アルゴリズム 75－1
（2000．11．10）

プログラムの構文チェックのためのデータ構造とアルゴリズム

嶋田 一行　　葛 崎偉

山口大学教育学部

これまでに、フローチャートやPADなど、様々な種類のプログラムの図式化が行われてきた。しかし、その目的はプログラムの流れを示すアルゴリズムの図式化であり、ソフトウェア開発中に行われる構文のチェックを行おうとすると、チェックのための検索アルゴリズムが作成しにくく、作成できたとしても高速な検索が可能なものにはならない。そこで本論文では、C言語で書かれているプログラムの構文チェックを効率よく行うために、プログラムが持つすべての情報を木構造の形で持たせたデータ構造を提案する。また、このデータ構造を用いた効率よいアルゴリズムを、構文チェックの例を挙げて紹介する。

Data Structure and Algorithm for Program Construction Analysis

Kazuyuki Shimata　　Qi-Wei Ge

Faculty of Education, Yamaguchi University

Till now, flowchart, PAD and etc. have been extensively used to express execution flows of programs. However in developing a program, especially from its previous version, checking programs' construction becomes ever important and for such checking flowchart and PAD are difficult to be used in designing efficient algorithms. In this paper, we propose a method to express the whole structure of a program by a tree graph. Using this tree graph, we design algorithms to efficient check the construction of programs that are written by C program language.

1　はじめに

情報技術の発展と共に、ソフトウェアの需要が益々高まってきている。ソフトウェアの開発における課題として、生産性や品質等の向上があげられており、それらを向上させる方法論が注目され、様々な提案が行われている [1]。

ソフトウェアの開発には、新規に開発する場合と、既存のソフトウェアの修正や改良による場合がある。既存のソフトウェアから始める場合、そのソフトウェアがソフトウェア内の個々のプログラム要素一つ一つを把握し、全体としての設計や仕様の確認をした上で、追加修正部分を含めた状態で要求されている仕様と一致するかを確かめなければならない。そのため、ソフトウェアの設計や仕様を記してあるドキュメントが必要となり、ドキュメントが不十分な場合は、設計や仕様の確認をする作業があるため、ドキュメントがある場合よりも開発時間がかかってしまう [2]。そのため、開発を迅速に行うために何らかの形でソフトウェアに関するドキュメントが必要になるのである。ドキュメントが不足あるいはない場合、手元にあるソフトウェアのソースファイルから、必要とされるドキュメントを作り出すことになる。

現在、プログラムを別の形で表現する方法としてはいくつかのものがある [3, 4]。例えば、フローチャートやＮＳチャート、ＰＡＤ図 [5, 6] や関数関連図などである。これらの表現方法により、ソフトウェアにおけるアルゴリズムを図式化することで、プログラムのブロック単位での流れを追うということは容易にできるかもしれない。しかし、プログラム全体の流れを追いつつ、ある変数なり、制御について検索をかけようとすると、既存の表現法では、検索アルゴリズムを作成しにくく、また作成できたとしても高速な検索を可能とするアルゴリズムにはならない。

そこで本研究では、ソースファイルレベルのプログラムの構文を忠実に表した木構造による新たなデー

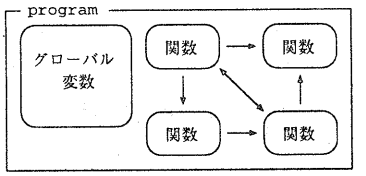

図 1: プログラムの構成

タ構造を提案する。これによって、ドキュメント生成のための解析だけでなく、ソフトウェア開発中の構文チェックも可能となる。またこのデータ構造を用いて解析をする際、高速に行えるアルゴリズムを紹介する。

2 C言語について

本研究でプログラム解析の対象となるC言語 [7] は、UNIXというオペレーティング・システムを記述するために作られ、UNIXと共に広く普及し、現在UNIX以外のソフトウェア開発に使われ、UNIX以外のOS上での処理系も多数存在する [8]。

