# 解析木インタプリタ PATIE0 のアーキテクチャ†

闞 暁 薇†† 板 野 肯 三†††

対話性の高いプログラミングシステムを実現するための一手法として，解析木をプログラムの内部表現としたインタプリタのハードウェアを実現することを目標に，PL/0 用のプロトタイプを，レジスタ転送レベルで設計した．解析木インタプリタのアーキテクチャは，機能別にトラバースユニット，コントロールユニット，データユニットという3つのユニットに分解し，さらに，ハードウェアの構成としては，これらをパイプライン的に結合して，実行の高速化を計った．パイプライン処理を効果的に行うために，解析木のノードをトラバースする方法を工夫し，実行に先だって必要なノードがプリフェッチできるようにした．本論文では，このような方針で設計した解析木インタプリタのハードウェアの構成と制御方式について説明し，レジスタ転送レベルでの性能の測定とその評価を行う．

## 1. はじめに

高級言語によるプログラミングの対話的な作業は，プログラムの誤りを発見するためのテストや，デバッグを行う時の編集と実行の繰り返しで占められている．しかし，コンパイラをベースにしたプログラミングシステムでは，ソースプログラムの変更や対話的な実行の指示など，対話的な作業にとって本質的なもの以外に，複雑なパラメータを与えてコンパイラ・リンカ・エディタといった道具を呼び出すという“作業”が必要である．また，これらの処理が終わって次の作業に移るまでには，かなりの待ち時間が必要であり，プログラマの思考が中断されてしまうことが多い．Make[1] のような道具を使えば，他の道具の起動は簡略化できるが，コンパイルやリンクに必要な時間自体は減少しない．

コンパイラをベースにしたプログラミングシステムにおけるもう1つの問題は，プログラムを停止して変更した際に，実行形式のプログラムを作り直す必要があり，これと同時に，変更前の実行状態が失われてしまうことである．このため，プログラムの実行は，変更によって影響を受けない部分も含め，最初からやり直さなければならない．実行中のプログラムの状態を可能な限り保持したまま，もとのソースプログラムに変更を加えて実行を継続することは，開発中のプログラムが巨大なプログラムであるときには，特に重要であることが多い．たとえば，大きなプログラムの実行を細かなレベルで監視している途中で，その一部を変更して実行の継続を試みたいということはよくある．

このような要求に対応する手法として，著者らは，構造エディタと解析木インタプリタを組み合わせた方法を試行してきた．この手法は，対話的な実行のための枠組みを提供するという立場からすると，インクリメンタルコンパイラ[2] よりも強力である．しかし，ソフトウェアで実現された解析木インタプリタを実行系として用いているので，コンパイルしたコードを実行するのに比べて，実行速度が約百分の一と低くなってしまう[3]．そこで，これまでに試作したプログラミングシステム COSMOS[3)-6)] では，動的複合実行方式[4] という手法を用いてインタプリタが実行するプログラムの範囲を必要な部分だけに限定し，残りはコンパイルしたコードを実行することで，実質的な実行速度を上げた．しかし，この動的複合実行方式では，プログラムに対するデバッグ機能の適用範囲をどこに設定するかが大きな問題であり，この問題を根本的に解決するには，インタプリタそのものを高速にして，実行時には，すべて，インタプリタで実行するしかない．そこで，著者らは，この問題を解決するもう1つの方法として，解析木インタプリタをハードウェア化することを考えることにした．

構造エディタと解析木インタプリタを組み合わせたプログラミング・システムにおいて対象となりうる言語には，特に制約はないが，現在，対象として考えているのは，C や PASCAL のような手続き型言語である．これまでに，COSMOS の上では，LISP や PROLOG も実現されているが，これらは，もともとインタプリタをベースにして開発された言語であり，比較的実現が容易である．この意味で，私たちの興味は，もともとコンパイラをベースにして開発されてき

† Architecture of a Parse-Tree Interpreter PATIE0 by XIAOWEI KAN (Doctoral Program in Engineering, Graduate School, University of Tsukuba) and KOZO ITANO (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学大学院工学研究科
††† 筑波大学電子・情報工学系

た言語をインタプリタの対象にするところにある．しかし，ハードウェアによるインタプリタをきめ細かく設計するには，Cは言語仕様が大きすぎるように思われた．そこで，準備として，簡単な言語である PL/0 を実行する解析木インタプリタ (PATIE 0: PArse-Tree InterpretEr for PL/0) のハードウェアを設計することで，本格的な手続き型言語に対応する設計の指針を得ることにした[6)~8)]．ただし，純粋な PL/0 は言語の記述能力が制限されすぎているので，配列や関数などを導入して記述機能の強化を行い，標準的なサンプルプログラムであるソートなどが記述できるようにした．

