対話型インタプリタ向きプログラムの内部表現と実行アルゴリズムの設計

闞 暁 薇† 板 野 肯 三††

解析木を実行するインタプリタの実行効率の向上を目指して，実行時に必要な情報の一部をコントロールグラフで表現して解析木に付加した“実行木”と呼ばれる新しいプログラムの内部表現を考案した．この実行木は，内部に解析木のデータ構造を内包しており，インクリメンタルパーサのための内部表現を兼ねている．本論文では，実行木の基本概念を提案し，その具体的な適用例としてC言語向きの実行木の設計と実行アルゴリズムを説明する．また，内部表現と実行の効率について評価を行った結果についても示す．

Design of the Internal Representation for a Program and the Interpretation Algorithm for an Interactive Interpreter

Xiaowei Kan† and Kozo Itano††

In order to enable the efficient interpretation based on a parse tree, a new concept “execution tree” has been proposed as an internal program representation attaching the control graphs produced from the semantic analysis. The execution tree is an extended data structure based on a parse tree which is also manipulated by the incremental parser. In this paper, the execution tree and relating interpretation algorithm are disclosed for C language. The representation cost and interpretation efficiency are also evaluated.

1. はじめに

対話型のプログラミングシステムを実現する一手法として，我々はプログラムを解析木の形で表現し，プログラマの指示に応じて部分的な修正と対話的な実行を行うことのできる編集実行系の設計と実現に関する研究を行ってきた．研究の初期の段階で実現されたプログラミングシステムCOSMOS[1)-4)]では，実行系であるインタプリタがソフトウェアで実現されていたため実行が低速であるという問題があった．そこで，高速な実行系を実現するために，インタプリタのハードウェア化を目指して研究を開始した．

この研究の過程で，種々の工夫を繰り返した結果，実行の対象になるプログラムの内部表現が高速な実行に最も大きな影響を与えることが改めて判明した[5)-8)]．COSMOSのインタプリタにおいても高速化のために種々の工夫を行っており，また，PL/0用に設計したハードウェアの解析木インタプリタPATIE-0[5),6),8)]においても多くの細かな工夫を行ったが，その大部分はアドホックなものであった．そこで，今回は，“すっきりした”形でアルゴリズムを表現することも目標とし，プログラムの内部表現とインタプリタの実行アルゴリズムを新しい枠組みで構成し直すために，解析木を内包している“実行木”と呼ぶデータ構造を考案した[7)]．実行木は実行時に必要な情報の一部をコントロールグラフで表現して解析木に付加し，効率よく実行できるように拡張したプログラムの内部表現である．解析木としてのデータ構造は内包されているので，プログラムの部分再解析ができるインクリメンタルパーサと部分実行ができるインタプリタの両者を矛盾なく自然に統合して設計することが可能である．

このような言語処理系を設計する手法としては，属性文法[9)]などのように構文規則単位で実行すべきセマンティクスを設計していくことが考えられるが，属性文法でインタプリタを定義しようとすると，構文規則に対して大域的なデータを使用できず，インタプリタの実行時の性能を上げられないことが問題となる．このため，我々は，構文規則単位でインタプリタのアル

† 筑波大学研究協力部
Department of Research Development, University of Tsukuba
†† 筑波大学電子情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba

ゴリズムを定義していくという枠組みは属性文法と同様な方式を使用し，動的意味の実現は大域的なデータを使用できる方式を採用してきた[2)~7)]．

今回設計した方式は，ハードウェアのインタプリタで実行することを具体的な目標としたプロジェクトの中で開発されたが，原理的にはソフトウェアで実現するインタプリタの場合にも適用可能な方式である．この意味で，実行木の概念は，インタプリタの実行アルゴリズムの実現がハードウェアで行われるかソフトウェアで行われるかという違いを越えて一般性を持っていると考える．

本論文では，実行木の概念について詳しく説明し，C言語に対して適用した例をもとに実行木の生成と実行アルゴリズムを示し，表現と実行の効率に関する評価を示す．

2. 実行木の概念

実行木は解析木を内包しており，実行の前に意味解析によって，実行時に必要な情報を数値や木構造などさまざまな形式で表現し，解析木に埋め込んで，解析木と一体化したデータ構造である．この実行木を定式化するために，言語の構文規則，意味解析のアルゴリズムである意味規則，インタプリタのアルゴリズムである実行規則の3つを一元的に定義するための枠組みとして，構文規則の定義の枠組みを拡張した“実行木規則”を考案した．

