目次

準備

言語の位置づけ

Ag は、C++によるソフトウェアに容易に組み込める Lisp 的な外観を持つ言語(およびその処理系等を含む総称)である。

字句解析系および評価系はクラスライブラリとして提供され、ほとんどの場合必要な部分のみを派生クラス内で定義すればよいだけである。したがって、新規・既存の C++プロジェクトを問わず極めて容易に導入可能である。また、STL コンテナテンプレートなど、C++側データとの連携も自在であることから、拡張言語としての対外的な用法にとどまらない。
Ag の言語的な特徴としては、Lisp 風味であることが第一にあげられる。ただし、外観はほとんど Lisp であるが、破壊的であることをデフォルトし且つ危険で小汚い言語仕様をモットーとしているため、あえて Lisp の亜種ではなくあくまで Lisp 風味の C++用組込スクリプト言語という位置づけである。

本文書における準備

本文書では、繰り返し現れる語については略記することとする。

Common Lisp	CL
Emacs Lisp	EL

なお、Lisp と記す場合、ことわりの無い限り scheme を含む Lisp 系言語全般またはそれらの全般的な特性を指すこととし、特定の Lisp 言語を指すものではない。

例題では、関数 print を使えるものと仮定する。print は、渡された値を仮想的なディスプレイに表示し改行を行う。戻り値は与えられた値そのものである。

```
(progn (print 'foo) (print 'bar)) ;==> bar

DISPLAY> foo
DISPLAY> bar
```

式の評価値を明示する場合には、;==>を使うものとする。

```
(+ 1 2) ;==> 3

(list 'foo 'bar 'baz) ;==> (foo bar baz)
```

変数オブジェクト

変数オブジェクトは Ag における変数の実体である。概念的には、シンボルが変数オブジェクト(変数を格納するオブジェクト)を束縛し、その変数オブジェクトが二次的に変数値を束縛していることになる。
特殊作用子 variable は局所変数または大域変数の変数オブジェクトを与える。

```
[x <- 'foo]

[var <- (variable x)]
```

関数 vref は変数オブジェクトに対するアクセサである。

```
(vref var) ;==> foo
[(vref var) = 'bar]
x ;==> bar
```

なお、variable と vref には入力マクロとして#&と#*が用意されている。

変数オブジェクトはポインタのように使えるが、普通のオブジェクトであるためいつの間にか参照先が無効化するというような事が起こらない（変数オブジェクトは存続中は対象のオブジェクトを束縛し続ける）。したがってポインタのように危険ではない。多くの場合、変数の参照渡しを行うために有用である。

```
(defun compute-square (var)
 [#*var = #*var * #*var])
(defun test (num)
 (compute-square #&num)
 num)
(test 3) ;==> 9
```

演算子と中置式

Ag には、中置式をサポートする組込マクロ iexpr が備わっている。

これを使えば、例えば

```
(+ (* a b) (/ c d))
```

を

```
(iexpr a * b + c / d)
```

と書くことができる。この例でも明らかなように、中置演算子*と/は、+に優先して結合している。

なお、マクロ iepr は入力マクロ[]によってより明瞭に記述できる。

```
[a * b + c / d]
```

中置式はもちろん S 式に変換される。

```
(expand '[a * b + c / d]) ;==> (+ (* a b) (/c d))
```

前置演算子は複数の引数を取ることができる。

```
[+ 1 2 3 * - 6 5 4] ;==> -18
```

二項（中置）演算子の定義

二項演算子を定義するには関数 defbinary を用いる。

(defbinary prio op-sym fn-sym :lock #f :serial #f)

ここで、prio は結合プライオリティで0から15までの整数である。値が大きいほど結合力が強くなる。プライオリティ0は通常、代入や束縛の演算子に用いる。+や-は9、*や/は10である。
op-sym は演算子のシンボル、fn-sym は演算の実体となる関数またはマクロのシンボルである。なお、op-sym と fn-sym が同一の場合,fn-sym は省略できる。

例として、加算と乗算を行う独自の中置演算子 my:+と my:*を定義する。

```
(defbinary 9 'my:+)

(defbinary 10 'my:*)
(defun my:+ (a b) (+ a b))
(defun my:* (a b) (* a b))
```

適用例:

```
[3 my:* 4 my:+ 5 my:* 6] ;==> 42
```

上の例はまた、関数+と-の呼び出しを行っているだけであるから、マクロを用いたほうが実行効率は良くなる。

```
(defmacro my:+ (&rest body) `(+ ,@body))

(defmacro my:* (&rest body) `(* ,@body))
```

しかしながら、上の defbinary の定義例では、実行上の問題は何もないが呼出効率上の致命的問題を抱えている。
例えば、次の S 式、

```
(+ a (* b c d) (* e f g))
```

に変換されると期待して

```
[a my:+ b my:* c my:* d my:+ e my:* f my:* g]
```

と書いても期待どおりには変換されない。
実際に expand を適用して変換させてみると、

```
(expand '[a my:+ b my:* c my:* d my:+ e my:* f my:* g])

;==> (+ (+ a (* (* b c) d)) (* (* e f) g))
```

と変換されて、関数+と*を用いた効率的な展開形になっていないことがわかる。

期待通りに効率の良い形式に変換させたければ、キーワード引数 serial に真値を設定すればよい。
serial が真値の場合、iexpr マクロは実体関数(マクロ)にオペランドを直列に渡す形式に S 式を構築する。

```
(defbinary 9 'my:+ :serial #t)

(defbinary 10 'my:* :serial #t)
(expand '[a my:+ b my:* c my:* d my:+ e my:* f my:* g])
;==> (+ a (* b c d) (* e f g))
```

最後に、キーワード引数 lock に真値を与えた場合、次回から演算子定義の上書きを禁止する。ただし、同じ定義であれば

再定義はスルーされる。

```
(defbinary 9 'my:+ :serial #t :lock #t) ;== #t

(defbinary 9 'my:+ :serial #t :lock #t) ;== #t <同一定義>
(defbinary 9 'my:+ :serial #f :lock #t) ;== #f <serial 属性が異なる>
(defbinary 10 'my:+ :serial #f :lock #t) ;== #f <プライオリティ値が異なる>
```

＊前置演算子＊

＊真性な末尾再帰＊

末尾再帰は動的にループ化される。Ag における末尾再帰の範囲は広い。

```
(defun foo (n &optional (acc 0))
 (if [n > 0]
   (bar (1- n) [acc + n])
   acc))

(defun bar (n &optional (acc 0))
 (if [n > 0]
   (foo (1- n) [acc + n])
   acc))
```

