となる．最初の形式では1行目で定義されたaの値が2行目で変わってしまうが，SSA形式ではそのようなことはないので，3行目の x_0 + y_0 を a_1 で置き換えることができる．

最適化の技術は，アーキテクチャの進歩にも合わせて進歩している．新しいアーキテクチャのマシンが作られてもコンパイラの最適化によってそれを活かさなければ意味がないからである．

たとえば，スーパーコンピュータとして，1命令で長いベクトルの演算をしてしまうベクトル計算機が作られても，プログラムのどの部分がそのベクトル命令で実行できるかを解析して，そのような目的コードを生成する最適化コンパイラ（ベクトル化コンパイラとも呼ばれる）がなければ意味がない．スーパースカラマシンのように複数個の命令を並列に実行できるようなマシンに対しては，でき上がった目的コードの命令を並べ替えて並列に実行できる割合を増やす必要がある．そのような最適化は命令スケジューリングと呼ばれる．

●目的コードの生成

コンパイラの最後の仕事は，対象マシン（ターゲットマシン）の機械語のコードを生成することである．

その一般的な方法は，中間表現の木がどんな形であったらどんなコードを生成すればよいかという対応関係を記述しておいて，ソースプログラムから得られた中間表現とのパターンマッチングによってコードを生成する方法である．しかし，ある中間表現にマッチするパターンは，一般には複数個あり得る．その中で最も良い組合せを選ぶ方法としては，各コードにそのコードのコスト情報を付けておき，そのコストの和が最小になるようなコード列を選ぶ方法がよく使われる．コストとして命令の長さ（バイト数）を与えておけば，短いコードが得られるし，命令の実行時間（サイクル）を与えておけば，速いコードが得られる．

上記のパターンマッチングのプログラムがマシンに依存しないかたちで書かれていて，あるマシンに対しては，中間表現とそのマシンの命令との対応関係を与えるだけでそのマシンのコード生成ができるようなコード生成系は，リターゲッタブル・コード生成系と呼ばれる．

中間表現に対して命令コードを生成するのは，複数の可能性の中から命令を選ぶことになるので，命令選択とも呼ばれる．命令選択の後に通常行われるのはレジスタ割り付けである．命令選択では，通常，レジスタの数は無限にあると考えて命令コードを生成しており，命令コードのレジスタ部は仮想レジスタ名になっている．レジスタ割り付けでは，各仮想レジスタ名に実レジスタを割り付ける．その場合，ハードウェアの持つレジスタを最大限に利用することによって効率の良いコードとなるようにするのであるが，その出来の良し悪しが生成されたコードの性能の良し悪しを大きく左右する．レジスタ割り付けのアルゴリズムとしてよく使われるものにグラフ彩色アルゴリズム（文献5）の396ページ）と呼ばれるものがある．

最後に，生成されたコードがアセンブラコードとして出力される．その際に，あるいはその前に，ちょっとした無駄なコードを修正することが行われることもある．これは覗き穴式最適化（peephole optimization）と呼ばれる．

スーパースカラ・マシンのように，命令レベルでの並列実行の機能を持ったマシンに対しては，その並列性を活かすように命令を並べ替える，いわゆる命令スケジューリングも重要な最適化の1つである．命令スケジューリングをいつするかも問題である．レジスタ割り付けの前にすると，並列性はよく抽出されるかも知れないが，レジスタを多く必要とすることになって，レジスタ割り付けで損をするかもしれないし，レジスタ割り付け後にすると，レジスタ間の依存関係によって命令の並べ替えが制限されてしまうかもしれないからである．レジスタ割り付けと命令スケジューリングを同時に行うことも考えられる．