2.1 実行木規則

実行木規則は，対象言語の構文規則に，意味規則や実行規則を規定するのに必要な拡張記号や拡張記号だけで構成した拡張規則を付加することによって定義する．例えば，

```
A   ::= B C   ……(1)
B   ::= D     ……(2)
C   ::= H $A……(3)
D   ::= E $F……(4)
$F  ::= $G $J ……(5)
H   ::= I J   ……(6)
```

のような実行木規則があるとき，規則中の$のついていない記号は構文を構成する記号であり，$で始まる記号が拡張記号である．ここで，規則(1),(2),(6)は純枠な構文規則であるが，規則(3)と(4)は構文規則に拡張記号 $A，$B がそれぞれ付加されている．規則(5)は，拡張規則である．この拡張記号に使う記号は，構文規則の中で定義されている記号と重複してはならない．

実行木規則の右辺に現れる拡張記号は，左辺に同名の記号が定義されていないときには，拡張規則で定義する．例えば，上の例では，(3)の $A は(1)のAで定義されており，(4)の $F は(5)の $F で定義されている．意味規則の中では，右辺の拡張記号に対しては，実行用の木構造を生成するアルゴリズムを定義するが，実行規則では，拡張記号を通常の構文規則の記号とは全く区別しないで，アルゴリズムの定義を行う．

2.2 実行木の生成

実行木の生成は，次のような手順で行う．まず，実行木規則中の構文規則部分に基づいて，プログラムを構文解析し，規則ごとにノードを生成して解析木を作成する．次に，この解析木に対し，意味解析を行う．意味解析は規則ごとに行い，その結果は，解析木中のその規則に対応するノードに保存していく．解析木のノードに対応する規則に拡張記号が存在する場合は，意味解析の結果に基づいて，実行時に便利なように必要なノード間を直接接続する．場合によっては，左辺が拡張記号であるような実行木規則を用いて，新たなノードを作成してこれを介して接続することもある．**図1**に，実行木の例を示す．ここで，実線で示されているリンクは，解析木としての構造を規定しているが，点線で示されたリンクは意味解析の結果によって作成されたリンクであり，実行木規則の中の右辺の拡張記号に対応している．この点線で表示されたリンクは，木構造を逸脱しており，グラフ型のデータ構造になる．

2.3 意味解析の方針

言語のセマンティクスはデータ構造や制御構造などに関係しているので，ここでは，これらに関する具体的な意味解析の方針について説明する．

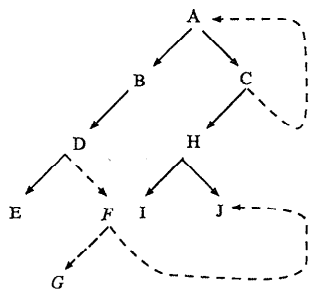


図 1 実行木の表現の例
Fig. 1 An example of the representation for an execution tree.

（1） データ構造

データ構造に関する情報としては，変数の型，割り当てられる領域のサイズ，メモリ上の位置などがある．これらを意味解析の結果として保存する場合，シンボルテーブルを用いる方式[10]や，さらに高速化のためにハードウェアのテーブルを用いる方式[11),12)]などがある．一方，参照側のノードの中に保存しておくという方式もありうる．しかし，本方式では，これらの情報は定義ノードに置き，参照ノードから定義ノードへリンクを張って，常に，定義ノードを通してアクセスを行う方式[2),3)]を継承することにした．ここで，参照ノードから定義ノードへリンクを張る処理が，拡張記号を実行木の中に展開する処理に当たる．

実行時の効率のみを考えれば，実行時の情報をなるべく参照ノード側に展開しておいた方が有利である．しかし，この方式では，対話的なデバッグのときに実行される，ブレイクポイントなどのデータのアクセスに関する“トラップ”の設定を行うのに，関係するあらゆる参照ノードにこの情報を分配しなければならない．しかし，本方式のように常に定義ノードを通してアクセスすれば，このような煩雑さを避けることができる．