この定義では、foo と bar が相互を呼び出しあって総和を求めているが、どちらも末尾再帰となっているため、スタックオーバーフローを引き起こすことなく動的にループ化される。

```
(foo 100000) ;==> 5000050000
```

更に、funcall と apply も末尾呼び出しになりうる。

```
(defun foo (n fn &optional (acc 0))
 (if [n > 0]
   (funcall fn (1- n) fn [acc + n])
       acc))
(foo 100000 #'foo) ;==> 5000050000

(defun foo (n fn &optional (acc 0))
 (if [n > 0]
   (apply fn (1- n) fn [acc + n] nil)
       acc))
(foo 100000 #'foo) ;==> 5000050000
```

言語の特色

ラムダリスト

Ag のラムダリストは表現力を優先させている。例えばレスト引数の後ろにキーワード引数を配置した場合、

```
(defun foo (&rest rest &key key) (list rest key))

(foo 1 2 3 :key 4) ;==> ((1 2 3) 4)
```

となる。

この例の foo 呼び出しは CL ではエラーとなるが、逆に CL の場合、

```
(defun cl-foo (&rest rest &key a b) (list rest a b))

(cl-foo :a 'bar :b 'baz) ;==> ((:a 'bar :b 'baz) bar baz)
```

というようなことができる。Ag で同じことを望む場合、

```
(defun ag-foo (&rest rest)
  (list-bind (&key a b) rest
    (list rest a b)))
```

とでもすればよいだろう。

```
(ag-foo :a 'bar :b 'baz) ;==> ((:a 'bar :b 'baz) bar baz)
```

キーワード引数はまた、:flag 指定子によりペアをもたないフラグキーワード、:rest 指定子により残余を値とするレストキーワード、:pair 指定子により通常のキーワードであるペアキーワードの三種を指定できる。

```
(defun foo (&key :flag a b :rest c :pair (d 0)) (list a b c d))

(foo) ;==> (#f #f nil 0)
(foo :a :d 1 :c 'bar 'baz) ;==> (#t #f (bar baz) 1)
(foo :b :c 'bar 'baz :d 2) ;==> (#f #t (bar baz :d 2) 0)
```

上の例では、a と b がフラグキーワードであり、引数に指定されると真となる。
c はレストキーワードであるので、引数に指定されると、それ以降の値がリストとして渡される。
d は通常のキーワードである。
なお、フラグキーワードとレストキーワードに対しては、

```
(&key :flag (a 1) :rest (r '(1 2))) ;; NG
```

のようにデフォルト値の指定はできない。

Ag で使用できるラムダリストキーワードは&aux &key &optional &rest の 4 つ。

位置についてのルール
1) 同一のラムダリストキーワードは連続してもよい。
(&optional a b &optional c)は(&optional a b c)と等価。ただし、
(&key :flag a b &key c)は(&key :flag a b :pair c)と等価。

2) &aux はどの位置に置いてもかまわない。ただし、&aux の前方にあるラムダリストキーワードは後方に置くことができなくなる。
3) &key の後ろには、&aux 以外のラムダリストキーワードを置くことはできない。
4) &rest の後ろには、&key と&aux 以外のラムダリストキーワードを置くことはできない。

＊便利な述語 got?＊
キーワード引数によって動作を変える次のような関数を考える。

```
(foo :print 'bar) ; bar を表示
(foo :quote 'bar) ; 'bar を返す
(foo) ; エラー
```

さて、このような関数は、あまり深く考えなければ

```
(defun foo (&key print quote)
 (cond {print (print print)}
    {quote `',quote}
       {#t (error)}))
```

となるだろう。実際、上の呼び出し例は期待通りに動く。L
ところが、この定義には致命的な欠陥がある。つまり、nil が与えられた場合、その nil が明示的に与えられたものなのか暗黙に与えられたものなのかは判別不可能である。

```
(foo :quote nil) ;==> エラー
(foo :print nil) ;==> エラー
```

これを防ぐために、おそらく殆どの場合は何か特別な値をデフォルト値にしておけば済むだろう。

```
(defun foo (&key (print 'NULL) (quote 'NULL))
 (cond {[print != 'NULL] (print print)}
    {[quote != 'NULL] `',quote}
       {#t (error)}))
```

ところがこれも、その特別な値を渡された場合には機能しなくなるのは同じである。
Ag では、そのような心配のない関数を記述するため、述語 got?を用いることができる。

```
(defun foo (&key print quote)
 (cond {(got? print) (print print)}
    {(got? quote) `',quote}
       {#t (error)}))
```

got?はラムダリストパラメータにのみ適用可能な特殊作用子で、そのパラメータが明示的に与えられたものであれば真、デフォルト値が与えられたものであれば偽を返す。

```
(defun got-test (&optional a b &key c :rest d)
 (list (got? a) (got? b) (got? c) (got? d)))
(got-test) ;==> (#f #f #f #f)
(got-test 1) ;==> (1 #f #f #f)
(got-test 1 2 :c nil) ;==> (1 1 1 #f)
(got-test :d 1 2 3) ;==> (#f #f #f 1)
(got-test :d) ;==> (#f #f #f 1)
```

ここで、got?が真を与える場合の真値として1が与えられていることに注意。
got?は1つ以上のラムダリストパラメータを取ることができ、少なくともその内の1つのパラメータが明示的に与えられているならば、その内で明示的に与えられたパラメータの個数(>=1)を返す。明示的に与えられたパラメータが一つもなければ、偽値が反される。
また、got?は多値を返し、二番目の返り値は明示的に**与えられなかった**パラメータの個数である。

```
(defun got-test-2 (&key :flag a b c)
 (values::list (got? a b c)))

(got-test-2) ;==> (#f 3)
(got-test-2 :a) ;==> (1 2)
(got-test-2 :b :c) ;==> (2 1)
(got-test-2 :a :b :c) ;==> (3 0)
```

名前空間

まず先に断っておかなければならない。Ag には厳格な名前空間もパッケージシステムも存在しない。
しかしながら、複数モジュールを単一パッケージとして機能させることや、モジュールローカル(C/C++でいうところの static)なシンボルを供給することで、名前の衝突をほぼ回避することが可能である。

モジュールローカルなシンボル

モジュールローカルなシンボルを得るには、シンボルの先頭に##を付加する。##は入力マクロであり、パーサがそれをモジュールローカルなシンボルに変換する。

```
(defparameter ##local_data ...)
```

ただし、##local_data がモジュールローカルであるためには、事前にモジュールを識別する情報を字句解析器に与えておく必要がある。
もっとも簡単な方法としては、

```
#.""
```

