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

闕　暁薇
筑波大学工学研究科

板野 肯三
筑波大学電子情報工学系

アブストラクト

C言語の解析木を直接実行するハードウェアインタプリタを実現するために、解析木の内部表現と実行アルゴリズムを設計した。解析木を高速に実行するために、解析木を部分的に変形し、解析木のオリジナル構造を保持する実行木を生成する。実行木には、データの参照から定義への結合や、ジャンプなどの制御フローのバイパスなどが含まれる。この方式に基づいて、プロトタイプをインプリメンテーションし、表現の空間コストや実行時の性能等について、評価を行った。

Design of the Parse Tree Representation and the Interpretation Algorithm for a Hardware Interpreter

Xiaowei Kan
Doctoral Program in Engineering,
University of Tsukuba
Tennoudai 1-1-1, Tsukuba-shi,
Ibaraki-ken, 305 Japan

and

Kozo Itano
Institute of Information Sciences and Electronics
University of Tsukuba
Tennoudai 1-1-1, Tsukuba-shi,
Ibaraki-ken, 305 Japan

Abstract

In order to realize a hardware interpreter to execute a parse tree of C language directly, we designed the internal representation of the parse tree and its interpretation algorithm. To accelerate the execution, the parse tree is partially reformed into an "execution tree", preserving its original parse tree structure. The execution tree comprises the bindings of data references to their definitions, and the bypasses of control flow of jump constructs. We also evaluated the space cost of the representation and execution performance at run time based on the prototype implementation.

１．はじめに

対話的なプログラミングシステムを実現する一手法として、プログラムの内部表現を解析木の形で行い、実行時にプログラムを部分的に修正しながら、インタプリタで実行を行う方式の設計と実現を行ってきた。最も初期のバージョンであるＣＯＳＭＯＳプログラミングシステム[4]では、インタプリタをソフトウェアで実現していたが、これを高速化するために、ハードウェアによるインタプリタの実現可能性を簡単な言語ＰＬ／０で評価し、十分な結果が得られた[7,8]。そこで、これらの結果をもとにして、実用的な言語であるＣに関して、解析木のインタプリタの設計を行った。

Ｃ言語のインタプリタを設計する過程では、解析木の各ノードにどのように実行すべきセマンティクスを配分していくかが大きな問題となる。すなわち、実行するという面からみると、解析木は、最適なデータ構造とは言えない部分もあり、もっと実行しやすい形に変形した方がよい場合もあることが分かった。一方、現在、著者らが採用している方式では、実行中のプログラムの部分再解析（インクリメンタルパージング）を行うことを可能にするために、プログラムのメモリ中での内部表現は、基本的には、解析木であることが必要である[4,5,6]。

そこで、プログラムの内部表現が解析木であることを保ったまま、実行時に有利な木構造を付加して、全体としては、内部表現を少し変形してやることを考えた。すなわち、インクリメンタルパーサは、すでに存在するプログラムの内部表現を属性付きの解析木であるとみなして処理を行える必要がある。しかし、同時に、インタプリタは、同じ内部表現を実行用に拡張設計された木であると解釈して実行を行う。この木の拡張された部分は、意味解析の際に、解析木に対する属性をして付加されるが、この属性が単なる値ばかりではなく、木構造になっている部分もあり、この木構造になっている部分を、インタプリタでは、もとの解析木と区別しないで実行する。このインタプリタが実行する木を実行木と呼ぶことにする。

このような方式をとったことにより、Ｃ言語のインタプリタは複雑ではあったが、結果的には、実行アルゴリズムの複雑さの一部を意味解析処理に移すことになり、ハードウェアとして実現されるインタプリタはより単純になったばかりでなく、実行速度も高速化することが可能となった。また、言語処理系の立場からみると、もとの解析木を、一種のコンパイル的な処理によって、実行に適した形に変換しているとみることもできる。

本論文では、Ｃ言語の解析木インタプリタのために設計した、プログラムの内部表現と、Ｃ言語の各種構文やセマンティクスに対応した、意味解析と実行のアルゴリズムを説明する。

２．解析木と実行木

インクリメンタルパージングを可能にするためには、プログラムの内部表現が解析木でなくてはならないという要求がある。一方、実行時のコストを軽減し、かつ、高速化するためには、内部表現は実行に向いた形（実行木と呼ぶ）に変形したいという要求がある。Ｃのインタプリタを設計するにあたって、この２つの要求を同時に満足させることができるように、プログラムの内部表現を工夫した。本節では、まず、この内部表現について説明する。

２．１　解析木

一般に、言語の構文に従って生成された解析木は、抽象構文木などに比較してプログラムの実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、メモリ上の表現コストだけではなく、実行のコストがかなり大きい。したがって、本設計でも、解析木の表現を圧縮する方式[6]をとることにした。意味解析のアルゴリズムや実行のアルゴリズムはノード単位で定義するが、これらのノード単位で定義された意味解析アルゴリズム（意味規則と呼ぶ）や実行アルゴリズム（実行規則と呼ぶ）が存在しないノードのうち子ノードが１つしかないもの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する。また、実行と表現のコストをより小さくするため、終端記号に対応するノードは生成しないことにした。

２．２　実行木

解析木のノードに実行規則を付加していくという方針だけでは、効率よく実行するのには無理があり、また、インタプリタのハードウェアの構成を単純化するのにも問題がある。そこで、必要に応じて実行するのに適した形の木を導入する。解析木のノードと解析木から作り出される実行木のノードの大部分は共通であるが、概念的には、インタプリタによってトラバースされる木が実行木であり、これは、解析木と完全に同じものではない（図１参照）。実行木が解析木と異なる部分は、識別子の参照から宣言部へのリンク（後述）や制御構造に関する工夫などがある。たと