命令スケジューリングをループの繰り返しにまたがって行うのはソフトウェア・パイプライニングと呼ばれる．ループで多く時間を費やすようなプログラムに対しては，

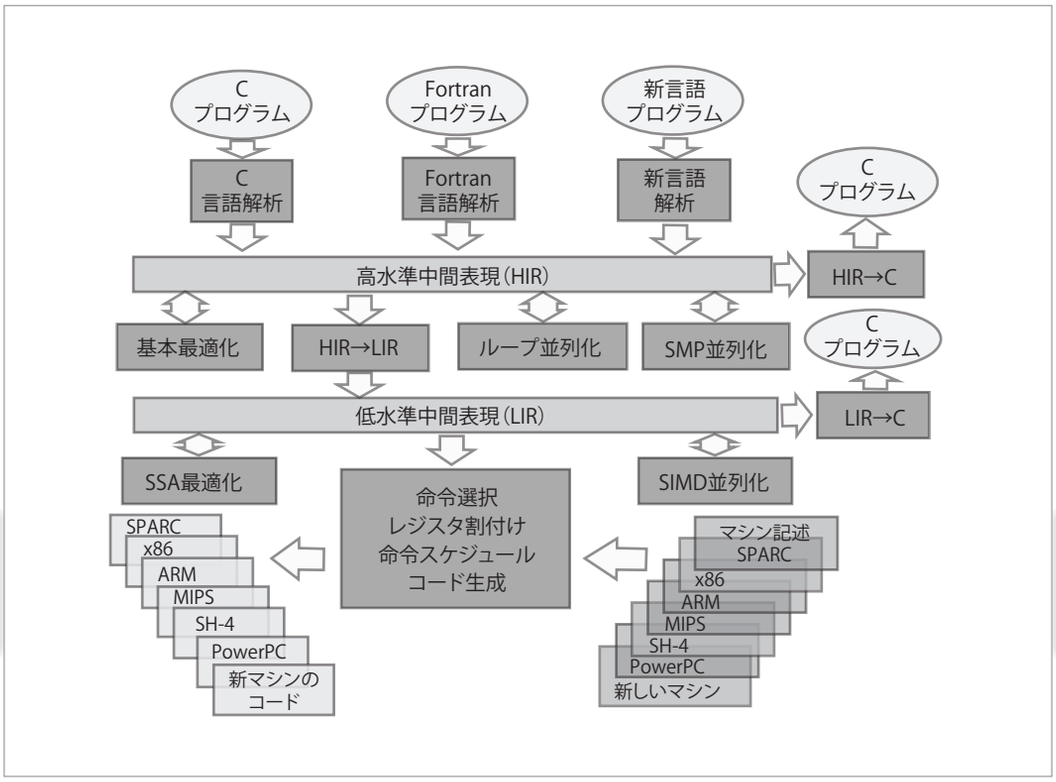

図-4　COINS の構成

その効果は大きい．

COINSの構造

COINS は，今まで述べたような構造を持ったコンパイラが容易に作成できるようなインフラストラクチャとして開発された[20]．その構造は**図-4**のようになっている．COINS は，次のような特徴を持っている．

(1) すべて Java で書かれている．

(2) 2 段階の中間表現（ソース言語のレベルに近い高水準中間表現 HIR : High-level Intermediate Representation と機械語のレベルに近い低水準中間表現 LIR : Low-level Intermediate Representation）を持ち，ソースレベルの処理もマシンレベルの処理も容易である．

(3) 新しいコンパイラの構成を容易にするコンパイラ・ドライバを備えている．

COINS の標準ドライバは C コンパイラのドライバのかたちをしているが，多くのオプション指定機能を持ち，それらのオプションを指定することで，種々のコンパイラとして機能する．

たとえば，ターゲットマシンを "x86" と指定し，実行環境を Windows マシンの "cygwin" と指定することで，cygwin で実行されるコードを生成するコンパイラとなる．また，いろいろな最適化機能とその適用順序を指定することによって，種々の最適化の実験をするコンパイラとすることもできる．

C 言語以外の言語のコンパイラは，この標準ドライバのサブクラスを作ることによって容易に作成できる．たとえば，Fortran コンパイラのドライバは標準ドライバのサブクラスであり，フロントエンドを呼び出す部分をオーバライドしているだけである．それだけで，標準ドライバと同じオプション指定機能を持ったドライバとなっている．

(4) リターゲッタブル・コード生成系を持つ．

(5) 優れたバックエンド・モジュールを持つ．

レジスタ割り付けなどにすぐれ，x86 のようなレジスタの少ないマシンに対しては，高度な最適化を適用しなくても，gcc-O2 に近い性能を出している．モジュール構成が明確で，コンパイラ自身のトレース機能も充実しており，機能拡張がしやすい．

