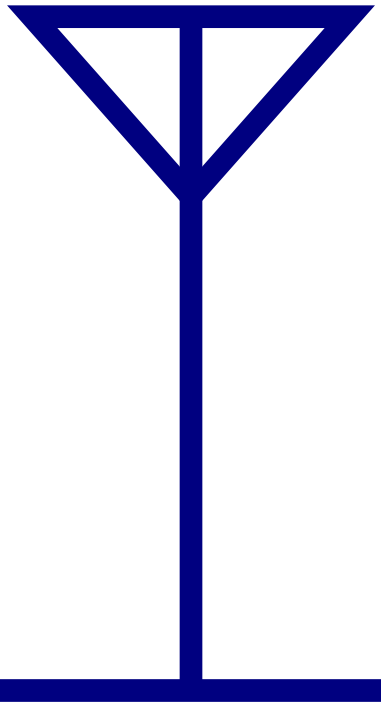
Journal of Hamradio Informatics No.1

うさぎさんでもわかる並列言語Chapel

Chapel the Parallel Programming Language



無線部開発班　**平成 29 年 1 月 30 日改訂**

http://pafelog.net

目次

第1章　はじめに

本稿で紹介する Chapel は、**型推論**と**テンプレート**と**区分化大域アドレス空間**を特徴とした並列言語である。概して並列処理は、プログラムの可読性と移植性を損ね、挙動が複雑で、所期の性能を得るのは困難を伴う。故にこそ、並列処理の詳細な実装を隠蔽し、自動的に高度な最適化を施す**高生産性並列言語**には価値がある。

1.1　環境の構築

Chapel のウェブサイト[1]で頒布されているアーカイブを入手して展開し、下記のように make でビルドする。

```
$ tar xzf chapel-1.14.0.tar.gz
$ make -C chapel-1.14.0
```

環境変数を設定して、コンパイラへのパスを通す。例えば bash の場合、.bashrc に以下の 2 行を追記する。

```
export CHPL_HOME=~/.chapel-1.14.0
cd $CHPL_HOME && source util/setchplenv.sh > /dev/null && cd - > /dev/null
```

1.2　最初の例題

テキストエディタで下記の通りに 1 行だけのプログラムを打ち込み、hello.chpl と名前を付けて保存する。

hello.chpl

```
writeln("Hello, world!");
```

シェルを起動して chpl コマンドでコンパイルする。実行可能ファイル hello が生成されるので、起動する。

```
$ chpl -o hello hello.chpl
$ ./hello
Hello, world!
```

なお、実行可能ファイル hello には、Chapel のコンパイラによって自動的に help オプションが追加される。

```
$ ./hello --help
```

chpl コマンドにも様々なオプションがある。例えば、savec オプションを指定すると、C 言語に変換できる。

```
$ chpl --savec hello hello.chpl
```

fast オプションを指定すれば、推奨設定での最適化が有効になる。リリース版をビルドする際に重宝する。

```
$ chpl --fast -o hello hello.chpl
```

[1] http://chapel.cray.com

第2章　字句と型

2.1　識別子

大文字と小文字は区別する。空白 (0x20) とタブ (0x09) と改行 (0x0A) と復帰 (0x0D) は、区切り文字である。

識別子　`ident := [A-Z_a-z] [$0-9A-Z_a-z]*`

2.2　基本型

Chapel は**強い静的型付け**を行い、型はコンパイル時に決まる。言語仕様に組み込まれた型を**基本型**と呼ぶ。

ボイド型　`void`
論理型　`bool`
整数型　`int uint`
実数型　`real`
虚数型　`imag`
複素数型　`complex`
文字列型　`string`

2.3　リテラル

論理値　`bool := 'true' | 'false'`
整数値　`int  := ('0x' | '0X' | '0o' | '0O' | '0b' | '0B')? [0-9]+`
実数値　`real := [0-9]* ([0-9] '.' | '.' [0-9]) [0-9]* ('e' [0-9]+)?`
虚数値　`imag := real 'i'`
文字列　`text := ('"' char* '"' | "'" char* "'")`

2.4　コメント

C 言語 (C99) と同様に、1 行のコメントは//の右側に、複数行のコメントは/*と*/で囲んだ中に、記述する。

comment.chpl

```
// 1-line comments

/*
multiline comments
 or block comments
*/
```

第3章　式と演算子

3.1　評価戦略

式は**正格**に評価され、演算子や関数は**値呼び**か**参照呼び**である。式が定数の場合は、積極的に**部分評価**する。

3.2　優先順位と結合

関数適用	`. () []`	**左結合**	**高**
実体化	`new`	**右結合**	
型変換	`:`	**左結合**	
累乗算	`**`	**右結合**	
集約演算	`reduce scan`	**左結合**	
否定演算	`! ~`	**右結合**	
乗除算	`* / %`	**左結合**	
単項演算	`+ -`	**右結合**	
ビットシフト	`<< >>`	**左結合**	
ビット論理積	`&`	**左結合**	
排他的論理和	`^`	**左結合**	
ビット論理和	`|`	**左結合**	
加減算	`+ -`	**左結合**	
レンジ	`..`	**左結合**	
順序比較	`<= => < >`	**左結合**	
等値比較	`== !=`	**左結合**	
短絡論理積	`&&`	**左結合**	
短絡論理和	`||`	**左結合**	**低**

3.3　オーバーロード

Chapel の演算子は、再定義することで新たなデータ型に対応する。これを**演算子のオーバーロード**と呼ぶ。

unary.chpl

```
proc !(str: string) return str + "!";
writeln(!"Hello, World");
```

上の例は、オーバーロードにより文字列を引数に取る演算子!を定義した。同様に、2 項演算子も定義できる。

binary.chpl

```
proc *(str: string, n: int) return + reduce (for 1..n do str);
```