とする。この入力に対してパーサは値を返さないことに注意。
この場合、その都度モジュール識別情報が他のどのモジュール識別情報とも重複しないように自動更新される。

```
#.""
[##a <- 100] ;(defparameter ##a 100)の中置式による表現
(print ##a)
DISPLAY> 100
(defun get-local-a () ##a)
#.""  ;モジュール識別情報の再設定
(bound? '##a)  ;==> #f   モジュール識別情報が固有の値に再設定されたため
(get-local-a) ;==> 100    get-local-aは再設定前に定義されているため
```

プログラムの実行中に複数回ロードされるようなモジュールであれば、上記の例のようなことが起こると都合が悪い。#.""を字句解析器に渡す度に、同じモジュールでもまったく違うローカル識別子が生成されるからだ。
そのような場合、モジュール識別情報を明示的に与えればよい。次の例では、module: fooという識別情報を与えている。

```
#."module: foo"
```

識別情報は文字列リテラルと同様の形式で記述できるので、例えば改行を含めて説明文のように使ってもよい。

```
#."
 module: foo
 author: xxx
"
```

簡単な例:

```
#."test-1"
[##a <- 'local-a-test-1]
#."test-1" ;同じ識別情報で再設定
(print ##a)
DISPLAY> local-a-test-1
#."test-2" ;異なる識別情報で再設定
(bound? ##a) ;==> #f
[##a <- 'local-a-test-2]
(print ##a)
DISPLAY> local-a-test-2
#."test-1" ;最初の識別情報に再び戻す
(print ##a)
DISPLAY> local-a-test-1
```

このように、同一のモジュール識別情報を与える限りにおいては、生成されるモジュールローカルなシンボルは常に一定であることが保証される。モジュールローカルなシンボルは、どのような方法を用いてもユーザレベルでは intern 不可能な名前のシンボルであるため、モジュール識別情報さえ重複しなkれば、名前の衝突は起こりえない。従って、複数のモジュール

にまたがる関連するモジュール群を記述する際には、同一のモジュール識別情報をモジュールの最初に設定しておけば、複数モジュール間にまたがる共通のモジュールローカルな関数･マクロ･グローバル変数といった資源を安全に共有できる。ただし、上記の保証は実行時においてのものであり、同じ識別情報を与えたからといって、プログラムを再起動して(symbol-name ##a)の値が再び同じになる保証はどこにもない。モジュールローカルなシンボルの名前は知る必要もないし、知ろうとする行為はおそらく無意味だ。

パッケージ向けシンボル

Ag ではシンボルの先頭以外には:を含めることができるため、関数などのまとまった機能群をパッケージとして外部に供給する際には、C++風の

FooLib::bar

といった命名法を用いるとわかりやすく名前の衝突も起こりにくい。

モジュール内で、このような外部供給用のシンボルを扱う際には、事前に

```
#.FooLib
```

としておくと、

```
'#<bar ;==> FooLib::bar
```

のように、#>bar という入力に対して FooLib::bar というシンボルが返されるようになる。#>はモジュールローカルなシンボルを与える##と同様の入力マクロである。

もちろん、

```
(defin #<bar ...)
```

のかわりに、明示的に

```
(defun FooLib::bar ...)
```

としても何ら問題はない。

ただし、

#>bar

で得られるシンボルを明示的に入力することはできない。入力マクロ#>はパッケージローカルなシンボルを生成する。パッケージローカルなシンボルは、モジュールローカルなシンボルと同様に、intern 不可能であるから、

#.FooLib

としてあるモジュールの中では、パッケージ共通のローカルな資源を名前衝突の恐れなしに共有することが可能となる。

＊入力マクロ･字句マクロ＊

```
' Exp --> (function Exp)
` Exp --> (back-quote Exp)
, Exp --> (unquote Exp)
,@ Exp --> (unquote-splice Exp)
#' Exp --> (function Exp)
#? Exp --> (? Exp)
#& Exp --> (delay Exp)
#* Exp --> (force Exp)
```

```
#: Exp --> (unique Exp)

#\C --> 文字Cの値

Sym1.Sym2 --> (cref Sym1 'Sym2)
Sym1.Sym2.Sym3 --> (cref (cref Sym1 'Sym2) 'Sym3)
...

Sym1..Sym2 --> (static-cref Sym1 'Sym2)
Sym1..Sym2..Sym3 --> (static-cref (cref Sym1 'Sym2) 'Sym3)
...

.と..はいくらでも混在してよい
Sym1..Sym2.Sym3 --> (cref (static-cref Sym1 'Sym2) 'Sym3)



Sym.#[ Exps... ] --> (@ Sym Exps...)
Sym.Sym2.#[ Exps... ] --> (@ (cref Sym 'Sym2) Exps...)

(Exp1 Exp2). Exp --> (Exp Exp1 Exp2)
(Exp1 Exp2 Exp3). Exp --> (Exp (Exp Exp1 Exp2) Exp3)
(Exp1 Exp2 Exp3 Exp4). Exp --> (Exp (Exp (Exp Exp1 Exp2) Exp3) Exp4)
...
Ex)
 (1 2 3).-  ;==> -4 ;;(- (- 1 2) 3)

(Exp1 Exp2).. Exp --> (Exp Exp1 Exp2)
(Exp1 Exp2 Exp3).. Exp --> (Exp Exp1 (Exp Exp2 Exp3))
(Exp1 Exp2 Exp3 Exp4).. Exp --> (Exp Exp1 (Exp Exp2 (Exp Exp3 Exp4)))
...
Ex)
(1 2 3)..- ;==> 2 ;; (- 1 (- 2 3))

{ }. { }..は ( ). ( ).. と同じである。

[Exp1 Exp2]. Exp --> [Exp1 Exp Exp2]
[Exp1 Exp2 Exp3]. Exp --> [Exp1 Exp Exp2 Exp Exp3]
...
Ex)
[1 2 3 4 5].+  ;==> 15 ;; [1 + 2 + 3 + 4 + 5]

(apply #'s::cat ["foo" "bar" "baz"]."-")
[Exp1].. Exp --> [Exp Exp1 Exp]
```

```
[Exp1 Exp2].. Exp --> [Exp Exp1 Exp Exp2 Exp]
[Exp1 Exp2 Exp3].. Exp --> [Exp Exp1 Exp Exp2 Exp Exp3 Exp]
...
[+ * - /].10 ;==> 109  ;; [10 + 10 * 10 - 10 / 10]
```

＊マクロ・関数・クロージャ＊

Ag においては、マクロは評価前に呼ばれるということを除いては関数と同一の扱いである。よって、通常の関数と同様に名前によって関数オブジェクトを取得することも可能である。

```
(defmacro mac (x) `(defun ,x () ',x))

(funcall #'mac 'foo) ;==> (defun foo () 'foo)
(function? #'mac) ;==> #t
```