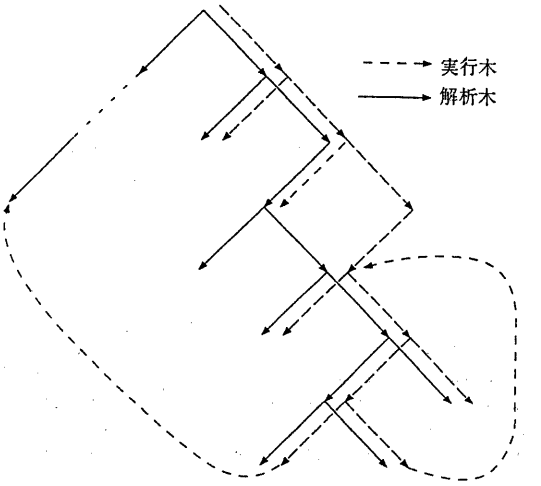

図1. 解析木と実行木

えば、C言語の場合は、switch文の中のcase文がラベルであり、構文として構造化されていないために、実行木としては、switchの構造を導入するなどの工夫をする。

2．3　解析木と実行木の融合

プログラムの物理的な内部表現をできるだけ小さくするために、実行木を独立に作る部分は極力少なくして、解析木をできるだけそのまま使うようにすることにする。したがって、プログラムの内部表現としては、解析木と実行木が共存することになる。大部分のノードは、両方の木から共有されているので、この状態で、一貫性がとれなくてはならない。そこで、同じノードでもパーサとインタプリタで、別の解釈を行うことにした。すなわち、パーサがノードを解釈するときは、対応する構文規則に従うが、インタプリタの場合は、別の解釈をする。たとえば、あるノードに子ノードへのポインタが2つ格納されていたとしても、このノードに対応する構文規則の右辺に非終端記号が1つしかなければ、2番目のポインタは、パーサからは認識されない。一方、インタプリタでは、2つのポインタを両方ともに認識するように設計することができる。解析木の中には、構文の再解析や意味解析の時にはアクセスされるが、インタプリタが実行するときにはアクセスされないノードもあり、また、逆に、実行木にしかないノードもある。現在、実行木のデータ構造は、ループの存在を許しているので、実行木は、厳密には、木構造ではないが、一応、本論文では、解析木とのアナロジで実行木と呼ぶことにする。

3．設計の方針

3．1　構文規則の再構成

まず、実行木のもとになっている解析木に効果的に実行規則を分配できることが望ましいので、解析木が実行に適したものになるように構文規則の再検討を行った。

（1）2進木

まず、ハードウェアインタプリタの実現を容易にするために、PATIE0の場合と同様に、構文規則の右辺の非終端記号の数を2に制限して、解析木を2進木になるように制限した。ANSI Cの構文規則の中には右辺に非終端記号が2個以上ある生成規則が5個あり、制限を満足させるために、これらを定義し直した。

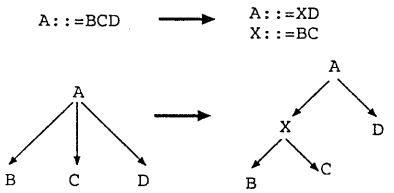

図2. 2進解析木

（2）右再帰性と左再帰性

次に、木のでき方は、構文規則に右再帰性があるか左再帰性があるかで大きく異なるが、これに関しては両者を許すことにした。同じセマンティクスを実行するにも、木の枝の短い方から順に木を辿った方が有利である。

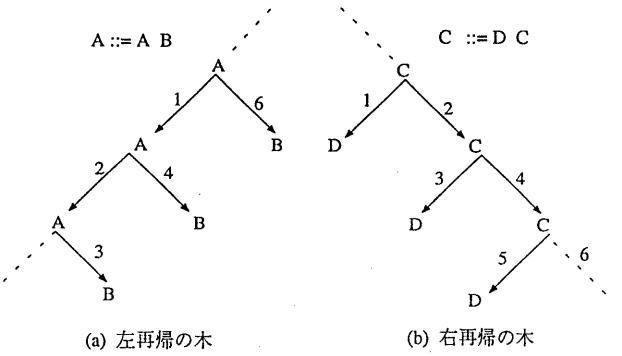

図3. 右再帰の木と左再帰の木

・右再帰性の規則：右再帰性を持つ構文規則に基づいて解析を行うと、図3(b) に示すように、右下がりの木ができ、ソースプログラムで記号が現れた順に実行するように設計したいときは、このタイプの規則を使用する。

・左再帰性の規則：左再帰性を持つ構文規則に基づいて解析を行うと、図3(a) に示すように、左下がりの木ができる。演算子の結合を自然に処理するには、このタイプの規則を使った方が設計しやすい。

この2種類の規則を混用するには、LL（1）パーサでは無理であるが、これまで、パーサには、LL（k）を用いているので、特に、問題は起こらない。

（3）実行規則を定義するための再構成

実行規則を解析木のノードに定義し実行木をつくる段階

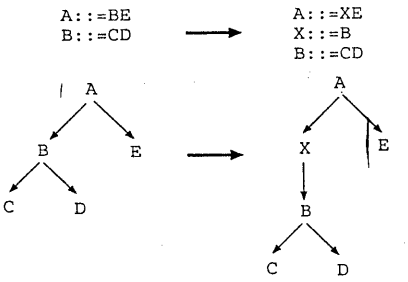

図4. 実行規則を定義するためのダミーノード

で、ダミーのノードが存在した方が設計を行い易い場合がある。このような場合には、構文規則を図4に示すように書き換えて、ダミーノードの挿入が可能にする。図4の例では、構文規則Bの前にダミーのx を挿入した状態を示し