第4章　変数と定数

4.1　変数

変数は、破壊的代入が可能で、その型はコンパイル時に決定する。下記は、int 型の変数 foo の宣言である。

var.chpl

```
var foo: int;
```

変数宣言では、変数の初期値を指定することもできる。下記は、int 型の変数 foo に初期値 12 を代入する。

var.chpl

```
var foo: int = 12;
```

初期値が明示的に与えられる場合は、型を省略できる。下記は、int 型の変数 foo に初期値 12 を代入する。

var.chpl

```
var foo = 12;
```

4.2　静的定数

定数の中でも、予約語 param を前置して宣言された定数は、*parameter* となる。本稿では、**静的定数**と呼ぶ。静的定数の値は、基本型か列挙型で、コンパイル時に確定する。下記は、int 型定数 foo に 12 を代入する。

param.chpl

```
param foo: int = 12;
```

静的定数の宣言では、定数値を明示する必要上、初期化の式を省略することはできないが、型は省略できる。

param.chpl

```
param foo = 12;
```

静的定数は、第 8.7 節に述べる通り、ジェネリクスの基盤である。また、定数への破壊的代入は禁止される。なお、予約語 config を予約語 param に前置して宣言する静的定数は、定数値をコンパイル時に変更できる。

param.chpl

```
config param foo = 121;
```

上記の静的定数 foo の値は、デフォルトでは 121 だが、コンパイル時に set オプションにより変更できる。

```
$ chpl param.chpl --set foo=12321;
```

4.3 動的定数

定数の中でも、予約語 const を前置して宣言された定数は、*constant* となる。本稿では、**動的定数**と呼ぶ。動的定数の値は、任意の型で、実行時に確定し、変更できない。下記は、int 型定数 bar に 12 を代入する。

const.chpl

```
const bar: int = 12;
```

動的定数の宣言では、定数値を明示する必要上、初期化の式を省略することはできないが、型は省略できる。

const.chpl

```
const bar = 12;
```

動的定数は、それ自体が *immutable* であることを示し、その参照先が *immutable* であることは保証しない。

const.chpl

```
class Bar {
  var mutable;
}
const bar = new Bar();
bar.mutable = 114514;
```

なお、予約語 config を予約語 const に前置して宣言する動的定数は、値をプログラム起動時に変更できる。

const.chpl

```
config const bar = 121;
```

上記の動的定数 bar の値は、デフォルトでは 121 だが、定数と同名のオプションにより起動時に変更できる。

```
$ chpl -o const const.chpl
$ ./const --bar=12321
```

4.4 型の別名

定数の中でも、予約語 type を前置して宣言された定数は、*type alias* となる。直訳すれば、**型の別名**である。

alias.chpl

```
type num = int;
```

型の別名は、変数宣言や関数定義を始め、あらゆる場所で利用できる。下記は、int 型定数 foo を宣言する。

alias.chpl

```
param foo: num = 121;
```

なお、予約語 config を予約語 type に前置して宣言する型の別名は、設定値をコンパイル時に変更できる。

alias.chpl

```
config type num = int;
```

上記の型の別名 num の値は、デフォルトでは int だが、コンパイル時に set オプションにより変更できる。

第5章　逐次処理

5.1　式と代入

式文とは、式の直後にセミコロンを付記した文である。式が評価され、その結果は受け取ることができない。
代入文は、左辺値の直後に**代入演算子** (= += -= *= /= %= **= &= |= ^= &&= ||= <<= >>=) と式を並べ、セミコロンを付記した文である。式を評価した結果を左辺値に代入する。左辺値は変数かフィールドである。

5.2　ブロック

複文とは、複数の文の並びをまとめ、その前後を括弧で閉じた文である。内部の文は先頭から順に実行する。
複文は、**レキシカルスコープ**を構成し、内部で宣言された変数や関数に対し、外部からの参照は禁止される。

block.chpl

```
{
  var a = 12;
  writeln(a);
}
writeln(a);
```

5.3　条件分岐

条件分岐とは、式を評価した結果により、実行する処理が切り替わる構文である。if 文と select 文がある。
if 文の条件式は bool 型である。else 節は不要な場合は省略可能である。条件式を囲む括弧は不要である。

if.chpl

```
if age < 20 then writeln("Adult Only") else writeln("Welcome");
```

予約語 then は条件式の範囲を明示する意味もある。then 節に複文を記述する場合は、then を省略できる。

if.chpl

```
if language == "Kotlin" {writeln("I love Kotlin");}
```

select 文は**多分岐**を行う。条件式と等しいラベルを持つ when 文を実行する。等値性は演算子==で検証する。

switch.chpl

```
select animal {
  when "cat" do writeln("meow");
  when "dog" do writeln("bowwow");
  otherwise writeln("gobblegobble");
}
```

当該のラベルがない場合、otherwise 文を実行する。when 文と otherwise 文は、固有のスコープを有する。

5.4 条件反復

条件反復とは、条件式が true である限り、処理を反復する構文である。while do 文と do while 文がある。while do 文では、まず条件式が評価され、true ならば do 節を実行する。do 節が終了すると条件式に戻る。

while.chpl

```
while i < 1024 do i *= 2;
```

do while 文では、まず do 節が実行され、条件式が true ならば do 節に戻り、false ならば反復を終了する。

while.chpl

```
do i *= 2 while i < 1024;
```

while do 文では、do 節が複文の場合、do を省略できる。なお、両文とも、条件式を囲む括弧は不要である。

5.5 要素反復

要素反復とは、イテレータが反復する限り、返値に対する処理を実行する構文である。第 6.8 節で解説する。

for.chpl

```
for i in 1..100 do writeln(i);
```

5.6 大域脱出

break 文は、条件反復や要素反復を強制的に終了する。continue 文は、条件反復や要素反復の先頭に戻る。条件反復や要素反復が入れ子の場合、break 文や continue 文の対象は、最内の条件反復や要素反復である。

break.chpl

```
for i in 1..10 do break;
```

ラベル文は、条件反復や要素反復に名前を与え、break 文や continue 文による**大域脱出**の対象に設定する。

label.chpl

```
label foo for i in 1..10 do for j in 1..10 {
  if i < j then continue foo;
  writeln(i, ".", j);
}
```