より機械語に近い仕様をもった言語を低級言語、それに対して高級言語という用語があるが、その意味でC言語は中級言語とも呼ばれている。すなわち、機械語レベルで行なうようなビット反転、ローテーションなどのような記述が容易にできる一方、構造化プログラミングに適した言語仕様もあわせもつのである。また、近年進められているネットワーク化のためのツールやインターネットツールもC言語で記述されているものが多く、種々のソフトウェア開発の現場でもC言語はよく使われている。このように、今後さらにC言語が普及していく要因は数多いので、本研究ではC言語で書かれたプログラムを対象としたデータ構造を提案する。

C言語は関数型言語と言われ、構造の基本単位は関数と考えられている。つまりC言語プログラムは幾つかの関数が複雑につながり合い1つのプログラムを構成していると考えられる（図1を参照）。そこで、関数単位で木構造を構成していくことができるのではないかと考えた。また、図1にあるように、複数の関数から使えるグローバル変数が関数とは別にあるため、関数とは別の木構造を持つ。よって、本研究で提案するデータ構造は、グローバル変数の情報を納めた木と、関数の情報を納めた木の2つからなる。

```
#include <stdio.h>
#include <stdlib.h>

int  count = 0;
int  t_count = 0;
char text[50];

void text_select(void);
void count_check(int step);
void text_check(void);

int main(int argc, char *argv[])
{
  int key_in,step,exit_flag=0;

  step = 0;

  while(1){
    printf("No? [1-3]\n");
    scanf("%d",&key_in);
    switch(key_in){
      case 1:
        text_select();
        break;
      case 2:
        count_check(step);
        break;
      case 3:
        exit_flag++;
        break;
      default:
        break;
    }

    if(exit_flag != 0){
      exit(0);
    }

    step++;
  }
}

void text_select(void)
{
  if(t_count < 10){
    strcpy(text,"under 10.\n");
  } else {
    strcpy(text,"out of 10.\n");
  }
  text_check();
}

void count_check(int step)
{
  count = count + step;
  printf("now step : %d\n",count);
  text_check();
}

void text_check(void)
{
  printf("%s\n",text);
  t_count++;
}
```

図 2: プログラムの例

関数内で使われる文として、if文、switch文、while文のような、プログラムの流れを変える制御文、変数、配列の宣言、四則演算や、ビット反転を行う演算子、定義された関数の呼び出しがある。そのため関数内部では、制御文、宣言、演算式、関数呼び出しをデータ構造の基本単位として表していく。

3 プログラムの木構造表現

ここでは、図2にあるプログラムを例にあげ、データ構造の説明をする。なお、構文木を式で表現する場合は、

- （親：子，子，…）　　(子は左側の子から)

の法則にしたがって表現している。

3.1 グローバル変数の構造

ここで扱うのは、関数の外で宣言されているグローバル変数などである。関数の内部で宣言されているローカル変数とは違い、グローバル変数は複数の関数から利用できる。

関数の外で宣言されるのは、変数と構造体・共用体・列挙型のタグがある。そこで [global-variable] で変数を、[global-tag] で構造体・共用体・列挙型のタグに関する情報を子として持つ（図3を参照）。子の並び方として、変数名、またはタグ名を親ノードとして中心にとり、親ノードよりアルファベット順