ている。
ダミーのノードを必要とする実行規則は、大きく分けて２種類ある。

・実行時の木のトラバース処理を先行制御するために、フロー制御用の実行規則を処理するハードウェアのメカニズムがなるべく単純になるようにしたい。このためには、１つのノードで１度に処理する仕事を均等に配分したい、また、トラバースとしての基本処理の種類を少なくしたい。このために、ダミーノードを導入して、実行規則の設計を最適化する。

・拡張される実行木をもとの解析木からリンクするための接続点を用意する必要がある場合がある。

これらに関する具体的な例は、４節で詳述する。

３．２　実行木用の付加木

３．１で説明したような構文規則の再構成という手段では、効果的な実行規則が定義できない場合もある。このような場合は、実行すべきセマンティクスを木の形に生成して、属性としてもとの解析木に付加して実行木を作り、これを実行する。この意味で、これは、一種の”コンパイル”を部分的に行っているのと等価であると考えることもできる。

（１）識別子
ソースプログラムの中の変数や定数、関数など名前で、定義と参照が結合されている概念がある。普通のコンパイルでは、これらの識別子は、データの実体への結合に変換される。これに対して、解析木では、定義部分の木が残っているので、参照のノードから、定義のノードへの結合をまず行い、定義ノードを通してデータの実体へと結合する。この結合は、実行木としてのリンクで実現される。この方式をとることにより、データの定義部が実行されることになるので、データの参照のトラップを制御フローのトラップと同一のメカニズムで実現することが可能になる。また、構造化されたデータに関しても、柔軟に対応できる。

（２）ループ
ループを実現するときに最も重要なことは、ループ本体の実行が完了した時点で、如何に効率よくループの先頭のノードへ飛べるかということである。このための方法として、ループの先頭ノードをループ本体を実行する前にスタックに保存し、ループ本体の実行が完了する時にスタックから先頭アドレスをポップしてループを再開するというやり方もあるが、ここでは、意味解析の時に、ループ本体の終わるノードに、ループの先頭ノードへのポインタを格納し、実行する時に、このポインタを用いて、子ノードを実行すると同じアルゴリズムで、ループの先頭へ飛んでループを再開するようにした。ループの実現は、実行木の子ノードへのポインタを使って行うので、この意味では、データ構造が、厳密には、木構造ではなくなっている。
この方法を採用したことにより、スタック操作を必要とせず、ハードウェアの構成が単純になるだけでなく、実行の高速性が得られると同時に、各ノードでの処理も均等に配分され、均整な実行アルゴリズムが得られた。

（３）ジャンプ
通常のマシン命令によるジャンプの場合は、ジャンプ先のアドレスをコンパイル時に決定しておき、ジャンプ命令でそのアドレスに直接ジャンプする。解析木インタプリタの場合は、ジャンプ先のアドレスを決定することに加え、木のトラバースの状態の一貫性を保たなくてはならない。ジャンプ先でのトラバースの状態は、もちろん、そこまでの実行木のノードを実際に一個づつ辿っていけば得られるが、辿るための方法は、通常の木の辿り方とは大きく異なるので実現が容易ではなく、さらに、実行に時間がかかり、ジャンプをしている効果が無くなってしまう。そこで、ジャンプを高速に行う手法として、ジャンプ先でのトラバースの状態を高速に作り出すための木をジャンプする側のノードの下に生成してやることにした。具体的なアルゴリズムは４章で示す。

４．　意味解析と実行アルゴリズムの設計

本設計ではANSI-Cの言語定義［１］をもとにした。実行時の記憶管理には、データフレームスタックを使い、この最初のデータフレームに、staticの変数、広域変数、定数などを割り当て、関数の再帰呼び出し時に割り当てる必要のあるautoの変数などの局所変数は、２番目以降のデータフレームに置くことにした。実行木の各ノードの物理的表現は、２ワードまたは３ワードで表現されている。第１ワードは、構文の生成規則に対応するノードのコードと、二つフィールドc1とc2から構成される。この二つのフィールドには通常第１子ノードと第２子ノードへのポインタをそれぞれ保存するが、場合によって、実行中に必要が属性を保存することもある。第２ワードには、意味解析や再解析を行う時に使用する、親ノードへのポインタが保存される。さらに、必要に応じて第３ワードを追加し、属性を保存する。

４．１　データ構造と演算

（１）変数と定数
３．２節の（１）で説明したような参照ノードと定義ノードを結合する方針で変数を実現した。広域変数と局所変数は別々のデータ領域で管理されているので、実行中に変数のデータ領域中のアドレスを計算するために、広域変数と局所変数を区別する必要がある。これは、定義ノードにフラグを付けて、実行中にこのフラグを検査することによって、区別をすることもできるが、実行を単純にして速度を上げるために、本設計では、実行木としてこの区別をすることにした。具体的には、意味解析の時に、広域変数の定義ノード毎に、新しいノードglobal_id_Dを生成し、定義ノードid_Dの第１子ノードとして付け加える。そして、広域変数を参照しているノードからは、このglobal_id_Dノードへリンクを張る。参照している変数が局所変数の場合に

は、id_Dにリンクする。ここで、global_id_Dノードには、広域変数領域のアドレスを計算する実行規則を定義させ、id_Dには、局所変数領域のアドレスを計算する実行規則を定義させておき、どちらを辿るかで、アドレスの計算処理を分離する。

