ＣのＡＮＳＩ規格案について

石畑　清

（東京大学理学部情報科学科）

プログラミング言語CのANSI規格（米国の国内規格）制定に向けての作業が進行中である。現在、作業は規格案[4]を非公式に配布して、各界の意見を求めている段階にある。本稿は、この規格案を概観することを目的とする。特に、従来のCとの違いを詳しく調べることに重点を置く。読者がCに関する知識を持っていることを仮定する。

１．　規格制定の背景

Cは、Unix開発の際（1970年代の初め）に、Ritchieによって設計された言語である。BCPLの流れをくんだシステム記述言語に属する。計算機に密着した記述といわゆる構造的プログラミングの両方を可能とする言語仕様を持っている。Unixの成功にともなって、Unixのほとんどのシステムプログラムの記述に使われているCが注目を集め、広い範囲で使われるようになってきた。

Cの使用範囲が広がるにつれて、しっかりした言語の定義が必要とされるようになってきた。現在までのところ、Cの公式の定義文書と見なされているのは[1]である。この本は、名著という評判が高いが、言語の定義文書としては必ずしも十分とは言えない。この本だけでは、言語の細かな点に関する知識を得ることは困難である。さらに、その後Cの仕様が徐々に拡張されてきたこと、[1]にはCによるプログラミングに不可欠なライブラリ関数の定義が与えられていないことなどから、規格の制定を求める声が強くなっていた。

このような要望に応じて、ANSIは1983年５月にCの規格を制定するための委員会（X3J11）を作った（SIGPLAN Notices，1983年９月号参照）。これに対して、一部には規格不要を主張する動きもあったが、大勢は好意的に見ているようである。なお、これより前、1977年からBell研究所では内部的なCの標準を作る作業が行なわれていた[3]。当然、ANSIの規格案にもBell研究所の作業の成果が強く反映されていると思われる。

２．　規格の構成

規格案は５部（Ａ～Ｅ）からなる。Ａは、規格の目的、規格が規定するものの範囲、用語の定義など、規格につきものの約束事をまとめたものである。Ｂ、Ｃ、Ｄが規格の主要部分で、Ｂが環境、Ｃが言語自体、Ｄがライブラリをそれぞれ扱っている。Ｅは付録で、文法規則一覧、ライブラリ一覧、索引などからなる。Ｅには、処理系定義とされている箇所など、プログラムの移植の障害となりうる点もまとめてある。

なお、規格案の言語仕様についての規定は、[1]を基礎としている。[1]で不足している点を適宜追加するという方針で書かれたらしい。また、ライブラリに関する部分は、/usr/group委員会のUnix 1984 Standardに基づいて作られている。

３．　環境

環境については、翻訳環境と実行環境の区別、文字セット、各種の制限値などが規定されている。翻訳環境と実行環境を区別したのは、プログラムを実行するための計算機と開発するための計算機が違う場合のことを考慮したものであろう。文字コード系が違う場合に適当な変換を加えることなどが定められている。

翻訳環境については、プログラムの解析の手順、特に構文要素の認識と前処理部分の処理の順序について詳しく定めている。

実行環境には、freestandingとhostedの２種類が用意されている。前者は、裸の計算機上でプログラムを動かす場合で、ライブラリを使っていないプログラムだけが動く。後者は、ＯＳの下で走るプログラムの場合で、Ｄで定められたライブラリを使うことができる。また、この場合は、実行が関数mainから始まること、mainの引数argc、argvを通して、プログラム起動時にＯＳから渡された文字列を受け取れることなどが定められている。

制限値は、プログラムの大きさや複雑さに関するものと、数値型の表現法に関するものの二つがある。前者は、処理系がプログラムの長さ、名前の個数、入れ子の深さなどに対して設ける制限を規制するためのものである。あまり小さな制限値を設けられると、プログラムの移植の際に困るので、それぞれ最低これだけは受け付けなければならないという値が決められている。数値型の表現に関しては、charを８ビット以上、short intやintを16ビット以上、long intを32ビット以上とすることなどが定められている。