本研究の具体的な目的は，PL/0 インタプリタのハードウェアを設計し，シミュレーションによって性能の評価を行って，実現性のあるアーキテクチャの基本的な枠組みを決定することにある．したがって，プログラミングシステム全体を実現することはせず，解析木インタプリタをどのようにハードウェア化すれば，対話的なデバッグに必要な機能を保持したまま，高速な実行性能が得られるかに焦点を絞って設計を行うことにした．結果的には，コンパイルしたコードを実行するのに匹敵する性能を持つ対話型実行向きのインタプリタの設計に成功したと考えている．本論文では，以下，このような方針で設計した解析木インタプリタのハードウェアの構成と制御方式について説明し，レジスタ転送レベルでの性能の測定とその評価を行う．

## 2. 解析木の表現と実行の方針

解析木の論理的な構造には，言語の文法定義が反映されるが，この解析木のノードの設計を文法レベルまで含めて実行に適した形に設計することは極めて重要である．ノードの表現形式は，通常の CPU から見た機械語の形式に対応しており，この部分の設計が実行性能に 5〜10 倍の範囲で影響することが，初期の予備的なシミュレーションで判明した．そこで，本章では，まず，ハードウェアによる高速な実行を行うための解析木の表現について説明し，ついで，解析木上での実行について，その概略を説明する．

### 2.1 解析木のノードの表現形式

解析木のノードを実行に適した形に最適化するために，まず，文法は，LL ($k$) とし，各生成規則の右辺の非終端記号の数を最大2に制限した．これによって，解析木の各ノードが持つ子ノードの数を2に限定し，ノードの大きさを固定にすることができる．ノードの保持する主要な情報は，ノードの種類（対応する生成規則）を示すコード，2つの子ノードへのポインタ，意味解析の結果などであるが，このうち，ノードの種類を示すコードと2つのポインタは，まとめて1語 (64 ビット) にすることで，ノードをメモリからフェッチする時のメモリアクセスの負荷を軽減する．

解析木は抽象構文木などに比較してプログラムの本質的構造を表現するには冗長なノードを多く含んでおり，メモリ上での表現のコストが大きい．解析木インタプリタはこの冗長なノードを含む木を辿るために，ノードのフェッチの処理は全体のボトルネックになる．したがって，ここでは，子ノードが1つしかなく，属性や意味規則が存在しないノードを親ノードの表現と一体化することで，解析木の表現を圧縮して，実行時の効率が下がらないようにする方法を採用している[5)]．

### 2.2 解析木と実行の例

繰り返しによるループなどの制御構造を除いて，実行するときに必要な解析木のトラバースは1回で済むように実行アルゴリズムを設計する．解析木と実行の例を以下に示す．

(1) 代 入 文

**図 1** に，PL/0 での代入文 "$x:=x+y$" の解析木を，生成規則とともに示す．点線によって接続されている枝は，圧縮されている部分であり，実際には存在しないので，実行する必要がない．また，この解析木には，ノードの親子関係を表現するリンク以外に，意味解析の際に作成されるリンクが示されている．例えば，"idR" は，変数の参照に対応するノードであり，本来は，解析木の末端のノードにあたるが，意味解析の時に，この変数の宣言に対応するノード "ValIdD" にリンクされる．このようなリンクによって，デバッグの時に変数ごとのブレークポイントの設定などの操作が容易になる．さらに，この図に示すように，実行時のトラバースのコストを小さくするため，実行の対象となっているノードを1回だけ走査するだけで済むように実行アルゴリズムの設計を工夫している．

(2) IF 文

**図 2** に IF 文の解析木とトラバースの経路を示す．各生成規則の右辺の非終端記号の数を2以下に制限したことにより，IF 文は，2つの生成規則に分けて定義する必要がある．このため，ノードもこれに対応して2つになる．これをトラバースする際には，"if" ノードでは，"cond", "if 2" の順にトラバースする

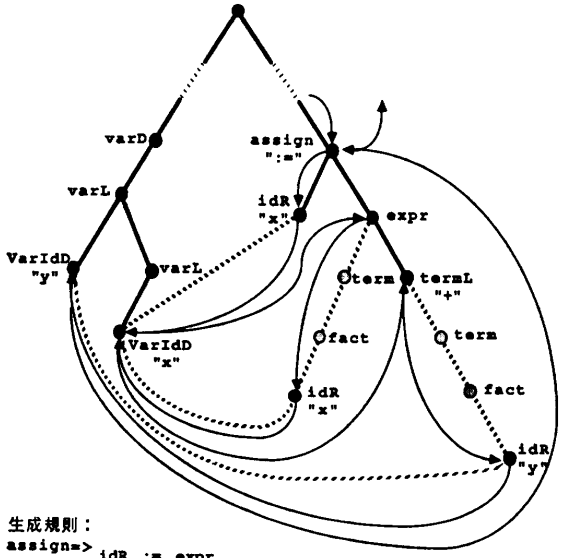
が，“if 2” ノードでは，条件式 “cond” の値により，2つの子ノードの内に1つを選択する．

（3） WHILE 文

**図 3** に WHILE文の解析木とトラバースの経路を示す．WHILE文も IF文と同様に 2 つのノードに分けて表現され，“while” ノードは，普通のノードと同様に取り扱う．“while 2” ノードでは，意味解析時に，2番目の子ノードへのポインタとして，“while” ノードへのリンクを設定しておく．そして，“while” をトラバースする時，条件式 “cond” の値が真の場合には，2番目の子ノードへトラバースすることでループが実現される．また，条件式の値が偽の場合には，“while 2” ノードから脱出することによって，WHILE文の実行が終了する．