さらに、定義ノードは実行木の葉ノードであり、子ノードが存在しないので、フィールドc1とc2に、対応する変数のタイプとサイズを格納しておく。なお、データフレームの中のオフセットまたは広域変数領域（最初のデータフレーム）中のオフセットは、オペランドに格納しておく。

実行時に、単純変数が参照されると、参照ノードでは何にもせず、第1子ノードへのポインタを用いて直ちに定義ノードへ飛ぶ。そこで、サイズやタイプに応じて、オペランドに格納されているオフセットを用いて変数のデータ領域中のアドレスを計算し、その値をメモリからフェッチする。フェッチしてきた値は、サイズ、タイプ、アドレスと組にして保持し、参照ノードの親ノードへ戻った時点で参照が完了する。構造体などの参照については、後述する。

定数の場合は、意味解析が異なるだけで、広域変数と同じ方式で実行する。

図5には、変数の定義と参照の木の例を示す。ここで、

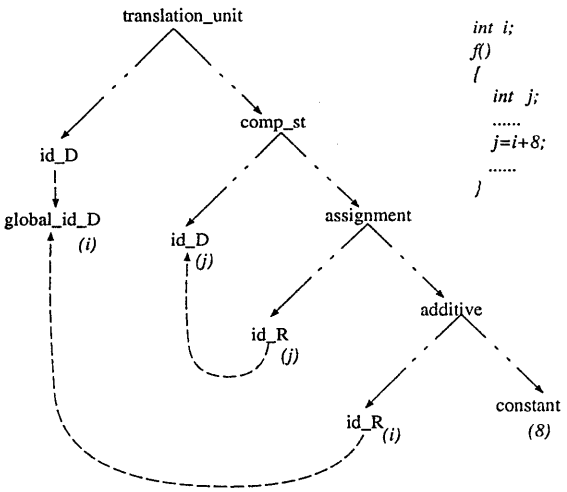

図5. 変数の例

iは広域変数であり、jは局所変数である。意味解析時に、解析木（実線で描かれた部分）上のiの定義ノードid_Dの下に、広域変数のアドレスを計算するための実行規則が定義されているノードglobal_id_Dを付け加えて、iの新しい定義ノードを生成する。iは広域変数なので、iの参照ノードid_Rから、この新しい定義ノードglobal_id_Dへリンクを張る。一方、jは局所変数なので、直接参照ノードid_Rと定義ノードid_Dの間にリンクを張る（これらの破線で描かれているリンクとglobal_id_Dノードは実行木としてのみ存在する）。実行時に変数の参照が起こると、これらのリンクを使って参照ノードから対応する定義ノードへ飛び、global_id_Dかid_Dのいずれかでアドレスの計算を行い、メモリの参照が行われる。

（2）構造体

Cの構造体に関するコンストラクトは、型を定義する部分、実体を定義する部分、参照を行う部分に分かれる。意味解析時に、型を定義する部分では、structの中で各メンバのオフセットを計算し、各メンバの定義ノードに設定する。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、実体の定義ノードに設定する。そして、参照部分のノードからは、定義ノードへのリンクを行う。このとき、参照名の内、構造体名ノードは、実体の定義ノードにリンクし、メンバ名ノードは、型を定義している木の中の対応するメンバのノードにリンクする。

実行時に、構造体が参照されると、実体名からは実体定義ノードへ木を辿り、データ領域中でのアドレスを入手し、メンバ名からは型定義ノードへ木を辿り、構造体中でのメンバのオフセットを入手し、これらを加えて、メンバのデータ領域中でのアドレスが計算される。メンバの型やデータのサイズは、型定義部から返される。

図6には構造体の宣言と参照の簡単な例を示す。この例では、意味解析時に、まず、構造体xは局所的に定義されているので、構造体の名前xは局所変数と同じように、id_Dを定義ノードとして使い、その第1フィールドにタイプ（structである）を格納し、第2フィールドにサイズ（ここではsize(int)+size(char)である）を格納する。オペランドには、データフレームの中にこの構造体のオフセットを格納する。それから、この定義ノードとxの参照ノードid_Rの間にリンクを張る。次に、各メンバの参照を定義に結合するために、メンバbの参照ノードid_Rのから定義ノードsmb_id_Dへリンクを張る。ここでは、メンバのアドレスは構造体名のアドレスにメンバの構造体の中のオフセットを足すことによって計算されるので、メンバbの定義ノードのオペランドには、変数と違って、このメンバが構造体中のオフセットを格納する。第1ワードのc1とc2には、変数と同様にそれぞれbのタイプとサイズを格納する。最後に、cは普通の局所変数として、定義ノードにタイプ、サイズとデータフレーム中のオフセットを格納してから、定義ノードと参照ノードの間をリンクする。

実行時には、まず、代入文c=x.b の右辺値を評価するために、xの参照を行い、データフレームの中の構造体xのオフセットをとってくる。変数の参照との違いは、構造体の名前の参照には値が必要なく、アドレスを計算してくるだけでよい。次に、メンバbの参照を行い、構造体の中のbのオフセットをとってくる。postfix_expノードで・演算を行う。即ち、この二つの値を足す結果をメンバ bのデータ領域の中のアドレスとしてメモリをアクセスし、その値をとってくる。それから、代入先のアドレスを計算するために、変数cの参照を行い、cアドレスをとってくる。最後に、assignmentノードでメンバ bの値をcに代入し、代入文の実行が完了する。

一方、構造体へのポインタで構造体の要素を参照する場合は、上記と大体同じアルゴリズムで行うが、アドレスを計算する時に、定義ノードから取ってきたアドレスを使うの代わりに値を使うという違いがあるだけである（後述の