４．　言語仕様

以下に見るように、この規格では今までのＣの仕様に多くの変更や拡張が加えられている。これらの変更を考える際、設計者たちは、二つの原則を念頭に置いていたようである[3]。一つは、型の検査を厳しくすることである。これは、Pascalなどの言語の影響である。ただし、Ｃはもともと計算機の動作に近いレベルでの記述を目的とした言語なので、型の検査によって何かができなくなるようなことがあってはならない。強いて型によって束縛せずに自由な書き方を許す部分も残されている。もう一つは、言語仕様をあまり複雑なものにしないということである。具体的に言うと、実行時ルーチンを使わないと実現しにくいような機能は排除されている。

以下、言語仕様の変更それぞれについて、変更した理由、例なども含めて解説する。変更の内容については規格案と今までのＣ、特に[1]を対照しながら述べる。筆者の個人的意見も適宜交えることにする。変更理由の説明は[3]から引用したものが多い。

４．１　構文要素

(a)　文字セット

プログラムを書くのに使える文字は、ASCIIの印字可能文字95種類のうち、$、@、`の三つを除いたすべての文字と、CR(復帰)、LF(改行)、HT(水平タブ)、VT(垂直タブ)、FF(改ページ)の五つの特殊文字である。このほかの文字を処理系が受け付けることは自由であるが、処理系に依存しないプログラムはこの文字セットだけを使って書くことが望ましい。もちろん、基礎となるコード系がASCIIである必要はない。行の区切りの表わし方は処理系ごとに定める。

文字のうち、やや特殊なものには代わりの書き方が用意されている。たとえば、

```
??<    →    {
??(    →    [
??/    →    \
```

などである。この置き換えは文字定数の中でも行なわれる。

(b)　予約語

五つの予約語――const、enum、signed、void、volatile――が追加された。一方、今までのＣにあった予約語entryは廃止された。entryは、将来の拡張用という名目で予約語となっていたが、使われる見込みがないので廃止された。

(c)　文字定数と文字列リテラル

文字定数や文字列リテラルの中で特殊文字や文字コードの値を指定する記法が追加された。

```
\a      ベルなどの注意を喚起するための文字
\v      垂直タブ
\xddd   16進法による文字コードの指定
```

逆斜線の直後の文字がエスケープ記法として定義されていない場合、今までは逆斜線を無視してその文字自体として扱うことにしていた。これに対して、規格案ではエラーということにしている。これは、将来記法を拡張する際に、それまでに作られたプログラムが動かなくなることを防ぐためである。

文字定数の中に複数の文字を書くことができるようになった。

(d)　文字列リテラルの連結

文字列リテラルが二つ並んでいる時（間に空白、改行、注釈などが入っていてもよい）は、二つの文字列の中身を連結した一つの文字列として扱うことになった。長い文字列を書きたい場合に有効であろう。

４．２　型

(e)　signed

今まで、char型の値が符号つきか否かは処理系定義とされていた。たとえば、PDP-11用の処理系では符号つき（-128～+127）、IBM-370用の処理系では符号なし（0～255）としていた。

もちろん、これで通常は十分なのだが、場合によっては、符号つきか符号なしかをはっきりさせたい場合がある。そこで、charのほかに、signed char、unsigned charの二つの型を設けた。signed charは符号つき、unsigned charは符号なしであることが保証される。単なるchar型は、いずれか一方と同じになるが、いずれになるかは処理系定義である。

なお、signed intなどと書くことも可能であるが、intと同じ型にすぎない。

(f)　数値型

数値型の種類が増えた。

今まで、数値型には、char、short int、int、unsigned int、long int、float、double（long floatでも同じ）の７種類があった。

このうち、charには、前項で述べたようにsigned char、unsigned charの二つが追加された。整数型には、unsigned short int、unsigned long intの二つが追加された。すなわち、charも含めていずれの長さにも符号ありと符号なしの両方の型が用意されることになった。浮動小数点型は、float、doubleにlong doubleが追加された。long doubleはdouble以上の精度を持つ。long floatと書いてdouble型を表わす書き方は廃止された。

定数の直後に型を指定する文字を置くことによっ

て、符号なし型の整数定数や、float型、long double型の浮動小数点定数を書くことができるようになった。たとえば、

```
500u    1234L    6.023e23f
```

switch文の式の型がintでなくてもよいことになった。一方、構造型のビットフィールドの型はintかunsigned intに限られることになった。型の許容範囲について細かい見直しが行なわれたらしい。