定数値に関しては，変数へのアクセスメカニズムと同じになるようにするために，定数領域の中に置くこととし，実行木のノード中に直接には保存しない．構造体などの場合には，メンバに関する情報は，構造体の定義部にまとめて置き，これに参照側からリンクする方式をとる．したがって，データの参照時には，必ず，そのデータの定義ノードを経て参照が起こる．

（2） 制 御 構 造

制御構造のうちループ型のものに関しては，ループの本体の実行終了時に，制御構造の先頭に戻ることが必要である．この実現としては，分岐先のノードのアドレスを木の中に埋め込んでおく方法が考えられる．他の選択としては，制御構造の先頭のノードで戻り番地をスタックに動的に保存し，戻るときにはこれを使用するというやり方が存在する．本方式では実行のメカニズムをなるべく単純にするために，前者の方式をとることにした．C 言語での while 文での例を以下に示す．

```
while    ::= while expr while1
while1   ::= stmt $while
```

この中で，while は終端記号，2番目の規則の最後に現れる $while は拡張記号，その他は非終端記号である．この規則に基づいて生成される実行木は**図 2** のようになる．ここで，拡張記号 $while はループの先頭へのリンクとなる．この例では，実行木規則が2進木を生成するようになっているが，このことは特に本質的な制限ではない．

他の制御構造としては，条件分岐やループの継続や脱出のためのメカニズムが存在する．これらの制御構造では，コントロールの流れがそれらを含む上位の制御構造から外れないので，その実現は比較的容易である．しかし，goto などの一般の制御構造を取り扱うにはさらに複雑な方式が必要であり，これに関しては，次章で詳述する．

（3） 手続きと関数

手続きや関数に関しては，データと同様に，呼び出し側から定義側に対してリンクを張る．このリンクが拡張記号に相当する．関数の返すデータの型，形式パラメータの数や型に関する情報は，関数の定義側に保存する．特に，関数の場合は，データとしての定義と参照という側面と，制御構造という側面の両方を持つが，ここでは，これらが共通の枠組みで表現されている．

2.4 実行のメカニズム

実行のメカニズムは，以前に設計された PL/0 のインタプリタ[5),6)]と基本的に同じ方式を使用する．インタプリタの実行アルゴリズムは，各実行木規則ごとに定義されており，実行中の状態は各規則に対して大域的なデータとして保持する．手続き呼び出しに伴う局所変数を保存するためのデータフレームなどに関しては，プログラムをコンパイルしてから実行する方式と基本的な違いはない．

コンパイルしてから実行する方式との間の最も大きな相違点は，実行木をトラバースするメカニズムが必要な点である．このトラバースは，あるノードの実行中に，そのノードの子ノードの実行を行うという点で再帰性を持つので，その実現にはスタックが必要である．このためのスタックを“トラバーススタック”と

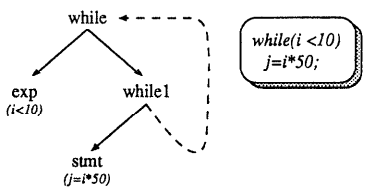


図 2 while 文の実行木の例
Fig. 2 An example of the execution tree for a “while” statement.

呼ぶ．具体的には，このトラバーススタックには，ノードの実行中の状態が保存される．例えば，図1に示すような実行木におけるノードAを実行する場合では，子ノードBの実行を行う前に，子ノードCへのポインタと中断された親ノードAの実行状態がトラバーススタックに保存される．

3. C言語用の実行木の設計

実行木の概念に基づいて，具体的にC言語の実行木とこの実行木を実行するインタプリタの実行アルゴリズムを設計した．C言語の言語仕様としては，ANSI-C[13]をもとにした．ハードウェアのインタプリタPATIECを実現するときは2進木である方が良いが[5),6)]，ソフトウェアで実現する場合には，このような制限は特にないので[2),3)]，ここでは一般的な立場で説明を行う．