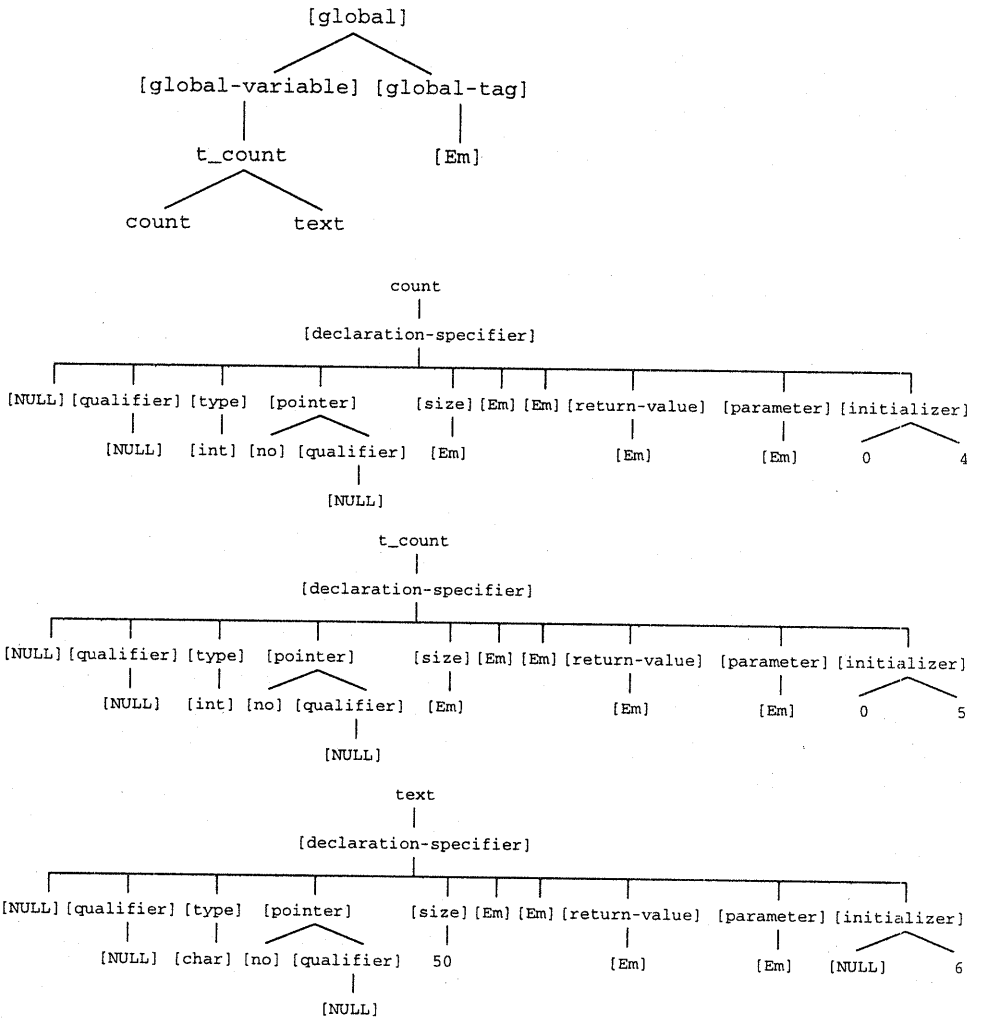

図 3: グローバルの構造

で出現が早ければ、親の左側の子とし、出現が遅ければ、右側の子として木を作成していく。

変数名、タグ名の下には、各々の宣言時の情報がくる。詳しいことは、後で述べる。

グローバルの構造

([global] : [global-variable] , [global-tag])

([global-variable] : 変数名)

([global-tag] : タグ名)

3.2 関数全体の構造

関数全体の構造で、関数に関する情報を木構造にする（図 4 を参照）。[declaration-function] の子として、プログラムにある関数がくる。この子の並び

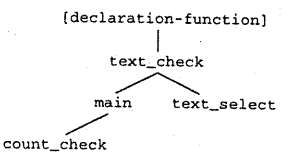

図 4: 関数全体の構造

は、関数名を親ノードとして中心にとり、親ノードよりアルファベット順で出現が早ければ、親の左側の子とし、出現が遅ければ、右側の子として木を作成していく。

関数名の子として、関数の中身が存在している。子の左から順に、関数自体の性質 [declaration-specifier]、その関数内だけで利用できるローカル変数 [local-