(g) 列挙型

Pascalにならって、列挙型が導入された。宣言は、次のような形をとる。

```
enum day { sun, mon, tue, wed,
           thu, fri, sat };
```

sunからsatまでが列挙型の値である。これを列挙定数と呼ぶ。列挙型の宣言は、構造型（struct）や連合型（union）の宣言と同様に、その型に属する変数を同時に宣言することもできるし、型の名前だけを宣言することもできる。名前のついた型は、後で変数宣言に使うことができる。たとえば、

```
enum { free, busy, error } status;
enum day workday, holiday;
```

status、workday、holidayは、いずれも変数名である。

列挙型の変数の実際の型は、処理系によってchar、intなどの中から適当に選ばれる。各列挙定数の値も処理系によって決められるが、プログラマが指定することもできる。たとえば、

```
enum permissions
        { read=4, write=2, execute=1 };
```

列挙型は、いくつかの決まった状態に名前をつけるような時に使う。このような場合、従来は整数型とマクロ定義を組み合わせて、

```
#define FREE  0
#define BUSY  1
#define ERROR 2
int status;
```

のように書くのが普通だった。これと比べると、列挙型には、定数宣言の手間が省ける、値の範囲が宣言から明確にわかるなどの利点がある。

ところで、Pascalの列挙型は、それだけで閉じた世界であり、整数との間の演算などは許されなかった。これに対して、この規格では、列挙型を整数型の一種と考えている。たとえば、

```
status = free + busy * error;
workday = 5 * sun;
```

などの、Pascalに親しんだ人間には無意味とも思える書き方が許されることになる。列挙型の変数がとりうる値をせっかく並べ上げても、それ以外の値を代入した時に処理系がエラーとして検出しないのでは、意味が半減してしまう。これは、PascalとCの発想の違いによるものだろうか。

規格案は、付録の中で列挙型の変数にその型の列挙定数以外の値を代入しようとした時に警告メッセージを出してもよいと述べている。これは、型を厳しく検査するというPascal流の考え方にほかならないが、残念ながら正式の規則ではないので、守られるとは限らない。

(h) void型

Cは手続きと関数を区別しない。手続きも関数の一種である。手続きとして使われる関数にも結果の型を指定しなければならない。ところが、これらの関数は結果を返さないのだから、どんな型を指定してもうまくいくわけがない。そこで、値を一つも持たない特別の型としてvoidを新設した。値を返さない関数は、この型を返すと宣言すればよい。

void型は、このほか引数を持たない関数の宣言にも使われる(後述)。また、void型に対するポインタ型は、どのポインタ型とも相互に代入のできる特別な型である。記憶域割り当てルーチンなどがこの型を使うことが考えられる。

4．3 宣言

(i) 名前の管理

名前は、次の四つのクラスに分けて管理することになった。

・普通の名前。変数名、列挙定数名、typedefによって定義された型名などを含む。

・ラベル名。

・構造型、連合型、列挙型のタグ（予約語structなどの直後の名前）。

・構造型や連合型の要素の名前。

ラベル名を独立させたのがこれまでとの違いである。同じ名前でも、別のクラスに属するものであれば二重宣言とはならない。使われる場所が違うので区別がつくからである。

なお、typedefによって定義された名前は、場合によっては変数名と区別がつかなくなる恐れがある。そのような場合はどちらか一方の解釈を選ぶように、規則が決められてはいるが、言語仕様としては問題である。処理系にとってもやっかいな問題だと思われる。

また、構造型のタグにも特別な規則が設けられた。タグだけを書いて、中身のともなわない宣言は、その名前の構造型を後で宣言するという意味になる。たとえば、

```
struct b;
struct a { int x; struct b *p; };
struct b { int y; struct a *q; };
```