(6) SSA 最適化モジュールを完備している．

3 種類の LIR-to-SSA 変換（minimal，semi-pruned，pruned の形式のいずれか），2 種類の SSA-to-LIR 逆変換（Briggs 法，Sreedhar 法）を備えている．SSA 形式に対しては，基本的な最適化約 10 種類と，新たに開発した高度な最適化の機能も備え，その他に最適化を補助する変換機能もある．それらの任意の組合せをコンパイルオプションで指定可能である．

(7) SIMD 並列化の機能を持つ．

インテル x86 の MMX/ SSE/ SSE2/ Enhanced 3D Now! の命令や PowerPC の AltiVec の命令，SPARC の VIS の命令などのような，マルチメディア対応の複雑な SIMD 命令を目的コードとして生成する．まだ完全な実用化のレベルには達していないが，x86/ SSE2 については，いくつかの例題について成功しており，gcc の 5 倍以上の性能が得られているものもある．

なお，それに関連して，各マシンの命令語の意味を厳密に記述したものがある（上記の機種で約 6 万 5 千行．文献 21）の 7.2.2 参照）．また，SIMD 命令の適用可能性をチェックするために，演算に実際に必要なデータサイズを推論するアルゴリズムを開発した [22]．

(8) ループ並列化，粗粒度タスク並列化の機能を持つ．

ループ内の各変数，配列要素のループ繰り返しを考慮したデータフロー解析，ループ並列化変換（帰納変数の変換，private 化，reduction 検出），ループの do-all 型並列化可能性判定，プログラムの粗粒度タスクへの分割，粗粒度タスク実行の動的スケジューラ，などの機能を持ち，これらの結果を OpenMP に対する指示文として付加した C プログラムの形で出力する機能がある．

それらの機能を使った粗粒度並列化コンパイラとして CoCo（COINS based Coarse-grain Parallelizing Compiler）がある．

(9) コンパイル過程の可視化ツールがある．

ソースプログラム，HIR，HIR の制御フローグラフ，LIR，LIR の制御フローグラフなどの対応をグラフィカルに表示するツール CoVis や，コード生成過程のトレース結果をブラウザで表示するものがある．

コンパイラ作成の例題用C0言語の文法

これから，簡単な言語のコンパイラを作ってみよう．その言語としては文献 4）にある TinyC をもとにして，それに関数呼び出しとローカル変数の機能を追加したものを使う．ここではその言語を C0 言語と呼ぶことにする．

C0 言語の文法は**図 -5** の通りである．図の中の「//」以降はコメントである．ここで文法の書き方を簡単に説明する．文法は

A → α

という形の規則からなる．これは，A は α の形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ので，生成規則と呼ばれる．このような生成規則からなる文法は文脈自由文法と呼ばれる．矢印→の左側 A は生成規則の左辺，右側 α は右辺と呼ばれる．左辺は，1 つの記号からなり，その記号は非終端記号と呼ばれる．右辺は，非終端記号やトークン（終端記号とも呼ばれる，左辺には現れない記号）を並べたかたちである．文法の書き方の習慣として，非終端記号には A や B などのアルファベット文字を使い，記号列（記号を並べたもの）には α や β などのギリシャ文字を使う．

ここでは右辺に，先に字句解析の説明のところで述べた正規表現も使っている．たとえば，図 -5 の 1 行目は「Program は，最初に ExternalDeclaration があり，その後に ExternalDeclaration の 0 回以上の繰り返しがあるかたちをしている」と読むことができる．これは，「Program は，ExternalDeclaration の 1 回以上の繰り返しがあるかたちをしている」と読むこともできる．その ExternalDeclaration のかたちは 2 行目以降で定義されている．

なお，以下の文法では右辺の正規表現の中で

(α)?