3.1 制御構造

C言語の制御構造としては，条件分岐，ループ，ジャンプなどがあり，これらの中から代表的なものを選んで説明する．

（1） 条件分岐

C言語の条件分岐の制御構造としては，ifとswitchが存在する．ifは以前に設計したPL/0言語とほぼ同様に実現できるので，ここでは，switchの実現について説明する．

ANSI-Cの構文に忠実にswitchの解析木を作ると，**図3**の実線で示すような木となる．ここで問題となるのは，C言語では，caseが一種のラベル付きの文であり，制御構造として定義されていない点である．したがって，この木をそのまま実行しようとすると，条件に該当するラベルが見つかるまで木を順次トラバースしていくしかない．そこで，switch中のcaseを抜きだして，独立のデータ構造を作成して付加し（図3の破線で示した部分木），実行時にはこのデータ構造をトラバースすることによって，不必要な解析木のトラバースをバイパスする方式を考案した．

このための実行木規則は次のようになる．

```
switch      ::= switch ( expr ) stmt
                                  $caseeval
$caseeval   ::= $stmt_L $caseeval
              | $stml_L1 $caseeval
              | ε
```

図3の破線で示されている部分木は，意味解析時に，$caseeval規則に従って作成し，switchノードの拡張記号$caseevalの位置からリンクする．

（2） ループ

C言語では，ループに関する制御構造は，while, do-while, forの3種類がある．whileやdo-whileに関しては，PL/0で設計したwhileに関する実現とほぼ同様なので，ここでは，forに関する設計について説明する．forの実行木規則は，

for ::= for (exp1 ; exp2 ; exp3) stmt $for

のように定義できる．ここで，アンダラインで示されている名前は，終端記号を表す．exp1, exp2, exp3は構文的にはすべて同じ式expであるが，説明のために，区別して表現してある．$forは拡張記号である．

2進木にしたいときは，実行木規則を細分化し，定義し直せばよい．右辺の各非終端記号の評価の順序は，exp1を実行した後，exp2, stmt, exp3をこの順序で繰り返す．この順序がなるべく木に反映される

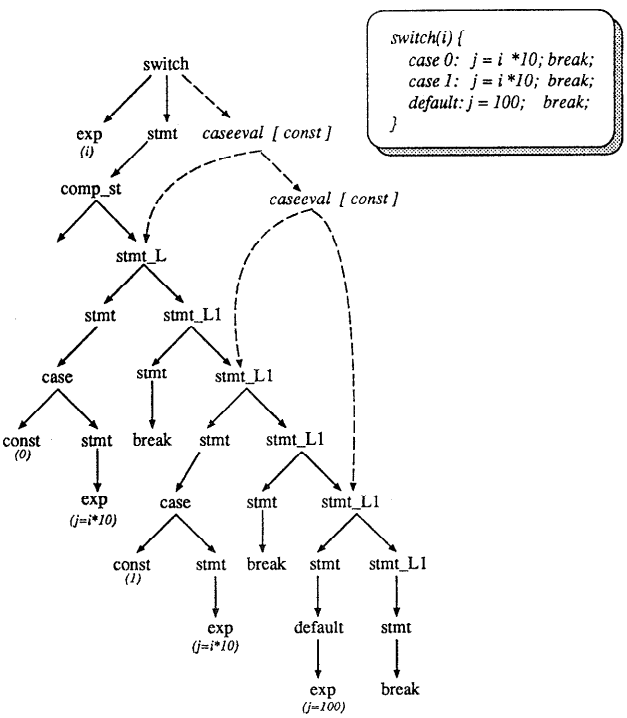


図3 switch文の実行木の例
Fig. 3 An example of the execution tree for a "switch" statement.