のように、二つの構造型が互いに相手に対するポインタを含んでいる時にこの書き方が必要になる。

なお、筆者にはどのような場合に同じ名前の変数

を２回以上宣言してよいかがわからなかった。名前のスコープに関する規定は改良の必要があると思われる。

(j)　構造型の要素の名前

今までは、構造型や連合型の要素の名前は全部違っていなければならなかった。たとえ、別々の構造型であっても同じ名前を含むことは許されなかった。これは、ピリオドや矢印による要素の選択の際、左側に書いた式の型を考慮せず、右側の要素名だけを見て処理していたからである。矢印の場合で言うと、左側には、その構造型に対するポインタ型だけでなく、任意のポインタ型や整数型の式を書くことができた。これは、型を厳しく扱うという大方針に反する。そこで、要素の選択は左側の式の型を考慮し、右側にはその型の要素だけしか書けないことにした。この結果、違う型の中であれば同じ要素名を使ってもよいことになった。

(k)　const

型指定子としてconstとvolatileの二つが新設された。

constは、変数の値が途中で変わらないことを示す。普通の変数であれば、初期値設定によって決まった値がずっと保持される。関数の引数であれば、実引数の値のままである。constを使った宣言の例は次のようになる。

```
int const power_of_10[] =
        { 1, 10, 100, 1000, 10000 };
```

変数をconstと宣言することの利点は、誤って値を変えられないようにすることもあるが、より重要なのは、記憶域割り当てを効率化することである。定数によって初期値設定されたconst変数は、それ自身定数にほかならないので、普通の変数とは別の領域、たとえばプログラムのオブジェクトコードの一部に割り当てることができる。一つのプログラムを複数のユーザが共用している場合、特に大きな配列を使っている場合に効果がある。

constは、普通の型指定子の位置のほかに、ポインタ型の指定の一部（＊の直後）に書くこともできる。これは、ポインタ変数の値が変わらないことを示す時に使う。たとえば、

```
char *const p = new_string();
```

とすると、ポインタ変数ｐの値は変わらないが、ｐの指している場所の内容は変わりうる。これに対して、

```
const char *q = "pointer_to_constant";
```

とすると、ポインタ変数ｑは別のアドレスを指すように変更できるが、ｑの指している場所の内容を変更することは禁止される。なお、文字列リテラルは、const charに対するポインタ型に属する。文字列リテラルの中身を変更することは禁止される。

const変数の値の書き替えの禁止は、規格の上ではlvalueとしての使用の禁止という形で表現される。すなわち、代入の左辺とすること、＋＋および－－演算子のオペランドとすることが禁止される。今までは、変数のアドレスをとる＆演算子のオペランドもlvalueであったが、規格案ではlvalueとしていない。したがって、const変数のアドレスをポインタ変数に代入することができる。

ポインタ変数を通してconst変数に代入するのは誤りであるが、処理系にとって検出が難しい。このように、処理系が検出しにくい誤りを規格案ではundefinedと表現している。処理系はundefinedな操作の結果について、いっさい責任を持たなくてよい。

(l)　volatile

volatileは、処理系による最適化を制限するために設けられた。volatileと宣言された変数を含む式は、規格に定められたとおりの順序で評価しなければならない。

volatileの宣言は、処理系の予想しない時に変数の値が変わる可能性があることを示すために使われる。たとえば、外部機器を制御するレジスタはvolatileと宣言しておかなければならない。また、ライブラリ関数signalを使って割り込み処理を行なう場合は、割り込み処理ルーチンの中で使う変数をvolatileと宣言しておく必要がある。このような細かな指定が必要とされるのも、システム記述に使われるＣならではであろう。

volatileの宣言がない場合、処理系は規格と同じ結果となる限りどんな順序で式を評価してもかまわない。式の一部を評価せずにすますことも許される。ただし、その場合は副作用がないことを証明しておかなければならない。

一般に、規格案は処理系にできるだけ大きな自由度を残すように努めている。式の評価中に副作用がいつ起こるかは保証されない。ただし、文が終わった時にはその中の副作用がすべて完了していることが保証される。副作用の完了が保証される点には、このほかに"&&"、"||"、"?:"、","の各演算子、if文、switch文、繰り返し文のそれぞれ式の部分、関数の呼び出しがある。

(m)　型も種類も指定しない宣言の禁止

今までは、関数の外での宣言に限って、

```
x;   f();
```

のように、型も種類も指定しない宣言が許されていた（int型と解釈される）が、これは禁止されることになった。ただし、本体のある関数の場合は従来どおり、

```
f()
{  .....  }
```