（4） 関数の呼び出し

**図 4** に関数の呼び出しに関連する解析木とトラバースの経路を示す．この解析木には，大きく分けると，関数の呼び出しと関数の宣言との2つの枝が示してあ



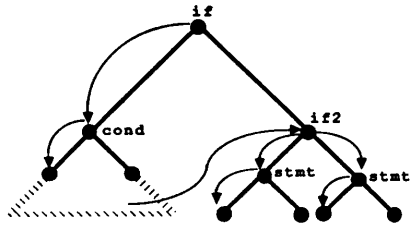

生成規則：

```
assign=> idR := expr
expr   => term | term termL
termL  => + term termL |
          - term termL
term   => fact | fact factL
factL  => * fact factL |
          / fact factL
fact   => idR | num |
          arridR [expr]|
          funcidR (args)
varD   => VAR varL
varL   => VarIdD | VarIdD,varL
```

トラバースの経路
圧縮されたリンクとノード
意味解析で作成したリンク

**図 1** 代入文の解析木の例 ($x:=x+y$)

Fig. 1 An example of a parse tree for an assignment statement.



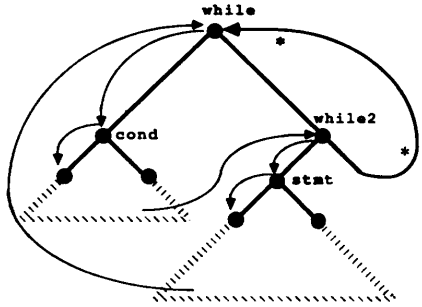

生成規則：

```
if  => IF cond if2
if2 => stmt ELSE stmt
```

トラバースの経路
解析木の枝

**図 2** IF 文の解析木の例

Fig. 2 An example of a parse tree for an IF statement.



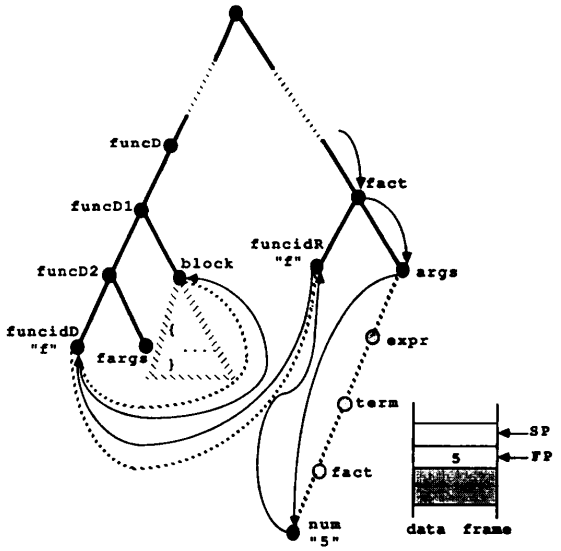

生成規則：

```
while  => WHILE cond while2
while2 => stmt
```

トラバースの経路
解析木の枝
意味解析で作成したリンク

**図 3** WHILE 文の解析木の例

Fig. 3 An example of a parse tree for a WHILE statement.




生成規則：

```
args  => expr | expr,args
func  => FUNCTION func1;
         | FUNCTION func1;func
func1 => func2 block
func2 => funcidD (fargs)
```

トラバースの経路
意味解析で作成したリンク
圧縮されたリンクとノード

**図 4** 関数呼び出しの解析木の例 (f(5))

Fig. 4 An example of a parse tree for a function call.

る．デバッグの時にブレークポイントの設定などの操作を容易にするために，意味解析の際に，関数名の参照に対応するノード “funcidR” から関数名の宣言に対応するノード “funcidD” へリンクを作成し，さらに，余分のノードをスキップするために，“funcidD” から関数の本体を表すノード “block” へもリンクを作成する．関数の呼び出す際には，まず，パラメータを評価して，その結果を新しく作成した関数呼び出しに対応したフレームに書き込む．次に，“funcidR” から，“funcidD” を通して “block” へトラバースし，その関数を実行する．

## 3. プリフェッチのモデル

解析木の各ノードの実行は，解析木のトラバース，ノードの再帰的な実行の制御，データ演算・転送という3つの処理に分けられる．PATIE 0 のハードウェアは，これらのそれぞれの処理に対応したトラバースユニット (Traverse Unit)，コントロールユニット (Control Unit)，データユニット (Data Unit) の3つのユニットをパイプライン結合して構成する（図 5）．