方がよいので，次のように実行木規則を定義し直すのが最善と思われる．

```
for   ::= for ( exp1 ; for1
for1  ::= exp2 ; for2
for2  ::= for3 ; stmt
for3  ::= exp3) $for1
```

図4に，for に対応する実行木の例とその実行順序を示す．矢印上の数字はトラバースの順番を示し，破線はループを実現するために，意味解析のとき作成したリンクを示す．

（3） ジャンプ

実行木を木構造に沿ってたどらないで直接特定のノードにジャンプすると，トラバーススタックの一貫性を保持できなくなるので，ジャンプする前にトラバーススタックの状態を調整する必要がある．break や continue の場合，木の上方へしかジャンプしないので，意味解析の時点でトラバーススタックからポップすべきスタックエントリの数に対応した“ポップカウント”を計算し，break や continue のノードに保存して，実行時にはポップカウントの数だけトラバーススタックをポップすることにした．

goto の場合は，break や continue と異なり，木の上方にジャンプするとは限らないので，ポップカウントを使うだけでは実現できない．そこで，木の上方へジャンプするメカニズムをポップカウントで実現し，これに木を下方にたどるメカニズムを組み合わせて goto によるジャンプを実現することにした．

木を下方にたどるメカニズムとしては，**図5**に示すように，トラバーススタックに積むべきポインタを埋め込んだノード $downtrv を必要な数だけ生成して goto ノードの下に付加する方式を採用した．これを生成するための実行木規則は，

```
goto       ::= id $downtrv
$downtrv   ::= $downtrv $stmt_L1
             | $stmt $stmt_L1
             | $stmt
```

のように定義できる．図5の例では，ポップカウントの値は1であり，まず，＊で示した stmt_L1 ノードの状態でスタックをポップし，goto ノードの下の2個の $downtrv ノードをたどることにより，必要なポインタがスタックに積まれる．

（4） ライブラリ

Cのライブラリに関しては，関数呼び出しの意味解析を行うときに，ライブラリであるかどうかの判定を行う．ライブラリであれば，専用のノードに変換する．そして，このノードが実行されたときに，ホストの計算機システムに割り込みを発生して，ライブラリの実行を行う方式を前提としておく．

3.2 データ構造

制御構造に比較すると，データ構造と演算に関するノードや部分木は実行される頻度が高いので，実行時の効率が低下しないように，実行木の設計には注意が必要である．

（1） 変　　数

C言語では，変数は大域変数と局所変数の2種類し

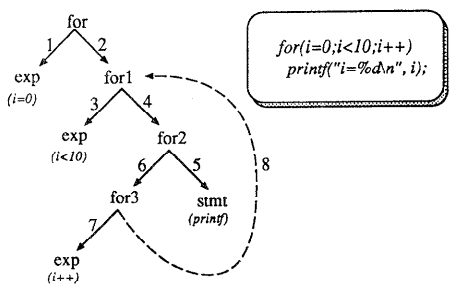


図4 for 文の実行木の例
Fig. 4 An example of the execution tree for a “for” statement.

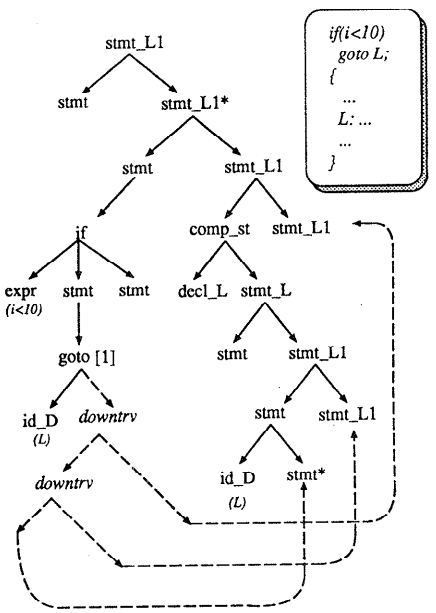


図5 goto 文の実行木の例
Fig. 5 An example of the execution tree for a “goto” statement.

かない．関数の呼び出しに応じて実行時に変数の値を保存するデータフレーム（励起レコード）は，この2種類のものが同時にはアクセスできればよいので，コンパイラが実行環境として前提にしている一般的な手法をそのまま使用する．

変数に関して意味解析時に行う処理は，実行木の中の変数の参照ノードから定義ノードへリンクを作成することと，定義ノード内に参照のときに必要なデータフレーム内のオフセット値や型，その変数の占める領域の大きさなどを保存することである．この具体的な例を**図6**に示す．この例に対応する実行木規則は，

```
comp_st      ::= { declarat_L stmt_L }
      ...
direct_D     ::= id_D |...
id_D         ::= id
postfix_exp  ::= primary_exp
               | postfix_exp ++ | ...
primary_exp  ::= id_R  |...
id_R         ::= id $id_D  |...
```