として、int型の関数を宣言することができる。

(n)　関数の外における変数の宣言

これは、[1]でもすでに述べられているが、再確認の意味で述べておく。

関数の外における変数の宣言には、extern、static、その他の３種類がある。その他とは、externもstaticも含まない宣言である。このうち、staticは一つのソースファイル内でしか使わない変数を宣言するもので、これには特に問題はない。

externのついた宣言は、変数の名前と型を宣言するだけで、記憶域は確保しない。プログラムの他の部分で、同じ名前の変数を宣言して記憶域を確保する必要がある。これに対して、externでもstaticでもない宣言は、記憶域を確保する。複数のソースファイルからなるプログラムの場合、この形の宣言を一つのファイルに置き、他のファイルにはextern宣言を置いてこれを参照することになる。

この方式は、今までUnixのＣコンパイラが採用していた方式と違う。Unixでは、関数の外で宣言された変数一つ一つをFortranのコモンブロックのように扱う。したがって、同じ変数を二つ以上のソースファイルの中で宣言しても、特に問題はない。リンク時に同じアドレスに割り当てられるだけである。このような背景があるので、UnixのＣコンパイラは、extern宣言とそうでない宣言を区別せず、いずれもexternとして扱っている。extern宣言には記憶域が与えられる。

ところが、多くのＯＳではコモンブロックの数が制限されているなどの理由からこの方法をとりにくい。記憶域の割り当ては一箇所だけにして、他のソースファイルからはこれを参照する形をとることが望ましい。そこで、規格案ではextern宣言とそうでない宣言を区別することにした。UnixのＣは、規格案のＣに拡張を加えたものと見なされる。

なお、関数の宣言は、externかstaticかのいずれかで、いずれでもない時はexternと解釈される。これは今までと変わらない。

(o)　初期値設定

初期値設定に対する制限が撤廃された。すなわち、連合型の変数に対する初期値設定とブロック内で宣言された構造型や配列型の変数に対する初期値設定が禁止されていたが、いずれも許されることになった。

連合型の場合は、どの選択肢を選ぶかが問題であったが、最初の選択肢に対する初期値設定と解釈することになった。

ブロック内で宣言された構造型の変数の場合は、

```
struct complex x = unit;
struct complex y = { 10.0, 5.0 };
```

の２種類の形式が許される。第１の形式の場合、unitはブロックの外側で宣言された変数である。この形式はブロック内の変数に対する通常の初期値設定と同じで、代入文を宣言に組み込んだ形にすぎない。構造型の代入が可能となったので、これを禁止する理由はない。右辺の式には、任意の形式のものを書くことができる。

第２の形式は、代入とは違う。こちらは、関数の外でだけ許されていた初期値設定の形式を関数の中でも使えることにしたものである。｛と｝の中の式は、すべてコンパイル時に計算できる式でなければならない。こちらの方が処理系にとっては扱いがやっかいであろう。

配列型の場合は、配列全体の代入が許されないので、構造型の場合の第２の形式だけが許される。たとえば、

```
int primes[4] = { 2, 3, 5, 7 };
char message[] = "error";
```

４．４　関数

(p)　関数の引数の型の指定

Ｃの型の扱いの中で一番の弱点は関数の引数と結果の型であった。ほとんどのＣの処理系は、関数の仮引数と実引数の型の整合を調べない。これによって多くの誤りが見過ごされる結果となっていた。

そこで、規格案では引数の型の対応を検査できるようにするために、関数の引数と結果の型だけを関数本体とは別に宣言する記法を導入した。これを関数のプロトタイプと呼ぶ。最初に例を示そう。

```
int max(int x, int y);
void print(FILE *, int);
```

maxは、二つのint型の引数をとってint型の値を返す。printは、FILE型へのポインタとint型の値を引数とし、結果を返さない。maxは仮引数の名前を示しているが、これは注釈としての意味しか持たない。printのように、仮引数名を示さない書き方も許される。この場合の型の書き方は、今までsizeofや型変換で使っていた書き方と同じである。