一般には，ノードのトラバースの順序は実行アルゴリズムによって決定されるため，あるノードの実行が完了するまでは，次のノードのトラバース処理は決定できない．このため，トラバースを先行処理する段階でどちらかの子ノードをアドホックに選択してプリフェッチすると，一定の割合で失敗が起こり，プリフェッチが確率的にしか成功しない．そこで，実行アルゴリズムの設計に制限を加え，条件分岐を行う必要のあるノードは例外として，次に実行すべきノードの決定が，現在のノードの実行の結果に依存しないで決定できるようにする．すなわち，実行時のトラバースの順序が実行の動的な状態に依存しないで，ノードの種類のみによってトラバースの順序が静的に決定できるようにする．このことは，実行のアルゴリズム（各ノードに対する実行規則の集合）を設計するときの制約になるが，PL/0 の解析木インタプリタの実行アルゴリズムを詳細に解析した結果，この制約があっても，大部分のノードに対する実行アルゴリズムの設計には影響がなかった．

基本的には，このような方針を前提にして，さらに，条件分岐ノードのプリフェッチを行うかどうかで，2つのプリフェッチのモデル（図5）を考案した．

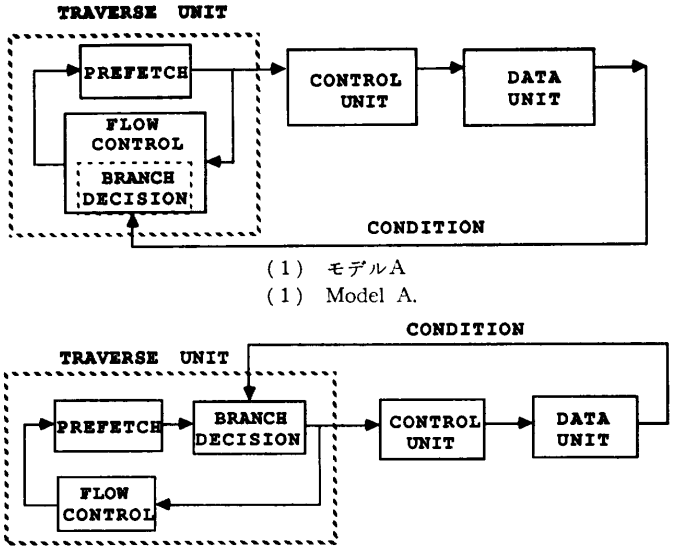
（1） モデルA

実行時の動的な状態に依存してフェッチの対象が変化する場合は，状態が確定するまでプリフェッチを停止すれば，失敗は起こらない．そこで，このモデルAでは，番件分岐などのノードについては，一切先行処理を行わず，条件の評価結果を待ってから実行すべきノードを決定することにし，動的状態に依存しないノードについては，あらかじめどちらの子ノードへトラバースするかを決めてフェッチを行う．

図5（1）で示すように，トラバースユニットはプリフェッチ部 (PREFETCH) とフローコントロール部 (FLOW_CONTROL) に分かれている．プリフェッチ部で解析木のノードがフェッチされて，次のコントロールユニットに送られる．これと同時に，このノードがフローコントロール部で処理されて，次にフェッチすべき解析木のノードのアドレスが決定され，プリフェッチ部に渡される．プリフェッチ部とフローコントロール部の処理はオーバラップさせない．もしも，条件分岐のノードをフェッチした場合，フローコントロール部の分岐の決定を行う部分で，データユニットから条件の評価結果の待ち同期をとる．

（2） モデルB

実行時の動的な状態としては，条件分岐などのように言語仕様によって要求されるものと，解析木の実行アルゴリズムをインプリメントするための技術的なものとがある．後者




（1） モデルA
（1） Model A.

（2） モデルB
（2） Model B.

図 5 プリフェッチのモデル
Fig. 5 Prefetch models.

は，モデルAで示したように，実行アルゴリズムの設計を制限することで，ある程度取り除くことができる．そこで，このモデルBでは，モデルAに加え，前者の条件分岐に関してもプリフェッチを行う．

モデルBでは，図5(2)で示すように，トラバースユニットはプリフェッチ部 (PREFETCH) とフローコントロール部 (FLOW_CONTROL)，分岐決定部 (BRANCH_DECISION) に分かれている．条件分岐ノードを処理する場合，プリフェッチ部では，条件分岐ノードの次に実行するノードを選択してプリフェッチし，次の分岐決定部に送る．分岐決定部では，プリフェッチされたノードが正しいかどうかの確認を行い，正しい場合にはコントロールユニットに送り実行する．プリフェッチするノードを誤った場合には，プリフェッチをキャンセルし，フェッチをやり直す．モデルAとBの違いは，モデルAでは，条件分岐のノードの処理の中で常に待ち合わせを行うのに対して，モデルBでは，条件分岐ノードの次のノードがどちらか適当にプリフェッチされて，そのノードで待ち合わせが起こるという点である．

ここで，条件分岐時のプリフェッチは，第1ノードを必ずプリフェッチするという方法を用いるモデルを B-1 とし，過去のトラバースの履歴を記録して，頻度の高い方をプリフェッチするというアルゴリズムを用いるモデルを B-2 とする．いずれのモデルでも，失敗したときは，キャンセルしてやり直す．モデル B-2 においては，実行の履歴を記録するために，トラバースユニット内部にノードのアドレスで検索可能なカウンタを連想記憶として用意することにする．このカウンタが溢れた場合は，LRU などで管理を行う必要があるであろうが，今回使った例は大きくなかったので，溢れは起こらなかった．