ただし、マクロは特別な属性を持つ関数であるので術語 macro?に反応する。

```
(macro? #'mac) ;==> #t
```

マクロでない関数オブジェクトは macro?には反応しない。

```
(macro? (lambda ())) ;==> #f
```

Ag では次のような定義は認められない。

```
(defun foo ((a b c) d &rest e) (list a b c d e))
```

しかし、Ag ではマクロ定義と関数定義を区別しないためこれでよい。
呼び出し例として

```
(foo '(1 2 3) 4 5 6)  ;==> (1 2 3 4 (5 6))
```

となる。
なお、Ag では便宜上、クロージャという語は関数オブジェクト一般(名前付き関数･無記名関数･マクロ)を指し、必ずしもフリーレキシカル変数を束縛しているとは限らないことに注意されたい。

＊スコープ＊

Ag は静的スコープと動的スコープを併せ持つ。
動的スコープとなる局面は次の通り。
1) 動的変数の参照
2) defun,defmacro,lambda 等の関数定義 (関数オブジェクト生成)
3) eval による式評価
これらの局面では、それらが適用された場所におけるレキシカルスコープが参照される。

1) 動的変数の参照
動的変数とは、関数オブジェクト生成時に静的に解決できない変数である。
レキシカル変数とフリー変数は静的に解決されるため、これら以外の変数があらわれた場合それは動的変数である。

次の例において、d のみが動的変数である。

```
(let ((a 'a))

 (defun foo (b)
  (let ((c 'c))
   (list a b c d))))
```

この場合、

```
(defparameter d 'global-d)

(foo nil) ;==> (a nil c global-d)
(let ((a 'new-a) (c 'new-c) (d 'new-d)) (foo nil)) ;==> (a nil c new-d)
```

このように、動的変数はそれが参照されるスコープによって参照先が変わる。

2) 関数オブジェクト生成時

例えば、CL 等であれば

```
(defparameter x 'global-x)

(let ((x 'local-x))
 (defun foo() (lambda () x)))
(funcall (foo)) ;==> 'local-x
```

となるが、Ag では

```
(funcall (foo)) ;==> 'global-x

(let ((x 'other-x))
 (funcall (foo))) ;==> 'other-x
```

となる。
これは、foo 中の λ 式は foo が呼び出された場所のレキシカルスコープに従って決定されるからである。
これを簡潔に示すために、ここでは expand というユーティリティを用いる。
expand はS式を受け取り、評価直前の形に展開された式を返す。
例えば
(expand '`(,(cadr x) ,(cddr y))) ;==> `(,(car (cdr x)) ,(cdr (cdr y)))
といった具合だ。
これを関数オブジェクト生成オペレータを含む式に適用する。

```
(expand '(let ((a (cadr x)))
           (lambda () (cddr a))))
;==> (let ((a (car (cdr x))))
    (lambda () (cddr a)))
```

λ 式の中の cddr マクロが展開されていないことがわかる。

このように、ALの仕様では、defun や lambda など、関数オブジェクトを生成するオペレータの内部は、それが必要とされたときに初めて展開され、関数オブジェクトが生成される。

さきほどの例を CL と同じ結果にしたい場合、ローカルな x を foo 中に明示的にインポートしてやればよい。

```
(let ((x 'local-x))
 (defun foo() (import-let (x) (lambda () x))))
(funcall (foo)) ;==> 'local-x
```

この時、import-let はスペシャル変数 x の実体をレキシカルな変数環境へと引きずり込んでいる。
import-let によって宣言されたレキシカル変数 x の実体は、スペシャル変数 x の実体と同じ。つまり、どちらも同じ実体のポインタを指していることになる。
したがって、その中で定義された foo は、スペシャル変数 x の実体を束縛するレキシカルクロージャとなることができ、その中で生成される λ 式中の x をスペシャル変数 x の実体にリンクさせることができるのである。

;;; λ 式の落とし穴

```
(defmacro mac () ''mac-1)
(defun fundef () (defun foo () (mac)) (foo))
(fundef)
(defmacro mac () ''mac-2)
(fundef)

(defmacro mac () ''mac-1)
(defun fungen () (funcall #'(lambda () (mac))))
(fungen)
(defmacro mac () ''mac-2)
(fungen)
```

＊真偽値＊

nil と#f を偽値とする。偽値でない全ての値は真値である。
真値は無限にあるが、明示的な真値を記述するため#t が用意されている。
なお、nil と#f は共に偽値であるが、同一ではない。

```
(false? nil) => #t  (false? #f) => #t
(eq? nil #f) => #f
```

#f は純粋な意味で偽を表し、nil は空リストの意味を含む。

```
(null? nil) => #t  (null? #f) => #f  (list? #f) => #f
```

この差が意味を持つのは、リストを返す関数の設計とその運用に限られるだろう。
論理値を扱う場合、通常は偽値として nil と#f のどちらが返されても問題ないようコーディングすべきである（例えば偽値に対する述語には null?ではなく false?を使う等）。

真偽値に関する1引数述語
null? f? false? not
t? true?

＊拡張されたバッククオート式＊

Ag は、CL や scheme と同様にバッククオート式をサポートする。
ただし、いくつかの拡張がなされており、入り組んだマクロをより直感的に記述することが可能となっている。
まず、やや複雑なマクロ定義の典型的として、グレアムの"On Lisp"で例題としてあげられている次のマクロ定義を例とする。

```
(defmacro abbrev (short long)
 `(defmacro ,short (&rest args)
   `(,',long ,@args)))
```

この典型的な「マクロ定義のためのマクロ」は Ag でもそのまま使えるが、Ag ではさらに次のように書くこともできる。

```
(defmacro abbrev (short long)
 `(defmacro ,short (&rest args)
   `(\,long ,@args)))
```

,',が\,に置き換わっているだけである。しかし\は大きな意味を持つ。

ここで、ネストされたバッククオート式の挙動について、ひとまずおさらいしておきたい。バッククオート作用子に対応するアンクオート作用子(,と,@)は「括弧」と同様、左端と右端から内にむかって順に対応する。
したがって、変数 a にシンボル foo が束縛された条件のもと、

```
``(,',a)
```

という式は、評価されると、まず（最適化されないかぎり理論上は）

```
`(,'foo)
```

となり、これがもう一度評価されると

```
(foo)
```

となる。この場合、ネストされたバッククオート式の中で a の中身を展開するために二段階を踏んでいることがわかる。この方法だと、ネストされたレベルだけ同じことを繰り返さなければならない、という事にもなりかねない。