5.7 値の交換

スワップ演算子 <=>は、演算子の両辺の**左辺値**を交換する。左辺値とは、代入文の左辺に指定する値である。

swap.chpl

```
var a = "golden axe", b = "silver axe";
a <=> b;
```

上記のスワップ演算子を始め、代入演算子は、返値を void 型と定義しており、実質的に文として機能する。

第6章　関数

第 6 章では、Chapel の *procedure* の仕様に言及する。単に *function* と呼ぶと、第 6.8 節の *iterator* も含む。本稿では特記ない限り、関数は *procedure* を指す。下記は、`int` 型の引数を取る `void` 型の関数 `foo` である。

proc.chpl

```
proc foo(x: int, y: int): void {
  writeln(x + y);
}
```

引数の型は引数の名前の後にコロンを挟んで記述する。返値の型は引数宣言の後にコロンを挟んで記述する。引数が複数ある場合はカンマ区切りで引数宣言を並べるが、引数がない場合は括弧そのものを省略して良い。

proc.chpl

```
proc hoge: void {
  writeln("foo");
}
```

関数呼び出しは、関数名の後に実引数を記述する。関数定義で括弧を省略した場合は、括弧を記述できない。

proc.chpl

```
foo(1, 2);
hoge;
```

実引数は、対応する仮引数の名前を指定して渡すこともできる。例えば、下記の 2 行は同じ効果をもたらす。

proc.chpl

```
foo(x = 1, y = 2);
foo(y = 2, x = 1);
```

仮引数にはデフォルト値を指定できる。デフォルト値を持つ仮引数は、関数呼び出しで実引数を省略できる。

proc.chpl

```
proc foo(x: int = 1, y: int = 2): void {
  writeln(x + y);
}
```

可変長引数の関数は、実引数の個数が可変である。可変長引数の実体はタプルである。第 9 章で解説する。

proc.chpl

```
proc foo(x: int ...): void {
  for param i in 1..x.size do writeln(x(i));
}
```

可変長引数の関数は、実引数をカンマ区切りで並べることもできるし、代わりにタプルを渡すこともできる。

proc.chpl

```
foo(1, 2, 3);
foo((1, 2, 3, 4, 5));
```

関数を終了するには return 文を実行する。この時、**返値**を指定する。返値の型は返値に合わせて指定する。

proc.chpl

```
proc foo(x: int, y: int): int {
  return x + y;
}
```

void 型の場合は、返値なしで return 文を記述する。内容が return 文のみの関数は、波括弧を省略できる。

proc.chpl

```
proc foo(x: int, y: int): int return x + y;
```

6.1 ラムダ式

ラムダ式は、**無名関数**を定義する式である。下記は、引数 x と y を加算する無名関数を定数 add に代入する。

lambda.chpl

```
const add = lambda(x: int, y: int) return x + y;;
```

無名関数に限らず、関数は全て *first class function* である。int 型の引数を加算する関数 plus を定義する。

lambda.chpl

```
proc plus(x: int, y: int): int return x + y;
```

first class function は、変数に代入できる。例えば、下記のプログラムは、plus 関数を定数 add に代入する。

lambda.chpl

```
const add: func(int, int, int) = plus;
```

変数に代入した関数は、実引数を与えれば呼び出せる。同様に、関数の引数に**高階関数**を渡すこともできる。

lambda.chpl

```
proc apply(op: func(int, int, int), x: int, y: int): int return op(x, y);
```

なお、組込み関数 func は、関数型を表現する。可変長引数であり、引数の型と返値の型を順番に指定する。

6.2 関数の修飾子

修飾子の前置により、リンカの挙動を制御できる。export 修飾子は、関数をプログラムの外部に公開する。

link.chpl

```
export proc foo(x: int): int return 2 * x;
```

inline 修飾子は、関数を強制的にインライン展開して、関数を呼び出す全ての式を、関数の内容で置換する。

link.chpl

```
inline proc foo(x: int): int return 2 * x;
```

外部で実装された関数を利用する場合は、extern 修飾子を使う。下記は sched_getcpu 関数をリンクする。

link.chpl

```
extern proc sched_getcpu(): c_int;
```

外部で実装された関数を利用する際は、プロトタイプ宣言を記述したヘッダファイルを指定する必要がある。

link.h

```
int sched_getcpu();
```

```
$ chpl -o link link.chpl link.h
```

6.3 引数の修飾子

修飾子の前置により、関数の引数の挙動を制御できる。param 修飾子は、引数を第 4.2 節の静的定数にする。type 修飾子は、引数を第 4.4 節の型の別名にする。下記の makeTuple 関数は、指定の要件のタプルを返す。

argument.chpl

```
proc makeTuple(param dim: int, type eltType): dim * eltType {
  const tuple: dim * eltType;
  return tuple;
}
```

in 修飾子を前置した引数は、仮引数に実引数の複製が渡される。引数への代入は、実引数に影響を持たない。

argument.chpl

```
proc foo(in x: int): void {
  assert(x == 1);
  x = 12;
}
var a = 1;
foo(a);
assert(a == 1);
```

out 修飾子を前置した引数は、仮引数に実引数の複製が渡されない。引数への代入は、実引数に書き戻される。

argument.chpl

```
proc foo(out x: int): void {
  assert(x == 0);
  x = 12;
}
var a = 1;
foo(a);
assert(a == 12);
```

inout 修飾子は、in と out の双方の性質を持つ。ref 修飾子を前置した引数は、*passed by reference* になる。

6.4 返値の修飾子

修飾子の後置により、関数の返値の挙動を制御できる。param 修飾子は、返値を第 4.2 節の静的定数にする。type 修飾子は、返値を第 4.4 節の型の別名にする。下記の makeType 関数は、指定の要件のタプル型を返す。

return.chpl

```
proc makeType(param dim: int) type return dim * int;
```

ref 修飾子は、返値を左辺値にする。値を単に取り出すことも可能だが、代入式の左辺にすることもできる。

return.chpl

```
var tuple = (0, 0, 0, 0, 0, 0, 0, 0, 0);
proc A(i: int) ref: int return tuple(i);
writeln(tuple);
for i in 1..9 do A(i) = i;
writeln(tuple);
```

上記のプログラムを実行すると、下記の出力を得る。関数 A の返値は、まるで変数であるかのように見える。

```
(0, 0, 0, 0, 0, 0, 0, 0, 0)
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

6.5 関数の条件式

関数呼び出しは、実質的に**テンプレート関数**のインスタンス化である。その条件は where 節で指定できるが、値が静的に確定する式に限る。where 節は第 4.2 節の静的定数と並び、*generic programming* の根幹をなす。下記の whichType 関数は、引数 x の型を定数 t に格納し、条件式で検査する。関数の呼び分けが実現する。

where.chpl