という形も使っている．これは α が 0 回または 1 回あることを表す．

この言語はほぼ C 言語の小さなサブセットであるから，この文法で定義されるステートメントなどの意味は，C 言語を知っている人には簡単に分かると思われる．14 行目はサブプログラム呼び出しのステートメントである．

なお，プログラム中の空白記号，改行記号，タブ記号などは，トークンとトークンの区切りを示す意味はあるが，その記号自体は意味のないものとして無視する．また "/*" から "*/" まではコメントとして無視する．

C0コンパイラのフロントエンド

COINS を使って新しい言語のコンパイラを作る場合はそのフロントエンドだけを作ればよい．フロントエンドではまず字句解析と構文解析をする必要があるが，それにはそれらの説明のところで述べた道具を使えばよい．

LL 構文解析と LR 構文解析では，後者の方が適用範囲が広いということで使われることが多かったが，最近の JavaCC などはその欠点を補うための機能が付加されていて，実用性が高くなっている．また，LL 構文解析は下向き構文解析であり，すでに解析した部分の情報をその後の構文解析の際に利用することが，上向き構文解析である LR 構文解析よりは少しやりやすい．したがって，構文解析しながら直ちに中間表現（COINS の場合は HIR）に変換してしまう方式をとる場合は LL 構文解析を使うことが考えられる．

しかし，それが適しているのは，比較的文法が簡単で

```
 1: Program → ExternalDeclaration ( ExternalDeclaration )*      // プログラムは外部宣言の列である
 2: ExternalDeclaration → VariableDeclaration | SubprogramDeclaration | SubprogramDefinition
                        // 外部宣言は変数宣言かサブプログラム型宣言かサブプログラム定義である
 3: VariableDeclaration → TypeId VarDecl ("," VarDecl )* ";"          // 変数宣言の例：int a, b, c;
 4: SubprogramDeclaration → SubprogramDeclarator ";"        // 例：int func2(int x, int y);
 5: SubprogramDefinition → SubprogramDeclarator "{" ( VariableDeclaration )* ( Statement )* "}"
                        // サブプログラム定義はサブプログラム型宣言の後に本体がついている
 6: TypeId → "int" | "void"                        // 型は int かまたは void
 7: VarDecl → ID ( "[" NUM "]" )?                  //VarDecl の例 1： index，例 2：array[100]
 8: SubprogramDeclarator → TypeId ID "(" ( ParamDecl ("," ParamDecl )* )? ")"        // 例：int func(int x)
 9: ParamDecl → TypeId ID ( "[" "]" )?           //ParamDecl の例 1：int x，例 2：int y[]
10: Statement → Variable "=" Expression ";"           // 文には代入文がある．その他に以下の文がある
11:     | "if" "(" ConditionalExpr ")" Statement ( "else" Statement)?        // if 文，"else" がついてもよい
12:     | "while" "(" ConditionalExpr ")" Statement          // while 文
13:     | "{" ( Statement )* "}"                  // ブロック文（文の列を "{" "}" で囲む）
14:     | CallExpr ";"                //call 文（サブプログラムの呼び出し）
15:     | "return" ( Expression )? ";"                //return 文，関数の場合は戻り値を指定する
16: ConditionalExpr → Expression ComparisonOperator Expression                // 条件式は比較式だけ
17: ComparisonOperator → "==" | "!=" | ">" | ">=" | "<" | "<="      // 比較演算子
18: Expression → Expression "+" Term            // 式は Term からなる．"+"，"-" の優先順位は一番低い
19:     | Expression "-" Term
20:     | Term
21: Term → Term "*" Factor          //term は Factor からなる．"*"，"/" の優先順位は "+"，"-" より高い
22:     | Term "/" Factor
23:     | Factor
24: Factor → "(" Expression ")"
25:     | "+" Factor
26:     | "-" Factor              // 単項演算子は優先順位が一番高い
27:     | CallExpr                // 関数呼び出し（関数サブプログラムの呼び出し）
28:     | ID ( "[" Expression "]" )?              // 例 1：index，例 2：array[index+1]
29:     | NUM              // 整数
30: Variable → ID ( "[" Expression "]" )?
31: CallExpr → ID "(" ( Expression ( "," Expression )* )? ")"                // 引数がない場合もある
32:
33: ID → Letter ( Letter | Digit )*        // 名前は先頭が Letter で，その後に Letter か Digit が 0 個以上
34: NUM → Digit ( Digit )*             // 整数は数字の列
```

図 -5　C0 言語の文法

ある場合である．文法が簡単であれば，この方式でフロントエンド全体を簡潔に記述することができる．文法が複雑になると構文解析しながら直ちに中間表現に変換する部分を簡単には記述できない場合が生ずる．その場合は，構文解析した結果をいったん抽象構文木に変換し，次にその抽象構文木をスキャンしながら目的とする中間表現に変換した方がよい．先に述べた LR 構文解析の道具の中には標準で抽象構文木が作成されるものがある．