## 4. パイプライン・アーキテクチャ

PATIE0 のパイプラインを構成する3つのユニットは機能的にほぼ独立しており，各ユニット間は単純なインタフェースで結合することができるように設計されている．以下では，このハードウェアの構成について説明する．

### 4.1 パイプラインの構成

図 6 に PATIE0 のハードウェアの具体的な構成を示す．トラバースユニットは，解析木をトラバースしながら，メモリ MEMORY から解析木を1ノードずつフェッチする．フェッチしたノードを次のユニットで実行する必要があれば，ノードの種類を表すコードと実行に必要な情報をインタフェイス用のレジスタNCODE と OPERAND1 を通してコントロールユニットに渡す．各ノードの実行は子ノードの呼び出しを含むいくつかのより細かいステップに分けられる．そこで，コントロールユニットでは，それぞれにマイクロ命令を対応させ，この命令の列を制御メモリ CMEM より読み出す．データの演算や転送といった実行が必要なときは，演算を指定するコードとノードからフェッチしたオペランドを，インタフェイスレジスタ ECODE と OPERAND2 を通してデータユニットへ送る．条件分岐ノードを実行するときは，データユニットで評価した条件値を COND レジスタを通して，トラバースユニットに送る．

コントロールユニットとデータユニットは，1レベ

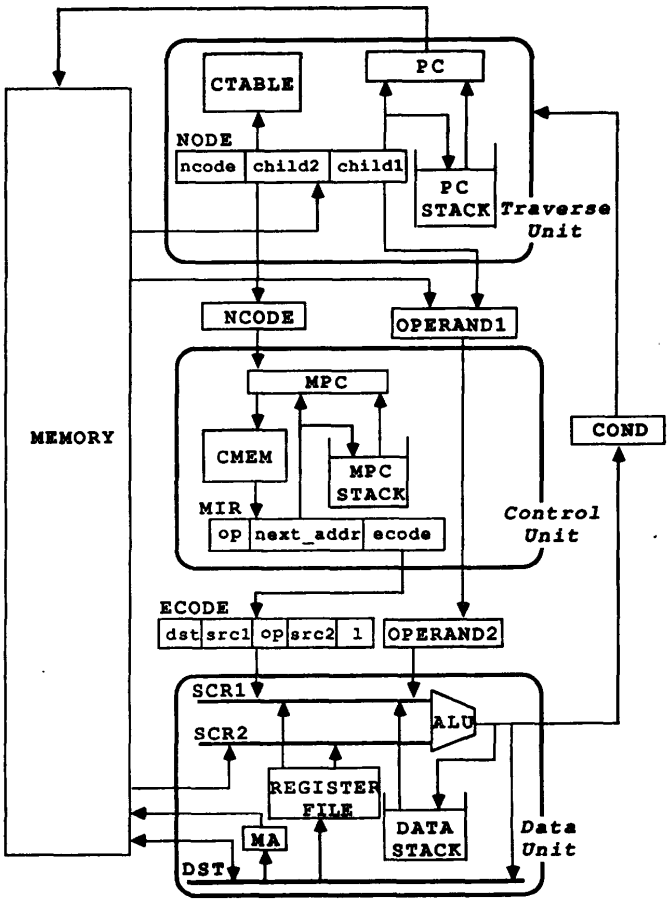



図 6 PATIE0 のハードウェア構成
Fig. 6 Hardware organization of PATIE0.

ルパイプライン型のマイクロプログラムのメカニズムと見なすこともできる．しかし，ここでは，コントロールユニットは，解析木のノードの再帰的な実行を制御するためのスタック付き状態マシンであると考える．また，データユニットはメッセージによって，コントロールユニットやトラバースユニットと結ばれているので，物理的なコントロールの面からみると，独立していると考えられる．

各ユニットに用意したスタックは，主に，子ノードの実行のために中断された親ノードの実行状態を保存するためのものである．専用のスタックではなく，メモリをスタックとして用いる方式もあり得るが，解析木の実行中，状態を保存する操作はノードをトラバースするごとに頻繁に行われるので，専用のスタック(後述の PC_STACK, MPC_STACK, DATA_STACK の3つ) を用意して実行状態を保存することにした．最終的に採用した方式では，各ノードの実行に要する時間は，数クロック程度であり，スタックのアクセスのスピードはインタプリタの性能に大きく影響している．ここで，3つのスタックをそれぞれのユニットに分散するのは，各ユニットの独立性を高め，構成のバランスをとるためである．

### 4.2 各ユニットの構成

パイプラインの各ユニットのハードウェアの構成を以下に説明する．

(1) トラバースユニット

トラバースユニットはフェッチすべきノードのアドレスを保持するレジスタ PC，フェッチしたノードを保持するレジスタ NODE，フェッチしない方の子ノードのポインタを保持するスタック PC_STACK，ノードの種類をデコードしてフローコントロール処理を決定するためのテーブル CTABLE などを持っている．このトラバースユニットは，単に，実行すべきノードのフェッチの先行制御を行うだけでなく，IF 文や WHILE 文の条件分岐の実行も行う．この IF 文や WHILE 文の条件分岐の実行は，コントロールユニットやデータユニットで行うべき処理を含んでいないので，トラバースユニット内で処理を完結させることができる．一般に，トラバースユニットでは，フェッチしてきたノードの実行が，次段のユニットの実行が必要であるかどうかの判断を行い，必要でない場合には，次段にノードを送らないようにして処理の最適化を行っている．