Ag の\も、,や,@と同様にアンクオート作用子である。しかし、\は必要であれば評価後に消滅することも選択できるし、消滅しないことも選択できる。,や,@は、それがアンクオート作用子として直接に作用したときにのみ消滅できることとは性質が大きく違う。\はアンクオート作用子として直接作用しなくても、消滅するかしないか（あるいはいつ消滅するかという存続期間までも）を記述できる。

```
``(\,a)
```

は、評価されると一気に

```
`(foo)
```

となる。\は評価後に消滅している。また、いくらネストしていても平気である。

```
`````\\\\,a    ;==>  ````foo
```

この演算子\が「マクロを定義するマクロ」のような複雑なマクロを記述する際の労力を相当量軽減することは明らかであるが、より複雑なマクロの記述のためにも有用だ。

変数 foo にシンボル bar、変数 bar にシンボル baz が束縛された条件のもと、次の式はどうなるだろうか。
```

```

＊大域変数・特殊変数＊

大域変数と局所変数は明確に区別される。
未束縛の大域変数に値を束縛するには、setq ではなく defvar や defparameter もしくは gsetq や中置演算子<-を用いる必要がある。
いったん値を束縛された大域変数に対しては、局所変数と同様に setq や setf で値の変更が可能である。

大域変数のうち、*で始まる(あるいは##*と#<*と#>*で始まる)変数は特殊変数となる。Ag に於いて、特殊変数へのアクセスはあらゆるスコープに対して動的に行われる。

(defparameter *svar* 'foo)
(defparameter gvar 'bar)
(defun test () (list *svar* gvar))
(test) ;==> (foo bar)
(let ((*svar* 100) (gvar 200))
 (test)) ;==> (100 bar)
(test) ;==> (foo bar)

＊アクセサ＊

アクセサ(accessor)とは、参照子としての基本機能の他に、次の3つの操作を可能とした作用子のことである。

1)束縛
(... :access-to-bind value)
value を束縛する。
例として、(car x :access-to-bind y)は[(car x) = y]に等価。
2)ポインタ取得
(... :access-to-addr)
参照先のポインタを得る。
例として、(car x :access-to-addr)は[& (car x)]に等価。
3)適用
(... :access-to-apply fnc)
ー引数関数 fnc に参照値を適用した結果を、新たに束縛させる。
例として、(car x :access-to-apply (lambda (x) (1+ x)))は[++ (car x)]に等価。

これらの詳細について、アクセサを実装しないならば具体的に知る必要はない。setf 等のマクロがこれらを隠蔽しているからである。

＊多値＊
```

多値を返すには関数 values を用いる
(values mainval [val2 ...])
第一引数として渡された値が主値となる。
ただし、返すことのできる値の数には制限がある（主値を含めて最大で17）

多値中の特定の値の補足 - values::capture
values::capture 特殊作用子は、返された多値から任意の値を捕捉する。
(values::capture (values 'a 'b 'c) 0) ;==> a
(values::capture (values 'a 'b 'c) 2) ;==> c
存在しなければ nil が返される。
(values::capture (values 'a 'b 'c) 3) ;==> nil

インデクスを指定しなければ、スルーされる。
(values::list (values::capture (values 'a 'b 'c))) ;==> (a b c)

インデクスが複数指定された場合、返り値は多値となる。また、インデクスは重複しても構わない。
(values::list (values::capture (values 'a 'b 'c) 2 0)) ;==> (c a)
(values::list (values::capture (values 'a 'b 'c) 1 0 1)) ;==> (b a b)

多値のリストへの変換 - values::list
values::list 特殊作用子を用いれば、返された多値をリストとして取得できる
(values::list (values 1 2 3)) => (1 2 3)
values::list はまた、リストとして取得する値を選択することもできる。
(values::list (values 'a 'b 'c 'd 'e) 0 2 4 6) ;==> (a c e nil)
第2引数以降は0から始まるインデクスである。対応するインデクスのとおりに
多値からリストが生成され、域外の値については nil とされる。
なお、このインデクスは任意の順でもよいし、同じインデクスが重複しても構
わない。
(values::list (values 'a 'b 'c 'd 'e) 3 2 1 3) ;==> (d c b d)

多値の新変数への束縛 - values::bind
(values::bind (vardecl1 ...) mvalues-exp [exp ...])
values::bind 特殊作用子を用いることで、新らしい局所変数を宣言し、多値のそれぞれの値を位置的に対応する変数に束縛することができる。これは意味的に CL の multiple-value-bind に等価な機構である。

(values::bind (a b c) (values 1 2) (list a b c)) ;==> (1 2 nil)

多値の判定 - values::multiple?
values::multiple?特殊作用子は多値に対する述語である。

```
(values::multiple? (values 1 2)) ;==> #t
(values::multiple? (values 1) ;==> #f
(values::multiple? 1)) ;==> #f
```
この述語は多値を返す。
```
(values::list (values::multiple? (values 1 2))) ;==> (#t (1 2))
(values::list (values::multiple? 1) ;==> (#f 1)
```

例として、標準ライブラリの prog1 マクロと等価な表現を示す。
```
(defmacro prog1 (exp1 exp2 &rest rest_exp)
 `(let (,#:?multiple ,#:result)
   ,exp1
   (values::bind (?multiple result) (values::multiple? ,exp2)
    (setq ,#:?multiple ?multiple
        ,#:result result))
   ,@rest_exp
   (if ,#:?multiple
     (apply #'values ,#:result)
    ,#:result)))
```

＊遅延評価＊
scheme 流の遅延評価オペレータ(delay,force)をサポートする。
delay は遅延評価のためのプロミスオブジェクトを生成する。
force は、force の最初の呼び出し時におけるプロミスの評価値を与える。二度目以降の呼び出しにおいては、キャッシュされた最初の呼び出し時の値を与える。
force にプロミスでない値を与えた場合、与えられた値をそのまま返す。

```
(let* ((a 0) (b 1) c
    (promise (delay [c = a + b])))
   c                ;==> nil
   [a = 100]
   (force promise)  ;==> 101
   c                ;==> 101
   [b = 200]
   (force promise))  ;==> 101

(force 0) ;==> 0
```

force は多値を扱うこともできる。

```
[p <- (delay (values (print 'hello) 'world))]
p.#force ;==> hello
```

```
DISPLAY> hello
(values::list p.#force) ;==> (hello world)
```

述語 promise?はプロミスオブジェクトに対し真値を返す。
```
(promise #&[a + b]) ;==> #t
```

述語 forced?は、プロミスオブジェクトを引数に取り、force によって評価されていれば真を返す。
```
(forced? #&[a + b]) ;==> #f
```

