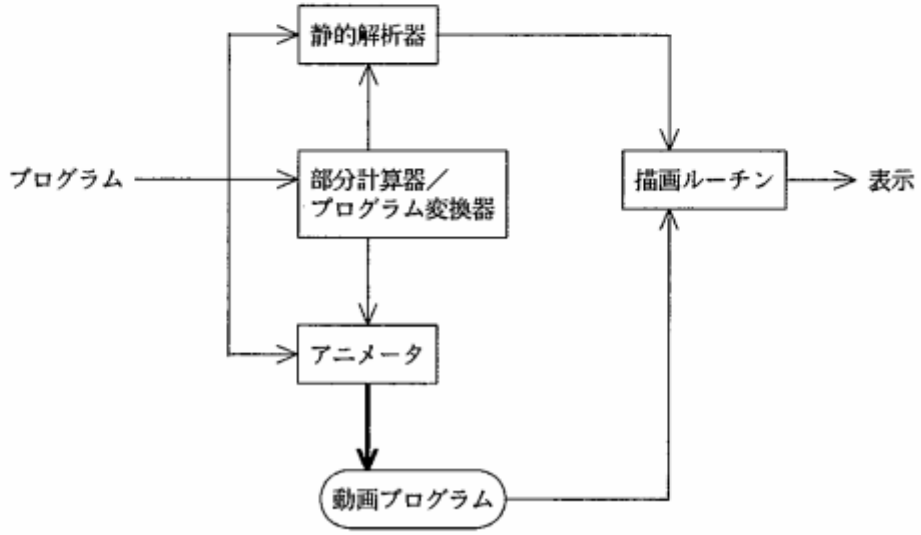
題　　名	並列プログラム変換／可視化システム（ＶＩＳＴＡ）
目　　的	プログラムの変換の技法を用いて並列プログラムの開発を支援する。また、プログラムの構造を静的／動的に表示することによってプログラムの理解を助け、デバックを容易にする。
概要及び特徴	1．部分計算によって変化するプログラム構造をグラフィック表示する 2．ＧＨＣプログラムを変換してシストリックアレイに相当するプロセス構造を導く様子を表示する。 3．レイヤードストリームのプログラム構造及び実行状況を表示する。
構　　成	