(2) コントロールユニット

コントロールユニットは，実行すべきマイクロ命令のアドレスを保存するレジスタ MPC，読み出したマイクロ命令を保存するレジスタ MIR，マイクロ命令のシークエンシングに使用するスタック MPC_STACK と，マイクロプログラムを格納している専用メモリ CMEM を持っている．マイクロプログラムを格納するためのメモリは，メモリアクセスのオーバヘッドを減少するために，主記憶上に置かず，専用の ROM (CMEM) としてコントロールユニット内に置く．コントロールユニットは，トラバースユニットから送られたノードのコードをそのノードに対応するマイクロ命令列の先頭アドレスとして使用して，マイクロ命令をフェッチする．つぎに，そのマイクロ命令をデコードして，次に実行すべきマイクロ命令のアドレスを決定する．データの演算や転送といった実行が必要な場合のみ，実行の指示を行うコードと必要なオペランドをデータユニットに送る．このとき，"begin" などのようにデータ構造に関する操作がないノードはデータユニットへ渡さず，コントロールユニットで実行を終了する．

(3) データユニット

データユニットは，コントロールユニットから送られた実行コード (ECODE) を受け取り，データの演算や転送といった処理を行う．データユニットは，データが格納されているメモリ MEMORY をアクセスするためのメモリアドレスレジスタ MA，演算用のスタック DATA_STACK，レジスタファイル，データやアドレスなどを転送するための3本の伝送バス (DST, SRC1 と SRC2) を持っている．レジスタファイルには，データレジスタ，アドレス計算用レジスタ，フレームポインタ，スタックポインタなどが置かれる．

データユニットで行う演算は，基本的に3オペランド型であり，

DST→SRC1 OP SRC2

の形式で表現される．ECODE は，5つのフィールドに分けられ，3つのオペランドと演算の種類およびリテラル(L)を指定する．演算の種類は合計 10 種類，オペランドの種類は 15 種類を定義している．

### 4.3 キ ュ ー

シミュレーションを行った結果によると，トラバース処理とこれ以降の処理 (再帰制御や演算) では，ノード単位での処理時間にばらつきがある．このばらつきのためにパイプラインの効率が悪くなる場合がある．

これを解決するためには，インタフェイスレジスタ NCODE と OPERAND1 をキューにしてやればよい．実行時の動的な状態によって，トラバースの経路が変化する可能性がないモデルAでは，キューの挿入は特に効果があると考えられる．そこで，モデルA（3章）の方式をベースにして，このキューの効果を評価することにした．このモデルは，モデルAの変形であるので，キューのサイズを $k$ として，対応するモデル A-$k$ とする．

## 5. 評　価

シミュレーションは，ハードウェア構文木インタプリタをC言語でレジスタ転送レベルで記述して SUN 3/50 M の上で行い，実行速度に関する性能の評価を行った．

### 5.1 レジスタ転送レベルの設計

ハードウェアの設計は，レジスタ転送レベルで行っている．まず，1相のクロックを用いることを前提としたので，レジスタは，エッジトリガ型のものを使っていると仮定する．メモリアクセスの遅延，演算回路やデコーダの遅延はゼロと仮定する．パイプライン間の同期は，クロックに同期したフラグレジスタを用いて行う．実際のハードウェアでは，非同期的な制御をすることもできるが，ここでは，設計を単純にするために，すべてレジスタ転送レベルの枠組みの中で行うことにした．

### 5.2 性能の測定

PATIE0 の性能測定を行うために，クイックソート (sort (200))，INTEGER の行列の乗算 (matrix (30*30)) とアッカーマン関数 (ackerman (3, 4)) の3つのサンプルプログラムを使用した．これらのサンプルプログラムに対応する解析木を作成して，PATIE0 の上で実行した．また，これらのサンプルプログラムを PASCAL コンパイラによってコンパイルして，SUN 3/50 の上で実行した．このコンパイラはCと同じくらい性能のよいコードを出すことが知られているので，比較の対象としては，十分なものと考えられる．PATIE 0 のクロックは 10 MHZ とし，実行時間はクロック数から計算した．ただし，このクロックは，レジスタ転送レベルでのクロックである．モトローラ MC 68020 のクロックは 15 MHZ であるが，こちらは，実時間での実行時間を測定した．使用したサンプルプログラムは，入出力がないので，CPU の実行時間のみの比較となる．このような条件で測定した結果を，表1から表4までに示す．