```
proc whichType(x: ?t): string where t == real return "x is real";
proc whichType(x: ?t): string where t == imag return "x is imag";
```

静的な式はコンパイル時に計算される。これを**部分評価**と呼び、例えば、階乗をコンパイル時に計算できる。

where.chpl

```
proc fact(param n) param where n >= 1 return n * fact(n-1);
proc fact(param n) param where n == 0 return 1;
param fac6 = fact(6);
```

上記は、**メタ関数**を定義することにより、言語の機能を後天的に拡張する**メタプログラミング**の例でもある。C++でも、下記のテンプレートを定義すれば階乗をコンパイル時に計算できるが、直感的でなく煩雑である。

where.chpl

```
template<int n> struct fact {
  enum {val = n * fact<n-1>::val};
};
template<> struct fact<0> {
  enum {val = 1};
};
constexpr int fac6 = fact<6>::val;
```

コンパイルエラーを出力する compilerError 関数など、ある種の関数は、メタプログラミングに有用である。

6.6 関数の型推論

型推論とは、静的型付け言語で、変数や関数の引数、返値の型を省略しても、自動的に補完する機能である。

inference.chpl

```
proc foo(x, y) return x + y;
```

引数 x と引数 y の型は、関数のインスタンス化の際に、実引数の型から推論される。返値の型も同様である。

inference.chpl

```
foo(1.0, 2.0);
```

引数の型を省略し、その型を関数の中で取得するには、**クエリ式**を利用する。下記の xtype がその例である。

query.chpl

```
proc foo(x: ?xtype) {
  select xtype {
    when real do writeln(x, " is real");
    when imag do writeln(x, " is imag");
  }
}
```

ただし、省略された引数の型を取得するには、クエリ式を利用せずに、予約語 type を利用する方法もある。

query.chpl

```
proc foo(x) {
  select x.type {
    when real do writeln(x, " is real");
    when imag do writeln(x, " is imag");
  }
}
```

クエリ式は、定義域が不明な配列を受け取り、定義域を取得する際にも利用する。配列は第 12 章に述べる。

query.chpl

```
proc foo(x: [?D] ?t) {
  writeln("domain is ", D);
  writeln(typeToString(t));
}
```

6.7 入れ子関数

関数の定義は、他の関数の内部に記述できる。これを関数の**ネスティング**と呼び、関数の秘匿に利用できる。

nest.chpl

```
proc factorial(num: int): int {
  proc tc(n, accum: int): int {
    if n == 0 then return accum;
    return tc(n - 1, n * accum);
  }
  return tc(num, 1);
}
```

6.8 イテレータ

予約語 proc の代わりに iter を前置して定義した関数は *coroutine* となり、処理の中断と復帰が可能になる。例えば、下記の関数 foo を何度も呼び出すと、最初は整数 1、次に整数 2、最後に整数 3 を返して終了する。

iter.chpl

```
iter foo(): int {
  yield 1;
  yield 2;
  yield 3;
}
```

coroutine は、yield 文を実行する度に処理を中断し、指定された返値を返す。return 文の記述は禁止する。*coroutine* の実行は、前に実行した yield 文の直後で再開する。典型的には、*coroutine* は for 文で利用する。

iter.chpl

```
for i in foo() do writeln(i);
```

for 文は、指定された *coroutine* を何度も呼び出し、返値をインデックス変数に格納して、処理を実行する。なお、連番を返すだけの *coroutine* は、**レンジ演算子**で事足りる。下記は、1 から 5 までの整数を出力する。

iter.chpl

```
for i in 1..5 do writeln(i);
```

自身を再帰的に呼ぶ関数を**再帰関数**と呼ぶ。第 6 章の *procedure* だけでなく、*coroutine* も再帰関数にできる。

iter.chpl

```
iter foo(n: int): int {
  if n > 0 then for i in foo(n-1) do yield i;
  yield n;
}
```

イテレータ型とは、*coroutine* として these メソッドを実装したクラス型もしくはレコード型の総称である。

iter.chpl

```
class Foo {
  var min: int;
  var max: int;
}
```

クラスとレコードの詳細は、第 8 章で解説する。上記のクラス Foo に、下記のメソッド these を実装する。

iter.chpl

```
iter Foo.these(): int {
  for i in min..max do yield i;
}
```

these メソッドの定義により、Foo クラスはイテレータとしての機能を獲得し、for 文で利用可能になった。

iter.chpl

```
for i in new Foo() do writeln(i);
```

for 文は、イテレータの these メソッドを暗黙的に実行するが、明示的に *coroutine* を指定しても構わない。

第7章　並列処理

タスク並列は粗粒度の並列化に適し、**データ並列**は細粒度の並列化に適す。両者の詳細は、別著[1]に述べる。

7.1　タスク並列処理

begin 文は、指定された文を実行するタスクを分岐させ、sync 文は、指定されたタスクの終了を待機する。

task.chpl

```
sync begin writeln("hello, task!");
```

cobegin 文は、複文である。波括弧の内部の各文をタスク分岐により並列処理して、全文の終了を待機する。

task.chpl

```
cobegin {
  writeln("1st parallel task");
  writeln("2nd parallel task");
  writeln("3rd parallel task");
}
writeln("task 1, 2, 3 are finished");
```

上記の cobegin 文は、begin 文によるタスク分岐と、sync 文による待機で、下記に示すように代用できる。

task.chpl

```
sync {
  begin writeln("1st parallel task");
  begin writeln("2nd parallel task");
  begin writeln("3rd parallel task");
}
writeln("task 1, 2, 3 are finished");
```

通常は、cobegin 文で事足りるが、タスクを再帰的に合流させる動作が不要な場合に、sync 文を利用する。

task.chpl

```
inline proc string.shambles(): void {
  proc traverse(a: int, b: int) {
    if b > a {
      const mid = (a + b) / 2;
      begin traverse(a, 0 + mid);
      begin traverse(1 + mid, b);
    } else writeln(substring(a));
  }
  sync traverse(1, this.length);
}
```

[1] http://pafelog.net/wing.pdf

なお、cobegin 文の内部に代入文を記述する場合は、with 節を記述して、代入する変数を示す必要がある。

task.chpl