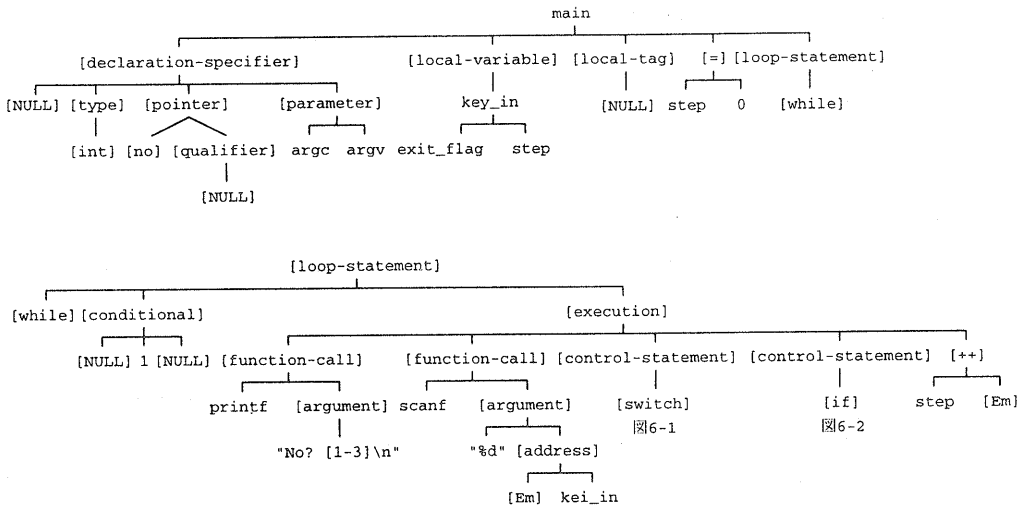

図 5: main 関数の構造 ([declaration-specifier] は一部省略)

variable]、ローカルなタグ [local-tag]、そして内部で使われている実行文が出現順に並ぶ。この関数自体の性質というのは、引数の数やその種類、戻り値の情報のような、関数定義（またはプロトタイプ宣言）の情報である。ローカルの変数の木やタグの木は、グローバルの場合と同じで、名前による 2 分木で構成する。

実行文の構造は、これから述べる。

関数全体の構造

([declaration-function] : 関数名)

(関数名 : [declaration-specifier] , [local-variable] , [local-tag] , 実行文)

3.3 制御文の構造

制御文については、C 言語では 5 つ (if, for, while, do-while, switch) に分けることができる。そのうち for 文, while 文, do-while 文については、式が真である間、それらの文にある実行文を繰り返し行うものである。これら 3 つのループ文を [loop-statement] で表し、各々を区別するために制御名として [for],[while],[do-while] とする。

if 文では else if や else によって、複数の結果に分かれてしまう。switch 文も、結果が複数の case や default に分岐しているので、if 文の変形と考えることが出来る。よって、この 2 つを [control-statement] で表現し、制御名として [if],[switch] とする（図 6 を参照）。また else if, else については、それらを含めて 1 つの if 文となるので、if 文の [control-statement] の下に、新たに [control-statement] をおき、制御名 [else-if],[else] とする。switch 文での case や default も、[control-statement] の下に、出現順で制御名 [case],[default] としてつながる。

[loop-statement],[control-statement] の両方の子として存在する [execution] の子は、制御文内の実行文が出現順に左から並ぶ。

制御文（for, while, do-while）

([loop-statement] : 制御名 , [conditional] , [execution])

([conditional] : 初期値設定 , 継続条件 , 増分)

制御文（if, switch）

([control-statement] : 制御名 , [conditional] , [execution] , $[\text{control-statement}]_{opt}$)

3.4 宣言の構造

宣言は、大きく分けると 2 つある。1 つは構造体・共用体・列挙型の構造の宣言で、もう 1 つは、変数・配列の宣言である。

まず最初に、構造体・共用体・列挙型の構造の宣言について説明する。これらは、構造体・共用体・列挙型であることを示す部分を除くと、同じ構造をとる。そのため、木構造も図 7・上のように、[struct-tag],[union-tag],[enum-tag] で区別し、それ以外は同じ物を使う。