**表 1** スタックとメモリのアクセス頻度
Table 1 Frequencies of stack and memory accesses.
総アクセス数（ノード当り平均アクセス数）

| | Sort | Matrix | Ackerman |
|---|---|---|---|
| PC_STACK | 1,293,592 (1.21) | 1,471,466 (1.26) | 288,604 (1.04) |
| MPC_STACK | 1,738,590 (1.62) | 1,979,378 (1.70) | 391,678 (1.41) |
| DATA_STACK | 324,796 (0.30) | 563,708 (0.48) | 72,030 (0.26) |
| PMEM | 1,071,299 (1.00) | 1,163,547 (1.00) | 299,033 (1.07) |
| DMEM | 362,799 (0.34) | 310,658 (0.27) | 97,862 (0.35) |

**表 2** 使用されたスタックのエントリ数
Table 2 Used entries of stacks.

| | Sort | Matrix | Ackerman |
|---|---|---|---|
| PC_STACK | 10 | 14 | 765 |
| MPC_STACK | 12 | 17 | 1,009 |
| DATA_STACK | 2 | 3 | 126 |

**表 3** 各ユニットの稼働状況
Table 3 Working ratios of pipeline units.
クロック数（稼働率）

| | Sort | Matrix | Ackerman |
|---|---|---|---|
| トラバースユニット | 3,496,694 (0.92) | 3,828,325 (0.93) | 958,941 (0.91) |
| コントロールユニット | 3,414,889 (0.89) | 3,546,379 (0,86) | 860,373 (0.82) |
| データユニット | 1,426,584 (0.37) | 1,584,008 (0.37) | 412,632 (0.39) |
| 総実行クロック数 | 3,819,304 | 4,113,880 | 1,052,325 |

**表 4** 実行時間の比較
Table 4 Comparison of execution times.
単位：msec

| モデル | Sort | Matrix | Ackerman |
|---|---|---|---|
| パイプライン無 | 834 | 896 | 223 |
| A-1 | 439 | 471 | 122 |
| A-2 | 404 | 426 | 108 |
| A-3 | 382 | 411 | 105 |
| B-1 | 433 | 471 | 124 |
| B-2 | 433 | 471 | 122 |
| MC68020 (15M) | 257 | 360 | 92 |

註：モデル A-$k$ では，$k$ はキューのサイズである．

### 5.3 性能の解析

まず，表1から分かるように，1つのノードを実行するのに，PC_STACK，MPC_STACK，DATA_STACK に関する操作は 0.3〜1.7 回程度行われる．このことは，ハードウェアの専用スタックを独立に用

意していることの妥当性を裏づけている．また，表2に示すように，アッカーマン関数などのような再帰呼び出しを多く行うプログラム以外は，実行時に必要なスタックのサイズはそれほど大きくない．したがって，ハードウェアとして実装するスタックの大きさは制限し，オーバフロー時に，割り込みを発生して主記憶との間でスワップすることが現実的であると考える．

表1の中の PMEM と DMEM は，それぞれ，トラバースユニットとデータユニットが，メモリをアクセスする回数を示している．PMEM に対する DMEM のアクセス頻度は，1/3 程度であり，これから，ノードの形をコンパクトに設計した効果があることが分かる．表3には，ウエイト状態を除いた各ユニットの実質的な稼働クロック数が示されている．このデータから，データユニットの稼働時間が他のユニットの半分程度であることが分かるが，データユニットでは，あらゆる演算の実行時間を1クロックとして設計しているので，現実にハードウェアを実現する場合に比べて評価が厳密でない．しかし，これらのことを考慮しても，3つのユニットのロードは概ね均衡していると考えられる．

表4には，プリフェッチのいくつかのモデルについて，シミュレーションを行った場合の，PATIE 0 の総合的な実行時間を示す．モデルAに関しては，キューの効果をサイズ1から3の場合それぞれに関して測定した．その結果を，A-1 から A-3 に示す．参考データとして，パイプライン処理を行わないインタプリタの性能を“パイプライン無”として示した．この結果から見ると，トラバースユニットとコントロールユニットの間にキューを置く効果が大きいことが分かる．ここにはデータを示さなかったが，実際，キューのサイズが3ぐらいまでは，パイプラインを構成するユニットの待ち時間が減少する．これは，ノードごとで異なっている実行時間がキューによって平均化されたためであると考えられる．

条件分岐ノードのプリフェッチを行うモデル B-1, B-2 に関しても，測定の結果を表4に示す．これらのモデルの性能が，プリフェッチを行わないモデルとそれほど変わらないのは，いくつかの原因が考えられる．まず，今回の設計では，メモリのアクセス時間が1クロックであると仮定している．このため，プリフェッチを行ったときのゲインが1クロックしかないのに比べ，プリフェッチに失敗した場合には，トラバースユニット内のスタックやレジスタを復元するための時間コストがかなり大きく，成功した場合のゲインを上回っている．また，条件ノードのプリフェッチを行う場合は，分岐決定部の処理を，フェッチの後に行う必要があるので，条件分岐ノード以外の実行でも，1クロックサイクル程度遅くなってしまい，これが全体にこのモデルを不利にしている．逆にいうと，モデルAでは，この分岐決定処理をプリフェッチ部とコントロールユニットの間に入れる必要がないので，有利になっていると考えられる．