関数本体の宣言にも、プロトタイプと同様の書き方を使うことができる。たとえば、

```
int max(int x, int y)
{ ..... }
```

となる。一方、従来の書き方も残されている。たとえば、

```
int max(x, y)
int x, y;
{ ..... }
```

両方とも、意味は同じである。

関数の呼び出しは、プロトタイプがある時とない時とで違ってくる。プロトタイプがない時は、引数の型の自動変換が起こる。char型やshort int型がint型に、float型がdouble型に変換される。一方、プロトタイプがある場合は、プロトタイプに書いてある型の変数に対する代入と同じ操作で引数が渡さ

れる。char型やshort int型からの自動変換は起こらない。

もちろん、関数の宣言（プロトタイプ、本体のいずれか）が、呼び出しの前にあれば、仮引数と実引数の型の対応が検査される。宣言なしで呼び出した場合は、

```
extern int 名前();
```

という形の宣言が仮定される。これは、引数の数も型も検査しないという意味になる(次項参照)。

プロトタイプを使う場合は、その関数を呼び出しているすべての場所からプロトタイプが見えるようになっていなければならない。また、本体より前にもプロトタイプの宣言を置かなければならない。これは、プロトタイプの有無によって、関数の働きが若干変わるからである。プロトタイプと本体は型の対応がとれていなければならない。

結局、関数の宣言には二つの流儀が成立することになる。一つは、今までどおり、プロトタイプを与えない方法である。この流儀では、引数の型や数の検査は期待できない。もう一つは、関数の本体の前にプロトタイプを常に与える方法である。この場合は、呼び出しの位置すべてにプロトタイプが見えるようにしておくことが必要である。この流儀によれば、関数の引数の型の検査は完全に行なえる。もちろん、プロトタイプが間違っていたりすれば元の木阿弥であるが、ヘッダファイルにプロトタイプを入れておくことにすればその危険は避けられる。

(q)　不定個の引数をとる関数の宣言

Cは、引数の型が可変であったり、引数の個数が不定だったりする関数を書くことも許してきた。printfがこの例である。このような関数は、次のようにして宣言することになった。

```
int printf(const char *format, ...);
```

最初の引数の型は決まっているが、それ以降の引数については型も個数も規定していない。...（ピリオド三つ）は区切り記号の一種である。

...による書き方は、少なくとも１個は引数が確定していないと使えない。引数の型が一つも決まっていない場合は、

```
int plus();
```

のように、括弧の中に何も書かない。これは従来のCにおける関数の宣言と同じ形である。これと紛らわしいが、一つも引数がない場合は、

```
static initialize(void);
```

のように、void型を引数として指定する。void型の引数というものはありえないので、引数なしという特別の意味を持たせたわけである。

引数のない関数の本体は、

```
cleanup()
{ ..... }
```

のように宣言する。プロトタイプの場合と書き方が違うことに注意されたい。不統一で気持が悪いが、従来のCと矛盾しないようにした結果なので、やむを得ない。

引数の数が決まっていない関数の中では、ライブラリ関数を使ってこれらの引数の値を取り出す。たとえば、

```
#include <stdarg.h>
int sum(int count, ...)
{   va_list p;
    int s = 0;
    va_start(p, count);
    while (count-- > 0)
        s += va_arg(p, int);
    va_end(p);
    return s;
}
```

va_listは、引数のリストを管理するための型である。関数の中では、最初にva_startを使って、va_list型の変数を初期化する。va_startの第２引数は、確定している最後の引数の名前である。個々の引数はva_argを使って一つずつ取り出す。va_argの第２引数には、引数の型を書く。

４．５　式

(r)　構造型や連合型の値の代入

今まで、構造型や連合型は複数の値に一つの名前をつけてまとめておくという役割しか持たなかった。値の操作は要素単位でしかできなかった。

規格案では、これを改めて、構造型や連合型の値全体を代入したり、関数の引数として渡したり、関数の結果としたりすることができるようにした。たとえば、

```
struct complex { double re, im; } p, q, r;
struct complex mult();
p = q;
r = mult(p, q);
```