以上のことから，ここではLL構文解析をしながら直ちに中間表現に変換していくフロントエンドと，LR構文解析をしながら抽象構文木を作成し，次にその抽象構文木から中間表現に変換していくフロントエンドの両方を作ってみることにする．ただし，後者は次回に説明する．LL構文解析のプログラムは，上に述べたような理由でJavaCCを使って作ることにする．

●JavaCCを使ったC0フロントエンド

最初に作るフロントエンドは，構文解析と意味解析との対応が比較的分かりやすく書けるLL構文解析の方法を使うこととし，その道具としてJavaCCを使うことにする．

＜LL構文解析とは＞

LL構文解析プログラムとして通常使われるのは再帰的下向き構文解析プログラム（Recursive Descent Parser）である．そのプログラムでは，各非終端記号に1つの関数プログラム（一般には再帰的（recursive））が対応する．たとえば，

A → B D

という生成規則（構文規則）で，BもDも非終端記号である場合には，

```
void A() { B(); D(); }
```

というプログラムが対応する．これがAの構文解析をするプログラムである．その中身は，Bの構文解析をし，続いてDの構文解析をするプログラムである．

ところで，

A → B | D

に対応する構文解析プログラムはどうなるであろうか．それは，次のような形である．

```
void A() { if (nextToken ∈ First(B) ) B();
           else if (nextToken ∈ First(D) ) D();
           else error(); }
```

ここで，nextTokenは，現在構文解析中のソースプログラム上の「次のトークン」であり，First(B)は，Bの先頭に出てくる可能性のあるトークンの集合である．したがって上のプログラムは，「次のトークンがBの先頭にあり得るものであれば，それ以降にはBの形をしたプログラム片があるはずだから，そのBの構文解析をし，そうでなくて，Dの先頭にあり得るものであればDの構文解析をする」と読むことができる．

このプログラムでは，次の1つのトークンだけを調べて，それによってそれ以降がBであるかDであるかを判定している．このような構文解析はLL(1)構文解析と呼ばれる．LL(1)構文解析がうまくいく（それでちゃんと構文解析できる）ような文法のことをLL(1)文法という．1つだけでなく，次と，その次の2つのトークンを調べて構文解析を進めるのはLL(2)構文解析と呼ばれる．それで構文解析できるような文法のことをLL(2)文法という．調べるトークンは先読み（look ahead）トークンと呼ばれる．

＜C0フロントエンドのJavaCC記述（その1：字句規則）＞

図-5のC0言語の文法記述のうち，33，34行は字句規則である．また，"if"などの予約語は字句規則に書くのが普通である．そのほかに字句規則として書かなければならないのは，C0言語の説明の最後にある「無視する記号」である．

コメント以外の字句規則の書き方は簡単である．

```
SKIP : { <SPACE: " "|"\t"|"\r"|"\n"> }
TOKEN : { <INT: "int"> | <VOID: "void">
          | <IF: "if"> | <ELSE: "else">
          | <WHILE: "while">
          | <RETURN: "return"> }
TOKEN : { <ID: <LETTER>(<LETTER>|<DIGIT>)*>
          | <NUM: (<DIGIT>)+>
          | <#LETTER: ["a"-"z","A"-"Z"]>
          | <#DIGIT: ["0"-"9"]> }
```

ここで，SKIPは無視する記号であり，TOKENが構文解析プログラムに渡されるトークンである．"\r"はMacintoshやWindowsのファイルの行末記号またはその一部として使われる記号である．"<"と":"の間にあるのがトークンの名前である．#で始まる名前は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．"(<DIGIT>)+"は<DIGIT>の1回以上の繰り返しを表し，"(<LETTER> | <DIGIT>)*"は，<LETTER>または<DIGIT>の0回以上の繰り返しを表す．["0"-"9"]は"0"から"9"までの文字集合の中の1文字を表す．