ポインタの設計を参照）。

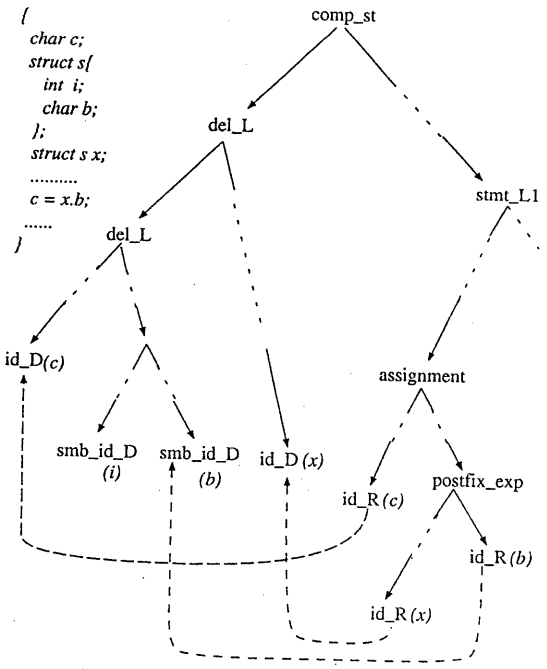

図6. 構造体の例

（3）配列

配列の参照は2ステップに分けて行う。まず、配列名は変数として参照することによって、配列のベースアドレスを計算してくる。次に、要素のオフセットを表す式を評価することによって、該要素のオフセットを計算してくる。そして、この二つの値を加えた結果をデータ領域のアドレスとしてデータ領域をアクセスし、要素の値を読み出して、参照が終了する。

二次元以上の配列の場合、意味解析の時に、アドレスの計算と同様に、次元の大きさを属性として配列参照ノードのオペランドに保存し、実行時に、この大きさに参照要素のオフセットを掛ければ、アドレスの計算が出来る。

（4）ポインタ

ポインタ変数の場合は単純変数と異なり、タイプがポインタとなっている以外に、サイズは、ポインタが指す実体のサイズを保持する。これは、ポインタに関連する演算を行うためときに必要になる。ポインタに対して整数の加減算がなされるときは、このサイズで調整をする必要がある。また、*などの場合は、ポインタ値をアドレスとして使用してメモリのアクセスを行う。

（5）演算子の優先順位

演算子の優先順位と結合規則を実現する方式に関しては2つの方式を考察した。まず、第1の方式は、意味解析時に、解析木の中に優先順位を反映した実行の順番に関する情報を属性として埋め込んでおき、実行時に、この属性に基づいて実行するというものである。第2の方式は、構文のレベルで演算子の優先順位と結合規則を表現しておく方法である。Cのver.7では、演算子の優先順位が構文で表現されていないので、初期のプロジェクトCOSMOSでは前者の方式を採用した。しかし、今回は、ANSI-Cを使うこともあり、構文規則に即してきれいに設計できるという利点も考慮して、後者の方式をとることにした。

4．2　制御構造

制御構造の実現は、実行木をトラバースするメカニズムの実現に深く関係する。インタプリタが実行木をトラバースする時には、その時点で辿らない方の子ノードに関する情報を保存しておく必要がある。これは、実際には、スタックに子ノードへのポインタを積むことで実現できるが、現在のインタプリタは、tail-recusion-optimizationの技法を導入して、不必要な情報を保存することを避けている。以下の説明では、このスタックのことを、トラバース状態スタックとして参照している。

（1）条件分岐

Cの条件分岐の制御構造は、if文とswitch文である。解析木を2進木にするという制限と、実行木を解析木にリンクするための接続点が必要であるため、if文とswitch文を設計する際にも、もとの構文規則を細分化して定義し直した。ここでは、switchの実現に関して詳しく説明する。

ANSI-Cの構文[1]に忠実にswitch文の解析木を作ると、図7の実線で示すような木になる。ここで問題なことは、Cでは、caseは一種のラベル付きの文であり、構文として構造化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いだされるまで、木を順次トラバースしていくしかない。この状況を改善する1つの方法として、case文を構造化した木を意味解析段階で作成し、実行木にすることにした。このために付加された木を図7の破線で示す。

具体的な意味解析処理としては、まず、case文毎に一つのcasetmpというノードを新たに生成する。その第一フィールドには対応するcaseのcaseボディの先頭アドレスを格納し、第二フィールドには、次のcasetmpノードへのポインタを格納し、オペランドにはこのcaseの分岐条件となている定数式の値を格納する。最後のcasetmpノードの第二フィールドには、defualt文へのポインタを格納する。この木のrootノードへのポインタをノードswitchlの第二フィールドに格納する。このようにして、switch文の全てのcase分岐に対応するcasetmpノードを作りだすことによって、switch文中のcaseラベルが構造化される。

実行に関しては、switchノードで、第一子ノードを実行することによって式の値を評価し終わってから、第二子ノードのswitchlへ行く。switchlでは直接第二子ノードへ行って、構造化されたcaseの木を辿りながら、分岐条件のマッチを逐次的に行う。条件に合うノードが見つかると、第一子ノードとなっているcaseボディへ分岐し、caseボーデ