などとすることが可能である。

一方、Cの配列名は、他の言語と違って、ほとんどポインタの代わりとしての役目しか持っていないので、配列同士の代入を導入することはほとんど不可能である。

(s)　sizeof

今まで、ポインタ同士の引き算やsizeofの結果はint型とされていたが、処理系によってはlong型としてもよいことになった。これは、int型にアドレスを表わすだけの長さがない処理系があるためである。

sizeofを構造型の変数に適用した時は、その変数の長さそのものではなく、この構造型を要素とする配列を作った時の隣の要素との間の距離を返すこと

にした。アドレスを整合させるために挿入される隙間を含めて勘定することになる。

(t) 単項の＋演算子

単項の＋演算子が新設された。この演算子は、処理系による演算順序の変更を禁止するために使うことができる。たとえば、

```
x + +(y + z)
```

とすると、ｙとｚを加えてからその答にｘを加える。こうしないと、たとえ括弧があっても、どの順序で加算を行なうかは処理系の自由となる。

４．６ 前処理系

(u) 条件つきコンパイル

条件つきコンパイルの指定は、今まで#if、#else、#endifの三つで行なっていたが、これに#elifを加えた。elifはelse ifの略である。意味は自明だろう。

名前が#defineで定義されているかどうかを#ifdefで判定することにしていたが、今後これは推奨しないことになった。代わりに、

```
#if defined(名前)
```

とする書き方を推奨する。definedは新しい擬似関数で、#ifや#elifの中でだけ使える。名前が定義されていれば１を、そうでなければ０を返す。

(v) マクロ展開

#defineによるマクロ定義の中に＃、＃＃の二つの記法が導入された。＃は直後の引数の両側に引用符を挿入して文字列リテラルの形にする。＃＃は、その前後の構文要素を連結して一つの構文要素とする。たとえば、

```
#define debug(n) \
        printf("x" #n " = %d", x ## n)
```

と宣言して、

```
debug(5)
```

と呼び出すと、

```
printf("x" "5" " = %d", x5)
```

となる。さらに、文字列リテラルの連結の規定によって、

```
printf("x5 = %d", x5)
```

と変形される。今までは、文字列リテラルの中までマクロ展開するかどうかがあいまいであったが、この記法の導入によって問題がなくなった。

(w) 前処理系の新機能

前処理指令の最初の＃は第１桁に置かなくてもよいことになった。ソースファイルの名前と行番号を表わす組み込みマクロ__FILE__、__LINE__が新設された。処理系に特有の指令を与えるための前処理指令#pragmaが新設された。

５． ライブラリ

Ｃは、裸の言語だけではほとんど使い物にならない。規格にライブラリに関する規定を設けるのも当然であろう。しかし、ライブラリは計算機やＯＳに強く依存するので、言語自体より規格化することが難しい。以下に見るように、この規格案ではほぼ最低限の機能をカバーする範囲に止めてある。

規格案に挙げられたライブラリ関数は、当然ながらUnixで提供されているものがほとんどである。しかし、言語自体が変更されていること、関数の引数の型をきちんと定義したことなどの影響もあって、現在のUnixのライブラリそのままというわけではない。

規格案のライブラリは12個のヘッダファイルからなる。ライブラリ関数も、それぞれこれらのヘッダのいずれかに属するものとされている。これは、各関数がマクロ（#defineによる）の形で提供されているためである。ただし、同名の関数も用意されているので、マクロの定義を#undefによって消すか、初めからヘッダを取り込まないことにすれば、本物の関数の方を使うことができる。

以下、ヘッダとその中で定義されている関数の名前を列挙する。

◎ assert.h

assert ： 引数を評価して０であればメッセージを出力して異常終了する。

◎ ctype.h

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit ： 文字の種類の判定。

tolower, toupper ： 大文字、小文字の変換。

◎ limits.h

各整数型の値の上下限と各浮動小数点型の表現法に関する定数のマクロ定義からなる。このヘッダだけはfreestanding環境でも使えることになっている。

◎ math.h

acos, asin, atan, atan2, cos, sin, tan, cosh, sinh, tanh, exp, frexp, ldexp, log, log10, modf, pow, sqrt, abs, ceil, fabs, floor, fmod ： 数学関数を提供する。

◎ setjmp.h

setjmp, longjmp ： 関数の実行を途中で放棄して、あらかじめ定めておいた地点へ飛び出す時に使う。

◎ signal.h

signal ： ハードウェアまたはソフトウェアによって発生した割り込みの処理を定める。割り込みの種類ごとに、発生した時に呼び出すべき関数を指定しておく。この関数から戻ると、割り込まれた地点から実行が継続される。割り込みの無視を指定する