なお，ここで注意しなければならないのは予約語とIDとの記述の順序である．字句解析プログラム生成系では一般に「先に書いたものが優先される」という原則がある（構文解析プログラム生成系でも同様の原則がある）．たとえば"int"という文字列はINTトークンにマッチするし，IDトークンにもマッチするが，この原則によってINTトークンと認識される．この記述の順序を逆にすると，予約語として認識されるものがなくなる．

ところで，"/*"で始まりそれ以後の最初の"*/"で終わるコメントの字句規則を正規表現で書くのは簡単ではない．「それ以後の最初の"*/"」を表現するのが難しいからである（文献3）の69ページ）．

正規表現から有限オートマトンを作り，それから字句解析のプログラムを作るときには「最長一致」を見つけるプログラムとしなければならない．その理由は，たとえばC0言語のソースプログラムを読んでいて ">=" があったときに，最初の ">" を読んだところで ">" というトークンを読んだことにしてやめてしまうと，">=" というトークンは決して認識されないことになる．そこで，">" を読んだところでは，">" というトークンを1つの候補としておいてさらに読み進める．その次が "=" でなければ，先ほどの候補に決定し，その次が "=" であれば ">=" というトークンを新たな候補とする．新たな候補が現れないと分かるまで読み続け，その時点で，今までの中で一番長い候補に決定する，というのが「最長一致」の原則である．

それに対して，「最初の "*/"」を見つけるのは「最短一致」で見つけなければならない．しかし，コメントの正規表現を簡単に書こうとして

```
"/*" (any-character)* "*/"
```

を与えると，最長一致で見つけるプログラムでは，ソースプログラムの中の最初の "/*" を見つけた後は，途中にいくつか "*/" があっても，最後にある "*/" までをコメントとしてしまう（プログラムの先頭と最後にコメントを書けば，プログラム全体がコメントと見なされてしまう）．最長一致の原則であっても「最初の "*/"」を見つけることになるような正規表現を書くことはできるが，それは大変複雑な正規表現になってしまう．

その問題を解決するために，"/*" を読んだ後は通常の正規表現とは別の状態であるとする方法がよくとられる．JavaCCでは，このコメントは以下のように記述すればよい．

```
SKIP : { "/*" : WithinComment }
<WithinComment> SKIP : { "*/" : DEFAULT }
<WithinComment> MORE : { <~[]> }
```

1行目は，"/*" を見つけた後はその "/*" を無視して WithinComment という状態に移行することを意味する．これ以後は先頭に <WithinComment> がついた行に書いてある正規表現だけに従う．2行目は WithinComment という状態で "*/" を読んだらそれを無視して DEFAULT の状態，すなわち通常の状態，に戻ることを意味する．3行目の ~[] は任意の1文字を意味する（[] は空の文字集合であり，~ はその否定（補集合））．MORE はまだトークンが完結しないことを意味する．したがって，WithinComment という状態に移行してからは，"*/" を読むまで読み続けたものが1つのトークンとなるが，それは SKIP であるから無視される．

＜C0フロントエンドのJavaCC記述（その2：構文規則）＞

JavaCCに文法を与えるとLL構文解析プログラムを生成する．JavaCCでは，与える文法をLL構文解析プログラムに近いかたちで書くことを要求する．たとえば，

A → B D

に対しては

```
void A() : {} { B() D() }
```

のかたちに書かなければならない．ここで，":" は " → " に当たると思えばよい．最初の "{}" は意味規則を書くときにメソッドのローカル変数の宣言を書く場所になる．A() の返す型は適当なものを与えることができるが，それは意味規則を書くときに決めることにして，いまは簡単にすべて void としておく．終端記号（トークン）の部分は "<" と ">" で囲ってそのまま書いておけばよい．たとえば，

9: ParamDecl → TypeId ID ("[" "]")?