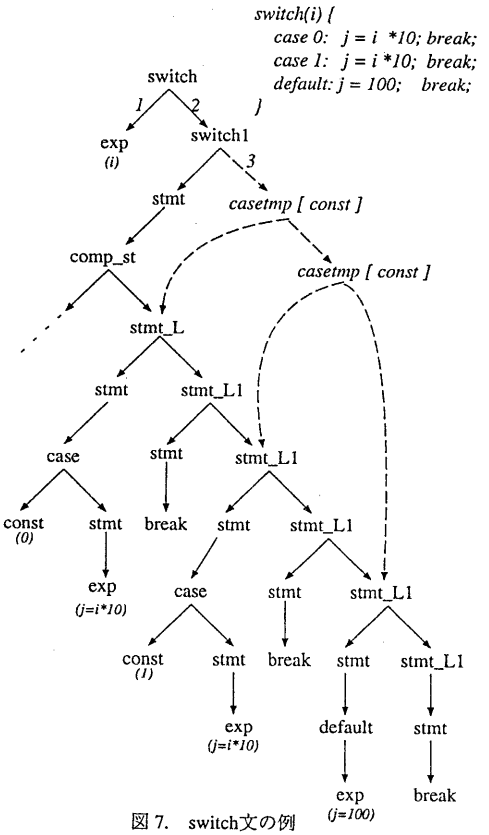

図 7. switch文の例

イを実行しにいく。この方式では、不必要に無関係なcaseのボデイをトラバースする必要がないため、分岐先を決定するために必要なトラバースのパスはかなり短くなった。

（2）ループ

C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に関する実現がほぼそのまま使えるので、ここでは、forに関する設計について説明する。

for文の構文は

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

である。ただし、ここでアンダーラインで示されている名前は、非終端記号を表し、exp1-3は構文的にはすべて式expであるが、説明のために、区別して表現してある。子ノードを２個以下にするという制限から、構文規則を細分化し、定義し直す必要がある。for 文のセマンティクスによると、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ので、この順序がなるべく木に反映されていた方がよい。そこで、次のように構文規則を定義し直した。

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

図８にはfor文に対応する木の例と木の実行の順序を示している。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、意味解析の時張ったリンクを示す。

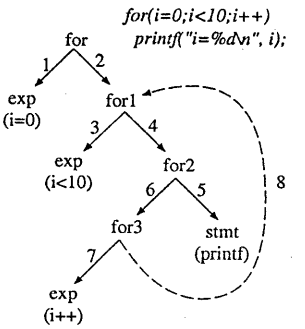

図8. for 文の例

このようなループを作ってよい条件として、トラバース状態スタックの一貫性が保たれていなければならない。この例では、８のトラバースを行う時点でのスタックの内容は、２のトラバースを行った直後の内容と同じになっている。

（3）breakとcontinue

解析木をそのまま辿らないで、直接特定のノードにジャンプすると、一般には、トラバース状態スタックの内容がトラバースの状態を示さなくなる。breakとcontinueの場合は、for とは異なり、ジャンプを行ったときに、スタックの内容に問題が生じる。しかし、breakやcontinueは、木を上方にジャンプするだけなので、トラバース状態スタックの内容を捨てるだけでよい。そこで、意味解析の時点でポップすべきスタックエントリの数をポップカウントとして計算し保存しておくことにした。

図９に、whileのループの中にbreakとcontinueが含まれている木の例と、インタプリタがbreakやcontinueする時点でのトラバース状態スタックの内容を示した。

breakの場合、意味解析時にはbreakノードから木を上に逆登りながら、上から下に辿るときにスタックにポインタをプッシュするノード（この例では、stmt_L1**）に出会ったら、ポップカウントを１だけインクリメントする。これを複文（compound statement）に出会うまで繰り返す。但し、この複文の祖父がループであれば、第一子ノードに余分に１だけインクリメントしなければならない。continueの場合は、終了の条件がループ文（while1,for2）に出会うことである。

breakあるいはcontinueの実行時には、ポップカウントを読みだして、その数だけスタックをポップしてやるだけで

よい。

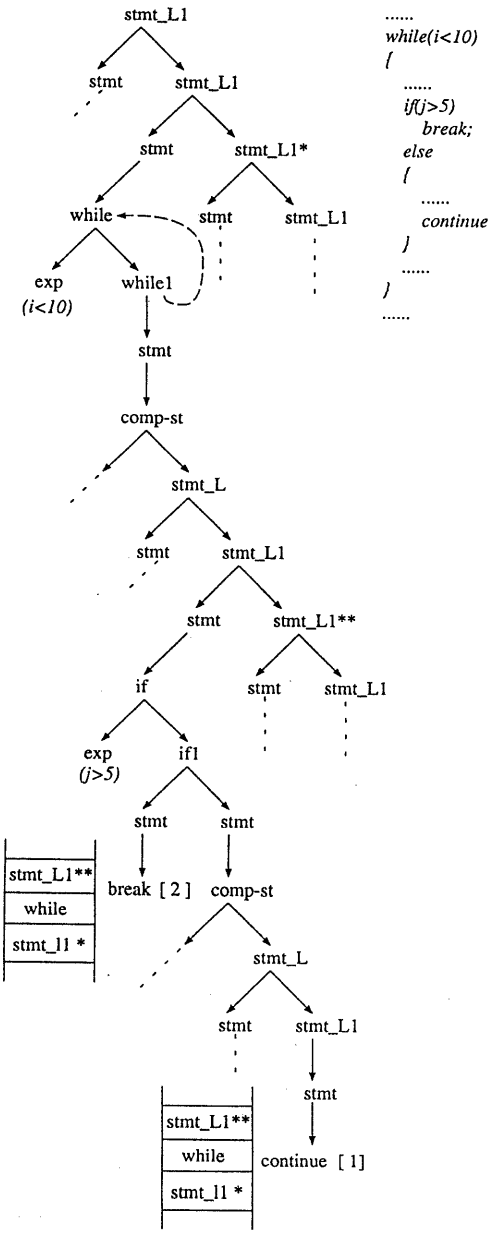

図 9. break文とcontinue文の例

（4）ｇｏｔｏ

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、木を上方にジャンプする機能はポップカウントで実現し、木を下方にジャンプする機能はスタックに必要な情報を積む機能を持つ木をgotoノードの下に生成しておきこれを実行することにして、これらを組み合わせて実行することにした。