```
cobegin with(ref a, ref b) {
  a = sched_getcpu();
  b = sched_getcpu();
}
```

coforall 文は、並列化された for 文である。反復処理を実行する多数のタスクを生成し、終了を待機する。

task.chpl

```
coforall i in 1..100 do writeln("my number is ", i);
```

coforall 文は begin 文と sync 文の糖衣構文である。上記のプログラムは下記のプログラムと等価である。

task.chpl

```
sync for i in 1..100 do begin writeln("my number is ", i);
```

7.2 データ並列処理

forall 文は、coforall 文と異なり、並列化の挙動を詳細に制御できる for 文である。第 7.4 節で解説する。

data.chpl

```
forall i in 1..100 do writeln("my number is ", i);
```

reduce 演算子は、多数の値を集計して少数の値に変換する演算子である。これを**リダクション演算**と呼ぶ。
reduce 演算子の前に演算子 (+ * && || & | ^ min max minloc maxloc) を、後にイテレータを併記する。

reduce.chpl

```
const sum = + reduce (1..1000);
```

上記のプログラムは、演算の競合を回避しつつ並列処理される点を除けば、下記のプログラムと等価である。

reduce.chpl

```
for i in 1..1000 do sum += i;
```

演算子 minloc は最小値の位置を、maxloc は最大値の位置を求める演算子だが、引数と返値がタプルである。

reduce.chpl

```
var (value, idx) = minloc reduce zip(A, A.domain);
```

なお、zip は引数に与えたイテレータを結合する。例えば、zip(1..2, 3..4) は (1, 3), (2, 4) を返す。
scan 演算子は、リダクション演算の途中結果を順番に返すイテレータを返す。これを**スキャン演算**と呼ぶ。

reduce.chpl

```
for sum in + scan (1..100) do writeln(sum);
```

scan 演算子は、reduce 演算子と同様に、効率的に並列処理される。時系列データの積分などに応用できる。

7.3 並列処理の禁止

serial 文は、条件式の値が true の場合は、do 節内でタスクの分岐を禁止して、逐次処理を行う文である。

serial.chpl

```
serial true do begin writeln("executed");
```

7.4 並列イテレータ

forall 文と coforall 文は、ともに並列化された for 文である点では同等と言えるが、その挙動は異なる。

iter.chpl

```
coforall i in 1..100 do writeln(i);
```

coforall 文は、begin 文の糖衣構文であり、イテレーション毎にタスクを分岐し、処理の終了を待機する。

iter.chpl

```
sync for i in 1..100 do begin writeln(i);
```

forall 文は、それ自体は並列化の機能を持たず、並列化に関する全ての判断をイテレータの裁量に委ねる。

iter.chpl

```
forall i in 1..100 do writeln(i);
```

forall 文は、以下に例示する *leader iterator* により並列化し、末端の逐次処理は *follower iterator* で行う。

iter.chpl

```
iter range.these(param tag): range(int) where tag == iterKind.leader {
  if this.size > 16 {
    const mid = (this.high + this.low) / 2;
    const (rng1, rng2) = (this(..mid), this(mid+1..));
    cobegin {
      for i in rng1.these(iterKind.leader) do yield i;
      for i in rng2.these(iterKind.leader) do yield i;
    }
  } else yield this;
}
```

上記の *leader iterator* は、レンジを再帰的に等分割しつつ並列化する。下記の *follower iterator* も実装する。

iter.chpl

```
iter range.these(param tag, followThis) where tag == iterKind.follower {
  for i in followThis do yield i;
}
```

以上のオーバーロードにより、forall 文で利用可能になる。実質的に、forall 文は下記の糖衣構文である。

iter.chpl

```
for subrange in (1..100).these(iterKind.leader) {
  for i in (1..100).these(iterKind.follower, subrange) do writeln(i);
}
```

第8章　複合型

複合型とは、第 2.2 節の**単純型**の対義語であり、複数の単純型や複合型の組み合わせで定義される型である。第 8 章で取り上げる**クラス**や**レコード**は、複合型である。他にも、タプルやレンジの実体はレコードである。複合型の中でも、クラスは**参照型**の性格を有する。new 演算子でインスタンス化し、delete 文で破棄する。参照型とは、変数や即値がインスタンスを参照する型である。明示的にインスタンス化を行う必要性がある。

class.chpl

```
class Foo {}
const foo: Foo = new Foo();
delete foo; // OK
```

複合型の中でも、レコードは**値型**の性格を有する。new 演算子でインスタンス化し、delete 文は禁止する。値型とは、変数や即値がインスタンスとなる型である。本来、明示的なインスタンス化や破棄は不要である。

record.chpl

```
record Bar {}
const bar: Bar = new Bar();
delete bar; // NG
```

参照型の変数は、何らかの値を代入する以前は、有効なインスタンスを参照しない。この状態を nil と呼ぶ。

nil.chpl

```
class Foo {}
var foo: Foo = nil;
writeln(foo);
```

8.1　メンバ変数

フィールドとは、クラスやレコードがインスタンス毎に有する変数であり、複合型という名称の所以である。フィールドの型に制限はなく、再帰的なクラスの定義も実現する。下記は、**リスト構造**を実装する例である。

field.chpl

```
class List {
  var value: int;
  var next: List;
}
```

特定のインスタンスのフィールドを参照するには、**ドット演算子**で、インスタンスとフィールドを指定する。

field.chpl

```
var cons = new List();
var cdr = cons.next;
cons.value = 114514;
```

Chapel では、フィールドへの参照は、左辺値を返すメソッドを暗黙的に呼び出す。下記はその実体である。

field.chpl

```
proc List.value ref: int return value;
```

フィールドと同名で、引数を持たず、左辺値を返すメソッドを**アクセサ**と呼び、明示的な定義も可能である。

field.chpl

```
proc Account.name ref: string {
  if name != "Tomori Nao" then name = "Tomori Nao";
  return name;
}
```

8.2 メンバ関数

メソッドとは、クラスやレコードに属する関数であり、必要であればインスタンスのフィールドを参照する。

method.chpl

```
class Add {
  const x, y: real;
  proc print(): void {
    writeln(x + y);
  }
}
```

特定のクラスやレコードのメソッドを呼び出すには、ドット演算子で、インスタンスとメソッドを指定する。

method.chpl