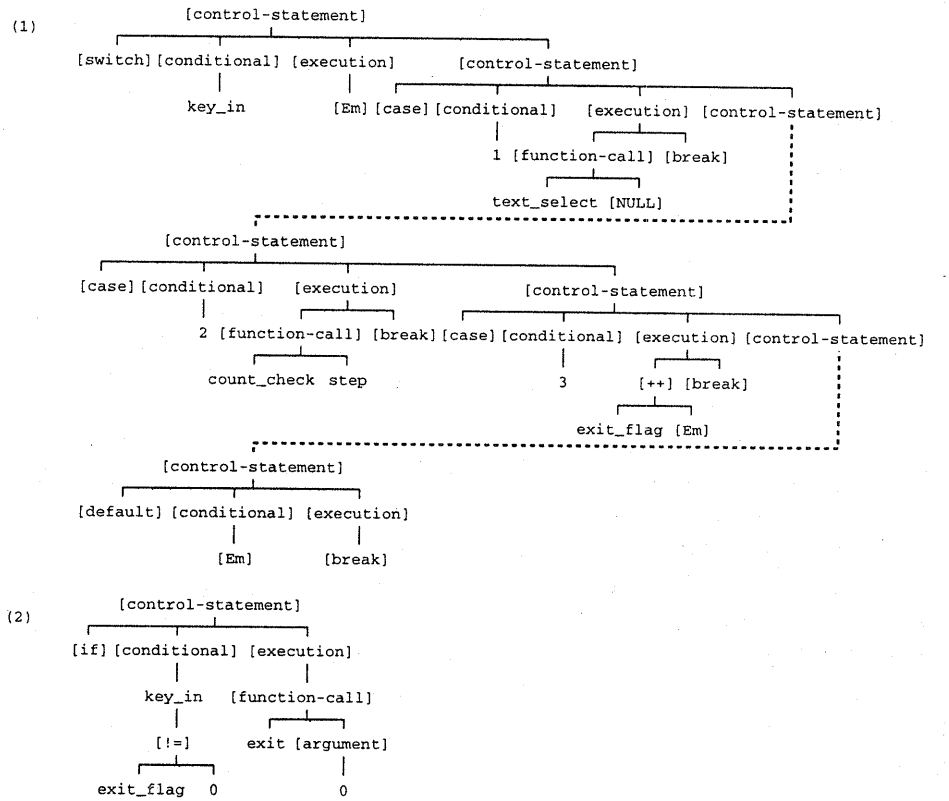

図 6: 制御文の構造

親としてタグ名を持つが、タグ名が省略されて宣言されている場合、構造体の変数名を用いて、[tag] 変数名をタグ名の変わりに使う。なお、タグ名が省略されていて、複数の構造体の変数が宣言されているときは、一番最初の変数名をタグ名の変わりに使う（図7下を参照）。

構造体・共用体

（ 変数名：[struct-tag] or [union-tag] or [enum-tag] ）

（ [struct-tag] or [union-tag] or [enum-tag] : 記憶クラス , [qualifier] , [member], 場所 ）

（ [member] : メンバ名 ）

（ メンバ名 : [declaration-specifier] ）

メンバ名は、名前のアルファベット順による2分木で構成する。持たしている情報は、次に述べる変数・配列のものである。また、場所は宣言された場所を示す。

次に、変数・配列の宣言について説明する。様々な

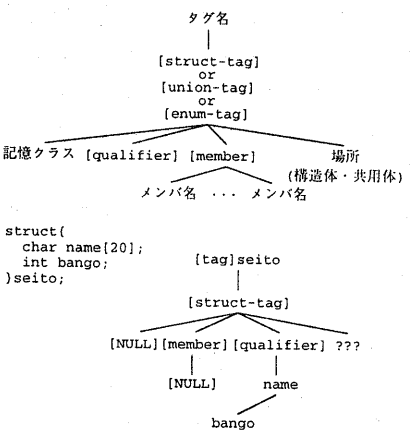

図 7: 構造体・共用体について

種類の変数・配列の宣言があるため、木構造に持たせている情報はかなり多くなっている。なお typedef による型定義も、これに当てはまる（このとき、記憶クラスとして [typedef] とおき、typedef によって新しく作られた typedef 名を親とする）。