のように定義できる．

配列の場合は，定義ノードに配列のサイズと要素のサイズの両方を設定することを除いては，単純変数と同じ意味処理が行われる．実行時に，配列が参照されたとき，インデックスと要素のサイズからアドレスの生成を行う．2次元以上の配列の場合には，要素のサイズとして1つ下位の次元の配列の占める領域のサイズが設定され，他は同様の処理を行う．

変数がポイント型であっても，同様の処理が可能であり，特に問題は起こらない．

実行時に，何れにしても，参照ノードから一度定義ノードへトラバースして，必要な情報を入手してからメモリへのアクセスを行う．

（2） 構 造 体

C言語の構造体に関する文法は，型を定義する部分，実行を定義する部分，参照を行う部分に分けられる．**図7**に構造体の宣言と参照の簡単な例を示す．この例に対応する実行木規則は，

```
comp_st    ::= { declarat_L stmt_L }
        ......
direct_D   ::= id_D | smb_id_D | ...
id_D       ::= id
sub_id_D   ::= id
        ......
assignment_exp ::= unary_exp
                 | postfix_exp . id_R
        ......
postfix_exp    ::= primary_exp
                 | postfix_exp . id_R | ...
primary_exp    ::= id_R | ...
```

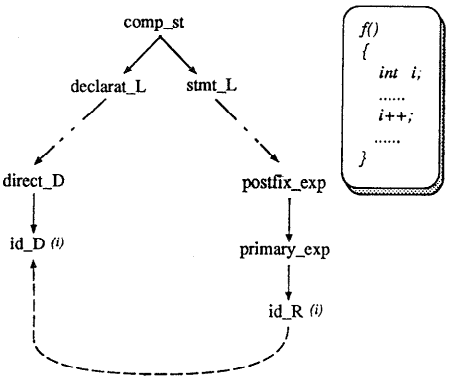


図 6 変数の実行木の例
Fig. 6 An example of the execution tree for a variable.

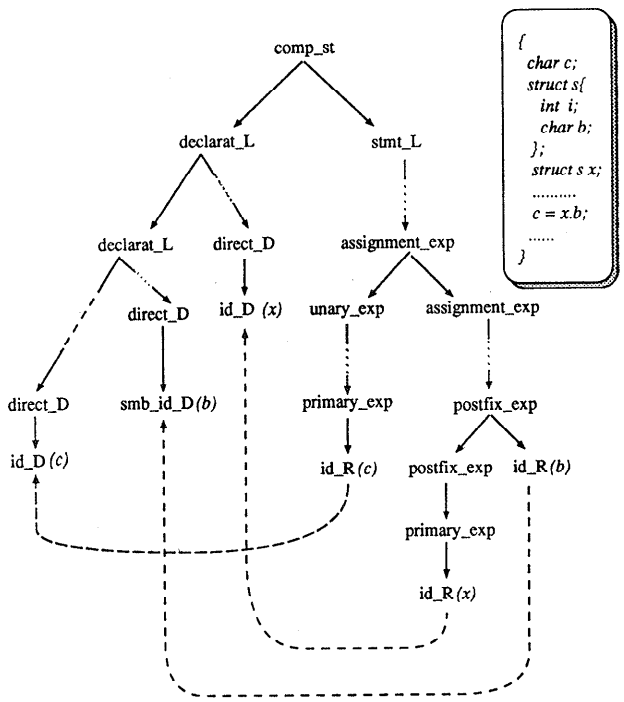


図 7 構造体の実行木の例
Fig. 7 An example of the execution tree for a structure.

```
id_R        ::= id $id_D
              | id $smb_id_D | …
```

のように定義される．

意味解析時に，型を定義する部分では，各メンバのオフセットを計算して，対応するメンバの定義ノードに型や割り当てられる領域のサイズとともに設定する．実体を定義する部分では，データフレーム中での構造体のオフセットを計算して，実体の定義ノード内に，構造体であることを示す情報と，この構造体に割り当てられる領域のサイズを設定する．構造体を参照するノードは，実体の定義ノードへのリンクを行い，メンバの参照ノードは，型を定義しているメンバの定義ノードにリンクを行う．このような方式をとることで，参照ノードを定義ノードにリンクするという単純変数で使用されている方式を統一的に使用することができる．