に対しては

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

と書けばよい．また，たとえば

A → B | D

に対しては

```
void A() : {} { B() | D() }
```

と書けばよい．前に述べた "if (nextToken ∈ First(B)) " などはJavaCCが生成してくれるから書く必要がない．ただし，Bの先頭とDの先頭に同じようなトークンが続くと，"if (nextToken ∈ First(B)) " でどちらかを選ぶことができなくなるのは明らかであるから，そのような生成規則は少し書き換える必要がある（後で述べるように書き換えないで済む場合もある）．たとえば，

2: ExternalDeclaration → VariableDeclaration
　　| SubprogramDeclaration | SubprogramDefinition
4: SubprogramDeclaration → SubprogramDeclarator ";"
5: SubprogramDefinition → SubprogramDeclarator
　　"{" (VariableDeclaration)* (Statement)* "}"

では，SubprogramDeclaration と SubprogramDefinition の両方の先頭に SubprogramDeclarator がある．このような場合は共通の SubprogramDeclarator で括り出して，以下のようにすればよい．

ExternalDeclaration → VariableDeclaration
　| SubprogramDeclarator
　　(";" | "{" (VariableDeclaration)* (Statement)* "}")

ところで，LL構文解析には1つの問題がある．それ

は左再帰性の問題である．たとえば，

A → A B

という生成規則に対して

```
void A() : {} { A() | B() }
```

と書いたとしたら，メソッドAが呼ばれたときには，何もせずに再びAを呼ぶことになるので，Aを呼ぶことを無限に繰り返すことになる．この生成規則の右辺の先頭に左辺と同じ非終端記号があるとそうなる．そのような生成規則は左再帰性があると呼ばれる．

C0言語の文法では，以下のところに左再帰性がある．

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

このままではLL構文解析はできないから，文法を書き直す必要がある．これらの生成規則から得られるものは，

Expression → Term ("+" Term | "-" Term)*

Term → Factor ("+" Factor | "-" Factor)*

から得られるものと同じであるから，文法をそのように書き直せばよい．

それ以外の部分はC0言語の文法のままで，すべてを上記のように書き直し，そのまえに先に述べたトークンの定義を付け加えたものをJavaCCに与えてみよう（読者にも自分でやってみることをすすめる）．JavaCCのインストールの仕方や使い方は，たとえば文献12)，23)を見ていただきたい．ただし，その文法記述ファイルの先頭には以下のような行がなければならない．

```
PARSER_BEGIN(C0Parser)
  public class C0Parser{}
PARSER_END(C0Parser)
```

これは，JavaCCの生成する構文解析プログラムのクラス名とそのクラスのフィールドやメソッドなどを書く部分である．今はその名前だけを書いておけばよい．

これをJavaCCに与えてみると，Warningメッセージが4つ出る．1つ目は上記の書き直した部分で，VariableDeclarationとSubprogramDeclaratorの両方とも先頭部分が

"int" <ID>

となる場合があるから，どちらかを選ぶことができない．lookahead 3以上を考えなさい，というメッセージである．VariableDeclarationの先頭から3つ目のトークンは"," か ";" であり，SubprogramDeclaratorでは "(" であるから，3つ先読みすれば区別をすることができる．そこで，その部分を

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

とするとこのメッセージは出なくなる．これは，先読みを3つしてそれが`variableDecl()`の先頭を示すものであったら，`variableDecl()`を選ぶことを意味する．なお，C0言語の文法では非終端記号を大文字で始まるようにしてあるが，Javaのメソッドは小文字で始める習慣なので，ここでは小文字に変更している．また，長い名前は紙面の都合も考えて短いものに変更している．

2つ目のメッセージは，もとの文法の

```
10: Statement → Variable "=" Expression ";"
14:      | CallExpr ";"
```

がともにIDで始まるというメッセージである．これも最初の代入文のところに`LOOKAHEAD(2)`を付ければ解決する．

3つ目のメッセージは，もとの文法の

```
11: Statement → "if" "(" ConditionalExpr ")" Statement
                    ( "else" Statement )?
```

から生ずる問題である．この文法はそもそもあいまいな文法である．たとえば，

```
if (a > 0) if (a < 0) b = 1; else b = 2;
```

は

```
if ( a > 0 ) { if ( a < 0 ) b = 1;}
else b = 2;                                    (1)
```

とも解釈できる（構文解析できる）し，

```
if ( a > 0 ) { if ( a < 0 ) b = 1;
                 else b = 2; }                 (2)
```

とも解釈できるからである．これは"else"をどの"if"と組み合わせるかが決まらない"dangling else"の問題といわれる．このようなあいまいさがないように文法を書き直すことはできるが，それは決して読みやすい文法ではない．通常，プログラム言語の文法では，生成規則はこのままのかたちにしておいて，付帯規則であいまいさがないようにする．通常の付帯規則は，「"else"はそれに最も近い"if"で，まだどの"else"とも組み合わさっていない"if"と組み合わせる」というものである．その付帯規則に従って構文解析すると上記の(2)のかたちになる．