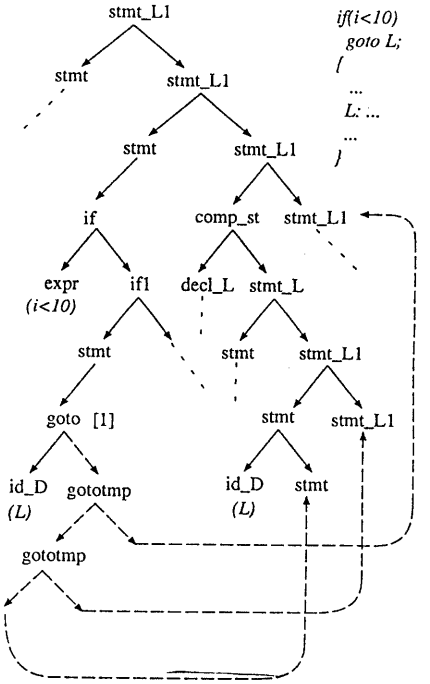

図10. goto文の例

図１０に、gotoの例を示す。ここで説明のために、gotoノードがある木の枝とgotoの飛び先のノードがある枝の合流点を合流点ノードと呼ぶことにする。

意味解析時には、まず、この合流点ノードからgotoノードまでの間のプップカウントを計算して、gotoノードに保存する。次に、合流点ノードから飛び先のノードまでの間をトラバースしたときにトラバース状態スタックに保存するべき情報を調べ、保存する順序で１エントリあたり１つづつgototmpという特殊なノードを生成していく。gototmpは、必要な数だけ生成してリンクし、トラバース状態スタックに保存される情報を子ノードへのポインタとして保存している。最後のgototmpノードには、gotoの飛び先ノードがリンクされている。図１０に示した例では、ポップカウントの値は１であり、付加されているgototempノードの数も１である。

実行時には、まず、gotoノードに格納されているポップカウントの数だけトラバース状態スタックをポップする。次に、gotoに下にあるgototmpを辿って１つづつ実行することによって、トラバース状態スタックに、合流点まで木を逆ってから下りてきたときと同じ状態を作り出す。そして、

最後に目的の飛び先ノードへとトラバースする。

（５）関数

関数の呼び出しとリターンも一種のジャンプであるが、この実現には、他の制御構造の実行のメカニズムに共通な部分と独自の部分があるので、ここでまとめて説明する。

意味解析時には、変数の参照などと同様に、関数の呼び出し用の関数名の参照ノードを関数名の定義ノードにリンクする。このために、関数名の参照ノードの第一子ノードに関数名の定義ノードへのポインタを格納する。関数名の定義ノードでは、実行に無関係なノードをバイパスするために、定義ノードの第２フィールド（ここは解析木で使われていない）に関数の本体となる木（関数のボディと呼ぶ）へのポインタを格納し、また、オペランドにはこの関数のデータフレームのサイズを計算して格納する。

returnに関しては、continueやbreakと同様の方式を使用する。すなわち、returnノードから木を上向きに関数のボディ定義ノードまで逆たどり、ポップカウントを計算し、これをreturnノードのオペランドに保存する。

実行時には、まず、関数の呼び出しノードで、それぞれの引き数を評価し、新しいデータフレームに格納した後、関数名の参照ノードの第１子フィールドを用いて関数名の定義ノードへ飛ぶ。ここで、新しいデータフレームが作成し、第２フィールドを用いて関数ボディへ飛び、木をトラバースしながら関数ボディを実行する。リターン時には、returnノードで、第一子ノードを実行することにってリターン値を評価してから、もとのデータフレームを復元する。最後に、ポップカウントの数だけトラバース状態スタックをポップし、関数の呼び出し側に戻る。

図１１には、関数の呼び出しに関連する木とその木を実行する時のトラバースの順番（矢印とその上の番号）を示す。この例では、returnノードの第一子ノードを実行し終る時点で、トラバース状態スタックの上には、stmt_l1*とstmt_l1**が積まれている。ここで、returnノードに保存されているポップカウントを用いてスタックを２回ポップし、stmt_l1**を次に実行するべきノードとし、関数からのリターンが行われる。

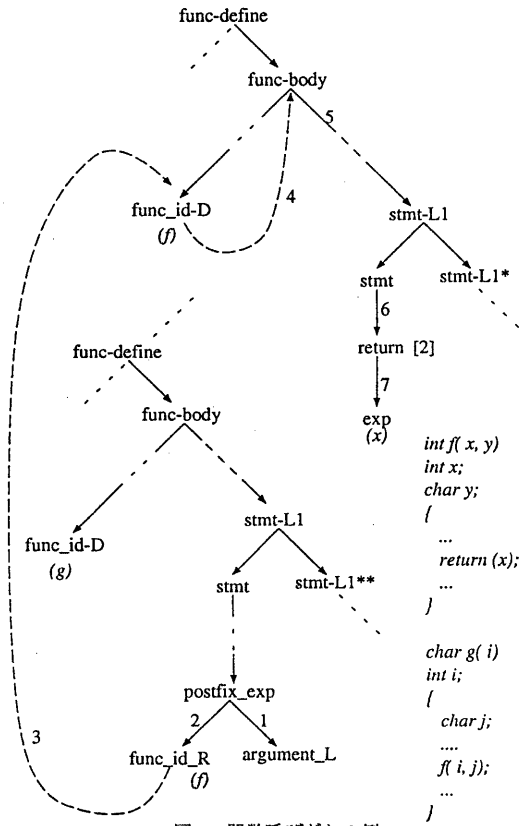

図11. 関数呼びだしの例

５．実現と評価

４章で説明した方式に従って、実際に、Ｃ言語用のプログラム内部表現である解析木と実行木を生成する言語処理系と、この内部表現を実行するハードウェアインタプリタのシミュレータを作成し、アルゴリズムの確認と、プログラムの内部表現のメモリコストや、実行時の性能に関する評価を現在行っているところである。