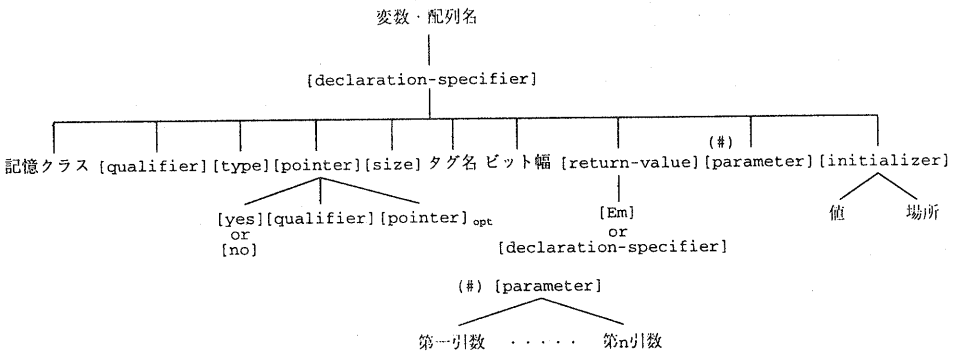

図 8: 変数・配列の宣言の構造

図 8 に示すように、[declaration-specifier] の子には、変数が持つ情報がくる。左から順に、変数の型や性質（記憶クラス,[qualifier])、配列ならばその大きさ（[size])、構造体・共用体・列挙型の変数であるときのタグ名、構造体のメンバで使われるときのビット幅、関数へのポインタ変数として宣言したときの戻り値や引数の情報（[return-value],[parameter])、そして、[initializer] として宣言された場所と初期化されるのならその値がくる。

変数・配列

（ 変数・配列名：[declaration-specifier] ）

（ [declaration-specifier]：記憶クラス , [qualifier] , [type] , [pointer] , [size] , タグ名 , ビット幅 , [return-value] , [parameter] , [initializer] ）

（ [pointer]：[yes] or [no] , [qualifier] , $[pointer]_{opt}$ ）

（ [size]： 1 次元の要素数, 2 次元の要素数, … , n 次元の要素数 ）

（ [return-value]：[Em] or [declaration-specifier] ）

（ [parameter]： 第一引数, 第二引数, … , 第 n 引数 ）

（ [initializer]：値 , 場所 ）

3.5 演算子の構造

演算子は、−500 のマイナスのような単項演算子、10＋20 のような 2 項演算子、3 項演算子 (条件演算子) の 3 つがある。

単項演算子と 2 項演算子は、親を演算子として、2 つの子を持たせる。ただし、2 項演算子の場合は、演算子の左側のオペランドを左側に、右側のオペランドを右側の子に配置するのに対して、単項演算子は少し異なる。単項演算子の場合は、その種類に応じて 3 種類の置き方がある。

3 項演算子は、別名条件演算子といい、式が真であるか偽であるかによってその後の処理が分岐する。そのため制御文の if 文の変形として考え、制御名 [?:true] と [?:false] で、真と偽の場合を分ける。

単項演算子

1. （ 演算子：[Em] , 対象オペランド ）
2. （ [cast]：データ型 , 対象式 ）
3. 前置型 （ 演算子：[Em] , 変数 ）
   後置型 （ 演算子：変数 , [Em] ）

2 項演算子

（ 演算子：左側オペランド , 右側オペランド ）

3.6 関数呼び出しの構造

ここでは、関数の呼び出しについて述べる。先に述べたように、C 言語は関数型の言語であるため、たとえば画面（標準出力）に 1 文字を表示する putchar 関数や、キーボード（標準入力）から 1 文字入力を受け付ける getchar 関数など、様々な機能を持つ関数を呼び出すことで一つのプログラムを構成していく。そのため、関数呼び出しも考えないといけない。

関数の中で関数を呼び出しているときは、親として [function-call] をおき、左側の子は関数の名前、右側の子は [argument] として使われたときの引数を持たせる。この引数の並びは出現順である（図 10 を参照)。

