テキスト処理プログラミングをマスタして LSIを効率的に設計しよう （基礎テクニック編）

デバイスの記事

関連データ

自作Verilog-VHDL変換ツールの内部構成

森岡澄夫

前回（本誌2004年7月号，pp.69-74）は，テキスト処理プログラミングを理解してツールを自作する意義と，本稿で題材として取り上げる自作Verilog HDL-VHDL変換ツールの基本操作について述べた．今回は，言語変換ツールの基本的な構造とプログラミングについて説明する．なお，ここで紹介する自作ツールは，本誌ホームページ（http://www.cqpub.co.jp/dwm/）からダウンロードできる．（編集部）

前回に続き，Verilog HDL-VHDL相互変換ツール（本稿ではこれを「CQ_V2V」と呼ぶ）の基本的なプログラミングのテクニックを紹介します．なお，今回作成する変換ツールは本誌ホームページ（http://www.cqpub.co.jp/dwm/）で公開されており，無償で広く公開されているコンパイラ群を使ってコンパイルできます．

1 変換処理の基本的な流れ

CQ_V2Vのような言語変換ツールが行わなければならない処理は，次の三つに集約されます．

1）文字列の置換
2）文字列の順序の入れ替え
3）さまざまな語句の追加・削除

そして，重要なポイントとして，入力としてやってきた文字列がたとえ同じものであっても，文脈によって扱いかたが大きく変わることが挙げられます．

● 文脈を把握したうえで文字列を操作

図1に，文脈によって変換のしかたが変わる例をいくつか示します．**図1（a）**の例は，VHDLのand，orという演算子をVerilog HDLに変換する際，意味（論理演算か条件演算か）によって，&&と&，あるいは||と|を使い分けな

```
VHDL          sig <= a and (b or c);
→Verilog HDL  assign sig = a & (b | c);

VHDL          if ((a ='1' and b ='1') or c ='1') then …
→Verilog HDL  (a == 1' b1 && b == 1' b1) || c == 1' b1) …
```

（a）同じ文字列でも，意味によって変換のしかたが違う

```
VHDL          process (clk) begin … end process;
→Verilog HDL  always (@posedge clk) begin … end
```

（c）文脈を読み取って適切な語を追加する必要がある

図1 文脈によって文字列操作のやりかたが変わってくる例
同じ文字列が入力されても，その文脈によって，文字列の置換・順序変更などのやりかたが変わってくる．また，(d)のように元の文とまったく異なった構造の文に書き換えなければならないこともある．

```
Verilog HDL   assign  sig = c ? a : b;
→VHDL         sig <= a when c ='1' else b;

Verilog HDL   always … sig = c ? a : b; … end
→VHDL         process …
                   if (c ='1')  sig <= a;
                   else         sig <= b;
                   end if;
              … end
```

（b）同じ文字列でも，出現場所によって変換のしかたが違う

```
VHDL          type STATE is (ST0, ST1, ST2);
              signal state_reg : STATE;
→Verilog HDL  parameter ST0 = 3' b001;
              parameter ST1 = 3' b010;
              parameter ST2 = 3' b100;
              reg [2:0] state_reg;
```

（d）意味が同じまま根本的に書き換える必要がある

ければならないことを示しています．**図1(b)**は，同じVerilog HDLの式であっても，出現場所（alwaysの中か外か）によって変換のしかたが変わる例です．

また，**図1(c)**，**(d)**は，書かれている意味内容を深く調べないと変換できない例です．**図1(c)**では，VHDLのprocess文のセンシティビティ・リストをVerilog HDLのそれに変形しようとしています．しかし，clkの前に追加するのがposedgeなのかnegedgeなのか，あるいは何も追加しなくてもよいのかは，process文の内部に何が書かれているかを調べないと判断できません．**図1(d)**はさらに強烈で，VHDLで頻繁に使用される列挙型がVerilog HDLには存在しないため，意味だけが同じで構文的には似ても似つかないものに変換する例です[注]．

以上のことから明らかなように，文字列を変換するには，その文字列が現れた場所における文脈，言い換えれば「いったいどのような意味で使われているのか」を調べることが重要になってきます．