全体として，パイプライン制御を行わない場合に比べて，パイプライン制御を取り入れた場合，全体の実行時間は，約 50% 以下になっており，かなりの効果が得られている．特に，キュー（サイズ3）を入れたモデル A-3 では，他のモデルよりも，高い性能が得られている．総合的に見ると，SUN 3/50 M 上でコンパイルされた機械語を実行する場合に対する PATIE 0 の実行時間は，1.31 から 1.71 程度であった．

## 6. おわりに

このプロジェクトの初期の設計では，ソフトウェアのインタプリタのアルゴリズムをそのままマイクロプログラムで実現していた．その場合，ソフトウェアのインタプリタの 10 倍程度の実行速度を得たものの，コンパイルされたコードの実行に比べ約 10% の性能であった．この後，簡単なプリフェッチのメカニズムやスタック操作の最適化などを少しずつ行い，コンパイルされたコードの実行に比べ，約 60% の性能を得るところまで行った．しかし，このままでは，これ以上の性能の向上は困難であると判断し，最初から設計をやり直し，現在のような形のハードウェアの構成に変更した．このモデルでも，現時点での最高速のマイクロプロセッサのスピードには及ばないが，解析木インタプリタの方もまだ高速化できる余地を残しており，一定の水準には達していると考えている．

今回の設計は，主に，将来 C や PASCAL などの本格的な手続き型言語を対象にしたハードウェア解析木インタプリタの設計方針を得ることを目的として行い，一応の成果が得られた．しかし，細かな点では，いくつかの問題が課題として残されている．例えば，メモリアクセスのモデルがプリミティブであり，メモリのアクセス時間が正確には評価できていない．対象言語として採用した PL/0 も単純な言語であるため，C や PASCAL などの本格的な言語を取り扱うときの問題がすべては解析しきれているとは言えない．性

能評価に使用したサンプルプログラムも限定されたものであり，また，ハードウェアの物理的な設計に依存する部分の評価が入っていないなど，より精密な評価を行う必要がある．しかし，これらのことを考慮しても，本格的な手続き型言語のためのハードウェアインタプリタを設計する十分な指針が得られたと考えている．

現在，この論文で提案したプロトタイプの設計とシミュレーションによって得られた指針に基づいて，C言語用のインタプリタを開発しているところである．また，解析木インタプリタの設計に合わせて，構造エディタの改造も必要であると考えている．

**謝辞** 本研究の基礎となった COSMOS の研究を行った，現在電子技術総合研究所研究員の佐藤豊氏には，貴重な助言をいただき，感謝します．また，多くの意見をいただいた大学院博士課程に在籍中の西山博泰氏に感謝します．

## 参 考 文 献

1) Feldman, S. I.: Make—A Program for Maintaining Computer Programs, *UNIX Programmer's Supplementary Documents*, Vol. 1, 4.3 BSD, U. C. Berkeley (1986).
2) Medina-Mora, R. and Feiler, P. H.: An Incremental Programming Environment, *IEEE Trans. Softw. Eng.*, Vol. SE-7, No. 3, pp. 472-482 (1981).
3) 佐藤 豊，板野肯三：構造エディタとソースコードインタプリタの統合的記述とその生成系，コンピュータソフトウェア，Vol. 4, No. 2, pp. 135-146 (1987).
4) 佐藤 豊，板野肯三：動的複合実行方式—直接実行系と翻訳実行系を統合した対話型実行方式，コンピュータソフトウェア，Vol. 2, No. 4, pp. 19-29 (1985).
5) 佐藤 豊，板野肯三：構造エディタにおける下降型パーサのための構文木の圧縮法，情報処理学会論文誌，Vol. 28, No. 3, pp. 310-313 (1987).
6) 佐藤 豊，板野肯三：構造エディタのためのインクリメンタル LL パーサの一構成法，情報処理学会論文誌，Vol. 28, No. 6, pp. 668-672 (1987).
7) Itano, K. and Kan, X.: PATIE: A Hardware Parse Tree Interpreter, Technical Note HLLA-E-18, 筑波大学電子情報工学系 (1988).
8) **闕 暁薇，板野肯三：** 構文木インタプリタ PATIE-0 のアーキテクチャ，情報処理学会研究会計算機アーキテクチャ，74-1 (1989).

(平成2年12月18日受付)
(平成3年 6 月13日採録)

**闕 暁薇**（正会員）

1960 年生．1982 年北京航空航天大学計算機科学系卒業．現在，筑波大学大学院博士課程工学研究科に在学中．コンピュータアーキテクチャ，言語処理系，プログラミング環境に興味を持つ．IEEE 会員．

**板野 肯三**（正会員）

昭和 23 年生．昭和 46 年東京大学理学部物理学科卒業．昭和 48 年同大学大学院修士課程修了．昭和 51 年同博士課程単位取得後退学．理学博士．筑波大学計算機センタ準研究員，同大学電子・情報工学系助手，講師を経て，現在，同助教授．コンピュータアーキテクチャ，オペレーティングシステム，プログラミング言語処理系に興味を持つ．ソフトウェア科学会，IEEE, ACM 各会員．