デ　　モ　　内　　容　（1／3）

(1) 部分計算によるプログラムのプロセス構造の変化を視覚化します。

(a) 関数型プログラムのコンパイル

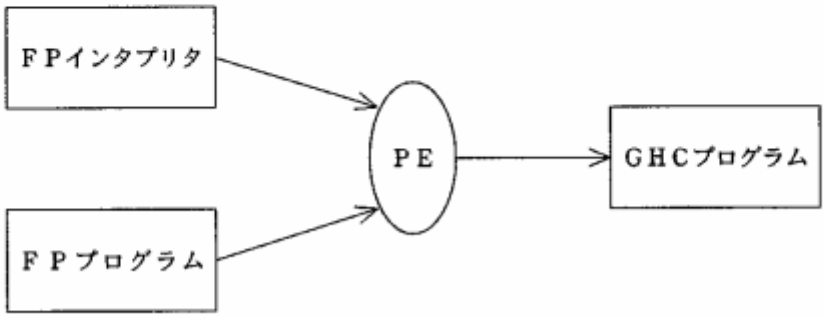
1. BackusのFP言語のインタプリタをGHC で記述します。

   ここで、インタプリタの構造を表示します。

2. FPのプログラムを記述します。

3. インタプリタにプログラムを与えて部分計算します。

   ここで、特化されたインタプリタの構造を表示します。



(b) 構文解析プログラムの生成と動作

1. 汎用の構文解析プログラムをGHC で記述します。

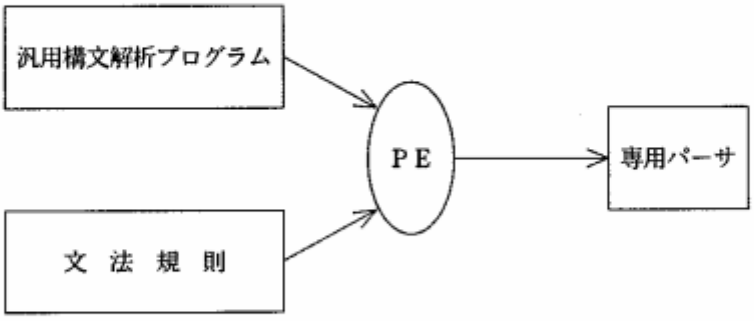
2. 文法規則を記述します。

3. 汎用の構文解析プログラムに文法規則を与えて部分計算します。

   ここで、与えた文法規則に専用化された構文解析プログラムの構造を表示します。

4. 文章を与えます。

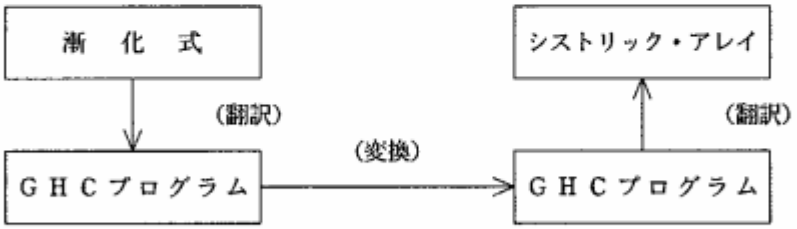
5. この文章を構文解析する様子を表示します。



デ　　モ　　内　　容　（2／3）

(2) プログラム変換によるシストリック・アレイの導出過程を視覚化します。

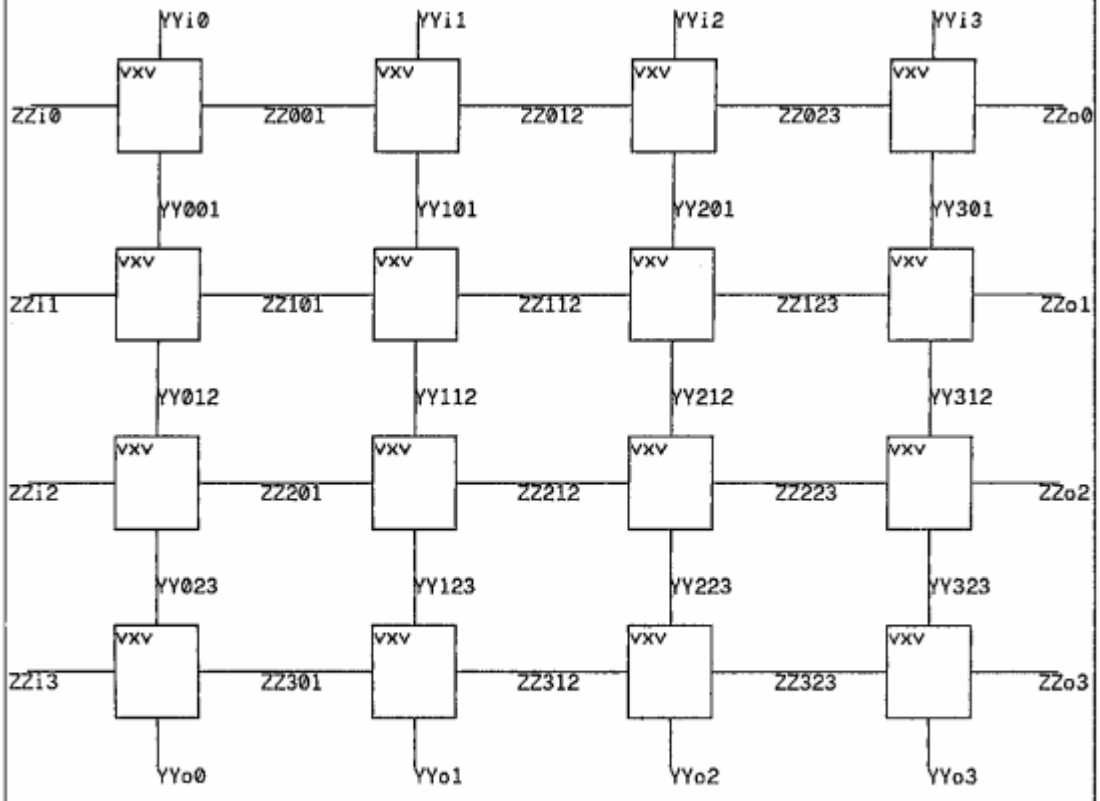
ＧＨＣのプログラム変換を応用して、漸化式による仕様定義からシストリック・アレイの構造を導出することができます。これは、漸化式を翻訳したＧＨＣプログラムを段階的に変換していって、恒久プロセスとストリーム・チャネルからなる、すなわちシストリック・アレイに対応するＧＨＣプログラムを得るものです。ここでは例として、a)有限インパルス応答フィルタ、b)行列乗算の２つを取り上げ、導出過程にそったプロセス構成の推移を表示します。



行列乗算の仕様：

$$Z = X * Y = \begin{bmatrix} x_1\ y_1 & \cdots & x_1\ y_n \\ \vdots & & \vdots \\ x_m\ y_1 & \cdots & x_m\ y_n \end{bmatrix} \quad \text{ただし} X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \quad Y = [y_1 \cdots y_n]$$

得られたシストリック・アレイのプロセス構成：



デ　　モ　　内　　容　(3／3)

(3) レイヤードストリーム型プログラムのプロセス構成を表示します。

レイヤードストリームはＧＨＣで全解探索を効率良く行うために考案されたデータ構造です。レイヤードストリームを用いた通信では送信側がストリームの要素を部分的にでも決定すれば、その部分が受信側でただちに参照可能となるため並列度の高い処理が可能となります。デモではレイヤードストリームを用いた４クイーンのプログラムについて静的なプロセス構成及び実行の様子のグラフィック表示を行います。

対象プログラム：

```
foruQueens(Q4) :-true |
        q(begin,Q1),
        q(Q1,Q2),
        q(Q2,Q3),
        q(Q3,Q4).

q(In,Out) :-true |
        filter(In,1,1,Out1),
        filter(In,2,1,Out2),
        filter(In,3,1,Out3),
        filter(In,4,1,Out4),
        Out = [1*Out1,2*Out2,3*Out3,4*Out4].

filter(begin, __, __,Out) :-true | Out = begin.
filter([],__, __,Out) :-true     | Out = []
filter([I*__ | Ins],I,D,Out) :-true       | filter(Ins,I,D,Out).
filter([J*__ | Ins],I,D,Out) :- D =:= I-J | filter(Ins,I,D,Out).
filter([J*__ | Ins],I,D,Out) :- D =:= J-I | filter(Ins,I,D,Out).
filter([J*Ini  | Ins],I,D,Out) :- J\=I,D =\= I-J, D =\ =J-I  |
        D1 := D+1, filter(Inl,I,D1,Out1),
        filter(Ins,I,D,Outs)
        Out = [J*Out | Outs].
```

静的プロセス構成の表示例：