関数の呼出

（ [function-call]：関数名, [argument] ）

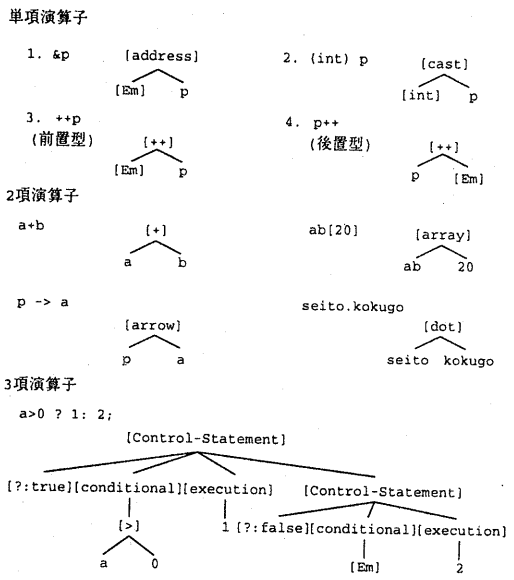

図 9: 演算子の構造

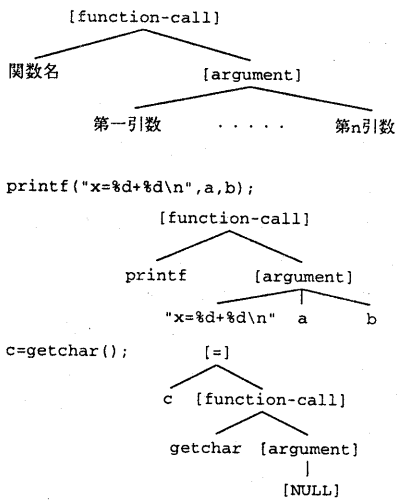

図 10: 関数呼び出しについて

([argument] : 第一引数, 第二引数, … , 第 n 引数)

3.7 その他

C 言語には、そのほかにもいくつかの構文があるので、それについて述べる。

関数を実行して、呼び出した関数へ制御を戻す場合に、C 言語では return 文を用いる。この return 文を示すものとして、[return] がある。子には、return 文で返される式がくる。この「式」が関数の戻り値になる。式を持たない return 文の場合、その子には [NULL] を置く。

switch 文では break 文が分岐後の処理打ち切りに使われる。switch 文以外にも、for 文, while 文, do-while 文のループ文に break 文は使われていて、このときはループの打ち切りに使われる。この break 文を示すものとして、[break] を使う。これは子を持たない。break 文と同じように、ループ文の処理の流れを変えるものとして、continue 文がある。continue 文を使うと、continue 文以降のその回のループ処理をスキップすることが出来る。これは [continue] として示している。

[goto] と [label] は、goto 文とそれに使われる識別子ラベルであることを示す。goto 文を使うと、識別子ラベルがあるところに処理の流れを移すことが出来る。[goto] の子には、行き先となる識別子ラベル名がくる。[label] の子には、他のラベルと区別するための識別子ラベル名がくる。

4 木構造を利用した構文チェック

本章では、これまでに紹介したデータ構造を使ったプログラムの構文チェックについて説明する。構文チェックの内容によって、プログラムの木構造データにおける調べ方は以下の 4 種類が考えられる。

1. switch 文に default が含まれているかを調べる時のように、実行文中のある文の中に対応する文が存在するかしないかを調べる
2. 配列のサイズを超える所にアクセスしているかを調べる時のように、変数の木から対象のデータを探すための 2 分探索を行い、実行文にあるデータと比較する
3. 配列のサイズよりも初期値の数が多い配列があるのかを調べる時のように、変数の木や構造体のタグの木にあるすべての変数やタグの情報を調べる
4. 変数が初期化される前に使用されているかどうかを調べる時のように、すべての実行文を調べ、必要とされる文が適切な位置に存在するかを調べる