こともできる。

kill : 他のプログラムに割り込みをかける。プロセスという概念が定義されていないので、中途半端な存在である。

◎ stdarg.h

va_start, va_arg, va_end : 可変個数の引数を受け付ける関数の中で、それらの引数の値を取り出すための関数である。実例はすでに述べた。

◎ stdefs.h

ライブラリに共通の型や定数の定義が入っている。

◎ stdio.h

remove, rename, tmpfile, tmpnam, fclose, fflush, fopen, freopen, setbuf, setvbuf, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, getc, getchar, gets, putc, putchar, puts, ungetc, fread, fwrite, fseek, ftell, rewind, clearerr, feof, ferror, perror : 入出力やファイルに対する操作を提供する。入出力は、今までの標準入出力ライブラリと同様に、ストリームを経由して行なう。関数の多くは周知のものであろう。

◎ stdlib.h

atof, atoi, atol, strtod, strtol : 文字列中の数の表現を数値に変換する。

rand, srand : 乱数。

calloc, free, malloc, realloc : 記憶域の割り当て。

abort : プログラムの異常終了。

exit : プログラムの正常終了。

getenv : ＯＳが管理する環境変数の参照。

onexit : exitの時に実行すべき関数を登録する。

system : ＯＳのコマンドを実行する。

◎ string.h

memcpy, memset, strcpy, strncpy, strcat, strncat, memcmp, strcmp, strlen, strncmp, memchr, strchr, strcspn, strpbrk, strrchr, strspn, strtok : 文字列の操作を行なう。strで始まる関数はNULで終わる文字列を扱う。memで始まる関数には長さも与えなければならない。

◎ time.h

clock, time, asctime, ctime, difftime, gmtime, localtime : 時刻を操作する。

６． 規格案に対する感想

どんな言語でも、使われ始めてから長くたつと、改良を要する点が出てくる。しかし、一方ではその言語で書かれた既存のプログラムを守らなければならないので、むやみな変更はできない。この二つの要請から、言語の仕様はだんだん汚なくなっていくのが普通である。Ｃもその例に漏れない。Ｃが実用言語になった証拠と見ることができるかも知れない。また、委員会組織で言語の規格を作ると、各人の希望を取り入れて仕様が拡大し、収拾がつかなくなるとは、よく言われることである。この規格案についてもこの批判はあたっていると思われる。

Ｃの場合、型のないBCPLを先祖とすることもあって、今まで型の束縛は比較的ゆるかった。型は、誤りを発見するよりも記述を便利にするための手段と考えられていたように思われる。それが広く使われるようになると、ソフトウェア生産性向上のために型による誤りの発見に重きが置かれるようになってきた。関数のプロトタイプなどを見ると、Ｃらしさがだいぶ失われた感もあるが、これも世の趨勢であって、やむを得ないであろう。

筆者は、規格案の目指す方向には賛成であるが、現在の規格案そのものには賛成しかねる点が多々ある。基礎となった[1]があまり形式ばった書き方をしていなかったのを無理に形式の整った文書に直したものという印象を受ける。言わなくてもよいことを述べていたり（たとえば、変数名の綴りを間違えたら何が起こるかわからないと述べている）、必要な規則が抜けていたりする点が多いように見受けられる。プログラミング言語の文法書というものは、もともとあまり読みやすくないものだが、この規格案は、あることがらがどこに述べられているか探すのが困難だったりするので特に読みにくい。正式の規格として認められるまでには時間がかかるのではないだろうか。

今後の予定としては、1986年３月に、ANSIから正式の規格案の形で発表され、公式の意見集めの段階に入ることになっている。

参考文献

[1] Kernighan,B.W. and Ritchie,D.M.: The C programming language, Prentice Hall, p.228, 1978.

[2] Ritchie,D.M., Johnson,S.C., Lesk,M.E. and Kernighan,B.W.: The C programming language, Bell System Technical Journal, Vol.57, No.6, Part 2, pp.1991-2019, 1978.

[3] Rosler,L.: The evolution of C - past and future, AT&T Bell Laboratories Technical Journal, Vol.63, No.8, pp.1685-1699, 1984.

[4] X3J11, American National Standards Institute: preliminary draft proposed C standard, X3J11/85-045, p.157, April 1985.