```
const add = new Add(x = 1, y = 2);
add.print();
```

定義済みのクラスやレコードにもメソッドを追加できる。その場合は、クラスやレコードの名前を明記する。

method.chpl

```
proc Add.add(): real return x + y;
```

8.3 生成と破棄

インスタンス化では、当該のクラスやレコードと同名のメソッドを実行する。これを**コンストラクタ**と呼ぶ。コンストラクタは明示的に定義することもでき、その場合、引数は自由に設定できるが、返り値は禁止する。

constructor.chpl

```
class Add {
  const x: real = 0;
  const y: real = 0;
  proc Add(x: real, y: real) {
    this.x = x;
    this.y = y;
  }
}
```

コンストラクタを省略した場合は、フィールドの初期値を引数に取るコンストラクタが自動的に定義される。

constructor.chpl

```
var add2 = new Add(x = 1, y = 2);
```

この場合、引数にはデフォルト値が設定され、フィールドの変数宣言の初期化式の値がデフォルト値になる。

constructor.chpl

```
var add1 = new Add(x = 1);
var add3 = new Add(y = 2);
```

クラスやレコードは、インスタンスが消滅する際に実行する処理を定義できる。これを**デストラクタ**と呼ぶ。デストラクタは明示的に定義できる。ただし、引数や返り値は禁止する。下記は、デストラクタの例である。

destructor.chpl

```
proc Foo.~Foo() {
  writeln("deleted");
}
```

8.4 自身の参照

クラスやレコードが、インスタンスを明示的に参照する際は、予約語 this を用いる。使用例を以下に示す。

this.chpl

```
proc Account.setName(name: string): void {
  this.name = "@" + name;
}
```

上記の代入文の左辺の name は Account クラスのフィールド name を、右辺の name は引数 name を参照する。参照型の性格を持つクラスに限らず、値型の性格を持つレコードでも、this は参照である点に注意を要する。

this.chpl

```
record Account {
  var name: string;
}
```

上記のレコード Account のインスタンスを引数に、フィールド name を変更する関数 setName を考えてみる。

this.chpl

```
proc setName(account: Account, name: string): void {
  account.name = name;
}
```

外部からは、フィールド name は不変に見える。レコードは値型で、引数にはコピーが渡されるためである。

this.chpl

```
var nao: Account;
setName(nao, "Tomori Nao");
writeln(nao);
```

予約語 this はコピーではなく参照であるため、意図する通りに自身のフィールドに代入することができる。

8.5 ファンクタ

this メソッドを定義したクラスやレコードは、実質的に関数としても機能する。これを**ファンクタ**と呼ぶ。

functor.chpl

```
class Add {
  proc this(a: int, b: int): int return a + b;
}
```

上記のクラス Add は、2 個の整数を引数に取り、加算を行う this メソッドを実装する。定数 add に格納し、第 6.1 節の *first class function* と同様に引数を与えて呼び出す。123+45 を計算し、出力として 168 を得る。

functor.chpl

```
const add = new Add();
writeln(add(123, 45));
```

8.6 継承と派生

継承とは、あるクラスやレコードが、他のクラスやレコードのフィールドやメソッドを引き継ぐことである。

inheritance.chpl

```
class Foo {
  proc foo(): string return "foo!";
}
```

上記のクラス Foo は foo メソッドを実装する。これに対し、下記のクラス Bar も foo メソッドを実装する。

inheritance.chpl

```
class Bar: Foo {}
```

クラス名の直後にコロンを挟んでクラス名を記述すると、それを**基底クラス**とする**派生クラス**の宣言になる。派生クラス Bar は基底クラス Foo のフィールドやメソッドを有する。必要に応じ、メソッドは再定義できる。

inheritance.chpl

```
proc Bar.foo(): string return "bar!";
```

上記は、Foo クラスから継承した foo メソッドの挙動を再定義する例である。これを**オーバーライド**と呼ぶ。

8.7 総称型

クラスやレコードのフィールドで、第 4.2 節の静的定数や第 4.4 節の型の別名を宣言すると、**総称型**になる。

generics.chpl

```
record Stack {
  type eltType;
}
```

総称型は、型を引数に取る型であり、リスト等のデータ構造の実装を特定の型から分離する際に有用である。

generics.chpl

```
var a: Stack(eltType = uint);
var b: Stack(eltType = real);
```

上記の変数 a と b は、両者ともに Stack 型だが、相互に異なる型を持つ。typeToString 関数で確認できる。

generics.chpl

```
writeln("a is ", typeToString(a.type));
writeln("b is ", typeToString(b.type));
```

上記のプログラムを実行して、下記の出力を得ることから、Stack 型は、型を引数に取る総称型だとわかる。

```
a is Stack(uint(64))
b is Stack(real(64))
```

総称型の引数となる型を *parameterized type* と呼ぶ。*parameterized type* は型に限らず、静的定数でも良い。

8.8 構造的部分型

特定の関数を実装したクラスやレコードは、特定の仕様を満たす型と認定する。これを *duck typing* と呼ぶ。

duck.chpl

```
class Duck {}
class Kamo {}
```

上記のクラス Duck と Kamo は相互の継承関係にないが、同型の引数を宣言した quack メソッドを実装する。

duck.chpl

```
proc Duck.quack(): string return "quack!";
proc Kamo.quack(): string return "quack!";
```

下記の関数 duck を定義する。型が省略された引数 x を受け取り、引数 x に対し quack メソッドを実行する。

duck.chpl

```
proc duck(x): string return x.quack();
```

下記のクラス Ibis を定義する。前掲のクラス Duck や Kamo を継承しない。quack メソッドも未実装である。

duck.chpl

```
class Ibis {}
```

Duck と Kamo と Ibis のインスタンスを duck 関数に与える。Ibis クラスの場合はコンパイルエラーとなる。

duck.chpl

```
duck(new Duck()); // OK
duck(new Kamo()); // OK
duck(new Ibis()); // NG
```

関数が参照するメソッドやフィールドに基づき、引数の型を暗黙的に制限する仕組みを**構造的部分型**と呼ぶ。

第9章　タプル

タプルは、値をカンマ区切りで並べたデータ型である。要素の型を揃える必要はなく、配列より軽量である。括弧で囲むことで生成する。個別の要素を参照するには、1 から始まる番号を添えて this メソッドを呼ぶ。

tuple.chpl