●「構文木を作って再トレース」がプログラムの基本構造

こうした文脈の解析を行いつつ，言語変換を行っていく手順について，以下に概略を説明します．

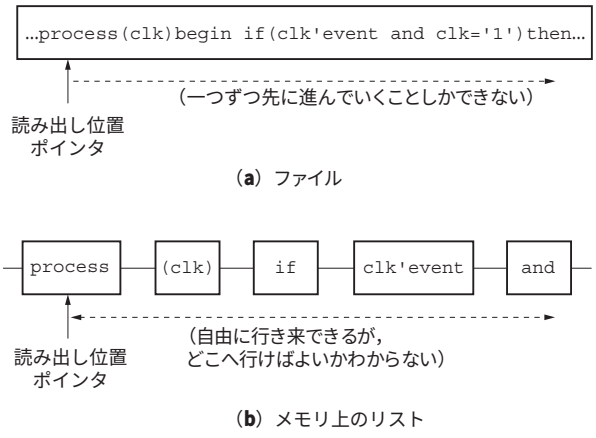
●基本手順1

まず，変換したいHDLが書かれたファイルをいったんメモリ上へ読み込みます．なぜなら，HDLがファイルに書かれたままでは，字句を最初から順番に読んでいくことしかできないので，文脈判断がやりにくいからです（**図2(a)**）．例えば，**図1(c)**でセンシティビティ・リスト中の語clkをファイルから読んだとき，そこにposedgeを付けるか否かは，HDL記述のもっと後のほうを読まなければならず，その場で判断できません．ファイルの内容がメモリ上に読み込まれていれば，clkを見つけたときに，その前に戻ったり後を先読みしたりしてようすを調べることができます．ファイルだとそうはいきません．

●基本手順2

メモリ上へHDL記述を読み込むとき，構文木というものを作ります．読んだ文字列をリストで（つまり，直線状につなげた形で）保持するのではありません（**図2(b)**，**(c)**）．構文木を作る理由は，先読み・後戻りするときや文字列順を入れ替えるときなどに，読みたい字句がメモリ上のどこにあるのかがすぐにわかるからです．

構文木の一例を**図2(c)**に示します．構文木は，現れる文字列を木の形につなげ，まとまった意味を持つ文字列群



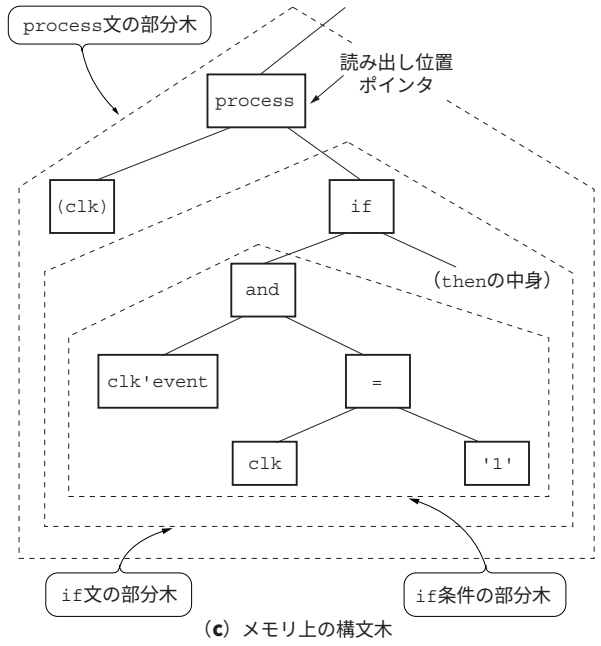


図2　HDLをファイルから読み込んでメモリ上に構文木を作る

(a)の場合，HDLがファイルに書かれた状態であり，先読みや後戻りができず，文脈判断の作業を行いにくい．(b)では，メモリ上に情報を読み込めば，好きな字句をいつでも読みにいける．しかし，直線状に字句がつながっているので，どこを読みにいけばよいかの判断が難しい．(c)のように構文木を作れば，読みたい字句がどこにあるのかがすぐにわかる．

注：**図1(d)**では，列挙型をワンホット・エンコーディングによる論理ベクトル型へ置換する，ということを行っている．この変換にあたっては，多くの処理が必要になる．まず列挙型の要素数を調べ，そこから各要素へアサインするバイナリ値を決定し，最後にこの列挙型を使用している全信号（ここではstate_reg）のタイプ宣言を論理ベクトル型へ置換する．