実行時には，構造体が参照されると，まず，実体の参照ノードから実行の定義ノードへ飛び，データフレーム中でのオフセットを得る．次に，メンバの参照ノードから対応するメンバの定義ノードへ飛び，構造体中でのメンバのオフセットを得て，これらを加えて，メンバのデータフレーム中でのアドレスを計算しメモリにアクセスする．

4. 言語処理系の実現と実行木の評価

実行木によるプログラムの具体的な表現と実行時の効率を評価するために，C言語を対象とした言語処理系を試作した．インタプリタの実現をハードウェアで行うかソフトウェアで行うかで実行木の最適な表現形式が異なるが，以前に実現したCOSMOSではC言語のソフトウェアでの実現に関してある程度の経験を得た．そこで，今回は，ハードウェアでの実現を試行した場合で評価を試みることとした．

（1） 言語定義

C言語の定義は，基本的には3章の方針にもとづいて行ったが，ハードウェアの場合には，2進木の方が高速に実行できるインタプリタのメカニズムが設計しやすいので，2進木での実現を試みた．このために，構文規則の中には右辺の非終端記号を2個に制限して，構文規則の中には右辺の非終端記号を2個に制限して，構文規則を定義し直した．ANSI-Cの場合は，2個以上ある規則が5個あり，これらを変形した結果，構文規則の数は203個となり，もとのANSI-Cの定義から14個増えた．実行規則を定義するために導入した拡張記号は3個であった．

（2） サンプルプログラム

実行速度を評価するために，クイックソート（sort(200)），整数の配列の乗算（matrix(30＊30)）とアッカーマン関数（ackerman(3, 4)），C言語用の字句解析器（scanner）の4つサンプルプログラムを使用した．

（3） パーサと意味解析

言語定義からパーサとインタプリタを生成することも将来の課題として考えているが，現時点では，それぞれ独自に実現を行った．実行木を生成する言語処理系は，パーサと意味解析器から構成されている．ANSI-C[13]の中で左再帰性を持つ構文は，左再帰を除いた構文に変形し，これに基づいてLLで構文解析を行う．これに対して，木は変形前の構文に基づいて左下がりの木を生成する．このため，右下がりの木と左下がりの木を混在させることが可能となっている．また，このパーサの中で構文解析中に，解析木の圧縮を同時に行う．圧縮のアルゴリズムは，基本的にはCOSMOSで用いられているもの[14]と同様の手法を用いた．

（4） 性能の測定と評価

実行木の表現を，上記のサンプルプログラムで測定した結果を**表1**に示す．この結果から分かるように，木は17%から24%程度に圧縮できた．また，実行木の表現の全体の中で拡張ノードの割合は1%であり，拡張ノードの導入による表現コストはほとんど無視できる程度である．一方，実用的なプログラムと思われるC言語の字句解析器の例では，実行木中でインタプリタのアクセスの対象になるノードは約8割程度であり，実行されていないノードは，宣言に対応する部分である．

インタプリタのシミュレータは，C言語を用いてレジスタ転送レベルで実現した．クロックを20 MHzとして実行時間をクロック数から見積ったところ，同じ

表1 内部表現の測定
Table 1 Measurement of the internal representation.

単位：ノード数

	sort	matrix	ackerman	scanner
圧縮前	716	678	417	38441
圧縮後	162 (22.8%)	165 (24.3%)	91 (21.8%)	6694 (17.4%)
拡張ノード	0 (0%)	0 (0%)	0 (0%)	87 (1.2%)
実行されるノード	92 (56.8%)	95 (57.6%)	76 (83.5%)	5867 (87.6%)

表 2 実行速度の測定
Table 2 Measurement of the exectuion speeds.
unit: msec

	sort	matrix	ackerman	scanner
Patie_0	219	235	56	—
PATIE_C	269	248	48	50
GCC†	76	142	81	15
CC†	71	127	82	16