＊特殊オペレータ＊
* defvar
```
(defvar var exp [reset=#f])
```
defvar は大域変数 var を定義し、exp を評価した値を束縛する。
第三引数は省略可能で、真値を渡した場合には、すでに定義済みであっても値を書き換える。デフォルトは偽であり、すでに定義済みの大域変数については何も行わない。
返り値は var に束縛されている値である。
なお、CLと同様の defparameter マクロも用意されているが、ただし、こちらも返り値は評価値であることに注意。

* defconst
大域定数を定義する。
書式は defvar に同じで、すでに定義済みの大域定数に対しても値の書き換えが可能である。
Ag においては、定数は defconst 以外での値の書き換えが禁止された変数のことを意味する。

* if
```
(if test a [b=nil])
```
test を評価し、真値なら a を偽値なら b を評価する。CL 互換。

* repeat
```
(repeat exp1 ...)
```
exp1 以下を右に順次評価し、終端にたどり着けば再び先頭の exp1 から同様の評価を繰り返す。
この無限ループは後述の return 特殊オペレータによって実行を終結できる。

* return
(return [retval=nil]) によって、retval を戻り値として repeat 構文を抜けることができる。

* tagbody
```
(tagbody {tag|exp} ...)
```
tagbody は、左から右に式を順次評価する。
ただし、ここで式とはコンスであり、コンスでなければそれはタグであるとみなされる。
戻り値は、最後に評価された式である。ただし、評価する式がない場合、あるいは最後のタグ以降に式が評価されなければ nil が返される。(CL非互換)。
タグへのジャンプは後述の go 特殊オペレータによって行える。

[ex]
(tagbody) => nil
(tagbody label-1 (progn 10)) => 10
(tagbody (progn 10) label-1) => nil
(tagbody (+ 1 2)) => 3
(let ((x 0))
 (tagbody (go tag2) tag1 [x = x + 1] tag2 [x = x - 1])
 x) ;==> -1

* go
(go tag)
tagbody 中の対応するタグにジャンプする。

* block
(block label [exp ...])
block は、exp 以下を順に評価し、最後に評価された値を返す。
概念的にはラベル付きの progn である。
後述の return-from によって対応するラベルの block 構文から抜けることができる。
CL互換。

* return-from
(return-from label [retval=nil])
対応するラベルを持つ block 構文から retval を戻り値として抜ける。
CL互換。

* list-bind
list-bind は、Common Lisp の multiple-value-bind に相当する。
ラムダリストと全く同じ表現が可能である。
(list-bind (a (b c &optional (d 0)) &rest e)
       '(1 (2 3) 4 5)
           (list a b c d e))   ;==> (1 2 3 0 (4 5))

* error
(error [val ...])
エラーを生成する。引数が与えられた場合、各引数に show を適用することにより得られる文字列が、エラーメッセージにスペースで区切られた形で付加される。

＊間接ポインタ＊
間接ポインタは低レベルの機構であり、適用には極めて注意が必要であることに注意されたい。
ポインタといっても、必ずしも具体的な計算機上のアドレスを指しているわけではない。間接ポインタに対し、後述する直接ポインタはアドレスを必ず直接指す。
間接ポインタは参照先のオブジェクトをバインドしたりしない低レベルの静的オブジェクトである。よって、ポイント先のオブジ

ェクトが消滅した後にもポインタはそのまま残る。そのようなポインタを扱えばどうなるか。C++プログラマには自明であるので説明する必要はないだろう。
ポインタは危険であっても極めて有用である。が、当然のことながら、ポインタは完全に把握可能な閉じた文脈で使用されるべきである。

addr で変数のポインタを取得し、pref で参照する。
Cで言えば、addr は&単項演算子、pref は*単項演算子に相当する。
なお、pref はアクセサである。

```
(let* ((a 0) (ptr (addr a)))
    (pref ptr) ;==> 0
    (pref ptr :access-to-bind 100)
    a)          ;==> 100
```

同じことを、中置記法においては単項演算子*と&によって以下のようによりわかりやすい表現に置き換えることができる。

```
(let* ((a 0) (ptr [& a]))
    [* ptr]          ;==> 0
    [* ptr = 100]
    a)             ;==> 100
```

単項&演算子を用いれば、変数だけでなく、car や cdr といったアクセサに対しても同様の表現で間接ポインタを取得できる。ただし、アクセサが間接ポインタを返せるかどうかはアクセサの仕様による。間接ポインタを返せない場合はエラーが発生する。

```
[a <- '(1 2 3)]
[p <- [& (car a)]]
[* p = 100]
a ;==> (100 2 3)
(setq a nil) ;;リスト消滅
[* p = 10] ;;NG おそらくセグメンテーション違反で死ぬ
```

＊スコープ関連＊

;スコープオブジェクト生成
* save-scope copy-scope
現在の局所スコープ(ネストしている局所スコープも含む)を保存またはコピーする。
保存した場合、現在のスコープに属する局所変数への参照が保存される。
コピーの場合は、局所変数に束縛された値がコピーされる。
返値はスコープオブジェクト。
ex) [scp = (save-scope)]

* save-scope-without copy-scope-without

局所スコープを保存またはコピーし、スコープオブジェクトを返す。
ただし、スコープオブジェクトに含めたくない変数名を (複数) 指定できる。
指定する変数名は、そのスコープに実際に存在しなくてもかまわない。
[scp = (let (a b c) ... (save-scope-without a c))] ; aとc以外の局所変数を保存

;スコープ適用
*scope
(scope scope-object-exp [exp ...])
scope-object-exp を評価して得られるスコープオブジェクトによって得られるスコープを現在の局所スコープに加える。
ただし、これは動的に追加されるスコープであるため、追加されたスコープに属する変数を参照するためには dynamic-let を用いてあたかも普通の局所変数であるかのように扱うか、あるいは fetch 特殊オペレータによってその変数のポインタを取る必要がある。
ex)
```
[scp <- (let ((a 0) (b 0))
      (defun status () (list a b))
      (save-scope))]
(defun foo () (scope scp (dynamic-let (a b) (list [-- a] [++ b]))))
(status) ;==> (0 0)
(foo)    ;==> (-1 1)
(foo)    ;==> (-2 2)
(status) ;==> (-2 2)
```

* scope'
scope と同様の形式であるが、変数の値の変化をスコープオブジェクトに反映させない。
ex)
```
[scp <- (let ((a 0) (b 0))
      (defun status () (list a b))
      (save-scope))]
(defun foo () (scope' scp (dynamic-let (a b) (list [-- a] [++ b]))))
(status) ;==> (0 0)
(foo)    ;==> (-1 1)
(foo)    ;==> (-1 1)
(status) ;==> (0 0)
```