これらのシステムを実現するにあたっては、まず、言語の定義を、構文規則、意味解析用の規則、実行規則などを一体の形で記述し、これをもとに、それぞれのシステムの実現を行った。現在、Ｃ言語を定義している構文規則の数は、１９２個（解析木のノードの種類の数でもある）あり、もとのANSI-Cでの定義から１４個増えている。また、実行木用に新たに導入したノードの種類は４個である。言語定義からパーサとインタプリタを生成することも考えているが、現時点では、それぞれ独自に実現を行っている。

プログラムの内部表現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。パーサは、ＬＬ（ｋ）で、解析木を圧縮するアルゴリズムは、基本的には、cosmosで用いられている手法と同様な手法を用いた。圧縮率は、いくつかのサンプルプログラムで測定した結果、１７％から２３％程度に圧縮でき、一応十分な結果であると考えている。

内部表現の評価としては、いくつかのレベルで考えなくてはならない。まず、意味を張り付け易くするために、解析木自体にまず、ダミーノードを導入している点、実行用にのみ使われる付加木を付ける点、ノードそのものの表現のコスト、圧縮との関係などである。解析木の中で占めるダミーノードの割合、解析木と実行木の共有率、実行木を作るために付加されたノードの割合などは、それぞれ０．３%、９９%，１%であり、実行木を作るために付加されたノードが全体の表現の中で大きな負担にはなっていないと思われる。全体として、意味を張り付けるのに冗長なノードは圧縮によって取り除き、必要なノードは付け加えるという方針は概ね成功していると考えられる。意味解

析時に木を逆向きに辿ることが必要なので、子ノードへのポインタの他に親ノードへのポインタも持っており、この点で、ノードそのものの表現のコストが高くなっているが、インクリメンタルな再解析を可能にするためにはどうしても必要であり、やむをえない。

現在の意味解析器は、cosmosの実現と異なり、変数や関数名など識別子の処理をするために、シンボルテーブルを使用している。意味解析用のアルゴリズムは、基本的には、個々の構文規則に対して設計されており、解析木を辿りながら、処理を行う。しかし、実行用の付加木を生成する制御構造の処理に関しては、現在のところ、構文規則からは独立して手続き的に実現している。処理系はＣ言語で実現し、一万行程度の規模である。

インタプリタのシミュレータはＣ言語でレジスタ転送レジスタで記述し、sparc1の上で行い、実行速度に関する性能の評価を行った。実行速度を測定するために、クイックソート（sort(200)）、integer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の４つサンプルプログラムを使用した。クロックは20MHZとして実行時間をクロック数から見積ったところ、同じプログラムをコンパイルしてsparc1上で実行した結果に比較して、１／２倍程度の性能を得た。まだ、細かいところを十分チューニングしていないので、とりあえず得られた性能としては、十分なものであると考えている。

６．おわりに

Ｃ言語のハードウェア解析木インタプリタのための、プログラムの内部表現と実行アルゴリズムの設計を行った。以前に行ったＰＬ／０の場合には、設計のポリシーがややアドホックであったが、今回は、実行アルゴリズムを設計するための枠組みとして、実行木という概念を導入し、解析木と実行木を概念的に区別することで、内部表現に内在する複雑性の整理を行った。この枠組みを用いて、Ｃ言語に関するインタプリタのためのアルゴリズムを実際に設計してみた経験からも、結果的に有効であったと考えている。

Ｃ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、キャスト演算、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。

現在のプログラムの内部表現と実行の方式は、特定の言語の構文やセマンティクスに依存しない部分が多く、普遍性のある方式である。したがって、この意味で言語定義を形式的に行っておき、この記述から形式的にパーサやインタプリタを生成することを容易にするものであるものと考える。

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と考える。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。

参考文献：

[1] Brian W.Kerighan and Dennis M.Ritche: The C Programing language,secand edition, Prentice-HALL,1988.

[2] Alfred V.Aho,Ravi Sethi and Jeffrey D.Ullman: Compilers - Principles,Techniques,and Tools, Addsion Wesley,1986.

[3] Medino-Mora,R.and Feiler,P.H.: An Incremental Programming Environment, IEEE Trans. Softw. Eng.,Vol.SE-7,No.3,pp.472-482,1981.

[4] 佐藤豊、板野肯三：動的複合実行方式ー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、２巻、４号、pp.19-29、1985.

[5] 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、情報処理学会第３１回全国大会、1f-7、pp.447-448、1985.

[6] 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、２８巻、３号、pp.310-313、1987.

[7] 闕　暁薇、板野肯三：構文木インタプリタPATIEO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)。

[8] 闕　暁薇、板野肯三：解析木インタプリタPATIEOのアーキテクチャ、情報処理学会論文誌、３２巻、９号、pp.1113-1121、(1991)。