ここでは、先に挙げたプログラムを例にし、「switch 文に default が含まれていないか」という構文チェックで、1 の調べ方について説明する。

switch 文の構造は、制御文のところであげた構造を使っている。このとき、[control-statement] で制御名が [switch] となるのが、switch 文である（図6を参照）。いくつかの [control-statement] がつながって switch 文を構成しているので、default があるとすれば、[control-statement] の下に制御名として [default] が存在する。

調べる際、対象ノードと比較ノードの2つを決める必要がある。対象ノードは調べるときの目印となり、比較ノードはそれの比較対象である。この2つを比較することで、構文チェックを行う。この場合、対象ノードは [control-statement] の制御名 [switch] であり、比較ノードは子の制御名 [default] である。

この構文チェックは、次の探索アルゴリズムを利用すればできる。

<< 探索アルゴリズム >>

1° 関数全体の木から、関数を一つ選ぶ。
2° 関数の構文木から、実行文にある対象ノードを探す。
3° 対象ノードがあれば、その対象ノードがある文の中に比較ノードがあるかを調べる。
4° 比較ノードの有無を知らせる。
5° 対象ノードがすべて見つかるまで、同じ関数の構文木にある他の対象ノードを探す。
6° 対象ノードが見つからなければ、まだ調べていない別の関数で対象ノードを探す。
7° すべての関数で探し終えたら、終了。

このデータ構造でプログラムを木構造にすると、1つの文が1つの木となり、また対象ノードと比較ノードが同じ木の中にあるので、比較ノードを調べる範囲が少なく、探索にかかる時間も少ない。

残りの3つの調べ方について、簡単に説明する。

2番目の調べ方は、実行文に書かれているデータが正しいものかどうかを調べるため、宣言時の情報と比較することになる。この情報は、変数名や関数名を頼りにして探すことになるが、変数や関数は名前による2分木として作られているので、2分探索を用いて素早くデータを探すことができる。

3番目の調べ方は、宣言時の情報を調べることになるので、実行文は調べなくてもよく、変数の木とタグの木にあるすべての変数の情報を調べればよい。そのため、探索にかかる時間は少ない。

4番目の調べ方は、すべての実行文を調べ、必要とされる文が適切な位置に存在しているかを調べることになるので、プログラムの流れを見ることになる。従って、探索時間についてはプログラムの流れを図式化するフローチャートや PAD などとあまり変らない。

5 おわりに

本研究では、ソースファイルにあるすべての情報を忠実に表した木構造によるデータ構造を提案し、またそれを用いた構文チェックを行うための効率的アルゴリズムを紹介した。

現在、このデータ構造とアルゴリズムを用いた構文チェックのソフトウェア開発を進めている。実用化に向けて、まずはプロトタイプの完成を目指し、そしてプロトタイプで得たデータを元にデータ構造やアルゴリズムの改善を行う必要がある。

参考文献

[1] 秋山義博ほか: ソフトウェア開発支援システム, システムと制御, 第31巻, 第9号, pp. 653-659 (1987).

[2] D. J. Robson, K. H. Bennett, B. J. Cornelius and M. Munro : "Approaches to Program Comprehension", *The Journal of Systems and Software*, Vol. 14, No. 12, pp. 79-84 (1991).

[3] 岡田謙一, 北川 節 : PAD によるソフトウェア開発システム, 情報処理学会論文誌, Vol. 26, No. 5, pp. 898-904 (1985).

[4] 大原茂之 : 木構造チャートによるプログラム開発・保守技法, 情報処理学会論文誌, Vol. 27, No. 10, pp. 1019-1026 (1986).

[5] 情報処理学会 : コンパクト版 情報処理ハンドブック, オーム社 (1997)

[6] 二村良彦 : プログラム技法 -PAD による構造化プログラミング-, オーム社 (1984)

[7] Samuel P. Harbison, Guy L. Steele Jr. : C, A Reference Manual THIRD EDITION, Prentice Hall Software Series (1991).

[8] JTM 企画株式会社, 荒瀬遥 : 初めての人のC言語入門, 西東社 (1991).