JavaCCでは，この生成規則の右辺に相当するものを

次のように書くとこの付帯規則に合ったかたちになる．

```
<IF> "(" conditionalExp() ")"stmt()
 ( <ELSE> stmt() | {} )
```

これは「("else" Statement)?」を「("else" Statement | ε)」のかたちにしたものである．ここでεは何もないことを意味する．εにあたる場所には{}を書かなければならない．その中には後で述べる意味規則を書くことができる．JavaCCでは(α | β)の形に対しては，次のトークンがαの先頭のトークンであればαを選んでしまうから，これで付帯規則に合うことになる．

4つ目のメッセージは，もとの文法の

27: | CallExpr
28: | ID ("[" Expression "]")?

がともにIDで始まるというメッセージである．これもCallExprのところにLOOKAHEAD(2)を付ければ解決する．

以上の修正をすれば，JavaCCに与えるファイルが完成し，その結果の構文解析プログラムを得ることができる（完成したものは文献24）にある．JavaCCに与えるファイルには".jj"というサフィックスを付ける習慣があるようなので，c0.1.jjという名前になっている）．

この後，上記のファイルに書いた構文規則に対して，意味解析をしてCOINSの高水準中間表現HIRの木のかたちを作っていく部分の記述を付け加えれば，C0コンパイラのフロントエンドができあがるが，紙面の都合でそれは次回にまわすことにする．

おわりに

今回は，連載の第1回目として，コンパイラの一般的な構造，そのようなコンパイラを作成するためのインフラストラクチャとしてのCOINSの構造を説明し，それを使って簡単なコンパイラを作ってみるための例題の言語としてC0言語をとりあげた．C0コンパイラのフロントエンドは2通りの作り方をしてみる予定であるが，今回はその1つのLL構文解析による方法で，構文解析プログラムを得るところまで説明した．次回はそのフロントエンドを完成するところまで説明する予定である．

謝辞　COINSコンパイラ・インフラストラクチャは，平成12年度から16年度までの文部科学省科学技術振興調整費の援助を受けて作成したものである．これは，COINSプロジェクトメンバならびに関係者の方々の多大な努力によって開発された．

参考文献

1）中田育男：コンパイラ，産業図書 (1981).
2）佐々政孝：プログラミング言語処理系，岩波書店 (1989).
3）中田育男：コンパイラ，オーム社 (1995).
4）渡邊　坦：コンパイラの仕組み，朝倉書店 (1998).
5）中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
6）湯浅太一：コンパイラ，昭晃堂 (2001).
7）Levine, J.R., Mason, T. and Brown, D.: lex & yacc, O'Reilly & Associates, 2nd edition (1990). 村上　列訳：lex & yacc プログラミング，アスキー出版局 (1995).
8）五月女健治：yacc/lex プログラムジェネレータ on UNIX，テクノプレス (1996).
9）http://www.gnu.org/software/flex/
10）http://jflex.de/
11）http://javacc.dev.java.net/
12）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).
13）http://www.gnu.org/software/bison/
14）http://cis.k.hosei.ac.jp/~nakata/lectureCompiler/JayJflex/
15）http://sablecc.org/
16）http://www.notava.org/notavacc/
17）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, pp.84-99 (2003).
18）http://www005.upp.so-net.ne.jp/kmori/kmyacc/
19）http://gcc.gnu.org/onlinedocs/gccint/RTL.html
20）http://www.coins-project.org/
21）http://www.coins-project.org/COINSdoc/
22）鈴木　貢，藤波順久，福岡岳穂，渡邊　坦，中田育男：マルチメディア SIMD 命令活用のためのデータサイズ推論 , 情報処理学会論文誌：プログラミング，Vol.45, No.SIG5 (PRO21), pp.1-11 (May 2004).
23）http://cis.k.hosei.ac.jp/~nakata/lectureCompiler/JavaCC/
24）http://www.coins-project.org/IPSJ-mitisirube/
25）http://gcc.gnu.org/

（平成18年3月3日受付）