†: sparc1 上での実行

プログラムをコンパイルして sparc1 上で実行した結果と比較して 1/3 程度の性能を得た．この結果を**表 2** に示す．ここで，PATIE0 と PATIEC はそれぞれ PL/0 と C 言語用のハードウェアのインタプリタである．PATIE0 の性能評価[5]のときと同様に，アッカーマン関数の実行は sparc1 の方がレジスタウインドウのスワップのために低速となる傾向が見られた．関数の再帰呼び出しがそれほど多くない場合には，sparc1 の方がやや高速であった．

5. おわりに

解析木をベースにしたインタプリタのためのプログラムの内部表現として実行木という概念を提案し，具体的な実行木の生成のアルゴリズムと実行木のインタプリタの設計を行った．今回提案を行った実行木という概念は，インタプリタの設計を整理する上で大きな効果があった．

具体的な言語処理系の実現はC言語を対象として行ったが，C言語の言語仕様のうち，プリプロセッサは今回の実現から除外した．また，ビット演算，キャスト，浮動小数点数の演算などの一部の演算の実現が現時点では完了していないが，設計の基本的な部分を評価するのには本質的な影響はなかったと考えている．実行木によるプログラムの内部表現とインタプリタの実現方式は，特にC言語の構文やセマンティクスに依存しているわけではないので本手法の一般性は高いと考えている．

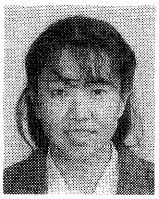
内部表現の空間コストや実行時の性能に関しては，まだ，十分な最適化を行っていないので，まだ，改良の余地が残されている．また，本方式ではインタプリタも含めた言語定義からパーサやインタプリタを自動的に生成する生成系の実現が可能であり，この生成系の実現を含めて，今後の課題としていきたいと考えている．

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筑波大学大学院博士課程の西山博泰氏に感謝します．

参考文献

1) 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).
2) 佐藤 豊，板野肯三：COSMOS における構造エディタおよびソースコード・インタプリタの実現法，第 31 回情報処理学会全国大会論文集，1f-7, pp. 447-448 (1985).
3) 佐藤 豊，板野肯三：構造エディタとインタプリタの統括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. 39-50 (1985).
4) 佐藤 豊，板野肯三：C 言語指向構造エディタ SSE，(原田編) 構造エディタ，pp. 59-70，共立出版 (1987).
5) 闕 暁薇，板野肯三：構文木インタプリタ PATIE0 のアーキテクチャ，情報処理学会研究報告，89-ARC-74, pp. 1-8 (1989).
6) 闕 暁薇，板野肯三：解析木インタプリタ PATIE0 のアーキテクチャ，情報処理学会論文誌，Vol. 32, No. 9, pp. 1113-1121 (1991).
7) 闕 暁薇，板野肯三：ハードウェアインタプリタ向きの解析木の形式と実行アルゴリズムの設計，情報処理学会研究報告，92-PRG-6, pp. 61-70 (1992).
8) 酒井 仁，板野肯三：構文木インタプリタの LSI 向き構成法，電子情報通信学会研究報告，CPSY 91-3, pp. 25-34 (1991).
9) Vogt, H. H., Swierstra, S. D. and Kuiper, M. F.: Higher Order Attribute Grammars, *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 131-145 (1989).
10) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers—Principles, Techniques, and Tools*, Addison-Wesley (1986).
11) Itano, K.: PASDEC: A PASCAL Interactive Direct-Execution Computer, *Proc. Inter. Conf. on High Level Language Computer Architecture*, pp. 161-169 (1982).
12) Itano, K. and Sato, Y.: Architecture of the Universal Direct-Execution Computer UDEC, *Proc. of the Hawaii Inter. Conf. on System Sciences*, pp. 264-273 (1987).
13) Kernighan, B. W. and Ritchie, D. M.: *The C Programing Language*, the 2nd edition, Prentice-Hall (1988).
14) 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).

(平成 4 年 4 月 28 日受付)
(平成 5 年 1 月 18 日採録)



關　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．1987 年～92 年筑波大学大学院博士課程工学研究科単位取得後退学．現在筑波大学理工学研究科準研究員．言語処理系，プログラミング環境，コンピュータアーキテクチャに興味を持つ．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．