```
var names = ("Tom", "Ken", "Bob");
const ken = boys(2);
names(3) = "Robert";
```

タプル型は、要素の型をカンマ区切りで並べた組で表現する。もしくは、要素の個数と型の乗算で表現する。

tuple.chpl

```
const tup1: (int, int, int) = (1, 2, 3);
const tup2: 3 * int = (1, 2, 3);
```

タプルは、要素の型が揃っている場合、these メソッドを呼び出すことにより、イテレータとして機能する。

tuple.chpl

```
for boy in ("Tom", "Ken", "Bob") do writeln(a);
```

複数の変数や定数を宣言する際は、変数名や定数名をカンマ区切りで並べたタプルによる宣言も可能である。

tuple.chpl

```
var (a, b): (string, int);
var (c, d): 2 * int;
```

同様に、複数の変数に対する代入も、タプルを利用して纏めることができる。これを**アンパック代入**と呼ぶ。

tuple.chpl

```
(a, b) = ("John", 24);
```

タプルによる変数宣言は、引数宣言にも適用できる。下記は、タプル (y, z) を引数に取る関数を定義する。

tuple.chpl

```
proc foo(x: int, (y, z): (int, int)): int return x * (y + z);
```

第 6 章の可変長引数の実体はタプルであるが、反対に、タプルを展開して固定長引数に与えることもできる。

tuple.chpl

```
proc mul(x: int, y: int): int return x * y;
const mul23 = mul((...(2, 3)));
```

上記 2 行目の ... は**タプル展開演算子**であり、本来は固定長引数である関数 mul に、実引数 2 と 3 を与える。

第10章　範囲

レンジは、整数の範囲を表現するデータ型である。第 11 章の**領域**に比べて軽量で、実体はレコードである。

range.chpl
```
const rng = 1..100;
```

レンジ演算子 .. で生成する。最小値と最大値を省略することにより、半無限区間や無限区間も表現できる。

range.chpl
```
const from1toInf: range(int) = 1..;
```

必要に応じて、要素の個数は #演算子で、周期は by 演算子で、基準点の調整は align 演算子で指定できる。

range.chpl
```
writeln(10..30 by -7 align 13 # 3);
```

要素の個数は size メソッドで取得できる。最小値と最大値は low メソッドと high メソッドで取得できる。

range.chpl
```
assert((1..100).size == 100);
```

レンジ型は、要素の型と有界性と不連続性を指定して表現する。

range.chpl
```
range(type idxType = int, boundedType = BoundedRangeType.bounded, stridable = false)
```

boundedType は有界性を表現し、デフォルトは bounded だが、最小値と最大値を省略すれば、無限になる。

bounded	最小と最大の両方が有限の有限区間
boundedLow	最小が有限で最大が無限の半開区間
boundedHigh	最小が無限で最大が有限の半開区間
boundedNone	最小と最大の両方が無限の無限区間

stridable は不連続性を表現し、デフォルトは false だが、by 演算子を使用した場合に限り true になる。

range.chpl
```
assert((1..3 by 2).stridable);
```

レンジは、these メソッドを呼び出すことで、イテレータとして機能する。典型的には、for 文で利用する。

range.chpl
```
for i in 1..5 do writeln(i);
```

these メソッドは、第 7.4 節の *leader iterator* と *follower iterator* をオーバーロードし、並列化に対応する。

第 11 章　領域

領域は、空間や集合を表現するデータ型である。第 10 章のレンジに比べて高級で、実体はレコードである。
矩形領域は、レンジをカンマ区切りで並べて生成し、任意次元の矩形空間や、矩形配列の定義域を表現する。
連想領域は、包含する具体的な要素をカンマ区切りで並べて生成し、集合や、連想配列の定義域を表現する。

domain.chpl

```
const rectangular = {0..10, -20..20};
const associative = {"foo", "bar"};
```

矩形領域の場合、dims メソッドでレンジのタプルに変換できる。個別のレンジは dim メソッドで取得する。

domain.chpl

```
var dom: domain(2) = {1..10, 1..20};
writeln(dom.dims());
writeln(dom.dim(2));
```

矩形領域の範囲を設定するには、レンジのタプルを引数に、setIndices メソッドを実行する。

domain.chpl

```
var dom: domain(2) = {1..10, 1..20};
dom.setIndices((-100..100, 1..200));
```

矩形領域型は、次元数と要素の型と不連続性を指定して表現する。

domain.chpl

```
domain(rank: int, type idxType = int, stridable = false)
```

rank は次元数で、rank メソッドで静的定数として取得できる。idxType は、各次元の要素の型を表現する。

domain.chpl

```
param rank = {1..10, 1..20, 1..30}.rank;
```

stridable は不連続性を表し、デフォルトは false である。連想領域型は、要素の型を指定して表現する。

domain.chpl

```
domain(idxType = string)
```

矩形領域と連想領域は、暗黙的に these メソッドを呼び出すことで、両者ともにイテレータとして機能する。

domain.chpl

```
for xyz in {1..10, 1..10, 1..10} do writeln(xyz);
for boy in {"Tom", "Ken", "Bob"} do writeln(boy);
```

these メソッドは、第 7.4 節の *leader iterator* と *follower iterator* をオーバーロードし、並列化に対応する。

第12章　配列

配列は、第 11 章の領域を定義域に持ち、値域への写像を表現するデータ型であり、実体はレコードである。配列の定義域により、矩形領域を定義域に持つ**矩形配列**と、連想領域を定義域に持つ**連想配列**に分類できる。

array.chpl

```
const rectangular = [1, 2, 3, 4, 5, 6, 7, 8];
const associative = [1 => "one", 2 => "two"];
```

配列型は領域を括弧で括り、要素の型を指定して表現する。領域を直に指定する場合、波括弧は省略できる。

array.chpl

```
var A: [{1..10, 1..10}] real;
var B: [{'foo', 'bar'}] real;
```

配列の要素を参照する際は、要素のインデックスを引数として、左辺値メソッド `this` を暗黙的に実行する。

array.chpl

```
A[1, 2] = 1.2;
A(3, 4) = 3.4;
```

上記のインデックスはタプルでも指定できる。インデックスに領域を指定した場合、部分配列を取得できる。

array.chpl