;動的な変数参照
* dynamic-let
(dynamic-let (var ...) exp ...]
scope 構文の内側で用いることで、動的レキシカルスコープ内の変数を、あたかも最初から存在するレキシカル変数であるかのように参照させる。
ex)
```
[scp <- (let ((a 'other-a) (b 'other-b)) (copy-scope))]
(let ((a 'lexical-a) (b 'lexical-b))
```

```
 (scope scp
   (dynamic-let (a b) (list a b))))  => (other-a other-b)
```
なお、これはスコープオブジェクトによる動的スコープだけによらない。
```
(defun foo () (dynamic-let (a b) (list a b)))
(let ((a 'a) (b 'b)) (foo)) ;==> (a b)
```
同じことは eval によっても実現できるが、特定の変数捕捉が目的であれば
dynamic-let を用いるほうが高速である。
```
(defun foo () (eval '(list a b)))
(let ((a 'a) (b 'b)) (foo)) ;==> (a b)
```

* fetch
```
(fetch symbol-exp)
```
symbol-exp を評価して得られる変数名(シンボル)を現在のレキシカルスコープから検索し、対応する変数へのポインタを返す。変数が見つからなければ nil を返す。
```
[scp <- (let* ((a 'other-a)) (save-scope))]
(let ((a 'lexical-a))
 [* (fetch 'a)]      => lexical-a
 (scope scp
  [* (fetch 'a)]    => other-a
  (fetch 'b)))      => nil
```

* blind-let blind-let*
let 構文であり、形式は let 及び let*と同じである。
異なるのは、blind-let と blind-let*は、その外側のレキシカルスコープを隠蔽する働きがあることである。
```
[a <- 100]
[b <- 200]
(let ((a 0))
 a     ;==>  0
 (blind-let ((b 1))
  b   ;==>  1
  a   ;==>  100
 ))
```
なお、初期化式の中では外側のレキシカルスコープを 53C2 照できる。
よって、特定のレキシカル変数のみを参照可能にするには、
```
[a <- 100]
(let ((a 0) (b 1) (c 2))
 (blind-let ((b b) (c c))
  (list a b c)))            ;==> (100 1 2)
```
のようにすればよい。

＊割り込み＊
エラーや例外などを Ag では割り込みとして扱う。

割り込みはまた、動的末尾再帰や大域ジャンプなどにも使われる。
＊例外＊
Ag における例外とは、任意のタグと任意の値で構成される値の組によってなされる割り込みである。
例外による割り込みは、catch や try によって捕捉することができる。

＊エラー＊
エラーは通常、不正な引数が渡されたり構文に従わない入力パターンが与えられた場合などに発生する割り込みである。
(+ 1 nil) ;==> エラー発生

エラーを発生させる - error
エラーを意図的に発生させるには、error 特殊作用子を用いる。
(error [vals...])
ここで、vals は任意の値の組である。与えられた値は、それぞれ関数 show によって文字列化され、それらを空白で連結した文字列をエラーの詳細としてエラーメッセージに加える。
例えば、
(error 'illegal "parameter")
であれば、"illegal parameter"という文字列がエラーメッセージの一部となる。

エラーの例外化 - throw-if-error
エラーを、例外のようにそのままの形で捕捉することはできない。そのかわり、エラーを任意の例外に変換することはできる。
throw-if-error 特殊作用子は、第1引数を評価し、そこでエラーが発生した場合、エラー割り込みを捕捉し、第2引数をタグ、第3引数を値とする例外を発生させる。
(throw-if-error exp [tag [val]])
ここで、tag と val は省略可能であり、tag が省略された場合は、デフォルトタグである 'error が渡されたとみなされる。val が省略された場合は、エラーメッセージを内容とする文字列がその値となる。
exp の評価中にエラーがおこらなければ、返り値は exp の評価値となる。
(throw-if-error [1 + 2]) ;==> 3
(throw-if-error [1 + nil] 'fail nil) ;==> シンボル fail をタグとし,nil を値とする例外が発生

catch
(catch tag-exp exps...)
catch 特殊作用子は、tag-exp を評価して得られる例外タグに一致する例外を捕捉する。例外が捕捉された場合、直ちに exps...の残りの評価を中断し、例外の値を戻り値として直ちにリターンする。
なお、例外タグが nil の場合、全ての例外を捕捉する。

unwind-protect
(unwind-protect exp [rest-exp...])
exp を評価し、次いで rest-exp 以下を順次評価する。
exp の評価中に割り込みが発生した場合、すべての評価が終わった後、割り込みを再開する。
そうでなければ、unwind-protect は exp の評価値を戻り値とする。
なお、rest-exp 以下の評価中に割り込みが発生した場合には、exp の評価結果(exp 中での割り込みの有無)にかかわらず、処理を中止し新たな割り込みに従う。

例:
```
(catch 'tag (unwind-protect (throw 'tag 'foo) (print 'bar))) ;==> foo
DISPLAY> bar
```

protect
```
(protect [exp...])
```
protect 特殊作用子は exp 以下を順次評価する。
ただし、それらの評価中にいかなる割り込みが発生しようともそれらの割り込みは全てブロックされる。つまり、protect 特殊作用子の呼び出しにおいてはいかなる割り込みも継続できない。
戻り値は割り込みの発生回数である。
例:
```
(protect (print 'hello-world)) ;==> 0
DISPLAY> hello-world

(protect (error "err1") (print 'hello)
     (error "err2") (print 'world)) ;==> 2
DISPLAY> hello
DISPLAY> world
```

try
```
(try trying-exp {[tag-exp [catch-exp...]]} [finally-exp...])
```
try 特殊作用子は、まず tag-exp を評価し捕捉すべき例外タグを得る(タグが nil の場合すべての例外に一致する)。
次に trying-exp を評価する。ここで例外が発生し、catch 節のタグに一致すれば catch-exp...以下を順次実行し、その後 finally 節に移る。捕捉できない例外であったり、あるいは割り込みであるばあいには、そのまま finally 節に移り、finally 節を抜けると同時に、割り込み状態に復帰する。何れの場合にも、finally 節は必ず実行されることに注意されたい。

catch 節や finally 節で新たな割り込みが起こったらどうなるか？その場合は、即座に新たな割り込み状態となる。

catch 節の catch-exp...以下では、暗黙に宣言された変数__exctag__と__excval__を参照することができる。これらは、捕捉された例外のタグと値を拘束している。
例外の発生の如何によらず、finally 節は必ず実行される。なお、finally 節では、暗黙に宣言された変数__interrupting__を参照できる。これは、例外が捕捉されれば偽であり、割り込みが継続中であれば真である。
finally 節の最後の式は、割り込みの継続中でなければ末尾再帰の対象となる。割り込み継続中であれば末尾再帰とはならない。なぜか？それは末尾再帰は割り込みだからである。これは通常使う上では意識することはない。