```
for (i, j) in A.domain do A(i, j) = i * 0.1 * j;
writeln(A(1..2, 1..3));
```

上記のプログラムを実行すると、下記の出力を得ることから、正しく部分配列を取得していることがわかる。

```
1.1 1.2 1.3
2.1 2.2 2.3
```

配列は、`these` メソッドを実装し、イテレータとして機能する。これは左辺値を返し、`for` 文で代入できる。

array.chpl

```
var boys = ['Tom', 'Ken', 'Bob'];
for boy in boys do boy += '-san';
writeln(boys);
```

矩形配列で、値が 0 の要素が多く、全要素を格納するコストが過大である場合、*sparse array* も利用できる。

array.chpl

```
var SpsD: sparse subdomain({1..16, 1..64});
var SpsA: [SpsD] real;
SpsD += (8, 10);
SpsD += (3, 64);
SpsA[8, 10] = 114.514;
```

配列の外見はレコードだが、内部的には、矩形配列や連想配列を実装するクラスのインスタンスを参照する。領域も同様で、配列や領域を引数に取る関数は *pass by value* だが、実質的には *pass by reference* に見える。

array.chpl

```
proc foo(arr: [] int) {
  arr = [2, 3, 4];
}
var A = [1, 2, 3];
foo(A);
writeln(A);
```

配列の代入は、右辺の配列の全要素を、左辺の配列の対応する要素に代入する操作である点に注意を要する。

array.chpl

```
var A: [1..10] int;
var B = A;
for a in A do a = 1;
writeln("A is ", A);
writeln("B is ", B);
```

上記のプログラムを実行すると、下記の出力を得る。これは、配列が参照型ではなく値型である証左である。

```
A is 1 1 1 1 1 1 1 1 1 1
B is 0 0 0 0 0 0 0 0 0 0
```

配列のエイリアスが必要な場合は、**エイリアス演算子** `=>`を利用する。部分配列のエイリアスも作成できる。

array.chpl

```
var A: [1..10] int;
var B => A[2..9];
for b in B do b = 1;
writeln("A is ", A);
```

上記のプログラムを実行すると、下記の出力を得る。配列 `B` に対する操作が配列 `A` に反映することがわかる。

```
A is 0 1 1 1 1 1 1 1 1 0
```

配列は、**区分化大域アドレス空間**に従い、*locale* と呼ばれる分散ノードに適切に分散配置する機能を有する。`dmapped` 演算子により、下記の矩形配列 `A` は格子状に分散配置され、矩形配列 `A` は周期的に分散配置される。

array.chpl

```
use BlockDist, CyclicDist;
var A: [{1..10,1..10} dmapped Block(boundingBox={1..10,1..10})] real;
var B: [{1..10,1..10} dmapped Cyclic(startIdx=(1,1))] real;
```

分配された配列や領域に対し、*locale* が自身の領域を取得するには、`localSubdomain` メソッドを利用する。

array.chpl

```
for locale in Locales do on locale {
  for (i, j) in A.localSubdomain() {
    writeln((i, j), " @ ", here.id);
  }
}
```

`Locales` は実行環境で利用可能な *locale* を格納する配列であり、`on` 文は指定された *locale* で文を実行する。

第13章　その他の型

13.1　列挙型

列挙型は、何らかのカテゴリを示す識別子の有限集合を表現するデータ型で、個別の識別子を**列挙子**と呼ぶ。

enum.chpl

```
enum Rabbit {Lapin, Lievre};
```

13.2　共用体

共用体は、状況によりデータ型が変化する変数に対し、適切な型を指定して参照可能としたデータ型である。下記の共用体 Pure の場合、フィールド r と i は同じメモリ領域に占位する。バイト境界は適切に調整する。

union.chpl

```
union Pure {
  var r: real;
  var i: imag;
}
var pure: Pure;
pure.r = 19.19;
```

最後に値を代入したフィールドのみが有意である。なお、クラスやレコードと同様にメソッドを定義できる。

13.3　同期型

下記は、sync 型と single 型の**同期変数**の利用例である。タスクは、同期変数 release$の更新を待機する。

sync.chpl

```
var count$: sync int = 32;
var release$: single bool;
coforall i in 1..32 {
  const mc = count$;
  if mc != 1 {
    count$ = mc - 1;
    writeln("wait");
    release$;
  } else release$ = true;
}
```

同期変数は、変数値の他に *full* か *empty* の状態を有する。初期値は *empty* で、代入により *full* に遷移する。同期変数への代入は *empty* 時のみ、参照は *full* 時のみ可能である。さもなければ、遷移するまで待機する。sync 変数のみ、変数値の参照で *empty* に遷移する。反対に、代入済みの single 変数は再代入を拒絶する。

第14章　モジュール

名前空間とは、変数や関数が可視となる範囲を module 文で区分し、衝突の可能性を回避する仕組みである。

module.chpl

```
module Foo {
  const piyo = "piyo";
  proc hoge(): string return "hoge";
}
```

上記の名前空間 Foo で宣言された変数 piyo や関数 hoge は、外部からの参照に**ドット演算子**が必要になる。

module.chpl

```
writeln(Foo.hoge());
```

use 宣言を記述すれば、記述した名前空間の内部では、対象の名前空間に対し、ドット演算子を省略できる。

module.chpl

```
use Foo;
writeln(piyo);
```

明示的な名前空間に属さない文は、ファイル名から拡張子.chpl を除去した名前の名前空間に暗黙的に属す。
名前空間は、プログラムの起点となる main 関数を明示的に定義できる。引数はなく、返値は int 型である。

module.chpl

```
proc main() {
  writeln("Hello, World!");
}
```

プログラムが起動すると、main 関数を定義した名前空間が初期化され、名前空間の直下の文が実行される。
依存関係に従い、全ての名前空間が初期化すると、main 関数を実行する。main 関数は重複して定義できる。

module.chpl

```
module Foo {
  proc main() {
    writeln("This is Foo");
  }
}
module Bar {
  proc main() {
    writeln("This is Bar");
  }
}
```

この場合、どの名前空間に属す main 関数でプログラムを起動するか、コンパイル時に指定する必要がある。

```
$ chpl --main-module Foo module.chpl
```