try の返り値は、finally 節の最後の式の評価値である。

＊ハッシュ表＊
ハッシュ表は、オブジェクトのハッシュ値を元に構成される辞書である。
ハッシュ表からキーを検索するにはインデクサを用いる。

(@ table key)
インデクサは、キーが見つかればデータと#t を組とする多値を返す。
キーが見つからなければ nil と#f を組とする多値が返される。つまり、
(values::capture (@ table key) 1)
は、キーが存在すれば真、存在しなければ偽だ。

```
[ht <- (make-hash-table)]   ;;(make-hash-table #'eq)に等価
ht.#['key] ;==> 多値 nil #f
[ht.#['key] = 'data]
ht.#['key] ;==> 多値 data #t
[str <- "hello"]
[ht.#[str] = 'world]
ht.#[str] ;==> 'world
ht.#["hello"] ;==> nil

[ht <- (make-hash-table :test #'Eq?)]
[ht.#[str] = 'world]
ht.#["hello"] ;==> world
ht.#["Hello"] ;==> nil

[ht <- (make-hash-table :test #'EQ?)]
[ht.#[str] = 'world]
ht.#["Hello"] ;==> world
[ht.#['hello] = "world"]
(maphash #'(lambda (key data) (print (list key data)))
      ht) ;==> nil
DISPLAY> ("hello" world)
DISPLAY> (hello "world")
```

＊アレイ＊
アレイは1次元以上8次元以内の多次元配列オブジェクトである。
アレイに対する参照はインデクサを用いる。
(@ array index..)
C で array[x][y][z]とするのと等価な表現は
(@ array x y z)
である。字句マクロを用いれば、より簡潔に
array.#[x y z]
と書くことができる。

```
[a <- (make-array 10 :set 0)] ;;要素数 10 の一次元配列の生成
a.#[0] ;==> 0
[a.#[0] = 100]
a.#[0] ;==> 100
```

```
[a <- (make-array 4 5 6)] ;;要素数(4,5,6)の3次元配列
a.#[0 1 2] ;==> nil
[a.#[0 1 2] = 100]
a.#[0 1 2] ;==> 100
```

```
array::new
array::define-index-rule 位置決めのための定数項と係数項の設定
array::size
array::set 指定値で埋める
array?
```

＊構造体＊

```
(defstruct foo
  a
  (b 0)
  (c 1))

[obj = (make-foo :a 'hello)]
(foo-a obj) ;==> 'hello
[(foo-a obj) = 100]
(foo-a obj) ;==> 100
obj.#foo-a  ;==> 100
```

```
struct::new
struct::name
struct::copy
struct::assign
struct::refer
struct?
```

＊パーサ＊

```
parser?
parser::new
parser::read
parser::rewind
```

＊ストリーム＊

stream?
io::getb
io::getl
io::gets
io::gets'
io::putb
io::putl
io::puts
io::puts'

io::tell
io::seek-set
io::seek-cur
io::seek-end

C++との連携

＊STL コンテナテンプレート＊

C++とより深い連携を実現するため、統一的なインタフェースで STL コンテナテンプレートのインスタンスを C++と Ag の間で相互に利用できる。C++側では Ag のコンテナオブジェクトから生の STL コンテナクラス（へのポインタ）を取得できるため、通常のコンテナ利用とほとんど変わらないコーディングスタイルを維持でき、従って C++側の生産性についてはほとんど影響を及ぼさない。更に、コンテナに格納されるデータ型についても、簡単に C++側で任意に追加できる。

デフォルトで、標準 STL コンテナテンプレートの全てのコンテナクラスをサポートする。つまり、std::vector,std::deque,std::list,std::set,std::multiset,std::map,std::multimap である。これらのコンテナクラスは C++のテンプレートと総称の強力な能力を用いて、ag_base_t を基底とする Ag のコンテナオブジェクトのクラステンプレートのインスタンスとして巧妙に具現化されるが、ユーザはそのラッパの具体的実装について知っている必要は全くない。

次の C++と Ag のコード例では、std::vector<uint16_t>を相互に利用する。C++の側で完全に生の std::vector<uint16_t>を操作している。このためには、僅かに AG_CT_VECTOR_CAST マクロを用いるだけである。生のコンテナをラップしている Ag オブジェクトの実装を知らなくてもこれは意味的に直感的に使える。

C++:
```
AG_USE_CT_VECTOR(uint16_t); // std::vector<uint16_t>の利用宣言
AGDEFUN_BEGIN();
AGDEFUN("vec-u16-square",_vec_u16_square)
{
 CheckArgNum(1); CheckArgIsDynamic(0);
 std::vector<uint16_t> *vec16 = AG_CT_VECTOR_CAST(uint16_t, argv[0].get_dynamic());
 if( ! vec16 ){
  // キャスト失敗
 }else{
  for(std::vector<uint16_t>::iterator p=vec16->begin(); p!=vec16->end(); p++)
```

```
    *p *= *p;
   return argv[0];
 }
}
```

```
Ag:
[vec <- new ct-vector <uint16>]
(ct::push-back vec 2 4 6 1 3 5)
(ct::translate-to-list vec) ;==> (2 4 6 1 3 5)
(ct::translate-to-list (vec-u16-square vec)) ;==> (4 16 36 1 9 25)
```

このように、Agと C++との間では、Ag側で対応するコンテナであれば全て C++で生のままの型で

インデクサ

ct-list,ct-deque,ct-vector,ct-set,ct-multiset,ct-map,ct-multimap、これら全てのコンテナはインデクサを持つ。
下記において、アクセサとイテレータ取得となる手続きについては、未割当の場合でも必ずアロケートされるため、失敗はしない。

シークエンスコンテナ共通

seq.#[i] <- アクセサ
seq.#[i :iterator] <- イテレータ取得
seq.#[i :reverse-iterator] <-イテレータ取得

SET コンテナ

set.#[val] ;==> 存在すれば真
[set.#[val] = #t(または任意の真)] ;==> val を挿入。返り値は新規に挿入したかの真偽。
[set.#[val] = #f(または nil)] ;==> val を削除。返り値は実際に削除したかの真偽。
set.#[val :iterator] <- イテレータ取得
set.#[val :reverse-iterator] <-イテレータ取得

MAP コンテナ

map,#[key] <- アクセサ
map.#[key :iterator] <- イテレータ取得
map.#[key :reverse-iterator] <- イテレータ取得